SOBRE O COMPORTAMENTO DE OBJETOS

Sergio Eduardo Rodrigues de Carvalho
Departamento de Informática, Pontificia Universidade Católica RJ
Rua Marques de São Vicente 225, Gávea, Rio de Janeiro, 22453
Telefone: (55)(21)274-4449 Fax: (55)(21) 511-5645
e-mail: sergio@inf.puc-rio.br

RESUMO

O comportamento procedural, o único em geral disponivel em linguagens orientadas a objetos, é necessário na construção de programas. No entanto, outras semânticas de comportamento, bem conhecidas e úteis na programação convencional, não tem representação na maioria daquelas linguagens. Neste artigo propomos a inclusão, em classes, de operações para o tratamento de mensagens assíncronas, para o tratamento de exceções e para a produção de sequências de valores para objetos controladores de repetição, oferecendo exemplos de seus usos.

ABSTRACT

The procedural behavior, the only one usually available in object oriented languages, is necessary in the construction of programs. However, other well-known behavioral semantics in the conventional programming world have no counterparts in object orientation. In this paper we propose the inclusion, in classes, of operations to handle asynchronous messages, exceptions, and the production of sequences of values for loop control objects, offering examples of their uses.

INTRODUÇÃO

Sistemas de programação orientados a objetos tem sido recentemente considerados pelo menos como alternativas viaveis para a construção de programas de qualidade. Tais sistemas tipicamente contem uma linguagem de programação, um executor de programas, uma

biblioteca de modelos para objetos, e não raro são acompanhados de sugestões para o projeto de programas, como são os casos de Smalltalk [7,8] e TOOL [1,2].

O conceito central em uma linguagem orientada o objetos é sem dúvida o conceito de *classe*. Como sempre é o caso em disciplinas emergentes, no entanto, existem hoje inúmeras semânticas associadas a este conceito. Uma das mais comuns estabelece que uma classe é um modelo para objetos, e que este modelo especifica tanto a *estrutura de dados* dos objetos quanto o seu *comportamento*.

A estrutura de dados dos objetos é uma estrutura de composição, onde classes já existentes modelam campos, ou componentes, da classe sendo definida. O comportamento dos objetos é especificado pelas ações que devem ser executadas para cada estímulo que objetos da classe podem receber.

Na grande maioria das linguagens orientadas a objetos este comportamento é de natureza exclusivamente procedural, ou seja:

- a operação que origina o estímulo tem sua execução suspensa até que as ações correspondentes a este estímulo, especificadas na classe do objeto receptor, sejam executadas;
- o ambiente de referências relativo a estas ações tem espaço reservado no topo da pilha de execução;
- dados de controle, como endereço de retorno e elo dinâmico, tambem são guardados nesta pilha;
- o controle de execução é transferido para a ação a ser executada;
- quando estas ações terminarem, o controle é dado de volta à operação que gerou o estímulo.

É tambem possivel que a descoberta da operação a ser executada sobre um objeto só possa ser feita em tempo de execução, já que pode depender da classe em que o objeto se encontra no momento.

Os argumentos deste artigo são:

- a semântica procedural ou metódica, embora útil, não reflete suficientemente bem todos os estímulos que objetos podem receber;
- existem outras semânticas para operações, até bem conhecidas no mundo da programação procedural, que podem ser adotadas com vantagens na programação orientada a objetos, e que hoje não existem em linguagens orientadas a objetos;
- a orientação a objetos, devido ao conceito de classe, aceita com elegância e naturalidade a especificação de outros comportamentos para os objetos modelados.

Cada um destes pontos será discutido em mais detalhes abaixo. Inicialmente proporemos uma sintaxe e uma semântica para classes, que deverá ser adotada na modelagem que se segue. Em seguida outros comportamentos para objetos serão apresentados: respostas a mensagens assíncronas, à ocorrência de exceções, à produção iterativa de novos valores. Todos estes comportamentos já existem implementados em linguagens de programação convencionais; o único objetivo deste artigo é propor sua colocação no escopo da orientação a objetos de tal forma que programadores já habituados a seus usos os reconheçam no novo estilo de programação.

A familiaridade que se obtem com esta adaptação é importante no aprendizado de sistemas orientados a objetos e no entendimento de programas; afinal, queremos sistemas de programação úteis a programadores da vida real, e não sistemas que dificultem sua utilização.

UM MODELO PARA CLASSES

Adotaremos aqui o modelo de Wegner [14], onde uma classe é uma implementação de um tipo abstrato de dados que pode ser beneficiada por um mecanismo de herança. Além disso, uma classe é uma estrutura estática, importante somente em tempo de compilação, que facilita a verificação da validade dos estímulos que seus objetos recebem. Esta visão "estática" de classes é bastante conveniente para a popularidade e para a facilidade de uso de linguagens orientadas a objetos, já que assim classes podem ser apresentadas como meras extensões de tipos, um conceito familiar a programadores convencionais.

Para consolidar este modelo, sugerimos a definição de classe informalmente apresentada abaixo:

```
class nome-classe
        inherits nome-classe,...
        exports campos,... operações,...
        imports nome-classe,...
        structure
                campo,...
        behavior
                operação,...
end class
```

As cláusulas **inherits**, **exports** e **imports** são a interface da classe com o mundo exterior. A cláusula de exportação lista os campos e operações visiveis fora da classe. Na cláusula de importação são declaradas todas as classes usadas na classe sendo definida, seja para modelar campos desta classe, seja para modelar objetos locais ou parâmetros de operações. Na seção de estrutura, declaramos a composição de campos necessária para descrever o estado de um objeto. Na seção de comportamento, declaramos todas as operações aplicaveis a objetos da classe. É aqui que postulamos maior variedade, como discutido a seguir. Para o comportamento procedural ou metódico, adotaremos a sintaxe abaixo:

```
method nome-método ( parâmetro,... ) returns nome-classe;
        declaração local,...
begin
        comando,... (incluindo return [ when expressão ];)
end method
```

sendo opcional a cláusula **returns**. O comando **return**, possivelmente condicional, fornece uma saida abrupta do método, destruindo seu ambiente de referência.

O TRATAMENTO DE MENSAGENS ASSÍNCRONAS

A construção de programas que ao serem executados apresentam uma interface gráfica para usuários é cada vez mais popular. O sistema operacional Microsoft Windows [11] foi

construido especificamente para este tipo de aplicação, e encontra-se hoje bastante difundido. Em uma aplicação típica, o usuário pode a qualquer momento pressionar um botão ou fazer uma escolha de menu, por exemplo, assim interferindo assincronamente com a execução da aplicação.

Em sistemas operacionais desta natureza, existe tipicamente um tratamento de mensagens assíncronas que envolve uma fila de mensagens e uma estrutura de repetição que constantemente testa a fila sobre a existência de mensagens. Havendo mensagens na fila, uma delas é retirada e seu tratador é invocado para execução. A pressão sobre um botão ou uma escolha de menu são exemplos de ações de usuário que causam a presença de mensagens na fila.

A construção de um programa para um destes sistemas em geral exige do programador preocupações com a fila de mensagens e com a construção de uma rotina única para o tratamento destas mensagens, rotina esta que consiste basicamente de um comando de seleção múltipla condicionado pelas mensagens a serem tratadas. Na verdade este tipo de programa apareceu na década de 60, quando linguagens para a programação de sistemas operacionais apareceram no cenário. É aqui que a orientação a objetos pode propiciar uma solução de muito mais alto nivel para o tratamento de mensagens assíncronas. A seguir apresentamos este tratamento, introduzido originalmente na linguagem TOOL [3].

A motivação é basicamente a seguinte: já que classes existem para modelar objetos, por que não considerar a existência de operações, dentro de classes, para o tratamento de mensagens assíncronas? Estas operações modelariam o comportamento assíncrono de objetos, com uma implementação como a já existente em sistemas tipo Windows, como a fila de mensagens mencionada acima. A primeira vantagem deste enfoque consiste na pulverização da rotina centralizadora de tratamento de mensagens, já que estes tratamentos estariam distribuidos pelas classes adequadas. Com isto ganha-se em clareza de programas e na reutilização destas classes. A segunda vantagem é o desaparecimento da estrutura de repetição controladora da fila, e da fila propriamente dita: o comportamento assíncrono

especificado pelos tratadores declarados nas classes coloca em muito mais alto nivel a construção de programas.

Com um mecanismo destes implementado de forma extensivel, por exemplo, pode-se escrever um programa de folha de pagamento em que a impressão de relatórios se faz de forma assíncrona: quando a impressão de um relatório se faz necessária, a operação em execução envia uma mensagem ao objeto "folha de pagamento", que eventualmente responde executando o tratador para este fim especificado em sua classe. O assincronismo neste caso se justifica, já que a impressão de relatórios em geral não retorna nenhuma informação para a operação que iniciou a impressão, que só precisa uma garantia de que o relatório vai ser impresso.

A semântica básica deste tratador seria a seguinte:

- a operação que envia a mensagem não tem sua execução suspensa (pelo menos do ponto de vista conceitual);
- o tratador só pode receber parâmetros de entrada;
- o tratador não retorna objetos e nem retorna ao ambiente de onde foi invocado.

Estas precauções são necessárias já que, não havendo sincronismo entre a operação que envia a mensagem e seu tratador, é até possivel que o objeto responsavel pelo envio da mensagem deixe de existir antes do tratador ser executado.

Como sintaxe, a seguinte estrutura poderia ser adotada:

message handler for nome-mensagem (parâmetro,...) **from** nome-objeto,...;
    declaração local,...
begin
    comando,...
end handler

A cláusula **for**, na assinatura do tratador, indica uma mensagem que pode ser respondida por objetos desta classe, pela criação do ambiente de referência adequado e pela execução de seus comandos. Esta cláusula é obrigatória. A cláusula **from**, opcional, indica

objetos que podem mandar esta mensagem para o objeto receptor. Por exemplo, a classe abaixo poderia ser usada na criação de uma interface com uma janela e dois botões:

```
class Interface;
        representation
                Window      w;
                PushButton  ok, cancel;
        end representation

        method R;
        begin
                w <- CREATE;
                ok <- CREATE;
                cancel <- CREATE;
                ...
        end method

        message handler for Clicked from ok;
                locais e comandos
        end handler

        message handler for Clicked from cancel;
                locais e comandos
        end handler
end class
```

Quando o método R é aplicado a um objeto da classe Interface, os botões 'ok' e 'cancel' são criados. Isto estabelece, no sistema TOOL, uma relação de propriedade: o objeto da classe Interface é o "dono" dos botões, e assim responsavel pelo tratamento das mensagens que estes objetos mandam a seus donos [3]. Uma destas mensagens, no caso de botões, é Clicked. Como são dois botões de significados claramente diferentes, dois tratadores diferentes são necessários - esta a razão para a cláusula **from** na assinatura de tratadores.

Para complementar o tratamento de mensagens a nivel de classe, e para assegurar um alto grau no encapsulamento de classes, um indicador genérico de donos de objetos deve existir no sistema: desta forma, os objetos de uma classe podem mandar mensagens para seus donos sem saber quem eles são. Para exemplificar, podemos considerar uma calculadora contendo em sua interface um botão entitulado ENVIAR, que quando pressionado por um

usuário, envia o valor no mostrador da calculadora para o dono dela, sem saber se este dono é uma planilha, e o valor deve ir para uma célula, ou uma folha de pagamento, e o novo salário de um empregado acabou de ser calculado. Este comportamento da calculadora poderia ser especificado pelo tratador abaixo:

```
handler for Clicked from enviar;
begin
        owner <<- EnviarValor ( disp );
end handler
```

onde EnviarValor é o nome de uma mensagem enviada pela classe Calculadora, e 'disp' é o valor no mostrador. Neste sistema cabe ao dono da calculadora ter, em sua classe de definição, um tratador para a mensagem EnviarValor. A existência de sintaxe e semântica para o envio de mensagens a receptores genéricos permite a construção de programas reutilizaveis seja individualmente, seja como componentes em outras aplicações, sem a necessidade de alterar código [4].

O TRATAMENTO DE EXCEÇÕES

Aqui tambem queremos tratar, a nivel de classe, um comportamento de objetos: o comportamento excepcional. Antes de propor este mecanismo, apresentaremos certas observações sobre o tratamento de exceções que serão adotadas no nosso modelo.

Inicialmente vamos aceitar, como é o caso em algumas linguagens convencionais, como CLU [9] e Ada [13], que a ocorrência de uma exceção em uma operação R deve causar a terminação de R, incapaz de cumprir seu contrato, na terminologia de Eiffel [10]. Aceitamos tambem que esta terminação pode ser precedida pela execução de uma seção especial de código, ainda no ambiente da operação sinalizadora da exceção: um tratador local para a exceção acusada. Existindo este executor de desejos finais, a exceção desaparece e a execução do programa continua normalmente.

Em seguida vamos aceitar que, não havendo este tratador local, e não havendo tambem tratadores a nivel de classe, propostos abaixo, a responsabilidade pelo tratamento da

exceção reside com a operação que invocou a operação sinalizadora. Assim, a operação chamadora deve estar preparada para tratar as terminações normais e anormais de uma operação chamada.

Tratadores locais, para qualquer tipo de operação, poderiam ser especificados de acordo com o modelo abaixo, comum em linguagens convencionais:

```
operation nome-operação (parâmetro,...);
        declaração local,...
begin
        comando,... (incluindo raise nome-exceção;)
        exception
                when nome-exceção,... then comando,... (incluindo raise);
                ...
        end exception
end operation
```

Na falta de um tratador local, delimitado por **exception** e **end exception**, a operação sinalizadora termina sem últimos desejos, a exceção continua acusada, e um tratador em outro nivel deve ser procurado. É aqui que o conceito de classe pode proporcionar uma solução intermediária entre o tratamento local e o tratamento na operação chamadora: a existência de tratadores de exceção definidos em classes.

Este tratador, se existente, seria invocado na falta de um tratador local à operação sinalizadora. Em termos, este tratamento seria ainda "local": a exceção seria levantada ainda no escopo da classe, antes de ser propagada à operação chamadora. Estes tratadores seriam usados para colocar em evidência, no nivel da classe, tratamentos comuns a um conjunto de operações da classe: o que fazer, por exemplo, quando os limites de um "array" que faz parte da estrutura dos objetos da classe são violados.

Como sintaxe para tratadores de exceção a nivel de classe, o modelo seguinte poderia ser usado:

```
exception handler for nome-exceção ( parâmetro,... );
        declaração local,...
begin
        comando,...
end handler
```

A semântica deste tratamento pode ser explicada considerando-se a aplicação de uma operação R a um objeto 'x' e os casos abaixo:

- nenhuma exceção é acusada em R, que termina normalmente;
- uma exceção 'e' é acusada em R, que tem um tratador local para 'e': o código deste tratador é executado no ambiente de R, a exceção é cancelada, R termina e a execução do programa continua normalmente;
- uma exceção 'e' é acusada em R, que não tem um tratador local para 'e', mas existe um tratador de classe para 'e': R termina, o código deste tratador é executado no ambiente da classe, a exceção é cancelada, e a execução do programa continua normalmente;
- uma exceção é acusada em R, e não existe tratador local ou de classe para a exceção: R termina anormalmente, e, para operações esperando um retorno de R, a exceção é propagada.

A propagação de uma exceção para uma operação chamadora pode ser considerada como uma exportação da classe acusadora; assim as exceções que podem ser propagadas a partir de uma classe devem ser colocadas na cláusula de exportação desta classe. Em [5] podem ser encontrados mais detalhes sobre o tratamento discutido acima.

A SEMÂNTICA DE CO-ROTINAS

Nesta seção apresentamos duas situações em que a semântica de co-rotinas é útil à linguagens orientadas a objetos: na construção de iteradores para a produção de valores para objetos de controle de repetição, e na construção de navegadores para estruturas dinâmicas de dados. Nestas duas situações, o objeto receptor da operação controla um comando de repetição. As operações são aplicadas a este objeto para produzir, fora do escopo da repetição, o próximo valor para este objeto. Com isto o tamanho do corpo da repetição fica

reduzido, o que é vantajoso no entendimento de programas. Alem disto, a declaração de iteradores em classes aumenta a reusabilidade das classes.

A seguir caracterizamos em mais detalhes iteradores e navegadores:

- um *iterador* produz o próximo valor de um objeto controlador de repetição construindo este novo valor a partir do valor corrente e do seu ambiente de execução (referências locais e código);
- um *navegador* de estrutura de dados obtem o próximo valor de um objeto controlador de repetição deslocando um cursor sobre uma estrutura existente, usando a posição corrente do cursor e seu ambiente de execução (referências locais e código);

No caso de navegadores a disciplina de acesso da estrutura de dados é fundamental para o acesso ao próximo valor. Como exemplo de iterador, podemos considerar o construtor de progressões aritméticas que existe na maioria das linguagens convencionais. Estes iteradores produzem, sem intervenção do programador, o próximo valor em um intervalo especificado. Como exemplo de navegador, podemos considerar o percurso sobre uma lista simplesmente encadeada, que a cada repetição produz o próximo elemento da lista.

A exemplo do que fizemos nas seções anteriores, o objetivo neste caso é a colocação de operações, em classes, que realizem as construções e os percursos mencionados acima. No caso de navegadores, as classes receptoras devem modelar estruturas de dados de natureza dinâmica.

Para iteradores, a sintaxe abaixo poderia ser adotada:

```
iterator nome-iterador ( parâmetro,... );
        declaração local,...
begin
        comando,... (incluindo yield [ when expressão ];)
end iterator
```

Sobre a semântica de iteradores, observamos inicialmente que seus parâmetros devem ser exclusivamente de entrada, já que iteradores existem com o bem definido propósito de

produzir o próximo valor para um objeto controlador de repetição; assim, a produção de efeitos colaterais pelo iterador não é aconselhavel. Pela mesma razão, iteradores não retornam objetos, como pode acontecer com métodos: eles simplesmente alteram, a cada invocação, o valor do objeto controlador, que acessam livremente em seus comandos pelo indicador genérico **self**.

A semântica básica de iteradores é aquela de co-rotinas, descrita por exemplo em [6]. Iteradores devem conter o comando **yield**, que quando executado, apenas suspende a execução de um iterador, fornecendo um novo valor para o objeto controlador. O contexto de execução do iterador não é perdido, já que é importante para a obtenção do próximo valor; tambem não é perdido o acesso do iterador ao objeto controlador.

A execução de um comando **yield** passa o controle para o primeiro comando da repetição controlada pelo objeto receptor do iterador; este endereço é guardado no ambiente do iterador. Quando o iterador é invocado novamente, na passagem do controle pelo cabeçalho da repetição, o comando a ser executado é o seguinte ao último **yield** , e seu endereço está tambem guardado no ambiente do iterador, que é mantido entre ciclos de produção de valor.

Um iterador termina quando produz o último valor da sequência especificada pelo seus comandos. Isto tambem sinaliza o término da repetição. Uma proposta para a inclusão de iteradores em TOOL é descrita em [12], e é baseada na linguagem CLU.

A seguinte estrutura de repetição poderia ser associada ao uso de iteradores:

```
loop for objeto <- nome-iterador ( argumento,... );
        comando,...
end loop
```

Os cuidados usuais devem ser tomados em relação ao uso do objeto controlador nos comandos do corpo da repetição. É razoavel supor que o uso de um iterador em uma repetição estabelece uma disciplina de produção de valores para o objeto controlador que não

admite alterações deste valor no corpo da repetição; assim, o uso deste objeto deve ser somente para leitura.

Para exemplificar o uso de iteradores, podemos considerar uma classe que define pontos como um par de coordenadas cartesianas, e a existência de um iterador nesta classe que produz pontos, a partir da origem, sendo dado um deslocamento horizontal, até um ponto dado. Este iterador poderia ser usado na repetição abaixo, supondo as declarações dos pontos 'p' e 'q':

```
loop for p <- OrigemADestino ( q, 0.5 );
        comando,...
end loop
```

e o iterador seria declarado como se segue:

```
iterator OrigemADestino (in Point p, in REAL delta_x);
        REAL teta;
begin
        -- calcular a inclinação do segmento:
        teta := p.y / p.x;
        -- produzir primeiro valor para objeto controlador:
        self.x := 0.0;      self.y := 0.0;
        loop  -- para a produção de novos valores:
                yield;
                -- preparar novo valor:
                self.x := self.x + delta_x;
                exit when self.x > p.x;  -- último valor já produzido.
                self.y := teta * self.x;
        end loop
end iterator
```

Navegadores são co-rotinas especiais, existentes em classes modelando estruturas dinâmicas de dados, e com a finalidade não de construir um próximo valor, como é o caso de iteradores, mas de acessar o próximo elemento da estrutura, de acordo com uma disciplina definida pela estrutura. A semântica descrita para iteradores se adapta integralmente a navegadores. Estes, no entanto, necessitam, dentro da repetição que controlam, de um identificador genérico para o elemento da estrutura de dados que acessam no momento; por

exemplo, o identificador **current**. É por meio deste identificador que os comandos do corpo da repetição acessam, um de cada vez, os elementos da estrutura sendo iterada.

Assim, se o objeto 'lp' for declarado como uma lista de pessoas, e se na classe que modela listas o navegador PrimeiroAUltimo existir, a repetição abaixo processaria, a cada iteração, a próxima pessoa da lista:

```
loop for lp <- PrimeiroAUltimo ();
        comando,...
end loop
```

sendo pessoas, nos comandos da repetição, referidas por **current**.

CONCLUSÕES

Em linguagens de programação convencionais várias estruturas de controle a nivel de unidade, com semânticas diferentes, já foram implementadas e são de uso comum, uso este justificado pela necessidade de ajuste mais fino na execução de programas. Em linguagens orientadas a objetos, no entanto, isto em geral não ocorre. O tratamento de mensagens assíncronas proposto acima é um exemplo desta ausência, que se torna mais evidente quando consideramos que já existem as primitivas, em modernos sistemas operacionais, para sua implementação, e que sistemas dirigidos por interfaces gráficas, usuários naturais desta semântica, são hoje tão populares.

Além disso, a falta do reaproveitamento destes mecanismos em linguagens orientadas a objetos parece ser agravada pela existência de uma interpretação simples do conceito de classe como modeladora de objetos, que permite especificar, como métodos são em geral especificados, outros comportamentos para os objetos modelados, com sintaxe semelhante e semântica adequada. A linguagem de programação TOOL, projetada para facilitar a construção de aplicações que contem interfaces gráficas para usuários, já contem, em sua primeira versão, métodos e tratadores de mensagens assíncronas.

Se concordamos que esta variedade de comportamentos é pelo menos conveniente, resta saber de que maneira eles seriam introduzidos em linguagens orientadas a objetos. Neste artigo propomos sua inclusão de forma suave, tentando reaproveitar conhecimentos já existentes no mundo da programação convencional, de maneira a tornar produtivos mais programadores em menor tempo.

BIBLIOGRAFIA

[1] Carvalho, S. E. R. "The Object and Event Oriented Language TOOL", Monografias em Ciência da Computação, Departamento de Informática, PUC/RJ, 6/93.
[2] Carvalho, S. E. R. "Decoupling Interface and Implementation: the TOOL Solution", Monografias em Ciência da Computação, Departamento de Informática, PUC/RJ, 37/92.
[3] Carvalho, S. E. R. "Asynchronous Behavior in the TOOL Programming System", Monografias em Ciência da Computação, Departamento de Informática, PUC/RJ, 18/93.
[4] Carvalho, S. E. R. "On the Reuse of Applications as Components", Monografias em Ciência da Computação, Departamento de Informática, PUC/RJ, 7/93.
[5] Carvalho, S. E. R. "Exception Handling in Object Oriented Languages: a Proposal", Monografias em Ciência da Computação, Departamento de Informática, PUC/RJ, 23/92.
[6] Ghezzi, C., Jazayeri, M., *Programming Language Concepts*, Wiley, 1982.
[7] Golgberg, A., Robson, D., *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, Palo Alto, CA 1983.
[8] Krasner, G., Pope, S., "A Cookbook for Using the Model View Controller User Interface Paradigm in Smalltalk 80", JOOP Aug-Sept 88, pp 26-49.
[9] Liskov, B., Atkinson, R., Bloom, T., Moss, E., Schaffert, J., Scheifler,R., Snyder, A. *CLU Reference Manual*, Springer Verlag, Berlin, 1981.
[10] Meyer, B. *Object-Oriented Software Construction*, Prentice-Hall, Englewood Cliffs, NJ, 1988.
[11] Petzold, C., *Programming Windows*, Microsoft Press, Redmond, Wash, 1990.
[12] Porto, S. C.; Carvalho, S. E. R. "Introducing Iterators in TOOL", Monografias em Ciência da Computação, Departamento de Informática, PUC/RJ, 41/92.
[13] *Reference Manual for the Ada Programming Language*, United States Department of Defense, Jan 1983.
[14] Wegner, P. "Classification in Object Oriented Systems:, SIGPLAN Notices, v21 #10, oct 1986.