Smalltracer: Um verificador de tipos e condições para Smalltalk

Silvio R. L. Meira
srlm@di.ufpe.br

Paulo H. C. Lisboa
phcl@di.ufpe.br

Universidade Federal de Pernambuco
Departamento de Infomática
PO Box 7851
50.739 Recife - PE

Abstract

This paper discusses the difficulty of today's object oriented languages and programming environments to keep together some benefits of the paradigm, such as fast prototyping and linguistic unity in a wider range of stages of the software life cycle. A type and condition environment, Smalltracer is proposed here as a possible solution for some of the problems that occur in Smalltalk and in the Smalltalk/V environment.

1. Resumo

Este trabalho discute a dificuldade das presentes linguagens orientadas a objeto de manter juntos certos benefícios advindos deste paradigma, como prototipagem rápida e unicidade linguística num maior número possível de fases do desenvolvimento. Apresenta-se aqui uma ferramenta, Smalltracer, como possível solução deste problema em Smalltalk.

2. Introdução

O aparecimento e domínio das linguagens orientadas a objeto teve como principal motivo os benefícios advindos deste paradigma, os quais não existiam nos métodos tradicionais de desenvolvimento de sistemas como, por exemplo, o modelo cascata e as linguagens a ele associadas. A seguir estão algumas das principais desvantagens dos modelos tradicionais:

1. Dificuldade de alterar um sistema concluído.
2. Existência de uma rígida sequência de etapas a serem seguidas, não refletindo o verdadeiro dinamismo do desenvolvimento.
3. Ausência de suporte ao desenvolvimento evolucionário via prototipagem e evolução.
4. Dificuldade na classificação das entidades ou componentes do sistema para reutilização futura.

O modelo orientado a objetos, por sua vez, supre essas carências, ao mesmo tempo que incorpora vantagens como:

1. Facilita o diálogo entre usuários e analistas por usar entidades (objetos) comuns à cognição humana.
2. Facilita a mobilidade entre as diferentes fases do desenvolvimento por usar um conceito único (objeto) em todas elas.
3. Provê um retorno rápido ao solicitante, uma vez que a codificação já é iniciada nas primeiras fases do desenvolvimento.

PDF compression, OCR, web optimization using a watermarked evaluation copy of CVISION PDFCompressor

4. Diminui o risco de não haver nenhuma implementação em um possível estouro de prazo de entrega.

Essas não são todas as vantagens do modelo, mas são suficientes para que se entenda a prototipagem rápida e a unicidade linguística, em várias etapas do desenvolvimento, como dois dos seus principais benefícios.

Os ganhos com a prototipagem rápida (além das vantagens 3 e 4 acima) podem ser melhor avaliados quando observamos o tempo gasto, no desenvolvimento de um sistema, com o levantamento dos requisitos e suas eventuais modificações, ao meio e final desse desenvolvimento. Um sistema de computação deve se adequar à necessidade que o gerou, porém nem sempre isto é realidade pela falta de conhecimento ou experiência computacional das pessoas que solicitam o sistema, que têm dificuldade de expressar o que realmente precisam. A prototipagem rápida se torna, então, a maneira mais simples e eficiente de resolver esse problema, uma vez que os solicitantes do sistema podem verificar se o protótipo reflete ou não a solução dos seus problemas, sem perda substancial de tempo pela equipe de desenvolvimento.

A unicidade linguística (vantagens 1 e 2) é uma extensão natural da unicidade conceitual (orientação a objeto) e é defendida em [HeE90], onde Henderson enfatiza que o uso de uma única linguagem de programação, que englobe o maior número de fases do desenvolvimento de um sistema, diminui o tempo total de desenvolvimento, assim como facilita a manutenção e extensão (flexibilidade).

3. Contexto Atual

A proliferação das linguagens orientadas a objeto e a incorporação, com pequenas variantes, desse conceito por várias linguagens tradicionais, promoveu o surgimento de formas diferentes de manipular e controlar os objetos. Todas as linguagens, entretanto, mantém os benefícios básicos advindos da orientação a objeto, com ênfase em um ou outro aspecto.

Existem linguagens como C++ e OWL, que se cercam de sintaxe de declaração de tipos para poder garantir um programa mais seguro, através de tipos fortes[1], todavia menos flexíveis à prototipagem.

Outras linguagens, como Eiffel vão mais além na sua preocupação com segurança e corretude[2] ao prover, além de tipos, um mecanismo de checagem de condições (assertions), como invariantes de classe, pré-condições, pós-condições, etc.[3] Meyer afirma, em [Mey88], que classes devem ser implementações de tipos abstratos de dados, onde o relacionamento entre elas e seus clientes precisa ser visto como um acordo formal (programação por contrato) que expresse os direitos e obrigações de cada um. Entretanto, para que esse contrato seja mantido, devemos reduzir os riscos de discrepância entre as implementações (classes) e suas especificações (tipos abstratos), ou seja, incorporar características dentro da linguagem ou ambiente de programação que cheguem a permitir, se possível, escrever especificações na própria linguagem (unicidade linguística).

Existe ainda, em oposição às anteriores, um grupo de linguagens orientadas a objeto que clamam a velocidade e flexibilidade na prototipagem de sistemas como sua principal característica. Essas linguagens têm tipos fracos, ou simplesmente não têm tipos, sendo Smalltalk e Self seus principais representantes.

4. A Necessidade

Visto a pluralidade de linguagens e enfoques, e as respectivas vantagens e desvantagens de cada uma, é claro que uma linguagem única, que agrupasse as diferentes virtudes das várias outras já citadas, seria ideal em muitos aspectos. Wegner, em [Weg90], entretanto, mostra como é difícil, para um ambiente de programação, fazer o casamento da flexibilidade, necessária para a prototipagem rápida, com a segurança exigida nos sistemas mais complexos.

---

[1] *Tipos fortes* é a característica de uma linguagem de obrigar que as variáveis e expressões de um programa resultem, apenas, em instâncias dos seus tipos declarados. Não interessando se essa checagem é feita em tempo de compilação ou de execução [Weg90]

[2] *Corretude* é a qualidade de um programa de estar de acordo com os seus requisitos.

[3] Eiffel também aceita assertions em *loops* e instruções *check*.

Autores como Hagman, em [Hag82], e Borning, em [Bol82], tentam unir, de forma perigosa, características existentes nas linguagens orientadas a objetos. Perigosa porque existe, nesses trabalhos, uma supervalorização dada à eficiência, em detrimento da clareza e flexibilidade. Hagman, por exemplo, propõe uma declaração (opcional) de tipos que não necessariamente deve ser respeitada pelo programador, caso seja, ganha-se eficiência. O perigo, portanto, é que o programador, ou cliente do programa, pensa estar trabalhando com um tipo quando, na verdade, pode não estar.

É comprovada, pela experiência de uso, e é de bom conhecimento de todos, as vantagens da prototipagem rápida em linguagens como Smalltalk (enfatizado em [GoR89] e [Dig91]) e Self [ChU91] [UnS87]. Não devemos, portanto, tolher essa virtude na linguagem que buscamos. Mas, não podemos, também, esquecer as dificuldades que aparecem em linguagens flexíveis quando o sistema deixa de ser um protótipo, ou um pequeno sistema, e evolui para um sistema de médio ou grande porte. Nessa situação o entendimento e administração do projeto normalmente se tornam caótico e o sistema será, então, propenso a ter uma série de defeitos não detectados na sua prototipagem que aumentarão exponencialmente com a sua evolução, por culpa, principalmente, da flexibilidade encontrada nessas linguagens de prototipagem.

[Hoa84] e [MSC89] mostram os benefícios advindos do uso de métodos formais para especificar sistemas em desenvolvimento, principalmente, quando esses são de médio ou grande porte. Eiffel tenta incorporar características que propiciem o uso desses métodos.

O conflito, entretanto, do uso desses métodos é que eles seguem uma metodologia muito rígida, propondo-se a iniciar o desenvolvimento de um sistema pela especificação dos seus requisitos, o que não é simples e está propenso a sofrer várias alterações posteriores, caso a prototipagem, numa fase posterior à especificação, comprove que os requisitos não refletem a real necessidade do usuário.

Booch é categórico ao afirmar, em [Boo91], que métodos rígidos de projetar sistemas promovem a progressão de erros. Isto por causa das dificuldades de se mudar a seqüência de desenvolvimento estabelecida nos modelos clássicos onde, ao se achar um erro na concepção do sistema, tentamos retardar a sua solução, ou o resolvemos de uma forma apenas paliativa, não impedindo seu reaparecimento ao final do desenvolvimento.

Dentro de um enfoque flexível para o desenvolvimento, pode-se dizer que até mesmo tipos são um empecilho na fase inicial. O uso de tipos aumenta a preocupação com a hierarquia das classes e a limitação que essa hierarquia causa nas entidades do programa, como variáveis, parâmetros, etc., pois as mesmas só poderão guardar instâncias do seu tipo, ou subtipos. O custo destas preocupações não é atraente nesta fase, uma vez que mudanças radicais no projeto podem ser necessárias após a prototipagem.

5. Uma Solução

Para tirarmos um melhor aproveitamento dos benefícios que cada linguagem apresenta, propomos uma modificação no processo de desenvolvimento de sistemas, além da união de alguns desses benefícios em uma única linguagem.

Na modificação da metodologia de desenvolvimento, propomos que requisitos provenientes do levantamento dos dados sejam descritos de uma forma simples, talvez informal, suficiente apenas para que se monte um protótipo do que se entende pelo sistema. Uma vez validado esse protótipo, caso o sistema final venha a ser de médio ou grande porte, devemos tentar enriquecer o protótipo com tipos e condições[4] que lhe garantam uma maior segurança operacional e semântica (fig. 1). A partir daí, podemos crescer gradativamente o sistema, com a certeza de termos controle sobre o que já foi desenvolvido. Não se deve esquecer que, ao incluir condições numa classe e respectivos métodos, estamos aumentando a convicção dela representar o tipo abstrato que se imagina e requer [Mey88]. Essa metodologia, inclusive, mantém a unicidade lingüística, pois como em Eiffel, permite escrever especificações na própria linguagem, dado que a fase de prototipagem já passou.

A linguagem de desenvolvimento proposta portanto é uma onde:

[4] *Condições*: invariantes, pré e pós-condições.

PDF compression, OCR, web optimization using a watermarked evaluation copy of CVISION PDFCompressor

1. O programador trabalha com uma sintaxe simples, mas poderosa, que permite adiar a discursão sobre tipos e condições para uma fase posterior do desenvolvimento.

2. Não existe demora nas compilações sucessivas que o desenvolvimento inicial (protótipo) de qualquer programa exige. Esse problema não existe nas linguagens interpretadas e é amenizado nos compiladores incrementais.

3. Existe um bom depurador para eventuais erros de programação[5].

As duas possibilidades básicas para obter uma linguagem com tais caracteristicas são:

- Trazer a flexibilidade das linguagens de prototipagem para alguma linguagem de médio ou grande porte, tarefa difícil visto a complexidade dos compiladores dessas últimas.

- Criar, em alguma linguagem de prototipagem, um framework que lhe adicione os mecanismos de segurança encontrados nas linguagens de maior porte.

A segunda possibilidade é mais interessante pela facilidade que algumas linguagens de prototipagem apresentam de alterar o seu próprio ambiente. Smalltalk foi a linguagem eleita para tal.

6. Smalltalk

Smalltalk é uma das primeiras linguagens orientadas a objeto e, provalvelmente, a que mais influenciou na disseminação dos conceitos de orientação a objeto.

Mais do que uma simples linguagem, Smalltalk é um ambiente de programação composto de uma sofisticada interface gráfica e ferramentas como editores, browsers de classe, vistoriadores de objetos, depuradores on-line, além de uma extensiva biblioteca de classes pré-definidas. Nesta linguagem, tudo são objetos e até mesmo classes seguem essa regra [Gor89].

[5] Esse trabalho não tem a intenção de entrar no mérito dessa questão, apesar de Smalltalk ter um excelente depurador de erros.

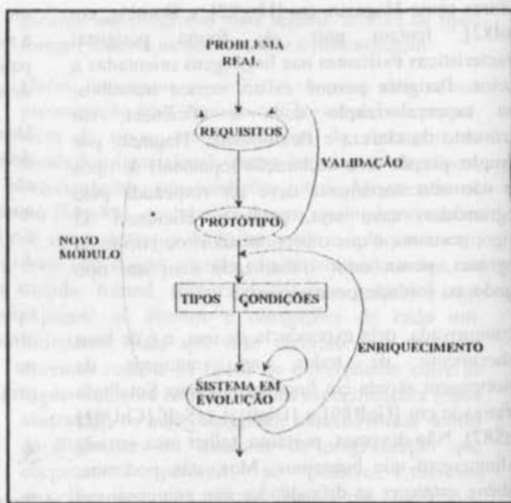

Figura 1

Smalltalk usa um interpretador que executa métodos semi-compilados.

7. Smalltracer

Criada para prover mecanismos opcionais de segurança a programas Smalltalk[6], a ferramenta Smalltracer provê um framework que não exige nenhuma alteração destes programas.

Smalltracer aceita declaração de tipos em variáveis de instância, classe, parâmetros e retorno de método, além de declarações de invariante de classe, pré-condição e pós-condição. Por razões que explicaremos a seguir, chamamos a pré-condição e pós-condição de pre-code e post-code na implementação do Smalltracer.

[6] Smalltracer está implementado em Smalltalk/V for Windows da Digitalk.

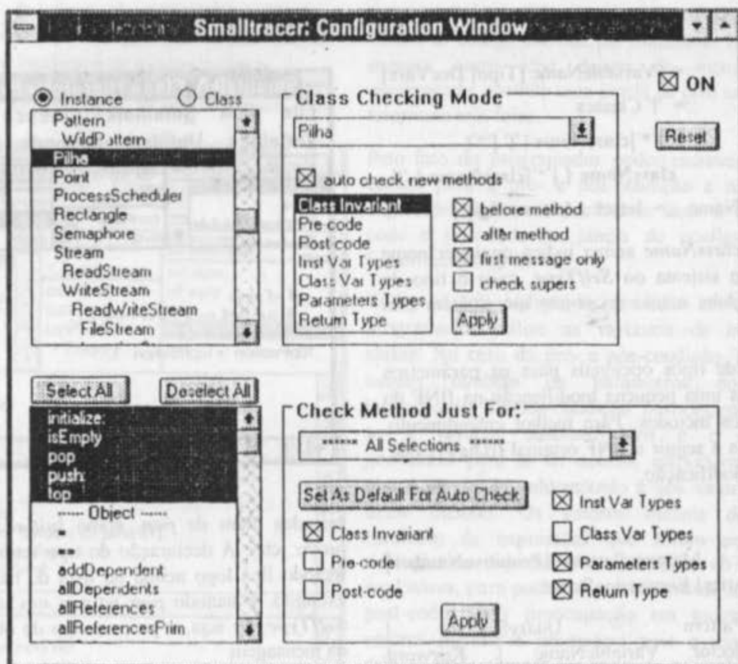

Figura 2

Operacionalmente, toda a idéia por detrás da ferramenta está na interceptação de mensagens selecionadas, previamente, pelo usuário [Bol190]. Podemos assim fazer a verificação dos tipos das variávies (instância, classe, parâmetros e retorno) e das condições da classe (pré-condição, pós-condição, invariante de classe) antes e depois da ativação do método.

A figura 2 mostra a janela de configuração do Smalltracer. Essa janela nos permite ajustar certas opções de funcionamento do Smalltracer à nivel de classe e método.

A parte superior da janela é responsável pelas opções relativas à classe. Podemos ajustar opções como invariante de classe e tipos das variáveis de instância e classe para serem feitos antes e/ou depois da ativação dos métodos interceptados pelas classes selecionadas, essas opções também permitem a verificação do que foi herdado das superclasses. Para todas as opções de funcionamento dessas classes, entretanto, pode-se optar pelas checagens apenas na primeira mensagem interceptada de cada objeto numa cadeia de ativações de métodos[1]. Existe também a opção de interceptação automática de métodos novos com modo de checagem determinado a partir de um padrão pré-estabelecido.

Na parte inferior da janela, temos as opções referentes aos métodos. Para os métodos selecionados como interceptados, pode-se escolher que checagens serão realizadas dentre várias opções, como invariante de classe, pré-condição, pós-condição, tipos das variáveis de instância, etc..

A BNF para declaração de tipos das variáveis de instância e classe é provida a seguir

[1] *Cadeia de ativação de métodos* é a sequência de métodos ativados quando uma mensagem é passada para um objeto

PDF compression, OCR, web optimization using a watermarked evaluation copy of CVISION PDFCompressor

- DecVars ::= {VariableName [Tipo] DecVars}
- Tipo ::= '!' Classes
- Classes ::= ['*']className | '(' ['*']
  className { |'*'|className } ')'
- VariableName ::= **letter { letter | digit }**

O símbolo ***className*** acima indica qualquer nome de classe do sistema ou ***SelfType***, caso o tipo da variável englobe o tipo do objeto que contém esta variável.

A inclusão de tipos opcionais para os parâmetros gerou apenas uma pequena modificação na BNF do cabeçalho dos métodos. Para melhor entendimento, apresentamos a seguir a BNF original ([Dig91] pag. 197) e sua modificação.

<u>Original</u>:

- Method ::= MessagePattern [PrimitiveNumber] [Temporaries] ExpressionSeries
- MessagePattern ::= UnarySelector | BinarySelector VariableName | Keyword VariableName {Keyword VariableName}

<u>Modificação</u>:

- MessagePattern ::= UnarySelector | BinarySelector VariableName ***[Tipo]*** | Keyword VariableName ***[Tipo]*** {Keyword VariableName ***[Tipo]***}

Percebemos pelas BNFs acima que mais de um tipo pode ser declarado para uma mesma variável ou parâmetro. Ainda mais, um tipo equivale implicitamente à declaração dos seus subtipos, sendo que o programador pode opcionalmente colocar um asterisco (*) à esquerda do tipo para indicar que ele é absoluto e não equivale a declaração dos seus subtipos.

Exemplos de declarações de tipos são dados nas figuras 3 e 4. Essas duas figuras apresentam uma extensão do folheador original de classes do Smalltalk.

A figura 3 mostra o método ***pop*** da classe Pilha. Essa classe tem ***contents***, ***max*** e ***topPosition*** como variáveis de instância e possui também alguns outros

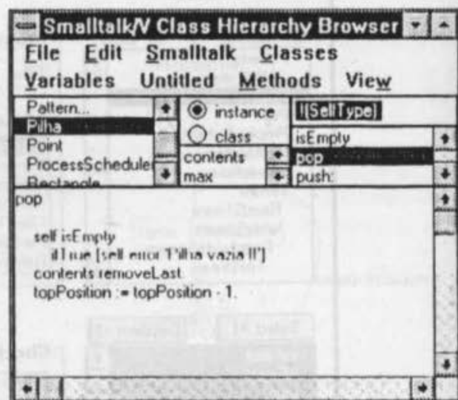

Figura 3

métodos além de ***pop***, como ***initialize:***, ***isEmpty***, ***push:***, etc.. A declaração do tipo retornado por um método fica logo acima da lista de métodos. Nesse exemplo, o método ***pop*** retorna um objeto do tipo ***SelfType***, ou seja, o próprio tipo do objeto receptor da mensagem.

A figura 4 exemplifica a declaração de tipos das variáveis de instância (área hachurada) e do invariante de classe (apresentado mais adiante) de Pilha.

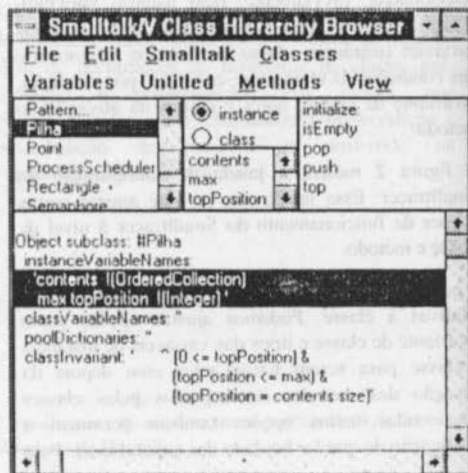

Figura 4

PDF compression, OCR, web ... evaluation copy of CVISION PDFCompressor

A declaração de tipos é opcional: variáveis, parâmetros e retorno de métodos sem declaração de tipos armazenam instâncias de qualquer classe.

Invariantes de classe, pré- e pós-condições são

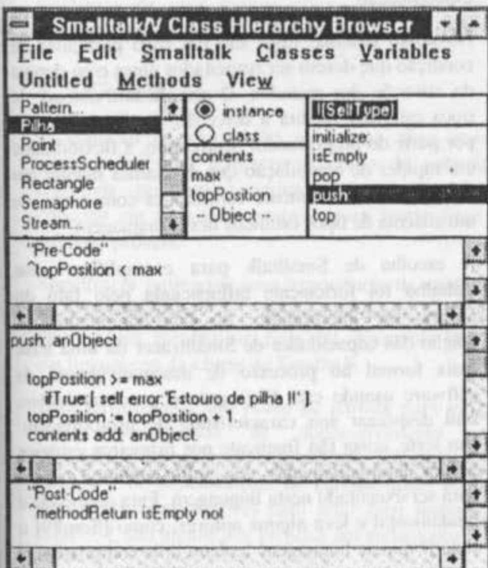

Figura 5

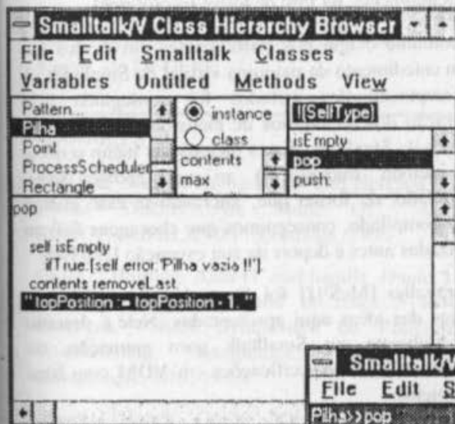

Figura 6a

implementados na ferramenta como expressões ou blocos de código escritos em Smalltalk. Esse código executa antes e/ou depois do método sendo interceptado, abrindo uma janela de erro caso o valor retornado seja falso.

Pelo fato do programador poder escrever qualquer código para a pré- e pós-condição e não apenas expressões boleanas, preferimos usar o nome pre-code e post-code na janela de configuração do Smalltracer (fig. 2).

Todos os três códigos de condição aceitam que o programador utilize as variáveis de instância e classe. No caso da pré- e pós-condição, podem ser usados também os parâmetros do método interceptado. Na pós-condição, particularmente, usa-se a variável ***methodReturn*** e o dicionário ***preValues*** para se ter acesso, respectivamente, ao objeto retornado pelo método e aos valores iniciais desse método. Os valores iniciais devem ser resultado de expressões que foram previamente avaliadas no pre-code e armazenadas no dicionário ***preValues***, para poderem posteriormente ser lidos no post-code. Essa preocupação em guardar certos valores iniciais é necessário pois a execução do método pode alterará-los antes da ativação do post-code.

A adição da opção ***View*** no menu principal da janela do folheador de classes permite modificar o pane de declaração de métodos e o da lista com o nome dos mesmos para que eles abram espaço para a declaração do pre/post-code e do tipo do valor retornado, respectivamente.

A figura 5 exibe um exemplo de declaração da pré- e pós-condição para o método ***push:*** da classe Pilha.

Na figura 6b temos o exemplo de uma janela de erro que se abriu pela violação do invariante de classe após a execução do método *pop* da classe Pilha. A causa do erro foi forçada por comentar a última linha do método (fig 6a).

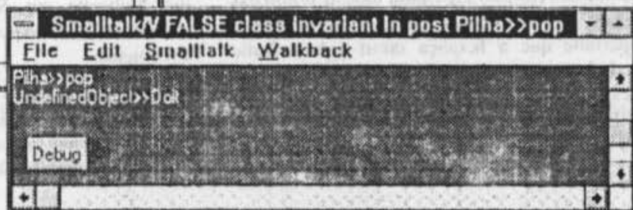

Figura 6b

PDF Compression, OCR, web optimization using a watermarked evaluation copy of CVISION PDFCompressor

Uma virtude da ferramenta, que complementa a funcionalidade já apresentada, é o fato desta ser ortogonal a Smalltalk e funcionar independentemente deste. O botão ON na janela de configuração (fig. 2) é o responsável pela ativação ou desativação de Smalltracer. Nenhum tipo, invariante de classe, pré- ou pós-condição é checado por Smalltalk quando Smalltracer está desativado. Smalltracer pode ser ativado e desativado durante uma sessão, sem prejuízo ou perda das declarações de tipos, invariantes ou pré/pós-condições existentes.

Não há declaração de tipos para as variáveis locais que, como o próprio nome indica, têm seu uso tão bem delimitado que não chegam a oferecer perigo para o programa como um todo. Para que um objeto armazenado por uma variável local saia do escopo do respectivo método dessa variável, ele tem de ser usado como 1) retorno do método, 2) parâmetro de alguma mensagem ou 3) ser armazenado em uma variável de 3.1) instância ou 3.2) classe. Qualquer uma destas formas possibilita ao Smalltracer checar o tipo e as condições que devem vigorar para este objeto. Existe ainda a possibilidade dos objetos serem transferidos via variável global ou de *poolDictionaries* e não terem portanto seus tipos checados. Esta possibilidade, todavia, não está sendo considerada por não configurar uma "boa" programação orientada a objetos, além do que casos específicos podem ser tratados pela pré/pós-condição.

8. Algumas Considerações

Tanta flexibilidade na declaração de tipos (tipos como opcionais, união de tipos, subtipos implícitos ou não, etc.) é necessária e provida por Smalltracer porque Smalltalk viola muitas das assumpções nas quais se baseia a maioria dos sistemas de tipos de linguagens orientadas a objetos e, conseqüentemente, precisa de recursos que não restrinjam a expressividade dos seus programas.

O uso da declaração *SelfType* para indicar o tipo das variáveis de um objeto como sendo o próprio tipo do objeto permite que a herança ou o polimorfismo funcionem de uma forma mais sadia.

A maioria dos sistemas de tipos para linguagens orientadas a objetos equipara tipos a classes. No nosso caso, tipos são baseados em classes (cada classe define um tipo) mas subclasse não tem necessariamente relação direta com subtipo. Isso se explica porque, freqüentemente, as classes em Smalltalk herdam implementação ao invés de especificação.

No nosso sistema, tipos são um caso particular de condição que devem ser respeitados antes e/ou depois da ativação dos métodos. O uso de um sistema de tipos estáticos facilita a detecção imediata de erros por parte do programador. Entretanto, a flexibilidade e a rapidez de compilação que desejamos manter em Smalltalk são totalmente conflitantes com o uso de um sistema de tipos estáticos nessa linguagem.

A escolha de Smalltalk para exemplificar esse trabalho foi fortemente influenciada pelo fato do mesmo ser interpretado, ou semi-compilado[2]. A adição das capacidades de Smalltracer dá uma base mais formal ao processo de desenvolvimento de software usando esta linguagem e o seu ambiente, sem desprezar sua característica de prototipagem. Um teste, coisa tão freqüente nos primeiros estágios de um desenvolvimento, não sofre nenhuma espera para ser executado nesta linguagem. Esta qualidade é fundamental e leva alguns autores, como [Som89] a sugerir que as linguagens tenham dois compiladores, um para desenvolvimento e outro para otimização, por causa do tempo gasto em compilação, principalmente, na fase de testes (protótipos).

O trabalho exigiu dos participantes envolvidos um bom entendimento da máquina virtual de Smalltalk. A interceptação dos métodos foi conseguida pela alteração dos dicionários de métodos existentes nas classes de Smalltalk. Esses dicionários ligam o nome do método (mensagem) ao seu código semi-compilado, de forma que, alterando-se esse código semi-compilado, conseguimos que checagens fossem realizadas antes e depois da sua execução [BoH90].

O trabalho [McS91] foi fonte de inspiração para muitas das idéas aqui apresentadas. Nele é descrito um ambiente em Smalltalk para animação, ou prototipação, de especificações em VDM com base em objetos.

---

[2] Smalltalk executa métodos semi-compilados que se apresentam numa forma mais próxima da linguagem de máquina que o código fonte do Smalltalk.

PDF compression, OCR, web optimization using a watermarked evaluation copy of CVISION PDFCompressor

Extensões desse trabalho para declarar tipos e condições nos objetos, e não apenas nas classes, já estão sendo consideradas. As classes parametrizáveis, em linguagens tipadas, apresentam correlação com essa idéia por permitirem que instâncias de uma mesma classe usem diferentes tipos nas suas variáveis de mesmo nome.

9. Conclusão

O conceito de orientação a objetos abre novas perspectivas para o desenvolvimento de sistemas. Devemos, portanto, experimentar e explorar esse conceito em toda sua extensão para tentar tirar o máximo proveito.

Smalltracer representa uma experiência de mudança na metodologia de desenvolvimento e nos mecanismos disponíveis para tal desenvolvimento, em linguagens da classe de Smalltalk.

Essa ferramenta é um ponto de partida para várias outras ferramentas de desenvolvimento que planejamos incorporar ao Smalltalk. Diagramas de objetos e de dependências de classes, além de um framework mais flexível para lidar com instâncias [GoR91], são alguns dos projetos em andamento.

Agradecimentos

Agradecemos as valiosas sugestões e discussões com José Fernando Tepedino Martins e Cássio Souza dos Santos, do DI-UFPE.

Bibliografia

[BoH90] Böcker, Heinz-Dieter and Herczeg, Jürgen "What Tracers Are Made Of", ACM ECOOP/OOPSLA'90 Proceedings, 1990.

[BoI82] Borning, Alan H. and Ingalls, Daniel H. H., "A Type Declaration and Inference System for Smalltalk", Ninth Symposium on Principles of Programming Languages, pag 133-141, albuquerque, NM, 1982.

[Boo91] Booch, Grady: "Object Oriented Design with applications", The Benjamin/Cummings, 1991.

[ChU91] Chambers, Craig and Ungar, David: "Making Pure Object Oriented Languages Practical", OOPSLA'91, 1991.

[Dig91] Digitalk Inc.: "Smalltalk/V Windows Tutorial and Programming Handbook", Digitalk Inc., January 1991.

[GoR89] Goldberg, Adele and Robson, David: "Smalltalk-80 the language", Addison-Wesley, 1989.

[GoR91] Gold, Eric and Rosson, M. B.: "Portia: An Instance-Centered Environment for Smalltalk", ACM - OOPSLA'91 Proceedings, 1991.

[Hag82] Hagmann, Robert: "Preferred Classes: A proposal for Faster Smalltalk-80 Execution", Smalltalk-80: Bits of History, Words of Advice, Glenn Krasner Editor, Addison-Wesley, 1983.

[HeE90] Henderson-Sellers, B. and Edwards, Julian M.: "Object Oriented Systems Life Cycle", Communications of the ACM, vol. 33, No. 9, September 1990.

[Hoa84] Hoare, C. A. R.: "Programming: Sorcery or Science?", IEEE, April 1, 1984.

[MeS91] Meira, Sílvio R. L. e Santos, Cássio S.: "SmallVDM: An Environment for Formal Specification and Prototyping in Smalltalk", V Simpósio Brasileiro de Engenharia de Software, Ouro Preto - MG, outubro 1991.

[Mey88] Meyer, B.: "Object-Oriented Software Construction", Prentice Hall, 1988.

[MSC89] Meira, S. R. L., Sampaio, A. C. Alves, Cavalcanti, A. L. C. and Borba, P. H. M.: "Métodos Construtivos para o Desenvolvimento Rigoroso de Software", Departamento de Informática - UFPE, 1989.

[Som89] Sommerville, Ian: "Software Engineering", Addison-Wesley, Terceira edição, 1989.

[UnS87] Ungar, David and Smith, Randall B.: "Self: The Power of Simplicity", ACM - OOPSLA'87 Proceedings, 1987.

[Weg90] Wegner, Peter: "Concepts and Paradigms of Object-Oriented Programming", OOPS MESSENGER, ACM Press, vol. 1, No. 1, august 1990.

PDF compression, OCR, web optimization using a watermarked evaluation copy of CVISION PDFCompressor