Programação Reflexiva sobre o Protocolo de Meta-Objetos Guaraná

Rodrigo Dias Arruda Senra

Dissertação de Mestrado

Instituto de Computação
Universidade Estadual de Campinas

Programação Reflexiva sobre o Protocolo de Meta-Objetos Guaraná

Rodrigo Dias Arruda Senra

17 de dezembro de 2001

Banca Examinadora:

- Prof. Dr. Luiz Eduardo Buzato
  IC—UNICAMP (Orientador)
- Profª Drª Ana Maria de Alencar Price
  II—UFRGS
- Profª Drª Cecília Mary Fisher Rubira
  IC—UNICAMP
- Prof. Dr. Hans Kurt Edmund Liesenberg (Suplente)
  IC—UNICAMP

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP

Senra, Rodrigo Dias Arruda

Se99p Programação reflexiva sobre o protocolo de meta-objetos Guaraná / Rodrigo Dias Arruda Senra -- Campinas, [S.P. :s.n.], 2001.

Orientador : Luiz Eduardo Buzato

Dissertação (mestrado) - Universidade Estadual de Campinas, Instituto de Computação.

1. Linguagem de programação (Computadores). 2. Framework (Programa de computador). 3. Programação orientada a objetos (Computador). I. Buzato, Luiz Eduardo. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

TERMO DE APROVAÇÃO

Tese defendida e aprovada em 17 de dezembro de 2001, pela Banca Examinadora composta pelos Professores Doutores:

Profa. Dra. Ana Maria de Alencar Pride
UFRGS

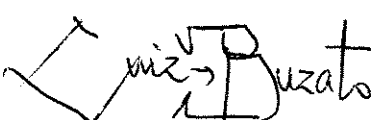
Profa. Dra. Cecília Mary Fischer Rubira
IC - UNICAMP

Prof. Dr. Luiz Eduardo Buzato
IC - UNICAMP

Programação Reflexiva sobre o Protocolo de Meta-Objetos Guaraná

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por Rodrigo Dias Arruda Senra e aprovada pela Banca Examinadora.

Campinas, 17 de dezembro de 2001.



Prof. Dr. Luiz Eduardo Buzato
IC—UNICAMP (Orientador)

Dissertação apresentada ao Instituto de Computação, UNICAMP, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

Para meus pais Nelson e Magali,
por tudo.

O fato de conseguirmos, por um instante, distinguir a verdade não ofuscada pelos nossos preconceitos, amar aos outros sem nada exigir em troca, e criar no êxtase que ocorre quando nos absorvemos completamente no que estamos realizando – o fato de termos tido esses vislumbres dá significado e direção a todas as nossas ações futuras.

O Homem à Procura de Si Mesmo Rollo May (1953)

Agradecimentos

Alexandre Oliva destaca-se dentre todas as pessoas que contribuíram, direta ou indiretamente, neste trabalho por três razões. Em primeiro lugar Oliva é a figura central na concepção e desenvolvimento do MOP de **Guaraná**, sobre o qual todo este trabalho se baseia. Em segundo lugar, Oliva contribuiu no conteúdo desta dissertação através de inúmeras discussões, sendo inclusive um dos principais revisores do texto. Em terceiro e último lugar, Oliva é amigo meu, por quem tenho profundo respeito e admiração.

Luiz Eduardo Buzato, meu orientador, me convidou a fazer parte do seu time, acreditou no meu potencial mesmo durante as adversidades, e respeitou meu espaço vital intelectual. Por tudo isso, e pelo que não foi dito, obrigado.

Agradeço a todos os colegas do LSD (Laboratório de Sistemas Distribuídos) pelas "viagens" intelectuais, pelo seu tempo e, sobretudo, pelas críticas.

Fábio F. Silveira e Acauan P. Fernandes são duas pessoas que eu não conheço pessoalmente, apenas conheço eletronicamente. Muito obrigado pelas perguntas. A necessidade de outrem daquilo que se produz é um poderoso motivador.

Oswaldo José Afonso Franzin e Marco Antônio Garcia Rossi, meus empregadores, sou grato a vocês por permitirem minha atuação na GPr Sistemas em tempo parcial, permitindo assim a minha pós-graduação. Sou grato a vocês por serem meus mentores na vida profissional extra-acadêmica. Não poderia ter tido melhores.

A todos meus outros amigos eu agradeço por continuarem meus amigos sempre que eu dizia: "não posso porque tenho que trabalhar na dissertação".

A meu irmão André, obrigado por ser meu melhor amigo.

A meus pais, pelo exemplo e pelo apoio. Muito obrigado!

Resumo

Esta dissertação traz contribuições teóricas e práticas.

No plano teórico, apresentamos uma unificação da terminologia de Reflexão Computacional, onde introduzimos o termo *para-objeto*. Após a compilação de uma série de critérios para se classificar protocolos de meta-objetos (MOPs), analisamos comparativamente os MOPs mais expressivos até o ano 2000 utilizando a terminologia e os critérios propostos por nós. Enfatizamos os MOPs implementados sobre a Máquina Virtual Java.

Na fronteira entre o plano teórico e prático, analisamos detalhadamente o MOP de **Guaraná**, utilizando a terminologia e critérios propostos.

O MOP de **Guaraná** é um protocolo de meta-objetos (MOP), idealizado por Alexandre Oliva, Luiz Eduardo Buzato e Islene Calciolari Garcia, que almeja simplicidade, flexibilidade, reuso de código de meta-nível e independência de linguagem de programação.

Nesta dissertação também propomos um modelo de programação para o meta-nível. Segundo este modelo, enunciamos os problemas típicos na programação de meta-nível, a partir dos quais enumeramos técnicas para contorná-los.

No plano prático é descrita a implementação de GDK: ***Guaraná*** *Development Kit*, constituído por um conjunto de ferramentas que implementam as técnicas propostas e que auxiliam a programação de meta-nível. Entre os componentes do GDK, existem utilitários para depuração e composição de meta-objetos.

Abstract

This dissertation brings theoretical and practical contributions.

In the theoretical sphere, we propose a unified terminology for Computational Reflection, introducing the term *para-object*. Moreover, we have compiled criteria to classify meta-object protocols (MOPs), which is used in a comparative analysis of the most expressive MOPs published till the year 2000. We give emphasis on those implemented on top of a Java Virtual Machine.

In the frontier of both theoretical and practical spheres, we make a detailed analysis of the **Guaraná** MOP, in which we apply the same terminology and criteria that we have previously defined.

The **Guaraná** MOP is a language independent meta-object protocol that aims at simplicity, flexibility and code reuse. It was conceived by Alexandre Oliva, Luiz Eduardo Buzato and Islene Calciolari Garcia.

Besides, we propose a meta-programming conceptual model, in which meta-level programming issues are raised, and followed by the techniques to tackle them.

In the practical sphere, we describe the implementation of GDK: ***Guaraná** Development Kit*, which consists of a set of tools that implement the proposed techniques and aid meta-level programming. Amongst GDK's components, there are tools to aid debugging and meta-object composition.

Conteúdo

Lista de Tabelas

Lista de Figuras

Capítulo 1

Introdução

Ao começar a redigir esta introdução, nos deparamos com a inevitável pergunta: Por onde começar? Durante uma curta pausa para reflexão nos surpreendemos tentando enumerar quais são as perguntas fundamentais.

Afinal, todo trabalho de pesquisa científica se originou do esforço da humanidade em responder a uma pergunta que desafiasse suficientemente a curiosidade do cientista ou, pelo menos, assim deveria ser. Não foi quando Sir Isaac Newton se perguntou: *Por que a maçã cai da árvore e a Lua sobe no céu?*, que havia sido dado o passo mais importante para a formulação da Teoria Gravitacional? Hoje, tal questionamento soa banal.

Ainda na busca pelas perguntas fundamentais, numa extravagância abstrata nos remetemos aos filósofos gregos, que não por coincidência simbolizam o esforço da humanidade em desbravar as fronteiras da ciência. E neste devaneio helênico, nos perguntamos: Quem sou eu? De onde vim? Onde estou? Para onde vou? E também não por coincidência, com uma ligeira adaptação, estas perguntas são de fato as pergunta fundamentais, delineando o contexto deste e de qualquer trabalho científico: Qual é o tema desta pesquisa? De que outros trabalhos se origina? Qual a contribuição deste trabalho em particular? Quais são os novos horizontes a serem explorados a partir dos resultados deste trabalho? Toda esta dissertação, deste ponto em diante, se dedica a responder estas perguntas.

1.1 Motivação

O tema desta dissertação se insere no contexto da Engenharia de Software, na busca de soluções para produzir software de melhor qualidade, cada vez mais fácil. Ainda que "fácil" seja um conceito difícil de definir com precisão, o termo contrasta bem com o algoz da Engenharia de Software: a complexidade.

Na luta pela administração da complexidade, observamos a ascensão do paradigma OO (Orientação à Objetos) que oferece ferramental para a luta contra a complexidade, coerente

com a diretriz fundamental da Engenharia de Software: a busca de baixo acoplamento e alta coesão. Contudo, o paradigma OO não representa uma solução completa e definitiva para todos os problemas envolvidos na produção de software.

Na produção de software, seja ela industrial ou artesanal, deseja-se atender aos requisitos dos usuários. No entanto, também se sabe que estes tais requisitos estão sempre mudando, muitas vezes até mais rápido do que o próprio ciclo de produção do software. Esta problemática tem motivado a busca de soluções através da incorporação do fator *adaptabilidade* ao projeto e à implementação de software, benefício este que a introdução do paradigma OO não conseguiu satisfazer por completo. E por quê ?

Em OO não existe um mecanismo explícito para garantir a distinção entre requisitos funcionais e não-funcionais, que é um dos principais fatores responsáveis pela complexidade das tarefas de análise e manutenção de software. A criação desta distinção através dos recursos nativos de OO aumenta a complexidade do software, pois eleva o acoplamento entre duas dimensões pouco coesas: a funcional e a não-funcional(gerencial).

Historicamente, há três abordagens distintas para lidar com os requisitos não-funcionais [64]: implementá-los no sistema operacional, implementá-los como primitivas da linguagem de programação ou como bibliotecas da linguagem de programação. Todas as três abordagens sofrem de baixa transparência e de difícil customização.

Até o momento da redação desta dissertação, soluções para incremento da *adaptabilidade* e *diferenciação entre aspectos funcionais e não-funcionais* têm sido buscadas na conjugação de duas tecnologias: **implementações abertas** [33] e **reflexão computacional** [44, 70].

A designação *implementação aberta* é conferida a uma abstração de dupla interface: nível base e meta-nível, onde o nível base consiste na funcionalidade essencial e primária de um dado sistema computacional. Já o meta-nível tem caráter gerencial e adaptativo. O primeiro é passível de uma existência autônoma e idealmente não toma ciência da presença do último, quando presente.

Por sua vez, a designação *reflexão computacional* compreende a capacidade de um dado sistema computacional em ter acesso a representação de seus próprios dados e, sobretudo, de que quaisquer modificações realizadas sobre esta representação hão de refletir em seu estado e comportamento. Com base nesta definição, é natural concluir que o conceito de reflexão computacional está intimamente ligado ao de implementações abertas. Quando estes dois conceitos são mapeados para o domínio de linguagens de programação orientadas a objetos, o padrão de interação entre nível base e meta-nível é denominado de **protocolo de meta-objetos** (MOP).

A arquitetura reflexiva Guaraná [55], sobre a qual está baseada esta dissertação, manifesta os conceitos supra-citados. O princípio fundamental que permeia toda arquitetura é o de que componentes do nível base nunca devem tomar ciência da existência de componentes de meta-nível. Entretanto, não existe um mecanismo embutido em Guaraná que isole o

acesso ao meta-nível a partir do nível base. Quando o princípio é respeitado, é atingida total transparência entre nível base e meta-nível, o que é mais do que desejável no que tange ao gerenciamento de complexidade. Sobretudo, Guaraná ainda provê facilidades de composição hierárquica [50] para código de meta-nível, o que incrementa o reuso e, por consequência, a qualidade do código produzido.

1.2 Evolução da Pesquisa

Após tomar conhecimento do projeto **Guaraná**, nos interessamos pelo desenvolvimento de uma plataforma de comunicação que se beneficiasse das facilidades reflexivas do MOP de **Guaraná**. Em particular, um de nossos objetivos era conciliar a construção de primitivas de comunicação síncrona e assíncrona em um único arcabouço de software, cuja inovação residiria na possibilidade de se chavear dinamicamente entre os modos síncrono e assíncro de comunicação, sem quaisquer alterações na aplicação cliente. Para tanto, o primeiro ano de pesquisa foi dedicado a investigar as manifestações de primitivas de comunicação em sistemas operacionais, linguagens de programação e plataformas de middleware.

Quando estávamos seguros o suficiente sobre *status quo* da pesquisa voltada à comunicação de dados, focalizamos o estudo sobre o MOP de **Guaraná**, que seria a plataforma escolhida para explorar reflexivamente um arcabouço de comunicação. Apesar da elegância no desenho e da qualidade da implementação do MOP, existiam ainda certos empecilhos que impediam o projeto e a implementação do arcabouço de comunicação almejado sobre as primitivas reflexivas do MOP de **Guaraná**.

Entre os problemas encontrados, podemos citar:

- Não estava definido claramente qual o modelo de atuação do meta-programador. Diversas questões não haviam sido abordadas na literatura pré-existente: Qual o ciclo de desenvolvimento de um meta-programa? Ao se projetar um meta-objeto, quais fatores devem ser levados em consideração?

- Não havia um vocabulário comum de conceitos reflexivos que descrevesse **Guaraná** e também outros MOPs similares. Cada descrição de MOP utilizava seu próprio jargão, dificultando uma análise comparativa entre MOPs distintos, assim como o estudo de Reflexão Computacional propriamente dita.

- A documentação de **Guaraná** enfocava os aspectos funcionais e estruturais do MOP, porém com a perspectiva de seus projetistas e não de seus futuros usuários. Essa lacuna tornava a aproximação de **Guaraná** por um usuário incipiente desnecessariamente difícil e desestimuladora.

- Uma vez dominados os conceitos do MOP, não haviam ferramentas disponíveis que facilitassem sua utilização, entre as ausências mais significativas podemos citar ferramentas de depuração do meta-nível e composição de meta-objetos.

Em face aos problemas encontrados, decidimos postergar a investigação sobre primitivas de comunicação reflexivas, redirecionando nossos esforços na tentativa de sanar estes problemas. Esta dissertação descreve estes esforços, bem como seus resultados.

1.3 Organização da Dissertação

A organização estrutural desta dissertação reflete seus objetivos.

O capítulo 1 descreve a motivação subjacente deste trabalho de pesquisa, sua evolução, sua estrutura e seus objetivos.

O capítulo 2 sumariza todos os conceitos comuns a diversos MOPs através da introdução de um vocabulário de definições unificado, que é utilizado na descrição comparativa dos MOPs estudados. Ao final do capítulo são enumeradas outras tendências relacionadas à reflexão computacional, relevantes no contexto da pesquisa.

O capítulo 3 apresenta o MOP de **Guaraná** de forma detalhada quanto aos aspectos estruturais e funcionais. A mesma terminologia convencionada no capítulo 2 é empregada. Sobretudo, esta descrição do MOP se diferencia da literatura pregressa pelo esforço em realçar diretrizes de utilização do mesmo, que de uma certa forma, justificam as decisões técnicas tomadas durante o projeto do MOP.

O capítulo 4 realça diversas armadilhas na programação reflexiva, que expressam os desafios à pesquisa nesta área incipiente. Para cada problema levantado são apresentadas as diversas alternativas do meta-programador, bem como são sugeridas técnicas de projeto e programação que ou solucionam ou contornam os problemas enunciados. Em suma, neste capítulo é desenvolvido um modelo de programação sobre o MOP de **Guaraná**.

O capítulo 5 ilustra as técnicas de programação apresentadas no capítulo 4, através da construção de uma série de ferramentas de meta-nível auxiliares. O conjunto de tais ferramentas chama-se GDK: ***Guaraná*** *Development Kit.* Ainda neste capítulo, são propostas outras ferramentas complementares ao GDK, que ilustram o potencial do MOP.

O capítulo 7 encerra esta dissertação com um resumo dos problemas enfrentados e das contribuições apresentadas ao longo da dissertação.

Capítulo 2

Reflexão Computacional: Teoria e Prática

No que tange à reflexão computacional, apresentamos neste capítulo algumas definições e conceitos básicos, critérios de classificação e um levantamento de suas mais expressivas manifestações em sistemas de software. No decorrer do texto, aproveitamos para convencionar a terminologia que permeará toda a dissertação.

2.1 Conceitos Básicos

O termo **reflexão** nos remete a dois conceitos distintos no domínio da linguagem natural. O primeiro conceito é reflexão como sinônimo de introspecção, ou seja, o ato de examinar a própria consciência ou espírito. O segundo descreve reflexão como uma forma de redirecionamento da luz. No domínio da Ciência de Computação, o binômio *reflexão computacional* encerra ambas conotações: introspecção e redirecionamento. A primeira denota a capacidade de um sistema computacional examinar sua própria estrutura, estado e representação. Essa trinca de fatores é denominada **meta-informação**, representando toda e qualquer informação contida e manipulável por um sistema computacional que seja referente a si próprio. Por sua vez, a segunda conotação, de redirecionamento, confere a um sistema computacional a capacidade da auto-modificação de comportamento. Ambos os conceitos, redirecionamento e introspecção, são tipicamente materializados em linguagens de programação sob a forma de interceptação na execução de primitivas da linguagem. Portanto, a equação resultante é *reflexão computacional = meta-informação + interceptação.*

Em contraposição com tal definição extra-simplificada, reproduzimos a formulação original da **Hipótese de Reflexão** por Brian Smith, cujo mérito foi ser o primeiro a abordar o tema com formalismo:

> *In as much as a computational process can be constructed to reason about an external world in virtue of comprising an ingredient process (interpreter) formally manipulating representations of that world, so too a computational process could be made to reason about itself in virtue of comprising an ingredient process (interpreter) formally manipulating representations of its own operations and structures.*
>
> **The Reflection Hypothesis,** Brian C. Smith (1982)

Por uma questão de brevidade, referenciaremos indistintamente os termos *reflexão* e *reflexão computacional* doravante.

Retornando aos conceitos de introspecção e redirecionamento, não existe dissociação entre os mesmos, mas sim uma relação de causa e efeito. Ou seja, a capacidade de auto-modificação depende de que quaisquer mudanças nas estruturas internas de um sistema computacional incorram em alterações no domínio por elas representado. Em contrapartida, qualquer auto-modificação efetivada deve garantir que as estruturas internas do sistema sustentem uma representação fidedigna do mesmo. Esse vínculo, que é batizado de **conexão causal**, foi teorizado originalmente por Brian Smith, e reafirmado por Pattie Maes na seguinte formulação:

> *A system is said to be causally connected to its domain if the internal structures and the domain they represent are linked in such a way that if one of them changes, that leads to a corresponding effect upon the other. So a causally connected system always has an accurate representation of its domain and it may actually cause changes in its domain as mere effect of its computation.*
>
> **The Causal Connection,** Pattie Maes (1987)

Uma forma de assegurar a validade da conexão causal em uma dada arquitetura reflexiva seria projetar tal arquitetura de forma que sua auto-representação fosse efetivamente utilizada em sua implementação. Dessa forma, por construção, já seria garantida a consistência entre o sistema propriamente dito e sua auto-representação. A dificuldade que surge no projeto de tais arquiteturas é conciliar expressividade e eficiência na auto-representação. A expressividade reflete o quanto a descrição do sistema dá margem para análise. A eficiência reflete o desempenho da auto-representação na efetiva implementação do sistema.

Dado que tais requisitos são frequentemente contraditórios, surge uma opção de abordagem reflexiva denominada **reflexão declarativa**. No caso de reflexão declarativa, a auto-representação não é uma completa descrição funcional do sistema, mas sim uma coleção de

restrições sobre o estado e comportamento do mesmo. Apesar de ser mais fácil conciliar expressividade e eficiência neste modelo, é muito mais difícil assegurar a propriedade de conexão causal. GOLUX [25] e Partial Programs [18] representam tentativas de provar a viabilidade de construção de linguagens de programação enfocando reflexão declarativa.

Com a introdução do conceito de reflexão, sistemas computacionais reflexivos passam a ser subdivididos em dois domínios: **domínio da aplicação** (externo) e **meta-domínio** (interno). O domínio da aplicação é independente e irrestrito, enquanto o meta-domínio está sempre vinculado com o domínio da aplicação. Essa dicotomia leva arquiteturas reflexivas a serem organizadas em uma pilha de níveis de abstração. O domínio da aplicação é representado usualmente por um único nível sem características reflexivas denominado **nível base**.

O meta-domínio é representado por um ou mais níveis reflexivos denominados **meta-níveis**. Cada meta-nível é responsável por manipular meta-informação do nível imediatamente inferior a si próprio, podendo tal nível ser o nível base (não reflexivo) ou outro meta-nível.

Apesar de o conceito de reflexão ser plenamente aplicável sobre diversos paradigmas de programação, como será ilustrado na seção 2.3, nosso interesse nesta dissertação está focado sobre a aplicação de reflexão sobre o paradigma da orientação a objetos (por brevidade, OO). Essa associação é mutuamente (e por que não dizer reflexivamente) benéfica. Por um lado, a programação OO se beneficia de facilidades reflexivas no que tange a criação de um acoplamento baixo e transparente entre requisitos funcionais e gerenciais. Por outro lado, a estruturação do meta-domínio e o gerenciamento de complexidade dos meta-níveis são beneficiados por construções OO. O sucesso da união de ambos os paradigmas é evidenciado pela pluralidade de linguagens de programação (vide Tabela 2.5) que surgiram e sobreviveram por terem adotado tal abordagem. O fruto imediato de tal casamento foi a gênese de uma terminologia especial, centrada no conceito de **meta-objeto**.

Um meta-objeto é antes de mais nada um objeto, ou seja, possui um estado e um comportamento associados. Além disso, um meta-objeto reside no meta-domínio e está vinculado diretamente a um ou mais objetos que pertençam a uma camada de abstração inferior. Tipicamente, em sistemas reflexivos cujo meta-domínio só possui uma camada, a denominação nível base e meta-nível é suficiente e inambígua. Portanto, nestes sistemas, os objetos não-reflexivos referenciados por um meta-objeto são denominados **objetos de nível base**. Estes conceitos são ilustrados na figura 2.1.

Em contrapartida, considerando sistemas reflexivos cujo meta-domínio possua diversos meta-níveis, o objeto vinculado a um meta-objeto de nível superior não pertence necessariamente ao nível base, uma vez que também possa ser um meta-objeto em relação a objetos de nível inferior ao seu. Este problema de nomenclatura tem sido contornado através da adição de prefixos *meta* para diferenciar objetos de níveis reflexivos distintos. Como as ar-

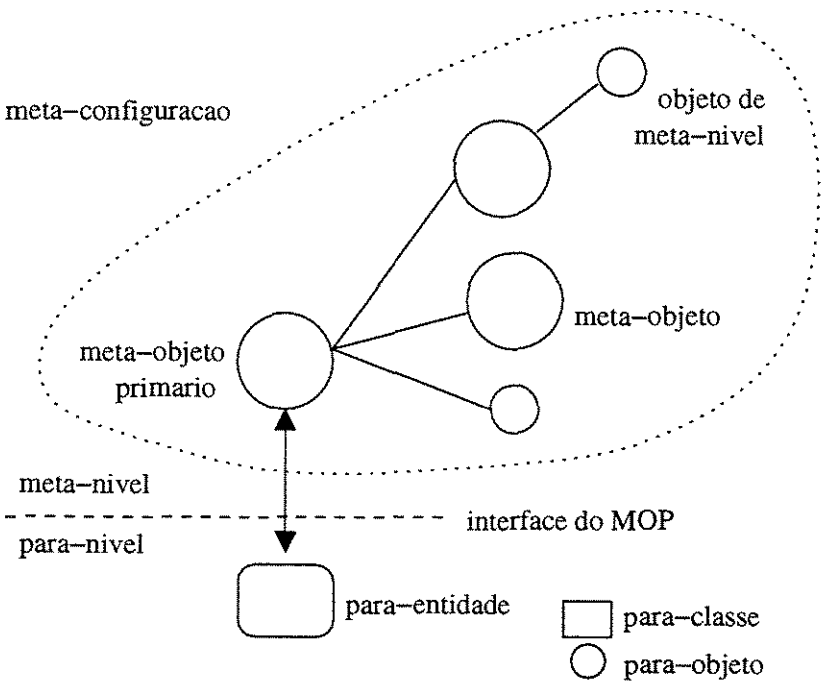


Figura 2.1: Elementos básicos de uma arquitetura OO reflexiva.

quiteturas reflexivas propostas, até a redação desta dissertação, se utilizam de exemplos com no máximo três meta-níveis; tal nomenclatura não incorre em um ônus sintático muito pesado. Não considerando a praticidade ou estilo, resta ainda um problema de nomenclatura. Não existe qualificador para se referenciar um objeto, realçando unicamente o fato de que ele seja subordinado a um outro objeto reflexivo, e independentemente de que ele próprio seja reflexivo ou não. Em suma, não existe um qualificativo para realçar subordinação sem mencionar a que nível pertence o objeto qualificado. Para preencher essa necessidade do discurso propomos o uso do prefixo *para*[1].

Dessa forma, em uma discussão sobre dois objetos, aquele que sofre reflexão será rotulado de **para-objeto** em oposição ao meta-objeto que exerce reflexão. O termo para-objeto possui a mesma semântica que o termo *referent object* quando aplicado no contexto de reflexão. Optamos por não utilizar o termo equivalente em português *objeto referente ou referenciado*, para evitar a ambiguidade no discurso. O último termo também é utilizado para denotar dois objetos de mesmo nível que estejam associados logicamente.

Da mesma forma, **para-nível** torna-se o nível que sofre reflexão e está subordinado a um meta-nível. Este prefixo é aplicável sobre qualquer entidade abstrata envolvida no discurso,

[1] O prefixo *para* vem do grego **para**, significando: 'proximidade', 'ao lado de', 'ao longo de', 'elemento subsidiário'. Também pode denotar 'similaridade' ou 'aquilo que está além'.

cujos exemplos mais frequentes são para-objeto, para-classe e para-nível. Utilizamos o termo genérico *entidade* ao invés de *objeto* para não criar um comprometimento em relação a que estrutura de dados do para-nível sofre reflexão.

Pelo mesmo raciocínio, **para-aplicação** seria a aplicação de para-nível, e no caso mais simples corresponderia à aplicação que pertence ao nível base da torre reflexiva.

Portanto, o "domínio da aplicação" ou simplesmente *para-aplicação* é constituído por um único para-nível: o nível base. Já o meta-domínio é constituído de uma pilha de níveis reflexivos que podem assumir ambos os papéis de meta-nível e para-nível, ou seja, exercer e sofrer reflexão.

Tipicamente, há objetos no meta-nível cujo propósito não é o de gerenciar para-objetos, portanto não podem ser classificados como meta-objetos. Esses objetos que residem no meta-nível, cuja finalidade é servir de apoio à execução de meta-objetos, serão referenciados nesta dissertação como **objetos de meta-nível**. Entretanto, é possível encontrar textos sobre Reflexão Computacional que tratam os termos *meta-objeto* e *objeto de meta-nível* indistintamente.

Ainda no meta-nível, o grupo de meta-objetos cooperantes vinculados direta ou indiretamente a um único para-objeto será rotulado de **meta-configuração** do referido para-objeto. Sendo assim, a meta-entidade denominada *meta-configuração* é composta por, ao menos, um meta-objeto primário (diretamente vinculado ao para-objeto) e, opcionalmente, por demais *meta-objetos* e *objetos de meta-nível*. É necessário cuidado para não confundir entidade com processo quando nos referimos à *meta-configuração* .

O processo de *meta-configuração* é o ato ou efeito de agrupar uma série de meta-entidades, vinculando-as a uma determinada para-entidade. Uma vez enunciada essa distinção, não há necessidade de que seja explicitada no texto, pois a interpretação correta pode ser inferida do contexto.

Finalmente, a somatória de todas as meta-configurações será denominada **meta-programa**, que deve ser igual a toda porção de código de uma dada aplicação que resida no meta-nível.

A interação entre o para-nível e o meta-nível é regida pelo **protocolo de meta-objetos** ou **MOP**[2], que consiste em uma interface de acesso à meta-informação e manipulação do para-nível a partir do meta-nível. De acordo com Lisbôa [41], existem quatro aspectos principais a serem considerados em um dado MOP: **vinculação**, **reificação**, **execução** e **modificação**. Por uma preferência de estilo, optamos por rebatizar o termo **modificação** por **intervenção** mantendo a semântica original. Alguns autores utilizam o termo **effectuation** quando se referem a este aspecto.

Esses aspectos são definidos na tabela 2.1. Figurativamente, a *vinculação* pode ser entendida como a cola entre para-nível e meta-nível, a *reificação* como a leitura do para-nível

[2]do inglês *Meta-Object Protocol*

Tabela 2.1: Aspectos principais de um MOP

Vinculação	Reflete a natureza da ligação entre meta-entidades e para-entidades. Por exemplo, qual é a cardinalidade do relacionamento meta-entidade/para-entidade, ou seja, quantas para-entidades podem estar subordinadas a uma única meta-entidade? Quantas meta-entidades podem gerenciar uma dada para-entidade? A vinculação pode ser estabelecida em tempo de compilação (estática) ou em tempo de execução (dinâmica), estabelecendo a jurisdição de uma meta-entidade no âmbito das para-entidades.
Reificação	É o processo de disponibilização de meta-informação oriunda do para-nível em um formato acessível ao meta-nível. Descreve tanto o estado das para-entidades quanto a dinâmica de troca de mensagens entre as mesmas.
Execução	Descreve a capacidade das meta-entidades de interagirem com o para-nível como se fossem para-entidades requisitando serviços.
Intervenção	Descreve a capacidade do meta-nível de controlar a estrutura, estado e comportamento do para-nível.

pelo meta-nível, a *intervenção* como a escrita no para-nível pelo meta-nível, e finalmente a *execução* como o uso do para-nível pelo meta-nível.

Não existe, ainda, consenso sobre qual é o conjunto ideal de características que devam constituir qualquer MOP de propósito geral e independente de linguagem de programação. Existem, entretanto, diversas propostas de incorporação de recursos reflexivos em linguagens de programação. Antes de proceder com uma revisão de tais linguagens (Seção 2.3) é mister apresentarmos alguns critérios de classificação de recursos reflexivos em linguagens de programação.

2.2 Classificação

Não existe, ainda, uma taxonomia definitiva para categorizar reflexão. Em virtude dessa necessidade, iniciamos esta seção com um paralelo entre os quatro aspectos básicos de um MOP (Tabela 2.1) e um sub-conjunto da coleção de critérios descritos na tabela 2.2 reunida por Cazzola [10], nominalmente: temporalidade, associatividade, transparência e abrangência. E o critério adicional *reflexividade*, proposto por Rideau [61].

Tabela 2.2: Mapeamento entre critérios de classificação e aspectos básicos de um MOP.

Critérios \Aspectos	Vinculação	Reificação	Intervenção	Execução
Temporalidade	√			
Associatividade	√			
Transparência	√	√	√	√
Abrangência	√	√	√	√
Reflexividade	√	√	√	√

Para facilitar a compreensão dessa discussão, adotou-se a nomenclatura a seguir. Um **aspecto** é uma perspectiva da interação entre para-nível e meta-nível (descrito na Tabela 2.1). Um **critério** é uma forma de avaliar os diferentes aspectos isoladamente, subdividindo-se por sua vez em diversas **categorias**.

2.2.1 Critério Temporalidade

O critério **temporalidade** está associado ao aspecto vinculação, que se dá entre elementos do meta-nível e do para-nível, criando três possíveis categorias: **durante-compilação**, **durante-carga** e **durante-execução**. A título de completude, os termos originais na língua inglesa que designam tais categorias são **compile-time**, **load-time**, e **run-time**, respectivamente.

A primeira categoria abrange todas as arquiteturas em que a vinculação entre para-entidades e meta-entidades é estabelecida ainda em tempo de compilação de código. A segunda abrange arquiteturas em que a vinculação só é estabelecida durante a carga de código em memória volátil. A terceira quando o código já está em execução. Da primeira à terceira existe uma gradação crescente da demanda por recursos computacionais.

A vinculação *durante-compilação* permite isolar o mecanismo reflexivo de qualquer dependência com o ambiente de execução (máquina virtual), seja um interpretador em software ou processador em hardware. O código de meta-nível atua diretamente no controle do processo de compilação, indiretamente influindo no comportamento do sistema como um todo. Nesta abordagem, o MOP é codificado dentro de um compilador ou pré-processador da linguagem alvo, cuja maior deficiência é não suportar a vinculação de meta-entidades com para-entidades cuja identidade só esteja definida em tempo de execução . Por exemplo, em arquiteturas que seguem esta filosofia não é possível vincular meta-objetos a para-objetos que venham a assumir um estado específico.

Em contrapartida, essa deficiência desaparece na abordagem *durante-execução*, cujo maior mérito é oferecer possibilidades irrestritas de vinculação parametrizadas por estado ou comportamento de um dado para-objeto. Portanto, esta última abordagem implica em

Tabela 2.3: Critério Associatividade: Mapeamento entre para-entidades e meta-entidades

Baseado em classe	classe $\mapsto$ meta-objeto
Baseado em instância	objeto $\mapsto$ meta-objeto
Baseado em referência	referência $\mapsto$ meta-objeto
Baseado em mensagem	invocação $\mapsto$ meta-objeto
Baseado em canal	objeto.invocação(objeto) $\mapsto$ meta-objeto

enxertar o MOP no ambiente de execução, seja ele software ou hardware.

Quanto à abordagem intermediária *durante-carga*, sua viabilidade é mais teórica do que prática, uma vez que desconhecemos qualquer arquitetura que a tenha adotado exclusivamente. Entretanto, com a popularização da utilização de bibliotecas de ligação dinâmica em linguagens compiladas e interpretadas, é possível que esta abordagem ganhe maior atenção. Acreditamos que a abordagem *durante-carga* seria melhor classificada como um mero caso particular da abordagem *durante-execução*.

Com relação aos demais aspectos do MOP: *reificação*, *execução* e *intervenção*; a nenhum destes se aplica o critério temporalidade, por se manifestarem sempre em tempo de execução independentemente do momento em que se dá a vinculação.

2.2.2 Critério Associatividade

O critério **associatividade** determina quais entidades de um dado nível de abstração podem sofrer reflexão, ou seja tornarem-se para-entidades. A título de simplificação vamos definir o operador reflexão $\mapsto$, de forma que seu operando esquerdo seja a para-entidade e seu operando direito seja a meta-entidade.

Segundo este critério, as categorias geradas pelas combinações entre para-entidades e meta-entidades para o aspecto vinculação são definidas na Tabela 2.3.

Vale ressaltar que a frequente manifestação da combinação de categorias *baseado em classe* e *baseado em instância* recebe a designação *modelo meta-objeto*, a qual não foi incluída como uma categoria distinta para evitar sobrecarregar o termo e incorrer em ambiguidade. A título de complementação, existe também a designação **modelo meta-classe**, consistindo em uma associatividade baseada em classe onde a meta-entidade vinculada à classe é denominada meta-classe. Neste modelo, classes atuam tanto como para-entidades de meta-classes quanto como meta-entidades de objetos.

O modelo meta-classe está vinculado à linguagem Smalltalk [19] e implementações reflexivas derivadas [8, 30] da mesma.

A categoria *baseado em mensagem* pode ser encarada como uma variação sobre a categoria *baseado em classe*, uma vez que não leva em consideração a instância originadora ou receptora da mensagem, apenas qual fração de comportamento do para-nível foi ativada.

Também a categoria *baseado em canal* deve ser encarada como um híbrido entre as categorias *baseado em mensagem* e *baseado em objeto*, pois tanto o método invocado quanto os para-objetos originadores e receptores da mensagem são levados em consideração na determinação da meta-entidade responsável. Esta categoria se originou da aplicação de reflexão à construção de infra-estrutura para sistemas distribuídos [1].

Um outro desdobramento existente é a categoria **baseado em referência** [20] na qual a meta-entidade estará vinculada a uma referência para uma para-entidade e não a para-entidade em si. Essa categoria pode ser emulada em MOPs que suportem associação *baseada em objeto* através de técnicas de **wrapping**. No contexto desta dissertação, *wrapping* consiste em encapsular um para-objeto em outro, mantendo uma relação de delegação entre os mesmos. Essa montagem permite emular a associação baseada em referência através da vinculação de um meta-objeto ao para-objeto encapsulador ou externo.

A título de simplificação do discurso, quando nos referimos a uma para-entidade **alvo** de uma meta-configuração, fazemos referência a associação reflexiva existente entre ambos.

É importante ressaltar que discutimos a natureza da para-entidade associada, porém nada foi mencionado quanto a *cardinalidade* da associação entre para-entidade e meta-configuração.

2.2.3 Critério Transparência

O critério **transparência** está associado à visibilidade da transição para-nível $\rightarrow$ meta-nível. Quando esta transição é visível pelo próprio para-nível diz-se que a reflexão é **explícita**, caso contrário **implícita**.

A forma explícita de reflexão é geralmente indesejável, pois permite a criação de código no para-nível dependente de código no meta-nível; o que implicaria em um indesejável alto acoplamento entre para-nível e meta-nível. Sendo assim, reflexão explícita é contraria aos mesmos objetivos que motivaram a formulação da abordagem reflexiva: isolamento entre meta-nível e para-nível mimetizando o isolamento entre requisitos funcionais e não-funcionais.

Uma forma de se medir o nível de transparência de uma dada arquitetura reflexiva seria observar o número de alterações necessárias ao se expor um para-nível sucessivamente a diferentes meta-níveis, que por sua vez não tenham sido construídos especificamente para o para-nível escolhido. No caso ideal de **transparência total** (implícita sobre todos os aspectos do MOP), não seria necessária nenhuma alteração no para-nível.

2.2.4 Critério Abrangência

O critério **abrangência** foi proposto por Ferber [28], sub-dividindo reflexão em **estrutural** e **comportamental**. Este critério, além de ser aplicável aos quatro aspectos do MOP, é ortogonal aos critérios anteriores. Sobretudo, as duas categorias *estrutural* e *comportamental*, diferentemente das categorias dos critérios anteriores, não são mutuamente exclusivas, podendo co-existir.

Classifica-se como reflexão estrutural a atuação de um dado aspecto do MOP sobre a estrutura das para-entidades. Complementarmente, classifica-se como reflexão comportamental a atuação de um aspecto do MOP sobre o comportamento das para-entidades. No domínio de orientação a objetos, essa bi-dimensionalidade surge naturalmente da própria definição de objeto como entidade que possui um estado e um comportamento associados.

No que tange aos aspectos *vinculação*, *reificação* e *intervenção* seria necessário determinar se cada construção da linguagem de programação alvo é passível de ser mapeada e manipulada (leitura e escrita) pelo meta-nível, respectivamente. Devido à grande diversidade de linguagens, e por consequência uma grande diversidade de construções, evitamos uma enumeração exaustiva adicionando duas categorias: *abrangência parcial* e *abrangência total*. Estas duas categorias são ortogonais às já apresentadas para o critério abrangência. Esta abordagem é inadequada para o exame de um MOP isoladamente, mas serve ao propósito de uma análise comparativa entre diversos MOPs.

2.2.5 Critério Reflexividade

Para finalizar nossa coletânea de critérios de classificação falta abordar o meta-programa como um todo. Rideau [61] categorizou meta-programas a partir da análise de parâmetros de entrada e saída. Seu enfoque está baseado em tratar meta-programas como processos transformadores código $\rightarrow$ código em oposição a transformação código $\rightarrow$ dados.

Nesta dissertação vamos batizar este critério de **reflexividade**. A descrição de algumas das categorias geradas pelo critério são expostas na tabela 2.4 através do operador reflexividade $\Re$.

De acordo com a sintaxe proposta por Rideau, $n \ \Re \ p$ expressa um meta-programa que possui n parâmetros de entrada e p parâmetros de saída, ambos parâmetros interpretados como código em oposição a dados. Segundo o proponente, as categorias enumeradas não são exaustivas mas atendem ao propósito de ilustração, podendo ser estendidas pela recombinação dos exemplos aqui citados.

Tabela 2.4: Critério Reflexividade: Transformações de código em código.

Categoria	Exemplos	Atividade
0 $\Re$ 0	não reflexivo	–
1 $\Re$ 0	interpretador	executa o programa parâmetro
	inspecionador	documenta o programa parâmetro
	analisador	verifica corretude do programa parâmetro
	testador	testa automaticamente o programa parâmetro
	depurador	executa programa explicitando estruturas internas
0 $\Re$ 1	pré-compilador	compila recursos em forma de código
	gerador	gera código a partir de descrições de alto nível
1 $\Re$ 1	compilador	traduz linguagem de alto-nível em outra de nível inferior
	otimizador	adapta código com intuito de melhorar o desempenho
	estilizador	melhora layout de código aumentando a legibilidade
2 $\Re$ 0	interpretador genérico	interpreta código segundo interpretador
	analisador genérico	analisa corretude de código segundo especificação
2 $\Re$ 1	remendador	aplica remendo(patch) em código

2.2.6 Os Quatro Princípios de Kiczales

Existem ainda quatro princípios, citados por Kiczales [33], que devem ser levados em consideração na análise e classificação de um MOP. São eles: controle de escopo, separação conceitual, incrementalidade e robustez.

O princípio **controle de escopo** de Kiczales assemelha-se ao critério *associatividade*, pois ambos se aplicam ao aspecto vinculação definindo que entidades são passíveis de reflexão. O enfoque do primeiro é definir uma cardinalidade de vinculação entre uma dada meta-entidade e para-entidades com propriedades iguais. Já o enfoque do segundo é definir qual é o universo de para-entidades, ou seja, qual conjunto de entidades que é passível de vinculação a qualquer meta-entidade. A título de exemplificação, quando analisamos se um dado MOP suporta vinculação de meta-objetos a classes(*baseada em classe*) ou a instâncias de classe(*baseada em instância*), estamos abordando a sua associatividade. Quando analisamos se um MOP *baseado em instância* permite vinculação de um meta-objeto a uma única instância em particular ou a várias instâncias de uma mesma classe, estamos abordando o *controle de escopo*. Muitas vezes este último é também denominado **granularidade** [10].

O princípio **separação conceitual** de Kiczales defende a existência de mínima interdependência entre os elementos da interface de meta-nível. A despeito da importância deste princípio, sua subjetividade impossibilita a sua aplicabilidade através de um critério classificatório.

No princípio **incrementalidade**, Kiczales propõe que por ocasião da adição de nova funcionalidade não deve ser necessária a re-implementação de funcionalidade já existente. Todos os MOPs construídos sobre o paradigma de OO já satisfazem teoricamente tal princípio, e representam a maioria dos MOPs propostos desde o surgimento de 3-KRS. Tal fato pode ser verificado pela distribuição de MOPs em relação aos diversos paradigmas de linguagens de programação retratada na tabela 2.5.

Por último, Kiczales afirma no princípio **robustez** que se deve minimizar o alcance de erros oriundos do meta-nível em relação ao resto do sistema. Dosar a relação entre o "poder + expressividade" e a "robustez" de um MOP é ainda um processo artesanal.

Na seção 2.3, fazemos uma revisão de diversas manifestações de reflexão em linguagens de programação e suas extensões. O objetivo principal desta revisão é servir de referencial comparativo para a nossa proposta de arquitetura reflexiva descrita no capítulo 3.

2.3 Manifestações

Com a introdução da arquitetura de John von Neumann em 1945, que unificou o armazenamento de código e dados em uma mesma mídia (volátil) de armazenamento, tornou-se possível que um processo computacional manipulasse formalmente representações de si

mesmo. Entretanto, a ausência de modelos formais para reger a construção de código auto-modificável levou ao banimento de tal prática de programação pela comunidade científica da época.

Entre 1974 e 1984, surgiram diversas linguagens de programação que introduziam mecanismos reflexivos de maneira *ad hoc*. Os principais problemas dessa abordagem foram: a falta de distinção entre para-nível e meta-nível e a necessidade de reengenharia do ambiente de interpretação da linguagem, mediante a necessidade de adição de novas facilidades reflexivas. Portanto, foi só com o advento do dialeto reflexivo 3-Lisp [70] que surgiu um formalismo mais apropriado. A Tabela 2.5 retrata a dinâmica de aparecimento de características reflexivas em linguagens de programação e suas extensões ao longo do tempo.

Como nosso propósito não é delimitar a fronteira entre linguagem de programação e extensão sobre linguagem de programação, há controvérsias quanto à classificação de certas linguagens/extensões entre autores opositores. Portanto, nos restringimos a documentação e análise da manifestação de características reflexivas.

Na tabela 2.5, as linguagens e extensões destacadas em negrito são as que apresentam formalismo reflexivo, em oposição às linguagens com características reflexivas *ad hoc*.

Dentre a miríade de linguagens apresentadas na Tabela 2.5, é interessante ressaltar aquelas que introduziram novos conceitos. FOL [83] e Meta-Prolog [7] são exemplos de linguagens baseadas em lógica com suporte à reflexão, onde foi introduzido o conceito de **meta-teoria** que difere do conceito tradicional de teoria, uma vez que se não se refere à dedução de um problema de domínio externo, mas à dedução de outra teoria.

Teiresias [14] e SOAR [38], por sua vez, são exemplos de linguagens baseadas em regras com uma arquitetura reflexiva, introduzindo a noção de **meta-regra**, ou seja, uma regra que rege a computação corrente e o processo de inferência, avaliação e conflito entre outras regras. Uma descrição mais detalhada dessas linguagens e das demais que suportam reflexão de forma *ad hoc* foi realizada por Pattie Maes [44].

Em oposição às supra-citadas linguagens que suportam reflexão *ad hoc*, apresentaremos a seguir uma coleção de linguagens de programação e extensões que manifestam características reflexivas de forma mais integrada e coesa, posteriores ao surgimento de 3-Lisp. Dentre essa coleção, as extensões reflexivas da linguagem Java serão apresentadas em separado e com maior profundidade, uma vez que possuem maior relevância no contexto dessa dissertação.

2.3.1 Linguagens e Extensões com Formalismo Reflexivo

Nesta seção, nosso interesse é fornecer uma breve retrospectiva histórica, cujo objetivo é retratar a evolução do formalismo reflexivo na pesquisa científica e sua incorporação em linguagens de programação. Além de complementar a fundamentação teórica desta dissertação, o objetivo deste esforço de revisão é apresentar as diferentes propostas reflexivas através

Tabela 2.5: Características reflexivas em linguagens de programação.

	Baseada em Lógica	Baseada em Regras	Imperativa Não-OO	Imperativa OO
1975				PLASMA [69]
1980	FOL [83]			Smalltalk [19]
				RLL [22]
1981				LOOPS [5]
				Actors [40]
1982		TEIRESISAS [14]		
1984		SOAR [38]	**3-Lisp** [70]	
1986	Meta-Prolog [7]			OBJVLISP [9]
1987				**3-KRS** [44]
1988				**ABCL/R** [80]
1989				Classtalk [30] e Actalk [8]
1991				**CLOS** [32]
1992				**AL-1/D** [26]
1993				**Open-C++** [66]
1995				**Java** [73]
				CodA [46]
1996				Neoclasstalk [62]
				Iguana [60]
1997				**MetaJava/metaXa** [20, 21]
				OpenJava [75]
				ReflectiveJava [84, 85, 86]
				Correlate [29]
1998				**Dalang/Kava** [81, 82]
				Guaraná [55]
1999				**OpenJIT** [23, 24]

de uma terminologia comum. Este esforço pode ser entendido como uma "normalização descritiva", visando facilitar o processo de análise comparativa.

3-Lisp

Proposta por Brian Smith [70] em 1982, 3-Lisp foi um dialeto da linguagem Lisp construído sobre 2-Lisp, que por sua vez foi um dialeto de Lisp mais estrito muito similar a Scheme.

3-Lisp inovou pela introdução de um modelo de reflexão procedural baseado em uma arquitetura de computação sequencial. Além de ter sido a precursora das linguagens imperativas não-orientadas a objetos com suporte a reflexão computacional, 3-Lisp já permitia reificação e intervenção de forma plena sobre todas as suas estruturas, respeitando a propriedade da conexão causal e suportando reflexão em tempo de execução. A evolução do modelo reflexivo de 3-Lisp provavelmente foi comprometida por estar intimamente ligado à filosofia de programação em Lisp, por não contemplar o paradigma da orientação-a-objetos e, principalmente, pela ausência de uma separação clara entre para-nível e meta-nível.

3-KRS

Proposta por Pattie Maes [44] quase cinco anos depois do advento de 3-Lisp, 3-KRS foi considerada a linguagem OO pioneira no suporte a reflexão. Assim como 3-Lisp, 3-KRS suportava reificação e intervenção de forma plena, sendo consistente com o princípio de conexão causal, o que se tornou possível pela adoção de um modelo de representação uniforme onde todas entidades da linguagem eram representadas como objetos. Cada objeto era emparelhado com um respectivo meta-objeto construído tardiamente sob demanda.

Apesar de delimitar uma fronteira entre para-computação e meta-computação, o MOP de 3-KRS apresentava um acoplamento muito forte entre para-objeto e meta-objeto, dificultando , assim, a construção de bibliotecas de componentes para o meta-nível.

ABCL/R

Quase concomitantemente ao surgimento de 3-KRS, ABCL/R foi concebida por Watanabe e Yonezawa [80] com uma abordagem reflexiva radicalmente diferente de 3-KRS. Em ABCL/R cada para-objeto é implementado por uma torre independente de meta-objetos que o descrevem a nível estrutural e comportamental. O MOP de ABCL/R é baseado em troca de mensagens assíncronas que permite concorrência de execução entre para-nível e meta-nível. Posteriormente, essa linguagem obteve continuidade através de ABCL/R2 e ABCL/R3 [45]. Esta última adotou um modelo de reflexão em tempo de compilação para evitar que haja sobrecarga relativa ao mecanismo de interceptação em tempo de execução. ABCL/R3 inovou com um compilador capaz de avaliar a definição de um dado meta-objeto para produzir

uma meta-entidade geradora de código nativo que atuasse em tempo de execução. Essa abordagem foi denominada **partial evaluation** [16].

CLOS

Common Lisp Object System, abreviado para CLOS, foi uma linguagem proposta em 1991 por Gregor Kiczales [32] para se tornar o dialeto Common Lisp OO padrão. Historicamente, CLOS é a sucessora de FLAVORS(Symbolics) e LOOPS(Xerox). Atualmente há várias implementações de CLOS, entre as quais é relevante citar MCL, KCL, CL(Allegro e CMU), Ibuki, Lucid, Medley, Genera(Symbolics), CLOE, LispWorks(Harlequin) e PCL. Dentre todos esses sucessores de CLOS, PCL é a implementação que possui o MOP mais detalhado.

Todos os elementos básicos da linguagem: classes, slots(variáveis de instância), funções genéricas, métodos, especializadores de métodos e combinações de métodos são objetos de primeira classe. Dessa forma, CLOS suporta plena reificação, já a intervenção está sujeita a um conjunto de restrições que evita a quebra de protocolo OO no que tange a hierarquias de generalização/especialização e a interface pública de classes/objetos. Entretanto, o MOP de CLOS impõe que todos para-objetos derivados de uma dada classe só possam estar vinculados a meta-objetos derivados diretamente de uma mesma classe. Essa restrição representa uma desobediência ao princípio da *separação entre domínios*, ou seja, a existência de plena desvinculação entre os requisitos funcionais de para-nível e os requisitos gerenciais de meta-nível.

Open-C++

Open-C++ foi elaborado por Shigeru Chiba [11, 13], unificando a estrutura do MOP de CLOS com o princípio de reflexão em tempo de compilação oriundo de Intrigue [39] e Anibus [31].

O objetivo dessa linguagem era disponibilizar um mecanismo eficiente e transparente para implementação de extensões de C++ através do controle do processo de compilação, que por sua vez abrange os seguintes aspectos da linguagem subjacente: definição de classes, acesso a membros, invocação de funções virtuais e criação de objetos.

Por um lado, essa abordagem provê ganho de eficiência por não incorrer em nenhuma sobrecarga em tempo de execução, tal como suporte a interceptação. Por outro lado, há uma penalidade em custo temporal e espacial no processo de compilação.

O MOP de Open-C++ gerencia a tradução de meta-código (extensões em Open-C++) em código C++ puro através de duas fases. Na primeira fase o MOP gera meta-objetos transientes, que só existirão durante a compilação. Na segunda fase, os meta-objetos são convidados a gerar fragmentos de código apropriados, que por sua vez podem ser tanto código C++ ou código Open-C++. A iteração entre as duas primeiras fases prossegue até que todos os fragmentos de código só possuam código C++.

Como na versão 2.0 de Open C++ o MOP ainda não estava integrado a um compilador C++, era necessária a existência de um terceiro passo para compilação de código C++ em código objeto da arquitetura alvo. Desconhecemos se tal integração chegou a ser efetivada.

Por estar fundamentada na estrutura do MOP de CLOS(baseado em classe), Open-C++ também viola o princípio da *separação entre domínios* ao obrigar que para-objetos com mesma generalização (classe) estejam associados a meta-objetos também com mesma generalização. Outra deficiência de Open-C++ é restringir a vinculação de um meta-objeto a um único para-objeto. Deficiência esta que é amenizada pela criação de um sistema de propagação automática e customizável de meta-objetos através de hierarquias de generalização/especialização presentes no para-nível. Ou seja, subclasses (para-nível) podem herdar meta-classes associadas a sua classe base.

Iguana

Iguana [60] é também um MOP que estende reflexivamente a linguagem C++ através de pré-processamento. Segundo seus idealizadores, Gowing e Cahill, um MOP especifica a implementação reflexiva de um modelo de objetos, sendo definido pela somatória das interfaces dos meta-objetos que o compõe. Portanto, quando Iguana alega suportar múltiplos MOPs isto deve ser entendido como o suporte simultâneo a múltiplos modelos de objetos. Segundo a terminologia defendida nesta dissertação, diríamos que uma meta-configuração pode implementar um novo modelo de objetos em Iguana.

O MOP de Iguana se destaca pela alta seletividade quanto à reificação, possuindo um conjunto de construções reificáveis bem mais amplo que o de Open-C++. Visando diminuir a sobrecarga em tempo de execução, é possível especificar dentre a vasta gama de elementos que compõe o modelo de objetos *default* da linguagem C++, quais são aqueles que devem ser passíveis de reificação para a composição de um novo modelo de objetos modificado reflexivamente. Tais elementos passíveis de reificação são agrupados em *categorias de reificação*, englobando ambos os aspectos estruturais e comportamentais do modelo de OO em C++.

A título de ilustração, podemos citar algumas categorias reificáveis (critério abrangência) em Iguana como: objeto, método, classe, busca de método (lookup), tabela de métodos virtuais, criação-ativação-destruição de objeto, entrega de mensagens, árvore de herança, etc. A abundância de categorias de reificação evidencia o compromisso entre poder e complexidade, que pode dificultar a produção eficiente de código reflexivo.

No que tange ao critério associatividade, Iguana disponibiliza quatro mecanismos de vinculação, que em sua própria terminologia são chamados de **seleção de protocolo**: default, de classe, de instância e de expressão. O primeiro designa apenas a ausência de vinculação, onde os para-objetos seguem o protocolo OO default de C++. O segundo e terceiro casos são particularidades de associatividade baseada em instância, onde é possível vincular uma meta-configuração a todas as instâncias de uma dada classe ou a uma única instância em particular, respectivamente. O quarto e último caso afirma a possibilidade de se vincular uma meta-configuração a uma expressão presente no para-nível, como por exemplo uma invocação de método.

Não encontramos documentação que expandisse o número de exemplos ou que delimitasse o conjunto de expressões válidas. Sendo assim, só nos resta o entendimento de que a seleção de protocolo por expressão engloba todos os demais casos anteriores, sendo coerente com a política de reificação ao oferecer máxima flexibilidade, ainda que sob a penalidade de alta complexidade.

Outra diretriz de Iguana relevante, no escopo de transparência, é sua opção por um modelo explícito. Ou seja, para-objetos podem conhecer, interagir e até substituir suas próprias meta-configurações. Uma vez que uma meta-configuração em Iguana é responsável pela implementação de um modelo de objetos, isto significaria que um para-objeto pode optar dinamicamente por mudar seu próprio modelo de comportamento. Acreditamos que neste quesito em particular existe o risco de se entrelaçarem características funcionais e não-funcionais, que na nossa concepção é muito mais deletério do que a tão evitada reificação total e sua consequente penalidade em desempenho.

CodA

Simultaneamente ao advento de Open-C++, Jeff McAffer elaborou CodA [46] no intuito de prover uma arquitetura de meta-nível que se opusesse às arquiteturas monolíticas.

A diretriz básica de CodA, inspirada em Actalk [8] e AL-1/D [26], é prover decomposição por comportamento lógico, onde aspectos do comportamento de um para-objeto são fatorados em um conjunto de meta-entidades denominadas **meta-componentes**, que por sua vez possuem uma **API**[3] simples e curta. Dessa forma, o MOP de CodA estimula a re-utilização de código e facilita o gerenciamento de meta-nível, especialmente por ser orientado a reflexão em tempo de execução (ao contrário de Open-C++), permitindo a recomposição dinâmica da meta-configuração de um dado para-objeto.

Na terminologia de CodA, a ação conjunta de um grupo de meta-componentes (meta-configuração) define um **modelo de objeto**, ou seja, o padrão de comportamento de um para-objeto. Apesar de estimular a composição e o reuso de meta-entidades, CodA não resolve

[3]A sigla API significa originalmente em inglês *Application Programming Interface*. Tipicamente, caracteriza-se por um conjunto de procedimentos ou funções logicamente relacionados.

inteiramente o problema da conciliação entre meta-componentes implementando modelos de objeto potencialmente conflitantes, mas efetivamente facilita a identificação do conflito.

Os meta-componentes básicos presentes em qualquer meta-configuração são: *Send*, *Accept*, *Queue*, *Receive*, *Protocol*, *Execution* e *State*. Basicamente, este conjunto é responsável por um esquema de comunicação flexível e extensível orientado a filas. Há suporte à troca de mensagens de forma síncrona ou assíncrona, e também à chamada de procedimento assíncrona(*promise* ou *future*) [42]. Até o presente, temos conhecimento apenas da implementação do MOP de CodA sobre Smalltalk, sobre a qual há suporte parcial à reificação e intervenção dos aspectos básicos do para-nível.

2.3.2 Extensões Reflexivas para Java

Uma vez que nossa proposta de ferramentas e técnicas de programação reflexiva foram implementadas e aplicadas sobre o MOP reflexivo Guaraná que por sua vez está correntemente vinculado à linguagem Java, optamos por destacar nesta seção outros trabalhos sobre reflexão que também adotaram a mesma linguagem hospedeira para seus MOPs. O intuito não é desmerecer trabalhos realizados em outros nichos, outrossim delinear nossa linha de pesquisa, facilitando o processo de comparação com abordagens similares.

É importante ressaltar que o MOP Guaraná foi projetado de forma a ser independente de linguagem de programação, desde que haja suporte a OO. Entretanto, até a redação final desta dissertação, a única implementação efetivamente levada a termo está baseada na linguagem Java.

Java

A linguagem de programação Java[4] possui suporte nativo à reflexão computacional denominado *Java Core Reflection (JCR)* [74]. Não obstante, o modelo de reflexão adotado só abrange os aspectos reificação e intervenção. Ambos com abrangência exclusivamente estrutural e parcial (devido a limitações impostas por requisitos de segurança).

As facilidades reflexivas permitem a criação de novas instâncias de classe, bem como a obtenção de acesso a atributos e métodos de instâncias já presentes na computação. Não existe distinção entre meta-nível e para-nível. Além disso, uma vez que não há manifestação do aspecto vinculação, não se aplicam os critérios associatividade e temporalidade. O suporte nativo à reflexão é tão precário e rudimentar que fomentou diversas incursões acadêmicas em um esforço de desenvolver MOPs mais elaborados para a linguagem.

[4]Java é marca registrada da empresa Sun Microsystems Inc.

ReflectiveJava

ReflectiveJava [84, 85, 86] defende a construção de um MOP que não incorra em modificações na linguagem Java, em sua máquina virtual (JVM) ou em seu compilador. Para tanto, o MOP de ReflectiveJava permite a construção de aplicações reflexivas mediante a utilização de um pré-processador. A partir de arquivos fontes escritos em uma linguagem declarativa de vinculação, o pré-processador gera fontes Java substituindo as classes alvo de reflexão por classes derivadas das mesmas, chamadas **classes de reflexão**.

As instâncias das classes de reflexão agregam internamente seus respectivos meta-objetos, para os quais explicitamente delegam os métodos especificados no arquivo de vinculação, antes e depois de invocar os métodos herdados. Portanto, a computação de meta-nível ocorre através de delegação (operações *metaBefore* e *metaAfter*) intercalada da computação de para-nível, que ocorre através de herança.

Ainda que a vinculação entre meta-objeto e seu respectivo para-objeto seja automática e ocorra em tempo de compilação, a existência das operações *getMeta* e *changeMeta* permite a verificação e alteração dinâmica da meta-configuração. Apesar da possibilidade de reconfiguração dinâmica, não há sugestões na descrição de ReflectiveJava sobre facilidades de composição de meta-objetos. O que nos leva a crer que as meta-configurações são unitárias.

Na ausência de reificação estrutural, mas desejando minimizar o acoplamento entre para-objeto e meta-objeto, foram introduzidos os mecanismos de **method Id** e **method Category**. Ambos consistem em rotular os métodos passíveis de reflexão através da linguagem declarativa de vinculação com números inteiros, de tal forma que, em tempo de execução, estes rótulos sejam informados ao meta-objeto durante o processo de interceptação. O primeiro identifica o método que está sofrendo reflexão, o segundo parametriza o curso de ação a ser tomado.

Conclui-se que ReflectiveJava apresenta um MOP com associatividade baseada em classe, porém com granularidade de método. A ausência total de transparência compromete uma eficaz utilização do MOP. Sobretudo, a mistura entre hierarquias de para-nível e meta-nível (classes de reflexão) viola a diretriz de não intrusão do meta-nível sobre o para-nível.

Apesar de sua grande portabilidade e independência de plataforma, O MOP de ReflectiveJava não apresenta um arcabouço prático, robusto ou seguro para a construção de aplicações reflexivas.

OpenJava

OpenJava [75, 76] assume o papel de um avançado pré-processador de macros que adere ao paradigma de reflexão computacional. Pode ser encarado como uma interface de meta-nível para estender a linguagem Java a nível sintático e funcional.

No que tange ao critério temporalidade, OpenJava se enquadra exclusivamente na catego-

ria de reflexão durante-compilação. Essa propriedade propicia as vantagens e desvantagens típicas de qualquer MOP baseado em tempo de compilação. Por um lado, há o benefício da ausência de sobrecarga em tempo de execução, que é contrabalançado pela limitada capacidade de alteração de comportamento.

Segundo a arquitetura de OpenJava, código-fonte com extensões sintáticas ou macros de meta-nível responsáveis pela interpretação/tradução são submetidas a um módulo tradutor que submete a *árvore sintática abstrata* a uma meta-entidade alocada para cada nó da árvore. Como a árvore é processada em uma ordem em particular (pré-ordem, pós-ordem, etc) através de um único passo, o escopo de atuação de cada meta-entidade fica impedido de utilizar informação de nós da árvore ainda não visitados.

Em suma, não é possível que a atuação de uma meta-entidade seja parametrizada por informação disponível em outras meta-entidades responsáveis por ramos da árvore ainda não visitados. Essa limitação vale para os quatro aspectos do MOP sob qualquer critério.

No que tange ao critério associatividade, OpenJava opta por um modelo baseado em classe. Sendo assim, é gerada uma meta-entidade (denominada meta-classe no jargão de OpenJava) para cada classe no nível base, esteja a classe presente na forma de código-fonte ou bytecode. Uma vez que o passo de compilação/tradução ocorre uma única vez, lembrando que é nesse passo que o código de meta-nível atua, só há a possibilidade de existência de um único para-nível, coincidente com o nível base.

Outra limitação importante existente em OpenJava, que afeta qualquer MOP atuante durante-compilação, é a distinção entre atuação sobre as para-entidades do ponto de vista do recebimento ou envio de uma mensagem. Devido à existência de polimorfismo em OO, é impossível determinar o tipo efetivo de uma referência para-objeto em tempo de compilação, somente o tipo do objeto por ela referenciado. Portanto, manipulações de meta-nível durante-compilação sofrem mais restrições quando atuam sobre a invocação de mensagens do que quando atuam sobre o atendimento de mensagens. Isso ocorre pois a associatividade, no primeiro caso, é baseada em referência; e, no segundo, baseada em instância (eventualmente classe, porém particularmente em Java a distinção não faz sentido).

Em decorrência dessa problemática, o processo de tradução de OpenJava está baseado em dois passos sequenciais. O primeiro é tradução referente ao atendimento de mensagens, o segundo é referente à invocação de mensagens. Nesta sequência, a primeira fase pode influir na atuação da segunda, mas a recíproca é falsa.

Javassist

Java Programming Assistant, ou simplesmente **Javassist** [12], é definida como uma ferramenta para auxiliar a programação na linguagem Java. Originalmente, *Javassist* foi projetada para automatizar a definição de classes. Neste contexto, um compilador especial era capaz de expandir "anotações" (similares a *templates*) em código-fonte através do uso

da API reflexiva da própria linguagem Java associado a um *ClassLoader* específico. Tais anotações não apenas simplificavam o processo de geração de classes, sobretudo permitiam a alteração da estrutura e comportamento das classes pré-existentes no escopo (*name space*) do *ClassLoader* específico.

Javassist foi incluída em nossa revisão bibliográfica de MOPs justamente devido a esta sua capacidade de manipular para-classes reflexivamente. No jargão de *Javassist*, os "assistentes" de criação ou alteração de para-classes assumem o papel de meta-objetos, possuindo vinculação baseada em classe em tempo-de-carga (*load time*). Justamente por basear-se em tempo-de-carga é que o MOP de Javassist é extremamente portável, uma vez que não implica em modificações no *inner sanctum* da JVM. Em contrapartida, as para-entidades manipuladas pelo MOP de *Javassist* estão restritas a co-existir num espaço de nomes específico, o qual é definido pelo *ClassLoader* utilizado. Em alguns nichos de aplicação, essa limitação pode quebrar a transparência entre para-nível e meta-núivel, potencialmente demandando alterações no código de para-nível.

Javassist permite atuação em tempo-de-execução. Contudo, considerando que Java é uma linguagem de tipagem estática; a aplicação de *Javassist* em tempo-de-execução se dá através de referências para tipos polimórficos.

Com relação à interceptação, *Javassist* permite a interpolação de *callbacks* antes ou depois da execução de métodos de uma determinada para-classe, ou no acesso a seus atributos. Estas *callbacks* desempenham o mesmo papel que meta-objetos. Esta abordagem apesar de simples e de fácil compreensão possui uma série de desvantagens: afeta intrusivamente a estrutura das para-classes com a mesclagem entre código de para-classe e de meta-objeto, não permite a composição de meta-objetos e não é transparente em relação ao para-nível.

Em suma, o MOP de *Javassist* possui três grandes vantagens: facilidade de compreensão, alto desempenho sem custo de transição para meta-nível, permite a definição dinâmica de classes, e alta portabilidade (independente de JVM). Por outro lado, o MOP de *Javassist* é inferior nos seguintes quesitos: não permite vinculação baseada em instância, não permite composição de meta-objetos, pode causar problemas de disjunção de espaço de nomes, alem de ser intrusivo no para-nível (não transparente).

Correlate

Correlate [3, 29, 63, 65] é uma evolução da biblioteca de classes XENOOPS (Execution Environment for OO Parallel Systems) [4] originalmente voltada para a criação de programas paralelos em arquiteturas com memória distribuída. Correlate é uma extensão que oferece concorrência por intermédio de uma arquitetura de meta-nível para uma linguagem de programação OO hospedeira não concorrente.

Apesar das versões iniciais terem se baseado em C++, Java tornou-se a linguagem hospedeira a partir da versão 3.0 de Correlate.

Correlate adota o conceito de **objetos ativos**, que são objetos possuidores do controle de sincronização sobre requisições externas em um ambiente concorrente. Os objetos ativos podem sofrer concorrência externa com outros objetos ativos, ou interna quando implementam **métodos autônomos**. Em oposição aos objetos ativos, existem os *objetos passivos*, os quais não apresentam as propriedades já citadas, servindo ao propósito de mapear os objetos construídos exclusivamente segundo as regras da linguagem hospedeira.

Estes conceitos são introduzidos na linguagem através da introdução de palavras reservadas como: *active* e *autonomous*. É importante ressaltar que o acoplamento entre objetos passivos (sem palavra reservada *active*) possuidores de fluxo de execução independente (*thread* própria) e objetos ativos deve ser controlado explicitamente pelo programador.

Sendo assim, as facilidades de sincronização de Correlate abrangem exclusivamente os objetos ativos. Portanto, objetos ativos não podem compartilhar objetos agregados passivos, somente objetos também ativos, cujo intercâmbio é feito através de um serviço de nomes embutido no ambiente de execução da linguagem.

O MOP de Correlate atua apenas sobre objetos ativos, suportando transparência total e implícita, apresentando portanto abrangência parcial e estática quanto à vinculação.

Quanto ao critério associatividade, o MOP se classifica na categoria *baseada em instância*, pois a criação de um para-objeto ativo implica na implícita criação de um meta-objeto associado, o que restringe a flexibilidade do MOP.

A temporalidade do MOP se classifica como *durante-compilação* quanto ao aspecto vinculação e *durante-execução* para os demais aspectos.

No que tange ao aspecto reificação, a abrangência do MOP é também parcial em ambos os níveis estrutural e comportamental, justificada através de uma redução de custo e consequente melhora do desempenho do meta-nível. A reificação de estado é obtida através do uso de computações sobre o estado interno do para-objeto, sem causar efeitos-colaterais.

A dinâmica de comunicação inter-para-objetos é reificada através de "objetos de interação", que expõe exclusivamente a porção do estado do para-objeto necessária para a análise da interação que ocorre no para-nível. Neste ponto, o MOP de Correlate se destaca dos demais por oferecer operadores que especificam modos de comunicação síncrona(operador $) ou assíncrona(operador @).

Há uma certa semelhança com a abordagem adotada por CodA, cuja diferença reside no meta-nível de CodA ser concorrente, enquanto o de Correlate é sequencial.

Em relação a facilidades para combinação de meta-entidades, este MOP suporta múltiplos meta-níveis. Entretanto, a utilização da torre de níveis reflexivos como forma básica de composição ortogonal de meta-funcionalidade induz construções excessivamente complexas. Não há suporte para composição de entidades através da geração de hierarquias de generalização/especialização, sob a justificativa de prevenir a manifestação de *anomalias de herança* [63].

Por fim, Correlate suporta um mecanismo de **object policies** inspirado em **Aspect Oriented Programming** [2, 34, 72] de forma a compensar certas limitações em seu MOP. O sistema de políticas de objeto se traduz em modelos(ou *templates*) independentes de quaisquer para-entidades, os quais serão interpretados por meta-entidades. As últimas definirão dinamicamente uma semântica (políticas) efetiva para os modelos. Ou seja, são uma forma de parametrizar a atuação de meta-entidades através de descrições abstratas que não definem intrinsecamente um comportamento (implementação) em particular. Este mecanismo serve também de "cola" entre para-nível e meta-nível.

Dalang/Kava

O MOP de Dalang [81, 82] se destaca dos demais MOPs baseados em Java por duas características: temporalidade durante-carga e portabilidade total, independente da presença de uma *JVM* em particular ou do código fonte. A sigla JVM significa *Java Virtual Machine* e designa o ambiente de execução da linguagem Java.

Com relação à primeira característica, Dalang se aproveita do mecanismo de carga dinâmica de classes, extensível em Java, para introduzir novas facilidades reflexivas no ambiente de execução. Para tanto, se utiliza uma técnica denominada **wrapping** ou **encapsulamento**. Nesta seção, preferimos utilizar o termo original *wrapping* ao invés do termo traduzido *encapsulamento*, pois o último pode suscitar confusão com outra conotação existente na terminologia OO.

A técnica de *wrapping* consiste em renomear uma dada classe sujeita à reflexão(*wrapped*), subordinando-a a uma outra(*wrapper*) que assume sua identidade(nome) e que simula sua interface com os demais componentes do para-nível, permitindo ainda atuação prévia ou posterior do *wrapper* em relação à delegação efetiva de uma mensagem ao objeto *wrapped*. Esse padrão de interação entre objetos é denominado **Wrapper** ou **Decorator** [17].

Com relação à segunda característica, Dalang adquire total portabilidade em virtude da linguagem Java e de seu ambiente de execução (JVM) suportar o mecanismo de carga de classes em toda e qualquer implementação. Extensões a esse mecanismo são implementadas através de hierarquias de especialização, cujos componentes são denominados **ClassLoaders**.

Quanto à associatividade, meta-objetos são *wrappers* para **para-classes**, que são simplesmente classes presentes no para-nível. A vinculação é definida estaticamente através de um arquivo de configuração, o qual especifica o mapeamento entre classes do para-nível e meta-objetos. A associatividade baseada em classe permite um refinamento de granularidade, possibilitando parametrizar a vinculação também por método.

A técnica de *wrapping* implica em um modo de comunicação síncrono entre para-entidade e meta-entidade através da relação de delegação existente entre ambas. Sendo assim, o MOP de Dalang possui abrangência comportamental em tempo de execução, ainda que o processo de vinculação se restrinja ao tempo de carga.

No que tange à transparência, apesar de meta-objetos "travestidos" de para-classes interagirem transparentemente com as demais para-entidades (objetos ou classes), existe a possibilidade de que a atuação do meta-objeto seja evitada(*by-pass*) mediante o conhecimento da nova identidade(nome) da para-classe "rebatizada". Portanto, consideramos a transparência de Dalang frágil, ainda que implícita.

Em função do requisito de portabilidade de Dalang, que implica na ausência de quaisquer modificações na JVM, torna-se necessário que aplicações sujeitas à reflexão tenham suas para-classes carregadas explicitamente no para-nível pelo ClassLoader reflexivo de Dalang. Essa restrição também enfraquece a transparência e aplicabilidade deste MOP.

No que se refere às facilidades de composição de meta-entidades, Dalang suporta tanto encadeamento (meta-configuração no padrão *chain of responsibility* [17]) quanto empilhamento de níveis reflexivos, onde a primeira alternativa é preferível por razões de simplicidade e economia de recursos.

Os projetistas de Dalang consideram a existência de três papéis assumidos por profissionais que atuam em sistemas reflexivos: o de desenvolvedor do domínio de aplicação (aspectos funcionais), o de projetista de componentes de meta-nível(aspectos não funcionais) e o de integrador de sistema (determinar a vinculação entre para-nível e meta-nível). Esta noção transcende o MOP de Dalang, sendo válido para todos os demais MOPs que suportam o conceito da *separação de domínios*, e sobretudo que ofereçam facilidades para construção de meta-entidades genéricas e recombináveis.

Existe uma série de problemas derivados da aplicação da técnica de *wrapping* para veiculação de reflexão, que foram levantados pelos pesquisadores de Dalang. Os mais relevantes são:

Eficiência A criação do *wrapper* através da compilação, em um processo a parte, de código-fonte para bytecodes penaliza o desempenho do sistema no quesito tempo.

Violação de Encapsulamento A possibilidade de quebra de encapsulamento entre meta-objeto e seu respectivo para-objeto, seja através do conhecimento da nova identidade do para-objeto ou da obtenção de uma referência direta ao para-objeto, enfraquece a corretude e consequente aplicabilidade do MOP.

Frágil Transparência A exigência de que as para-classes sejam explicitamente carregadas por um ClassLoader específico, revelam a intervenção do meta-nível sobre o para-nível.

Problema da Auto-Referência Dalang implementa o modelo **forwarding** de encapsulamento, no qual requisitos de **auto-referenciação**[5] são entregues ao próprio originador do requisito em oposição ao modelo **delegating**, onde os requisitos de auto-referenciação são entregues à entidade que encapsula o originador do requisito. A opção de Dalang pelo modelo *forwarding* implica na impossibilidade de reificação de comportamento auto-referente.

Conflitos em Hierarquias de Tipos A utilização da técnica de *wrapping* possibilita a manifestação de incoerências semânticas, especialmente em MOPs cujas para-entidades e meta-entidades estão também sujeitas a regras comportamentais derivadas de uma hierarquia de tipos (herança). No caso de Dalang, o *wrapper* pertence à hierarquia de meta-objetos enquanto o *wrapped* pertence a uma hierarquia diferente, frequentemente exclusiva do para-nível. Essa discrepância pode ter efeitos colaterais sobre o para-nível, como nos casos de coerção de tipos, polimorfismo e aplicação de reflexão estrutural(através de JCR por exemplo). Ainda relacionado à influência de hierarquias de tipos, existe o problema da coerência de comportamento entre para-objetos pertencentes a uma mesma hierarquia de especialização porém encapsulados por meta-objetos distintos, por sua vez potencialmente pertencentes a diferentes hierarquias de especialização (ainda que obrigados a possuírem uma raíz em comum: MetaObject). É importante ressaltar que este último problema é ortogonal à aplicação de técnicas de *wrapping*, existindo também em outros MOPs em que para-nível e meta-nível são OO.

Holzle [27] propôs a unificação entre objetos *wrapper* e *wrapped* como solução para os problemas acima derivados da aplicação da técnica de *wrapping*.

Dalang será sucedida por Kava, que ainda está em fase de implementação por ocasião da redação desta dissertação. Kava prevê um modelo mais eficiente de geração de *bytecodes*, sem necessidade de invocação de um compilador Java em um processo a parte, e permitirá a implementação de ambas *caller side reflection* (reflexão ativa ou *de saída*) e *receiver side reflection* (reflexão passiva ou *de chegada*) com vinculação baseada-em-classe e baseada-em-instância, enquanto Dalang suporta apenas reflexão passiva baseada-em-classe.

OpenJIT

Possuindo características similares à linha de pesquisa de Dalang/Kava, OpenJIT [23, 24] (1999) é um projeto de compilador "reflexivo" Java Just-In-Time (JIT) escrito em Java.

O compilador OpenJIT apresenta duas características relevantes: auto-compilação e disponibilização de componentes como objetos de primeira classe (*first-class values*). O recurso

[5]Entende-se por auto-referenciação a invocação de mensagens em OO sobre uma referência *this*.

de auto-compilação refere-se à capacidade do compilador de compilar a si mesmo durante a execução do programa usuário. Isto é possível dado que OpenJIT é escrito também em Java. Sendo assim, num primeiro passo, o código de OpenJIT é interpretado pela JVM, cuja interpretação resulta na compilação do próprio código bytecode em código nativo.

Já a disponibilização de componentes do compilador como objetos de primeira classe, coexistindo ao lado de componentes da *para-aplicação*, permite que para-objetos façam uso de facilidades de compilação em tempo de execução para os mais diversos fins. Entre eles: otimização, *partial evaluation*, depuração e extensão da linguagem.

Particularmente quanto a facilidades de extensão, OpenJIT provê o recurso de **compilets**, que permitem a especialização do *front-end* do compilador.

Os esforços de pesquisa em OpenJIT visam: melhorar seu desempenho sem comprometer sua portabilidade ou flexibilidade, definir parâmetros de utilização segura das facilidades de compilação durante-execução inseridas num contexto multi-tarefa (multi-threading) e analisar os requisitos de segurança e controle de escopo dos componentes JIT sujeitos a auto-modificação.

É importante ressaltar que o nível de abstração das para-entidades e meta-entidades manipuladas por OpenJIT difere sensivelmente do nível de abstração das entidades manipuladas pelos demais MOPs dessa seção. Enquanto a maioria dos MOPs possui associatividade baseada em entidades do paradigma OO (objetos,classes,métodos e mensagens), OpenJIT possui associatividade baseada em entidades relacionadas ao processo de compilação, tais como: bytecodes, laços de execução, blocos básicos, árvore de sintaxe abstrata, etc. Em virtude destas diferenças excluímos OpenJIT da tabela comparativa de MOPs (Tabela 2.6).

MetaJava/metaXa

Originalmente chamada MetaJava [20, 35, 36, 37], metaXa [21] suporta reflexão estrutural e comportamental através da extensão da capacidade reflexiva de uma JVM.

Essa abordagem de implementação reflexiva permite temporalidade durante-execução e ainda fomenta a separação entre código funcional e não-funcional. Essa separação é reforçada pela transparência implícita de metaXa, onde as transições do para-nível para o meta-nível ocorrem de forma síncrona.

No que tange ao critério associatividade, metaXa apresenta diversas opções de vinculação. É possível vincular meta-objetos a quaisquer uma das seguintes para-entidades: classe, objeto, referência, *thread* e *ClassLoader*.

No caso particular de MOPs implementados sobre a linguagem Java, não há muito sentido na distinção entre vinculação baseada em classe e baseada em instância, uma vez que para toda e qualquer classe é possível construir um instância que a represente. Isto significa que no domínio da linguagem Java a vinculação baseada em classe é um caso particular da vinculação baseada em instância. Todavia, uma dada implementação de um MOP pode se beneficiar ou não dessa característica.

De acordo com a documentação de metaXa, essa particularidade não é aproveitada, o que pode ser notado pelo tratamento especial dado as entidades classe, *thread* e *ClassLoader*. A rigor, as três para-entidades se enquadrariam no caso de vinculação baseada em instância, dispensando assim a adição de complexidade à interface do MOP.

Ainda relacionado ao critério associatividade, metaXa apresenta um controle de escopo flexível, permitindo que múltiplos meta-objetos possam estar vinculados a um mesmo para-objeto. Porém, sempre estarão combinados através de um encadeamento serial ordenado pela ordem inversa de vinculação, que a título de simplificação resume-se a um empilhamento de meta-objetos (padrão *chain of responsibility* [17]).

Quanto ao aspecto reificação, metaXa apresenta maior flexibilidade do que os demais MOPs já analisados. Existem seis mecanismos distintos provendo reificação em metaXa:

Reificação de mensagens na chegada (passiva): Este mecanismo notifica um dado meta-objeto sobre a recepção de mensagens em seu para-objeto vinculado, disponibilizando informações sobre a natureza da mensagem, como identificação do método invocado e seus parâmetros reais. Existe ainda, a opção pelo modelo de notificação, podendo ser em paralelo ou em série. Na primeira opção, a execução do método de para-nível é interpolada entre a execução de tratadores de meta-nível (pré-tratamento e pós-tratamento). Na segunda opção, a execução do método de para-nível é redirecionada ao tratador de meta-nível, cabendo ao último a decisão sobre a efetiva execução do método de para-nível ou não. A escolha do modelo de notificação é parametrizável por meta-objeto.

Reificação de mensagens na saída (ativa): Este mecanismo notifica um dado meta-objeto sobre o envio de mensagens (invocação de método) realizado pelo para-objeto vinculado sobre um outro para-objeto qualquer.

Reificação de acesso a atributos e trancamento de objetos: Este mecanismo notifica um dado meta-objeto sempre que algum atributo de seu para-objeto vinculado sofre leitura ou escrita. Particularmente na Linguagem Java, todo objeto possui um atributo implícito para fins de sincronização entre diferentes fluxos de execução(*threads*). Portanto, o acesso a este atributo implícito em operações de trancamento(*locking*) também é reificado. Todavia, metaXa apresenta uma limitação neste mecanismo de reificação,

que consiste na restrição de abrangência do mecanismo sobre atributos com escopo de proteção *private* ou *protected*.

Reificação da criação de instâncias: Este mecanismo é responsável por notificar um meta-objeto associado a uma classe de que um para-objeto foi instanciado a partir da classe vinculada.

Reificação de carga de classe: De acordo com este mecanismo, é possível que um meta-objeto seja notificado por ocasião da carga de uma dada classe no ambiente de execução da JVM, desde que tal meta-objeto tenha manifestado previamente o interesse pela notificação da carga de tal classe em particular. Ou seja, não é possível receber notificação da carga de uma classe cujo nome era desconhecido, ou sem que houvesse uma requisição explícita pela notificação da carga da classe específica. Este mecanismo de reificação se relaciona diretamente com o caso de vinculação baseada em *ClassLoader*. Nota-se que a vinculação é um caso especial, diferente de vinculação baseada em instância, uma vez que o mecanismo de reificação associado é muito mais restritivo do que o mecanismo de reificação usual (baseado em instância).

Reificação de código: Este mecanismo disponibiliza o código que implementa os métodos de um dado para-objeto para seu meta-objeto associado sob a forma de um vetor de bytecodes. Este é um mecanismo muito poderoso, porém de difícil manipulação.

Outra característica relevante do MOP de metaXa é a utilização da técnica de **class shadowing** ou "sombreamento de classes". Esta técnica permite a implementação da mudança de semântica de um dado para-objeto, sem afetar a semântica dos demais para-objetos que sejam instâncias da mesma classe.

A técnica consiste em criar uma réplica("sombra") da classe original sobre a qual serão implementadas modificações que afetam sua semântica. A partir de então, o para-objeto alvo torna-se instância da classe "sombra", enquanto os demais para-objetos continuam instâncias da classe original. Quando esta técnica é aplicada, para todo meta-objeto existe também uma classe "sombra" gerada para substituir a classe original de cada para-objeto vinculado. Apesar desta técnica de implementação oferecer transparência no âmbito do para-nível, podem surgir anomalias de herança caso para-objetos utilizem os recursos reflexivos básicos presentes na Linguagem Java. Esta técnica é uma variação sobre a técnica de *wrapping*.

Guaraná

Possuindo diversas semelhanças com metaXa, o MOP de Guaraná suporta reflexão estrutural e comportamental através da extensão da capacidade reflexiva de uma JVM. Essa abordagem é implementada pela interação síncrona entre para-nível e meta-nível através de transparência implícita, atendendo assim ao requisito de separação de domínios e permitindo o isolamento de aspectos funcionais e não-funcionais.

Uma vez que o MOP está embutido na JVM, é possível suportar temporalidade durante-execução conferindo aos diversos aspectos do MOP um teor dinâmico. Ou seja, os processos de vinculação, reificação e composição de meta-configurações atuam efetivamente em tempo de execução. Vale relembrar o compromisso existente entre flexibilidade e eficiência. Enquanto MOPs baseados em tempo de compilação são mais eficazes temporalmente, os mesmos possuem menor flexibilidade em comparação com MOPs baseados em tempo de execução.

Entende-se por eficácia temporal um melhor desempenho quanto ao consumo de tempo durante a execução. Na maioria da vezes, este desempenho é obtido por uma vinculação estática, o que diminui a sobrecarga de processamento em tempo de execução, sacrificando a flexibilidade da vinculação dinâmica.

O MOP de Guaraná optou pela flexibilidade em detrimento de maior eficácia temporal, uma vez que é possível melhorar o desempenho temporal através de técnicas de otimização, entre as quais vale ressaltar *partial evaluation* e *just-in-time compiling*.

No que tange ao aspecto associatividade, Guaraná oferece vinculação baseada em instância, se beneficiando da homogeneidade de representação em Java onde classes são também objetos. Essa escolha concilia abrangência de vinculação sobre as meta-entidades existentes mantendo a simplicidade, exceção feita sobre a vinculação a referências para a qual não há suporte em Guaraná. Entretanto, a ausência de suporte a associatividade baseada em referência não representa uma limitação de Guaraná, tendo em vista que a troca dinâmica da composição de meta-configurações permite a obtenção dos mesmos resultados obtidos por vinculação baseada em referência. Desta forma, Guaraná se mantém fiel à diretriz de manter a simplicidade sem sacrificar sua expressividade.

Ainda relacionado ao aspecto associatividade, Guaraná determina que a vinculação entre para-objeto e meta-objeto é uma relação um-para-um. Ou seja, um dado para-objeto possui um único meta-objeto a si vinculado, o qual recebe a designação de **meta-objeto primário**. Essa decisão também não representa uma restrição na flexibilidade do MOP, tendo em vista que o meta-objeto primário pode delegar suas obrigações reflexivas para uma composição hierárquica e arbitrária de meta-objetos subordinados. Na terminologia de Guaraná, quando um meta-objeto suporta a capacidade de delegação de suas responsabilidades reflexivas ele é chamado de **Composer**.

Quanto ao aspecto reificação, Guaraná oferece um mecanismo baseado em **reificação passiva** ou *interceptação de chegada* com notificação em série. Neste mecanismo a reificação

ocorre quando um dado meta-objeto primário é notificado assim que seu para-objeto vinculado sofreu uma operação de para-nível, nominalmente: leitura ou escrita de seus atributos (incluindo *locking*) ou invocação de um de seus métodos. Diz-se que há *notificação em série* pois o tratamento exercido pelo meta-objeto se interpõe entre a invocação do método no para-nível e a efetiva execução do método.

Neste modelo, cabe ao meta-objeto permitir ou não a efetiva execução do método de para-nível, desde que mantenha a coerência da computação de para-nível em caso negativo. Em caso positivo, quando há a efetiva execução do método de para-nível, a entrega dos resultados (se houverem) produzidos ao para-objeto que invocou o método é também reificada e submetida a aprovação do meta-objeto responsável. Em contra-partida, existe a **reificação ativa** ou *reificação de saída*. Neste caso, um meta-objeto é notificado da ocorrência de uma operação de para-nível quando seu para-objeto subordinado é o originador da operação. O MOP de Guaraná somente suporta **reificação ativa** por ocasião da instanciação de objetos. Quando um para-objeto cria um outro para-objeto de qualquer classe, tanto o meta-objeto vinculado ao para-objeto que requisitou a instanciação quanto o meta-objeto vinculado a classe do para-objeto instanciados são consultados (na ordem apresentada) sobre a composição da meta-configuração do para-objeto recém-instanciado. Este mecanismo permite a propagação em tempo de execução de uma dada meta-configuração ao longo das diversas cadeias de instanciação de uma dada aplicação. Guaraná não suporta reificação ativa quanto a invocação de métodos, presente em metaXa. Tampouco há suporte para reificação de código, considerando que a complexidade e a diferença de nível de abstração introduzidas por este mecanismo são incompatíveis com o resto do arcabouço de meta-programação oferecido pelo MOP de Guaraná.

Um desdobramento dos aspectos *intervenção* e *execução* é a capacidade do MOP de Guaraná na criação de invólucros de para-objetos ou **proxy-objects**. Tais entidades se comportam como para-objetos quaisquer, interagindo normalmente no âmbito do para-nível. Todavia, *proxy-objects* são instanciados a partir do meta-nível, não possuindo estado ou comportamento intrínsecos. Efetivamente seu estado e comportamento serão determinados dinamicamente pelo meta-objeto associado. A título de exemplificação, este é um recurso chave para a implementação transparente de serviços de replicação e persistência de para-objetos.

Sobretudo, devemos ressaltar que a diretriz fundamental do MOP de Guaraná é a impossibilidade de se determinar a partir de qualquer para-nível ou meta-nível a presença ou não de uma meta-configuração dado um para-objeto qualquer. O maior benefício desta diretriz é forçar que meta-configurações independentes sejam ortogonais por construção e, por conseguinte, possam sofrer composição de forma simples e arbitrária.

Tabela 2.6: Avaliação Quantitativa de MOPs baseados na linguagem Java

		Guaraná		metaXa		OpenJava		Javassist		Dalang/Kava		Correlate		ReflectiveJava	
Abrangência Estrutural/Comportamental		E	C	E	C	E	C	E	C	E	C	E	C	E	C
Temporalidade	durante-compilação			√		√	√					√		√	√
	durante-carga							√			√				
	durante-execução	√	√	√	√					√	√		√	√	√
Associatividade	baseado em instância	√	√		√						√	√	√		
	baseado em classe	√	√	√	√	√		√		√	√				√
	baseado em referência			√											
	baseado em mensagem														√
	baseado em canal														
	reificação total	√	√			√		√	√						
Transparência	vinculação	√	√									√		√	
	reificação	√	√			√					√		√		√
	intervenção	√	√			√		√	√				√		
	execução	√	√			√		√	√		√		√		
Meta-Composição		√	√	√						√		√			

2.4 Sumário

Iniciamos o capítulo definindo Reflexão Computacional no contexto de Ciência da Computação, e em particular no contexto deste trabalho de pesquisa.

Definimos que há suporte a Reflexão Computacional quando existe reificação de meta-informação e interceptação de comportamento.

No decorrer do capítulo, exploramos diversos conceitos, entre eles: conexão causal, reflexão declarativa, domínio da aplicação, meta-domínio, meta-objeto, objeto de meta-nível, meta-configuração, e MOP.

Propusemos aqui a introdução do prefixo "para", o qual serve para designar qualquer entidade que sofre reflexão, independentemente da possibilidade da mesma entidade exercer ou não reflexão. No jargão tradicional não exista tal designador, causando ambiguidade na descrição de ambientes com múltiplos meta-níveis.

Elegemos 4 aspectos e 5 critérios, ortogonais entre si, como arcabouço teórico de classificação dos recursos reflexivos em linguagens de programação. Nominalmente, os aspectos: vinculação, reificação, intervenção e execução; e os critérios: temporalidade, associatividade, transparência, abrangência e reflexividade.

Uma vez definidos os conceitos básicos e critérios de avaliação, apresentamos uma revisão bibliográfica dos trabalhos mais expressivos na área, dando ênfase àqueles voltados para a linguagem Java. Além criar um mapa temporal de referências bibliográficas para os principais MOPs, procuramos criar a distinção entre àqueles com formalismo reflexivo dos MOPs cujo suporte reflexivo é *ad hoc*.

O resultado da avaliação entre os MOPs baseados em Java indicou que o MOP de **Guaraná** é mais abrangente que os demais, o que pode ser verificado pela tabela 2.6. Dentre as omissões mais relevantes no MOP de **Guaraná**, ressaltamos: ausência de *reificação ativa* ou *interceptação de saída* (metaXa), ausência de objetos ativos (Correlate), e independência de JVM (Dalang/Kava). Entre as vantagens oferecidas por **Guaraná** ressaltamos: transparência implícita, composição hierárquica arbitrária de meta-objetos, temporalidade durante-execução para os 4 aspectos do MOP e vinculação baseada em instância e classe.

Da análise comparativa entre os MOPs baseados em Java, conclui-se que aqueles baseados na modificação da JVM (interpretador) são extremamente transparentes (transparência implícita), contudo não permitem modificações sintáticas na linguagem. Em contrapartida, os MOPs construídos com cumplicidade do compilador permitem a adição de sintaxe reflexiva à linguagem. Pela técnica de *wrapping* ou *shadowing* conseguem um bom desempenho em tempo de execução, uma vez que a reificação só é ativada nos casos efetivos de vinculação. Entretanto, nestes casos paga-se o preço de menor flexibilidade e menor transparência.

A análise de características reflexivas em linguagens de *Scripting* é feita no Apêndice A.

Capítulo 3

Descrição da Arquitetura Reflexiva Guaraná

Expandindo a descrição apresentada na seção 2.3.2 sobre o MOP do **Guaraná**, apresentamos neste capítulo um estudo detalhado de sua estrutura, dinâmica, implementação e utilização. A função deste capítulo é repassar ao leitor a essência do **Guaraná**, fornecendo substrato teórico e prático para o entendimento dos capítulos subsequentes. Entretanto, quaisquer particularidades de **Guaraná** não abordadas neste capítulo podem ser pesquisadas na documentação original do MOP [50, 51, 52, 53, 54, 55, 56, 49].

3.1 Definindo Guaraná

De acordo com seus criadores [52]:

> ***Guaraná*** *é uma arquitetura reflexiva que almeja simplicidade, flexibilidade, segurança e reuso de código no meta-nível.*
>
> **Definição de Guaraná** Alexandre Oliva (1998)

Dissecando a definição acima, podemos desmembrar três significados[1] distintos para o **Guaraná**: um conjunto de diretrizes de projeto, um protocolo de meta-objetos (MOP), e uma implementação sobre uma linguagem de programação.

Na presença de implementações alternativas, torna-se mister distinguir quando o termo **Guaraná** se aplica a uma implementação em particular sobre uma determinada linguagem de programação. Até a redação desta dissertação a única implementação completa de **Guaraná**

[1] No dicionário Aurélio a palavra *guaraná* também possui três significados. Ainda que sem relação direta com esta dissertação, mas a título de curiosidade, são elas: 1.Bras. Grande cipó da floresta amazônica, cuja cápsula fornece semente rica em substâncias excitantes. 2. Massa fabricada com essas sementes. 3.Bebida feita com o pó dessa massa.

está baseada em modificações sobre uma JVM (*Java Virtual Machine*) de domínio público. Esta dissertação utiliza **Guaraná** versão 1.7 como implementação de referência. Entretanto, já existem planos para outras implementações, inclusive contemplando outras linguagens de programação diferentes de Java.

Quanto ao significado de MOP, **Guaraná** pode ser definido como um protocolo de comunicação dentro do paradigma OO que rege a interação de objetos no contexto de reflexão computacional. Essa definição desvincula de **Guaraná** os aspectos inerentes da linguagem de programação que hospeda a implementação, distinguindo-o das demais propostas de MOPs. Esta dimensão será abordada nas seções 3.3 e 3.4

Quanto ao primeiro significado, do três é o menos intuitivo e o mais importante. **Guaraná** representando um conjunto de diretrizes transcende as fronteiras da linguagem hospedeira da implementação e de uma estruturação particular de MOP. À luz deste conceito se torna possível falar sobre a implementação de outros MOPs sobre **Guaraná**. Ainda que surjam outros padrões de interação entre entidades OO, as diretrizes permanecem as mesmas. São elas que conferem ao **Guaraná** os atributos de simplicidade, flexibilidade, reusabilidade e segurança. Tais diretrizes não somente distinguem **Guaraná** das demais propostas de MOPs, como também justificam a arquitetura do MOP.

Uma vez que a essência de reflexão computacional consiste em interceptação e introspecção (seção 2.1), as diretrizes de **Guaraná** definem os limites sobre os quais será possível prover interceptação e introspecção, sem que sejam violados os atributos de segurança, reusabilidade, flexibilidade e simplicidade. Por conseguinte, o MOP de **Guaraná** define mecanismos de interceptação e introspecção condizentes com as diretrizes. Finalmente, a implementação de **Guaraná** concretiza tais mecanismos permitindo sua utilização prática.

3.2 Atributos e Diretrizes

Existem quatro atributos que devem ser sempre satisfeitos, qualquer que seja o significado escolhido para o **Guaraná**. São eles: segurança, flexibilidade, reusabilidade e simplicidade.

3.2.1 Segurança

A segurança é extremamente negligenciada pela maioria dos MOPs. A explicação reside na própria essência de reflexão, cuja finalidade é revelar meta-informação e permitir alteração de comportamento por interceptação. Enquanto a maioria dos MOPs se preocupa exclusivamente com a construção de mecanismos reflexivos eficientes e flexíveis, **Guaraná** se distingue dos demais pela preocupação adicional em definir limites a estes mesmos mecanismos. Todavia, em virtude da amplitude semântica do termo segurança, vamos nos restringir nesta seção à segurança como sinônimo robustez.

A preocupação com robustez motivou a seguinte diretriz do MOP:

> ***Isolamento entre meta-objeto primário e para-objeto:***
> *As interações entre meta-objeto primário e seu para-objeto subordinado devem ser sempre intermediadas por uma terceira entidade ilibada, o núcleo do MOP, responsável por assegurar a integridade da interação.*

Em outras palavras, a interação entre para-objeto e seu meta-objeto representa uma vulnerabilidade do MOP. Uma vez que meta-objetos e para-objetos podem ser construídos arbitrariamente, não há como garantir sua incorruptibilidade. Portanto, antes que se possa determinar quaisquer propriedades invariantes sobre as interações entre para-objeto e meta-objeto, faz-se necessário introduzir ao menos uma entidade participante que seja incorruptível. Este papel é assumido por entidades provenientes do núcleo de execução de **Guaraná** (inclusive o núcleo propriamente dito), que intermediam os aspectos de *vinculação*, *reificação*, *intervenção* e *execução*.

Existem duas manifestações [56] concretas desta diretriz no mecanismo de interceptação de operações. Primeiramente, uma dado meta-objeto não pode construir arbitrariamente uma operação destinada ao seu para-objeto, mas apenas operações oriundas de uma fábrica de operações. A fábrica assume o papel da entidade incorruptível, com o respaldo do próprio núcleo do **Guaraná**, que a disponibiliza para o meta-objeto em questão. Em segundo lugar, uma vez de posse de uma operação, o meta-objeto não possui meios de entregar a operação diretamente para seu para-objeto. Ao invés, utiliza o núcleo como intermediário na entrega. Neste caso o núcleo assume o papel da entidade incorruptível. Na ausência desta intermediação, seria possível quebrar protocolo de objetos no para-nível, como por exemplo através da entrega de uma operação inexistente no para-objeto alvo.

Só é possível garantir a incorruptibilidade do núcleo blindando alguns de seus componentes chaves contra as facilidades reflexivas oferecidas pelo próprio núcleo. Como foi discutido no início da seção 2.1, observamos que o requisito de segurança força com que a auto-representação do sistema reflexivo não seja idêntica a implementação efetiva do sistema. Ou seja, com o MOP de **Guaraná** não é possível obter meta-informação sobre componentes críticos do núcleo, nem tampouco interceptar ou modificar seu comportamento reflexivamente. Nominalmente, os componentes do MOP que não são passíveis de reflexão são as classes *Guarana*, *Operation* e *Result* e quaisquer instâncias das duas últimas.

Surge naturalmente a pergunta: A impossibilidade de refletir sobre componentes do núcleo do MOP representa uma violação ao princípio da *conexão causal*? Pela interpretação literal da definição de *conexão causal* poderia se pensar que sim. Entretanto, o MOP respeita a *conexão causal* para todo o domínio reflexivo que consiste no somatório de para-níveis, que podem por sua vez se manifestar concomitantemente como meta-níveis.

Relembrando, uma meta-nível reflete sobre um para-nível, que por sua vez pode atuar como meta-nível de outrem. Esse empilhamento é recursivo. A base de tal recursão é o núcleo

do MOP, não reflexivo por construção. Na ausência desta premissa, a mera existência de uma meta-entidade refletindo sobre o núcleo do MOP implicaria num ciclo infinito recursivo para qualquer que fosse a computação executada. Portanto, o MOP respeita a *conexão causal* no domínio reflexivo, do qual está excluso o núcleo propriamente dito do MOP, responsável pelos 4 aspectos fundamentais: vinculação, reificação, intervenção e execução.

Ainda em relação a amplitude semântica do atributo segurança, além de robustez, o MOP de **Guaraná** provê recursos para evitar a intervenção de meta-código malicioso. Contudo, em um cenário reflexivo onde para-entidades são passíveis de sofrerem introspecção e interceptação, como evitar a ação de meta-entidades maliciosas? Através da seguinte diretriz:

> ***Privilégio da Anterioridade***:
> *Um objeto não reflexivo pode ser vinculado incondicionalmente a qualquer meta-configuração. Entretanto, um objeto reflexivo (para-objeto) só será efetivamente vinculado a uma nova meta-configuração com o consentimento da prévia.*

Dessa forma, meta-entidades maliciosas teriam que se sujeitar a aprovação da meta-configuração pré-existente no para-objeto alvo. Potencialmente, a meta-configuração pré-existente pode fracassar ao impedir sua própria substituição ou a admissão de uma meta-configuração maliciosa. Seja o fracasso causada por excesso de permissividade ou falha de julgamento, o que importa é a existência do mecanismo. É sempre possível, pelo *Privilégio da Anterioridade*, pré-instalar meta-configurações intolerantes a sua própria substituição em aplicações que exijam máxima segurança. Esta medida torna-se necessária uma vez que todo para-objeto não reflexivo aceita por *default* qualquer meta-configuração.

É importante ressaltar que o MOP de **Guaraná** permite, ao invés de exigir, sua utilização de forma segura. Ou seja, em função dos requisitos da para-aplicação ou mesmo da meta-aplicação, os mecanismos de segurança podem ser descartados em favor de maiores desempenho e simplicidade. A facultatividade da aplicação dos mecanismos de segurança aumenta a flexibilidade do MOP. A título de exemplificação, uma forma insegura de utilizar o MOP seria configurar meta-objetos para que aceitassem sempre sua substituição na meta-configuração sem qualquer autenticação por parte do meta-objeto substituto.

3.2.2 Flexibilidade

A flexibilidade do MOP advém, principalmente, da fatoração do menor conjunto possível de mecanismos não-redundantes, abrangendo as facilidades de introspecção e interceptação. O que pode ser sintetizado na diretriz:

> ***Minimalidade e Completude***:
> *A interface entre uma meta-entidade e o núcleo reflexivo oferece um conjunto minimal de primitivas reflexivas sem que haja perda de funcionalidade quanto à introspecção ou interceptação.*

O requisito de minimalidade implica tanto em um baixo-nível de abstração no enfoque de cada primitiva, quanto em um baixo-acoplamento entre primitivas. Este binômio é sinônimo de flexibilidade.

Esta diretriz não justifica a adoção exclusiva de reificação *passiva* ou *de chegada* na implementação do MOP, onde foi excluído o caso simétrico de reificação *ativa* ou *de saída*. Este trabalho de pesquisa nos leva a acreditar que a reificação *ativa* ou *de saída* deveria ser incluída no MOP para que o MOP possa se dizer verdadeiramente "completo". Apesar da diretriz, a adoção exclusiva de reificação passiva não é abrangente o suficiente, tampouco minimal.

Não é possível através do uso exclusivo de reificação passiva se obter o mesmo conjunto de resultados que se obteria com a adoção de reificação ativa. O contra-exemplo que prova esta afirmação é o problema da determinação da cadeia de invocações de métodos no para-nível, que será ilustrado no capítulo 4.

Ainda relacionado ao atributo flexibilidade, outro fator co-responsável por conferir ao MOP de **Guaraná** esta propriedade foi a adoção de reflexão *dinâmica* ou em *tempo-de-execução*, o que será ilustrado na seção 3.4.

3.2.3 Reusabilidade

Apesar da facilidade de composição explícita e hierárquica de meta-objetos contribuir para o atributo reusabilidade do MOP, o fator que mais favorece o reuso é descrito na diretriz:

> ***Maximização da Transparência***:
> *Sempre que possível os inter-relacionamentos entre entidades devem ser transparentes nos âmbitos intra-nível e inter-nível. No âmbito inter-nível, em qualquer momento da computação é impossível determinar a partir de qualquer para-nível se um objeto está ou não presente em uma meta-configuração, salvo o objeto decida revelar sua presença. No âmbito intra-nível, em qualquer momento da computação é impossível para um meta-objeto determinar sua posição relativa aos outros meta-objetos que o antecedem hierarquicamente na meta-configuração a que pertence, salvo os meta-objetos decidam revelar sua presença.*

O primeiro desdobramento da diretriz sugere o desacoplamento entre para-nível e meta-nível. O fato de um para-nível não poder exigir um meta-nível específico é condição necessária

(porém ainda não suficiente) para que seja possível aplicar qualquer meta-nível sobre tal para-nível. Se todo meta-objeto somente utilizar os mecanismos de introspecção disponíveis para angariar informação sobre o para-objeto vinculado, então a condição torna-se também suficiente.

O segundo desdobramento contribui para a composição arbitrária de meta-configurações pré-existentes, cujo efeito combinado sobre o para-objeto alvo seja o somatório (superposição) dos efeitos independentes das meta-configurações componentes. Notoriamente, o efeito de superposição depende de quão ortogonais são as meta-configurações componentes entre si. Todavia, a *maximização da transparência* favorece a ortogonalidade.

Um corolário direto desta diretriz é a impossibilidade de uma dado meta-objeto determinar se é o meta-objeto primário da meta-configuração a que pertence. Sendo assim, o implementador de um meta-objeto é forçado a adotar uma postura não intrusiva, de modo a evitar interferência sobre outros meta-objetos que possam co-existir (ou não) na mesma meta-configuração.

3.2.4 Simplicidade

Poderia se argumentar que o atributo simplicidade é incompatível com os demais atributos em face dos seguintes argumentos:

- A introdução de regras e limitações para atender os requisitos de segurança aumentam a complexidade do MOP, que seria inquestionavelmente mais simples se os mesmos não existissem.

- A reusabilidade só é conquistada através da fragmentação da arquitetura do MOP em uma hierarquia de componentes fracamente acoplados mas altamente coerentes. A teia resultante de inter-relacionamentos entre módulos é nitidamente mais complexa que uma arquitetura monolítica.

- A adoção de um conjunto de primitivas reflexivas focadas em um baixo nível de abstração não torna sua utilização mais simples do que se o de primitivas reflexivas com alto nível de abstração.

Entretanto, pode-se contra-argumentar que a flexibilidade simplifica a utilização do MOP, a reusabilidade simplifica sua manutenção e a segurança sua gerência. Paradoxalmente, o trinômio reusabilidade-flexibilidade-segurança prejudica a compreensão da arquitetura interna do MOP enquanto favorecem a da arquitetura externa.

Essa dialética da simplicidade transcende a fronteira do **Guaraná** e de protocolos de meta-objetos, sendo verdadeira em qualquer domínio da Engenharia de Software.

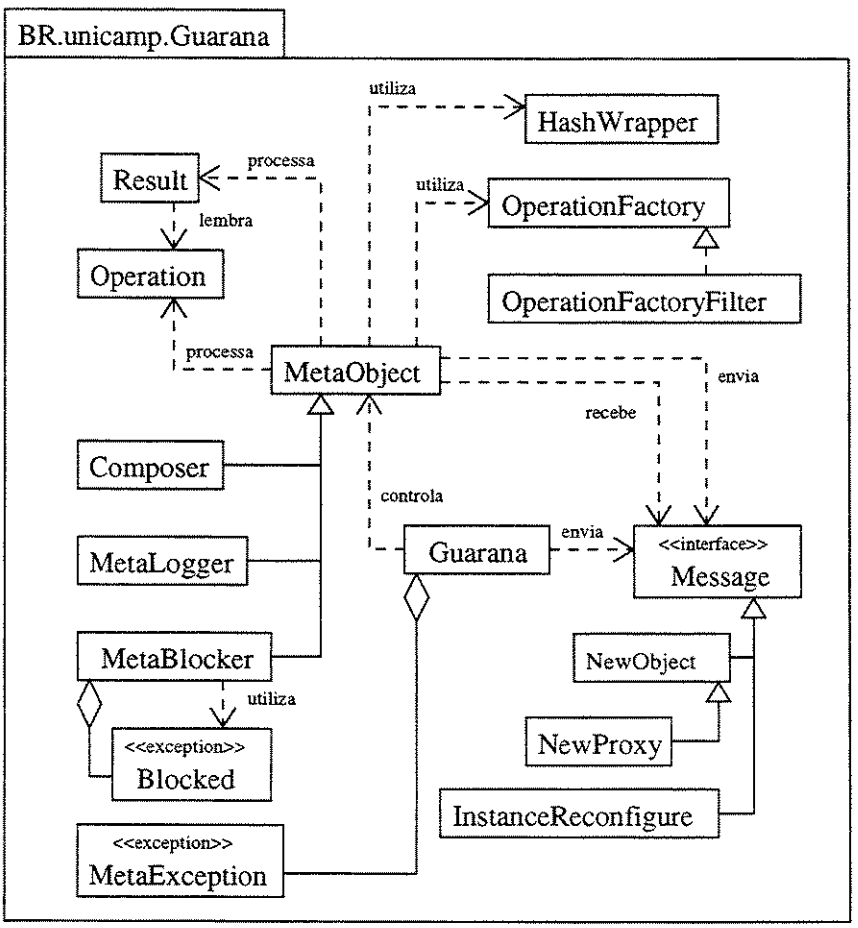


Figura 3.1: Diagrama de Classes

3.3 Estrutura do MOP

A estrutura do MOP do **Guaraná** está representada na Figura 3.1, utilizando-se diagramas de classes em **UML** [6, 48].

Como se pode observar, a entidade central do MOP é a classe *MetaObjeto*. Dessa observação, deriva naturalmente a questão: Por que a classe ***Guaraná*** não é a entidade central? Simplesmente, porque preferimos dar ênfase a perspectiva do usuário do MOP, ao invés da perspectiva do projetista do MOP.

A classe *MetaObject* é o ponto de partida para a definição e construção de entidades de meta-nível. Dividindo a figura 3.1 em quadrantes ordenados no sentido horário, simplificamos a análise dos relacionamentos entre *MetaObject* e as demais meta-entidades.

No primeiro quadrante (topo à esquerda), *MetaObject* relaciona-se com *Operation* e *Result*. Este relacionamento modela o aspecto *reificação* do MOP. Meta-objetos (instâncias de *MetaObject*) processam operações e seus resultados. É interessante observar, no âmbito do para-nível, que resultados são sempre **first-class objects** independentemente da linguagem de programação. Onde *first-class objects* são entidades que se comportam como qualquer tipo primitivo de uma dada linguagem de programação, podendo ser armazenadas em mídia permanente ou volátil, e passadas como parâmetro de entrada ou retorno de rotinas.

No caso de operações, entretanto, existe dependência da linguagem de programação. Através do processo de reificação essa assimetria desaparece. Tanto operações quanto seus resultados são convertidos em meta-informação e encapsulados em entidades do meta-nível, respectivamente instâncias de *Operation* e *Result*. Grosseiramente, podemos interpretar a reificação como um mecanismo de promoção de para-entidades a *first-class object* de meta-nível.

No segundo quadrante, *MetaObject* relaciona-se com *OperationFactory*, *OperationFactoryFilter* e *HashWrapper*. Vamos analisar inicialmente os dois primeiros.

Como o próprio nome já sugere, existe uma relação de dependência entre *Operation* e *OperationFactory*. Essa relação foi omitida do diagrama 3.1 para evitar poluição visual. Rememorando a discussão sobre o atributo segurança, instâncias de *OperationFactory* servem ao propósito de controlar o espectro de operações que um meta-objeto pode impingir sobre o para-nível. Ou seja, o meta-objeto utiliza a fábrica de operações que lhe foi fornecida para produzir operações destinadas ao seu para-objeto vinculado. É importante ressaltar, que é permissível ao meta-objeto primário compartilhar a instância da fábrica de objetos que lhe foi conferida entre os demais meta-objetos pertencentes a mesma meta-configuração.

O uso de *OperationFactory* é uma manifestação do aspecto *intervenção* do MOP, através do qual o meta-nível tem o poder de intervir no estado e comportamento do para-nível. Ainda neste contexto, *OperationFactoryFilter* e *HashWrapper* desempenham papéis de menor relevância. O primeiro atua como um filtro restritivo para uma fábrica mais permissiva. O último é necessário devido a uma particularidade da linguagem Java (hospedeira da imple-

mentação de referência). Sua finalidade é blindar o processo de reificação sobre estruturas de dados, presentes em uma dada meta-configuração, e que referenciem implicitamente o para-objeto vinculado a mesma meta-configuração. Na ausência dessa blindagem, um acesso a tal estrutura poderia gerar um ciclo infinito de reificação, que será discutido em detalhe na seção 4.3.2.

No terceiro quadrante, observamos o primeiro nível da hierarquia de especialização de *MetaObject* e as classes básicas utilizadas para reificar o comportamento excepcional do meta-nível. Quanto a hierarquia de especialização, existem três elementos básicos:*MetaBlocker*, *Composer* e *MetaLogger*.

Instâncias de *MetaBlocker* são as meta-entidades vinculadas por *default* a qualquer para-entidade, entre o instante de criação do para-objeto e sua efetiva vinculação a meta-entidade válida que requisitou tal vinculação. Em termos práticos, as instâncias de *MetaBlocker* oferecem aos para-objetos uma blindagem contra corrupção da vinculação legítima. Na ausência de *MetaBlocker*, seria viável que um meta-objeto malicioso assumisse o cargo de meta-objeto primário durante o processo de vinculação, sobrepujando o meta-objeto que legitimamente requisitou a vinculação.

A classe *Composer* é por sua vez a base de uma hierarquia de especialização, ainda que não representada no diagrama 3.1. Sua existência sugere que meta-configurações podem ser constituídas por mais de uma meta-entidade. Idealmente, instâncias de *Composer* ou derivados atuam como meta-entidades redistribuidoras de meta-informação entre outros meta-objetos. Considerando que são também meta-objetos, é possível construir uma hierarquia arbitrária de meta-objetos através do encadeamento de *Composer*s com diferentes semânticas de interconexão.

Poderia se argumentar o porquê da inclusão de *Composer* no núcleo de **Guaraná**? Considerando que o mesmo é uma particularização de um *MetaObject*, tal decisão não fere o atributo de simplicidade? Não. A razão da presença de *Composer* no núcleo de **Guaraná** é servir de referencial para a construção de diferentes semânticas de interconexão de meta-objetos. Através uso de *Composer* e derivados, é possível navegar em uma meta-configuração, distinguindo os meta-objetos que possuem uma semântica própria daqueles que atuam meramente como delegadores.

Já a classe *MetaLogger*, diferentemente das demais, não é imprescindível na constituição do MOP. Entretanto, atende a dois desígnios práticos. Serve como exemplo de meta-objeto simples e pleno de funcionalidade, assim como é útil no papel de ferramenta de depuração.

No quarto e último quadrante encontramos a classe *Guarana* e a hierarquia de especialização *Message*. A classe *Guarana* representa a interface para o núcleo do MOP e, sendo assim, não permite a criação de instâncias. Através desta interface se manifestam os aspectos *vinculação* e *execução* do MOP.

Quanto ao aspecto *vinculação*, ***Guaraná*** atende requisições de estabelecimento de vínculo entre para-objeto e meta-objeto.

Novamente, é enfatizado o atributo segurança do MOP, onde a classe *Guarana* atua como juiz de requisições de vinculação, assegurando o *Privilégio de Anterioridade* dos meta-objetos já vinculados em detrimento dos meta-objetos suplicantes.

A outra faceta da classe *Guarana* é veicular um meio de comunicação com entidades de uma meta-configuração, sem ferir a diretriz *Maximização da Transparência.* Em outras palavras, sem revelar a presença e identidade dos componentes da meta-configuração. Isto é possível através da semântica de comunicação em massa ou **broadcast**, cujo único parâmetro de identificação é a para-entidade alvo. O núcleo garante apenas que: se meta-entidade primária existir, então ela receberá a mensagem. Entretanto, a dinâmica de propagação da mensagem dentro da meta-configuração é dependente da semântica e estruturação da própria meta-configuração. O remetente da mensagem não possui meios de verificar se a mensagem alcançou algum destinatário, salvo este último assim o queira. O papel que cada mensagem assume será discutido na seção 3.4.

3.4 Dinâmica do MOP

A dinâmica do MOP, para efeito didático, pode ser fracionada nos seguintes processos:

Vinculação é a primeira associação de uma meta-configuração a um para-objeto ainda não-reflexivo, tornado-o reflexivo.

Atuação é toda e qualquer participação dos meta-objetos de uma dada meta-configuração sobre o para-objeto vinculado.

Propagação é toda vinculação originada automaticamente por uma meta-configuração, cuja finalidade é a instalação da própria meta-configuração sobre um para-objeto criado por uma para-objeto previamente vinculado à meta-configuração em questão.

Reconfiguração é cada alteração da constituição de uma meta-configuração já vinculada a um para-objeto.

Considerando a dinâmica de uma meta-configuração em particular, após a *vinculação*, qualquer combinação dos outros três processos pode ocorrer, cujo sequenciamento efetivo só é definido em tempo de execução.

Os quatro processos supra-citados possuem como denominador comum o envolvimento de para-nível e meta-nível.

Existe ainda o processo de **comunicação**, que se distingue dos demais por ocorrer exclusivamente no meta-nível, ainda que os destinatários da comunicação sejam especificados pelo

para-objeto a que estão vinculados direta ou indiretamente. Define-se então *comunicação* como sendo a troca síncrona de objetos de meta-nível (instâncias de sub-classes de *Message*) entre meta-objetos.

Nesta seção vamos examinar cada um dos processos separadamente.

3.4.1 Vinculação

O processo de *vinculação* é o único com ordem de ocorrência bem definida, por ser sempre o primeiro a ocorrer. A vinculação sempre define o meta-objeto primário do para-objeto alvo. Por construção, só existe vinculação propriamente dita entre o para-objeto e o meta-objeto primário. Por intermédio do meta-objeto primário, diz-se que o para-objeto está também vinculado aos demais meta-objetos componentes da meta-configuração. No caso de uma meta-configuração não-unitária, tipicamente o meta-objeto primário será instância de uma sub-classe de *Composer* (vide diagrama 3.1).

Para efetuar uma vinculação basta invocar o método **reconfigure(paraObject, oldMetaObject, newMetaObject)** da classe *Guarana*. Onde *paraObject* é uma referência para o para-objeto alvo, *oldMetaObject* deve ser nulo e *newMetaObject* é uma referência para o meta-objeto primário da meta-configuração a ser vinculada.

Antes que a vinculação seja estabelecida, o núcleo do MOP irá verificar se a *para-classe*, de quem o para-objeto foi instanciado, é reflexiva ou não. Em caso afirmativo, o meta-objeto vinculado a para-classe será notificado da intenção de vinculação através de uma mensagem **InstanceReconfigure**. Esta mensagem é um objeto de meta-nível que carrega os parâmetros *paraObject* e *newMetaObject* da requisição de vinculação. Este mecanismo permite que a meta-configuração associada a para-classe do para-objeto alvo seja consultada antes que a vinculação seja efetivada. Através desta consulta, a meta-configuração da para-classe pode aceitar, recusar ou substituir a meta-configuração a ser vinculada.

A cada nova vinculação estabelecida, o meta-objeto envolvido é notificado pelo núcleo através da primitiva **initialize(OperationFactory,paraObject)**. Neste momento, o meta-objeto recebe uma referência para a fábrica de operações válidas sobre o para-objeto que está sendo vinculado. Um erro comum é supor que a primitiva *initialize* está associada a construção do para-objeto. Com efeito, por ocasião do tratamento do método *initialize*, o meta-objeto alvo não tem como saber se o construtor do respectivo para-objeto já foi invocado. Ou seja, a primitiva *initialize* indica inicialização do meta-objeto e não do para-objeto. A detecção da inicialização do para-objeto é feita através da reificação da chamada ao construtor do para-objeto, a qual é notificada ao respectivo meta-objeto através da primitiva *handle(Operation,paraObject)*.

Por simetria, a cada desvinculação entre para-objeto e meta-objeto, o último é notificado através da primitiva **release(paraObject)**. Vale reforçar a inexistência de qualquer relação entre a primitiva *release* e a efetiva destruição (ou desalocação) do para-objeto.

A título de curiosidade, na implementação de referência de **Guaraná**, a vinculação é implementada através da adição de um campo oculto em cada objeto. Se este campo possui um valor diferente de nulo, então seu valor será uma referência para um meta-objeto vinculado. A manipulação deste campo ocorre exclusivamente no *inner sanctum* da JVM, não representando assim violação no atributo de segurança do MOP. Esta abordagem de implementação favorece o desempenho do teste se uma para-objeto é reflexivo ou não. Outra abordagem considerada viável, porém descartada devido ao desempenho inferior, seria a criação de um dicionário interno a JVM, representando as vinculações através de pares ordenados de referências. MOPs cuja implementação não modifica a JVM são forçados a adotar esta última abordagem.

Ao utilizarmos o termo *vinculação* nesta seção, estamos supondo implicitamente que o para-objeto alvo não é ainda reflexivo. Quando o para-objeto alvo já for reflexivo, o processo chama-se reconfiguração e será descrito na seção 3.4.3.

3.4.2 Atuação

Na terminologia do MOP de **Guaraná**, um para-objeto sofre e exerce **operações**. Um acesso de leitura ou escrita a atributos é uma operação sobre o para-objeto detentor de tais atributos. Da mesma forma, a invocação de um método pertencente a um para-objeto é também uma operação. Na terminologia OO utiliza-se preferivelmente *envio de mensagem* em detrimento de *invocação de método*. Ambas as expressões denotam o mesmo fenômeno. Nesta dissertação, preferimos a segunda à primeira para evitar ambiguidade, pois *envio de mensagem* também significa a comunicação síncrona entre meta-objetos através da troca de objetos de meta-nível, no MOP de **Guaraná**.

O processo de *atuação* consiste na manipulação de operações destinadas a um dado para-objeto por parte da meta-configuração vinculada. Sem perda de generalidade, vamos considerar apenas meta-configurações unitárias nesta seção. No intuito de facilitar a descrição, apresentamos diagramas de colaboração (figuras 3.2 e 3.3) e atividade (figura 3.4), seguindo a padronização UML [6, 48].

Na figura 3.2, observamos uma interação típica entre dois objetos de para-nível. O objeto *Caller* invoca o método *someMethod* presente na interface do objeto *Target*. Após o processamento do método, *Target* produz como resultado *result*, entregando-o a *Caller*.

A figura 3.3 introduz neste cenário um meta-objeto primário *MO* vinculado ao para-objeto *Target*. Neste diagrama de colaboração foi omitida a participação de entidades do núcleo de **Guaraná** internas a JVM. Essa omissão se justifica por estarmos modelando o nível de abstração em que atua o programador de meta-nível. As partes omitidas pertencem ao nível de abstração em que atua o programador do MOP propriamente dito.

Segundo o diagrama, a invocação de *someMethod* (passo 1) é interceptada e reificada pelo núcleo do MOP no objeto de meta-nível *op*, que por sua vez é uma instância da classe

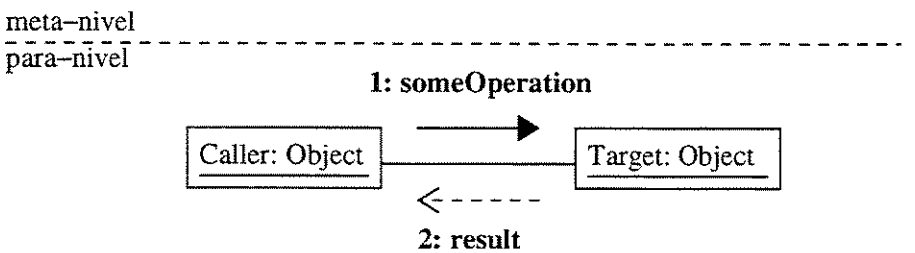


Figura 3.2: Diagrama de Colaboração – Interação básica não-reflexiva

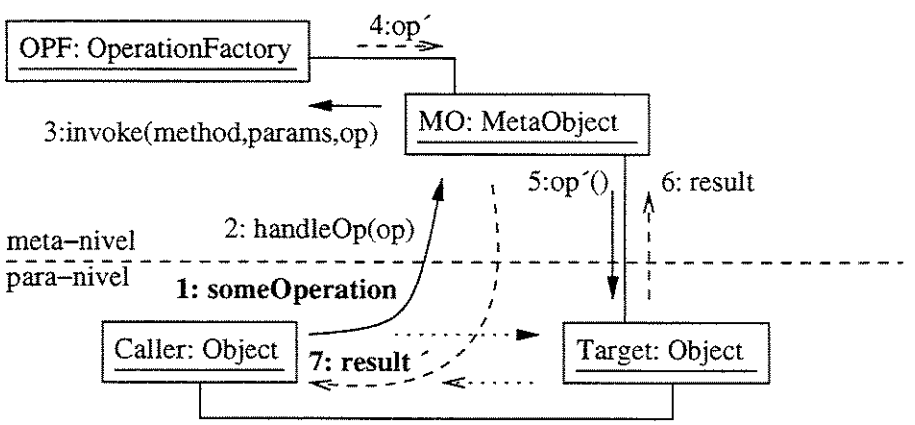


Figura 3.3: Diagrama de Colaboração – Interação básica reflexiva

Operation. No passo 2, o núcleo do MOP invoca o método *handleOperation* do meta-objeto primário *MO*, notificando a intenção do para-nível em exercer a operação *op* sobre o para-objeto *Target*.

Assim que *MO* é notificado da chegada de *op*, existem três desdobramentos possíveis:

- *MO* decide não intervir e repassar a operação incólume para *Target* através do passo 5;
- *MO* decide responder a operação diretamente através do passo 7, sem que haja nenhum repasse para *Target*;
- *MO* decide intervir e substituir *op* por uma outra operação antes do repasse para *Target*;

Considerando a última alternativa, devido aos requisitos de segurança já discutidos, *MO* não pode fabricar por si só uma operação destinada a *Target*. É necessário utilizar *OPF*, que é a fábrica de operações para *Target* autorizada pelo núcleo do MOP. Portanto, no passo 3 *MO* requisita a fabricação de uma nova operação *op'* que será repassada para *Target* no lugar da operação original *op* requisitada por *Caller*.

Antes de qualquer repasse, *MO* precisa informar o núcleo se tem intenção de ignorar, observar ou modificar o resultado produzido por *Target* após o repasse. Tais detalhes foram omitidos na figura 3.3, porém estão presentes na figura 3.4. Neste último diagrama estão explícitos tanto o fluxo de controle (flechas contínuas) quanto o fluxo de dados (flechas tracejadas).

É importante ressaltar que *MO* não possui meios de descobrir a identidade de *Caller* através de *op*, salvo *Caller* tenha explicitamente publicado uma referência para si mesmo entre os parâmetros de *op*. Essa limitação se deve ao modelo de *reificação passiva* ou *de chegada* adotado em **Guaraná**.

A atuação de um dado meta-objeto está codificada no tratamento das primitivas **handle(Operatrion,paraObject)** e **handle(Result,paraObject)**, correspondendo aos passos 2 e 6 da figura 3.3, respectivamente.

Um erro frequente na utilização do MOP é a invocação de algum método do para-objeto a partir do corpo do método *handle(Operatrion,paraObject)* de seu meta-objeto. Essa invocação gera uma operação de para-nível que será reificada e novamente submetida ao mesmo método *handle(Operatrion,paraObject)* do mesmo meta-objeto. Sendo assim, se não houver um desvio condicional o meta-programa entra em recursão infinita. Este cenário é típico do programador ainda incipiente no MOP, do qual deve-se suspeitar quando há erro de estouro de pilha da JVM. No intuito de evitar este ciclo de interceptação, alguns métodos típicos de qualquer para-objeto foram replicados na classe *Guarana* sem que houvesse ativação do mecanismo de reificação, com a ressalva de que o para-objeto alvo deva ser informado como

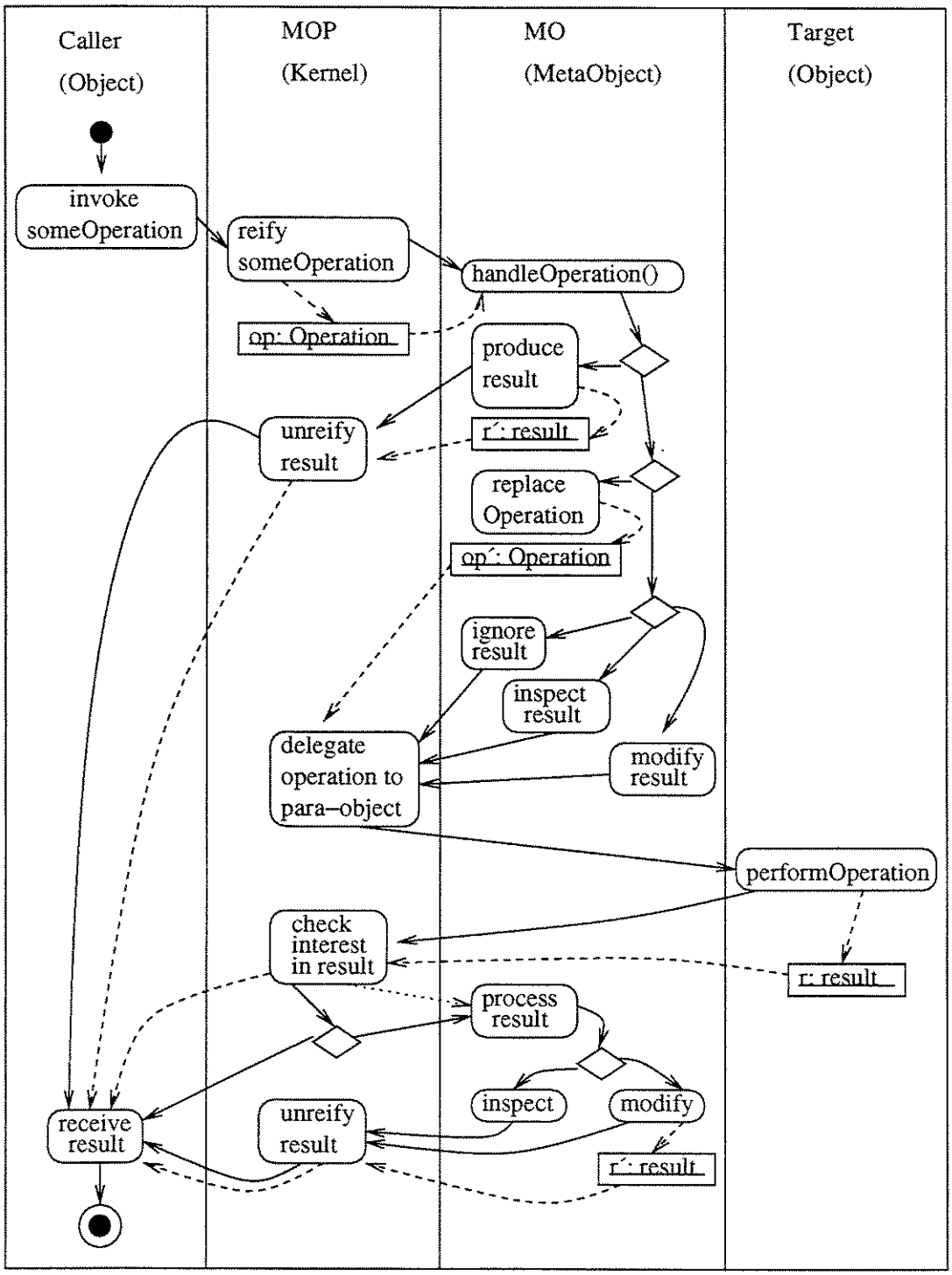


Figura 3.4: Diagrama de Atividade – Interação básica reflexiva

parâmetro. Para os casos não suportados pela classe *Guarana*, cabe ao meta-programador tomar precauções de forma a evitar a recursão infinita.

3.4.3 Reconfiguração

A mesma primitiva utilizada no processo de *vinculação* é também utilizada no processo de *reconfiguração*. Para efetuar uma reconfiguração basta invocar o método *reconfigure(paraObject, oldMetaObject, newMetaObject)* da classe *Guarana*.

O parâmetro *paraObject* especifica indiretamente através de uma referência para o para-objeto alvo, qual meta-configuração deve sofrer alteração. Essa forma de indexação obedece a diretriz de *Maximização de Transparência*, possibilitando àquele que requisita a reconfiguração desconhecer a constituição da meta-configuração a ser reconfigurada. O parâmetro *oldMetaObject* identifica a porção da meta-configuração alvo que deve sofrer alteração. O parâmetro *newMetaObject* identifica a natureza da alteração.

Pela diretriz de *Privilégio da Anterioridade*, a primitiva de *reconfigure* possui caráter meramente informativo. Cabe ao meta-objeto alvo analisar seus parâmetros, efetuando ou não o requerimento de alteração à sua discrição.

A título de exemplificação, analisemos alguns cenários típicos onde o meta-objeto *MO* vinculado ao para-objeto *PO* recebe :

reconfigure(PO,MO,MO')

Neste cenário, o parâmetro oldMetaObject indica que o próprio meta-objeto primário *MO* é a porção da meta-configuração a ser alterada. A alteração consiste em tornar *MO'* o novo meta-objeto primário, tal como é indicado pelo parâmetro newMetaObject.

Se *MO'* é um meta-objeto distinto de *MO* e ambos pertencem a meta-configurações unitárias, então a requisição *reconfigure* é um pedido de substituição de meta-configuração. Entretanto, *MO'* pode ser também a raiz de uma hierarquia de meta-objetos, dentre os quais faça parte *MO*. Neste caso, ao invés de uma substituição estará ocorrendo um incremento na meta-configuração. Sendo assim, a semântica da requisição *reconfigure* é definida dinamicamente em função da constituição de seus parâmetros e da meta-configuração alvo.

reconfigure(PO,MO,null)

Neste cenário, assim como no anterior, o parâmetro *oldMetaObject* identifica a meta-configuração *MO* como o alvo da alteração. No entanto, o valor nulo do parâmetro *newMetaObject* significa que a meta-configuração alvo deve ser removida e nenhuma outra colocada em seu lugar. Caso esta requisição seja aceita por *MO*, o para-objeto *PO* pode deixar de ser reflexivo.

reconfigure(PO,null,MO')

Este cenário é semanticamente equivalente ao primeiro, cuja única diferença reside no parâmetro *oldMetaObject* que agora é nulo. Aqui, o parâmetro nulo pode ser interpretado como máscara para qualquer que seja a meta-configuração instalada. Essa possibilidade é consoante com a diretriz da *maximização da transparência*.

Se o para-objeto *PO* não fosse reflexivo, recairíasse num cenário de vinculação ao invés de reconfiguração. Logo, pode-se dizer que a vinculação é uma caso particular da reconfiguração, em que parâmetro *oldMetaObject* é nulo.

reconfigure(PO,MO',MO")

Neste cenário, o meta-objeto alvo *MO"* é diferente do meta-objeto que efetivamente recebe a requisição *MO*. Se *MO* faz parte de uma meta-configuração unitária, então a requisição será ignorada. Caso contrário, *MO* é um *Composer*, possuindo um hierarquia de meta-objetos para os quais delega requisições. Dessa forma, a requisição de *reconfigure* será repassada para a hierarquia de delegação na expectativa de que o meta-objeto *MO"* seja encontrado.

Outra observação importante a cerca do processo de reconfiguração é quanto a estensibilidade do protocolo de reconfiguração. Todos os cenários apresentados anteriormente ilustram a semântica sugerida para a interpretação dos parâmetros de *reconfigure*. No entanto, qualquer semântica conferida aos parâmetros depende única e exclusivamente da efetiva implementação do meta-objeto que recebe a requisição. Sendo assim, torna-se possível criar novos protocolos e mecanismos de reconfiguração sobre a primitiva de *reconfigure*.

Ainda relacionado a estensibilidade do protocolo de reconfiguração, tecemos uma crítica quanto a ambiguidade semântica do valor nulo. Sob a perspectiva do meta-objeto alvo da reconfiguração, é impossível distinguir se o suplicante quis dizer "qualquer meta-objeto" ou "nenhum meta-objeto" ao utilizar o valor nulo em um dos parâmetros formais de *reconfigure*. Essa imprecisão dificulta a decisão de aceitação ou rejeição do pedido de reconfiguração. Esta ambiguidade poderia ser eliminada com o uso de constantes pré-definidas pela classe *Guarana*, tornando inválida a utilização do valor nulo na primitiva de reconfiguração.

É importante observar que em todos os cenários, o primeiro parâmetro sempre foi *PO*, que é o para-objeto vinculado a *MO*. O MOP de **Guaraná** garante que um meta-objeto nunca será requisitado a atuar sobre um para-objeto com qual não esteja vinculado. Por conseguinte, existem duas razões que justificam a existência do parâmetro *paraObject*, não só na primitiva *reconfigure*, mas em todas as primitivas da interface de um meta-objeto. A

primeira razão é a possibilidade de um meta-objeto primário estar vinculado diretamente a mais de um para-objeto, cuja recíproca é falsa. A segunda razão é possibilitar que o meta-objeto seja implementado **stateless**, ou seja, sem que haja a necessidade do meta-objeto armazenar e gerenciar internamente referências a para-objetos. Essa propriedade facilita a manutenção da consistência do meta-objeto num cenário de propagação.

3.4.4 Propagação

O processo de *propagação* rege como meta-configurações estendem sua influência por ocasião da criação de para-objetos. Quando um para-objeto é criado, existem dois contextos que podem desejar configurá-lo reflexivamente. O primeiro é o *contexto comportamental*, representado pelo para-objeto criador. O segundo é o *contexto estrutural*, representado pela para-classe a partir da qual o para-objeto foi instanciado.

O contexto comportamental possui maior prioridade na configuração reflexiva do para-objeto recém criado. Quando o para-objeto criador é reflexivo, seu meta-objeto primário é notificado sempre que um novo para-objeto é criado. Essa notificação se manifesta através da invocação da primitiva **configure(newParaObject,currentParaObject)** presente na interface do meta-objeto primário. Cabe ao meta-objeto primário definir a composição inicial da nova meta-configuração a ser instalada sobre o para-objeto recém criado. Entre as alternativas possíveis, o meta-objeto primário pode: propagar-se para a nova meta-configuração, delegar a decisão para os demais meta-objetos da meta-configuração a que pertence, ou até mesmo não contribuir, tampouco se propagar.

O contexto estrutural possui menor prioridade na configuração reflexiva do para-objeto recém criado. Quando a para-classe é reflexiva, seu meta-objeto primário é notificado sempre que um para-objeto instância dessa classe é criado. Essa notificação se manifesta através do recebimento da mensagem **NewObject** pelo meta-objeto primário da para-classe. Ao receber a mensagem, o meta-objeto primário da para-classe possui as mesmas alternativas que o meta-objeto primário do para-objeto criador possuía.

Existe uma diferença essencial entre o contexto comportamental e o estrutural. O contexto comportamental por ser o primeiro, sempre que se propagar se propagará através de *vinculação*, pois a meta-configuração do para-objeto recém criado estará vazia. Já no contexto estrutural há dependência da decisão do contexto comportamental e da existência ou não de meta-objetos na meta-configuração alvo. Desta forma, a propagação poderá transcorrer por *vinculação* se a meta-configuração alvo estiver vazia, ou por *reconfiguração* caso contrário. O último caso é discutido *a posteriori* na seção 4.3.3 sobre dupla propagação de meta-configuração.

3.4.5 Comunicação

O processo de *comunicação* entre meta-objetos, que não possuam referências entre si, é possível através da troca de mensagens. Tais mensagens são objetos de meta-nível, instanciados a partir de uma classe que implemente a interface **Message**.

Respeitando a diretriz de *Maximização da Transparência*, não é possível enviar este tipo de mensagem diretamente a um meta-objeto. Mensagens são sempre endereçadas por uma referência a um para-objeto, cujo destinatário efetivo é o meta-objeto primário associado ao para-objeto endereçado. Caso o para-objeto não seja reflexivo, a mensagem não será entregue, nem tampouco haverá efeito colateral de qualquer espécie. A inércia na ausência do destinatário é fundamental a fim de que não seja violada a diretriz de *Maximização da Transparência*, pois não deve ser possível testar a reflexividade de um para-objeto.

Todavia, o requisito de transparência introduz um empecilho ao meta-programador. Quando o último procura estender o MOP utilizando um protocolo próprio de mensagens, não há como saber se os meta-objetos destinatários receberam ou compreenderam a mensagem. Ou seja, a diretriz de transparência fortalece o MOP do ponto de vista de segurança, fomenta a independência e o desacoplamento entre meta-configurações ortogonais, porém pode se tornar inconveniente durante a meta-programação. Este problema é discutido posteriormente na seção 4.3.2.

Para enviar mensagens, utiliza-se o método **broadcast(paraObject,message)** da classe *Guarana*. Algumas mensagens são enviadas diretamente pelo núcleo do MOP, como foi mencionado nos casos de *InstanceReconfigure* e *NewObject*. Do ponto de vista da recepção, cada meta-objeto é notificado da chegada de uma mensagem através da primitiva **handle(Message,paraObject)**. Este mecanismo de comunicação comporta-se de forma síncrona, ou seja, uma invocação de *broadcast* encerra-se quando o último meta-objeto notificado retorne do tratamento de *handle(Message,paraObject)*.

3.5 Padrões de Projeto

Nesta seção, vamos revisitar os principais componentes do MOP de **Guaraná** abordando-os através da linguagem de **padrões de projeto** ou **design patterns** [17]. Por um lado, objetivamos facilitar o entendimento da arquitetura do MOP para aqueles já familiarizados com a terminologia de *padrões*. Por outro lado, o enquadramento dos componentes do MOP no arcabouço de padrões de projeto realça a qualidade do projeto original.

3.5.1 Padrões de Criação

Originalmente denominados **creational patterns**, os padrões de criação abstraem o processo de instanciação. Nessa categoria identificamos o componente *OperationFactory* como sendo

Tabela 3.1: OperationFactory mapeado no padrão Abstract Factory

Participante	**Componente do MOP**
AbstractFactory	classe *OperationFactory*
ConcreteFactory	instância de *OperationFactory* passada como parâmetro no método *initialize* de um dado *MetaObject*
AbstractProduct	classe *Operation*
ConcreteProduct	instância de *Operation* passada como parâmetro no método *handleOperation* de um dado *MetaObject*
Client	uma dada instância de *MetaObject*

uma manifestação do padrão **Abstract Factory**. Através da aplicação deste padrão, é possível parametrizar o conjunto de operações de para-nível que um meta-objeto pode aplicar sobre seu respectivo para-objeto. Onde o conjunto parametrizado pode ser qualquer sub-conjunto do universos de operações suportado pelo para-objeto.

Entretanto, a efetiva criação das operações é dissociada do meta-objeto por razões de segurança, cuja responsabilidade recai sobre a *OperationFactory*. O efeito desejado de dissociação justifica a escolha do padrão *Abstract Factory* em detrimento de outras alternativas como o padrão **FactoryMethod**. Por sua vez, a natureza dos objetos criados (operações de para-nível) não justificava tampouco a utilização do padrão **Builder**, que é mais adequado para modelar a criação de objetos estruturalmente complexos.

A tabela 3.1 estabelece o mapeamento entre os diversos participantes do padrão *Abstract Factory* e os respectivos componentes do MOP.

No que tange à propagação de meta-configurações, através da primitiva *configure*, cabe a cada meta-objeto que compõe a meta-configuração decidir se vai propagar-se ou não, e como. Em caso afirmativo, o meta-objeto é livre para implementar o padrão de criação que mais lhe convém. Quando o método de propagação escolhido for clonagem, identificamos uma manifestação do padrão **Prototype**.

Ainda relacionado a padrões de criação, podemos identificar a classe *Guarana* como possuidora de características equivalentes ao padrão **Singleton**, unicidade e acesso global, ainda que possua uma implementação diferente. Por razões de segurança, e em detrimento de flexibilidade, optou-se pela implementação dos serviços de *Guarana* através de métodos de classe.

Tabela 3.2: Composer mapeado no padrão Composite

Participante	**Componente do MOP**
Composite	instância de *Composer*
Leaf	instância de *MetaObject*
Component	classe *MetaObject* que define interface comum a todos elementos da composição
Client	instância de *MetaObject* ou núcleo do MOP (*Guarana*)

3.5.2 Padrões Estruturais

No que tange aos **padrões estruturais** ou **structural patterns**, a manifestação mais óbvia é o binômio padrão **Composite** – classe *Composer*.

A tabela 3.2 estabelece o mapeamento entre os diversos participantes do padrão *Composite* e os respectivos componentes do MOP.

Existem duas razões para tal escolha. A primeira razão é tornar indistinguível um conjunto de meta-objetos (meta-configuração) de um meta-objeto individual. Sendo assim, uma meta-configuração composta por diversos meta-objetos atua em uníssono como se fosse uma única entidade e, portanto, pode ser manipulada como tal pelas demais meta-entidades.

A segunda razão é disponibilizar um mecanismo mais flexível de composição do que o padrão **Chain of Responsibility**, cuja estruturação produz um encadeamento serial de elementos. Já o padrão *Composite* oferece um encadeamento hierárquico em forma de árvore.

Cabe aqui uma ressalva teórica. Enquanto *Composite* é classificado como *padrão estrutural*, *Chain of Responsibility* é classificado como *padrão comportamental*. Portanto, a primeira razão é de natureza estrutural, enquanto a segunda é de natureza comportamental.

Ainda em relação a padrões estruturais, instâncias de *MetaObject* podem assumir características equivalentes aos padrões **Flyweight** e **Proxy**.

No caso de *Flyweight*, um meta-objeto pode estar vinculado a múltiplos para-objetos, potencialmente todos os para-objetos existentes em uma dada computação. Isto se torna viável através da construção de meta-objetos **stateless**, onde a informação de estado é extrínseca ao meta-objeto, residindo exclusivamente nos para-objetos em questão.

No caso de *Proxy*, uma dado meta-objeto atua *a priori* como um intermediário de seu para-objeto vinculado, onde o primeiro assume a responsabilidade sobre o controle de acesso ao último. Este comportamento é construído implicitamente pelo núcleo do MOP em associação ao interpretador da linguagem (JVM).

3.5.3 Padrões Comportamentais

Além do *Composer* e sua relação com o padrão *Chain of Responsibility*, o padrão comportamental mais recorrente na arquitetura do MOP é **Observer**.

Essencialmente, o padrão *Observer* define uma relação de dependência um-para-muitos. Nesta relação quando o estado do objeto foco (participante **subject**) sofre alteração, todos seus objetos dependentes (participantes *observer*) são notificados.

Sendo assim a relação estabelecida entre um para-objeto e seu meta-objeto vinculado pode ser interpretada como uma relação *subject-observer*, onde os eventos que disparam as notificações são interações de para-nível na interface do para-objeto(subject). O mecanismo de *broadcast* de mensagens e os diversos tratadores *handleMessage* também se enquadram na filosofia de *subject-observer*.

3.6 Sumário

Neste capítulo exploramos o **Guaraná** nas três dimensões possíveis: a implementação (em Java), o protocolo de meta-objetos (o MOP), e a filosofia (conjunto de diretrizes).

A apresentação de **Guaraná** se fez através do exame de seus atributos mais notórios: segurança, flexibilidade, reusabilidade e simplicidade. No decorrer desta apresentação foram elencadas as seguintes diretrizes:

- Isolamento entre meta-objeto primário e para-objeto.
- Privilégio da Anterioridade
- Minimalidade e Completude
- Maximização da Transparência

O foco central do capítulo é a descrição estrutural e funcional do MOP de **Guaraná** e de seus componentes.

Quanto à utilização do MOP, alertamos para o risco de recursão infinita quando o gancho reflexivo é ativado, sem o devido cuidado, a partir de um meta-objeto. O que pode ocorrer sempre que um meta-objeto burlar os mecanismos de *reificação* e *intervenção* (oferecidos pelo MOP) no acesso a seu para-objeto vinculado.

Quanto à coerência do MOP com o princípio da *conexão causal*, explicamos que o comportamento das entidades é condizente com sua representação e vice-versa para todas entidades presentes no domínio reflexivo. Este domínio reflexivo é constituído pelo somatório de para-níveis, que se traduz por todas as entidades passíveis de vinculação reflexiva através da primitiva *Guarana.reconfigure()*. Sobretudo, o núcleo propriamente dito do MOP de **Guaraná** foi excluído intencionalmente do domínio reflexivo, sendo composto pelas classe *Guarana*, *Operation* e *Result*.

Quanto à completude do MOP, alertamos para a necessidade de inclusão do recurso de *reificação ativa* ou *de saída*, a fim de validar a diretriz de *Minimalidade e Completude*. Como prova dessa necessidade citamos a impossibilidade de determinação da cadeia de invocação de métodos no para-nível apenas por intermédio de *reificação passiva*.

Quanto ao desenho do MOP, criticamos a ambiguidade semântica do valor nulo (null) quando utilizado como parâmetro formal na primitiva *reconfigure* no contexto de uma reconfiguração. O meta-objeto alvo da reconfiguração não tem como discernir se o suplicante da reconfiguração se referia a "qualquer meta-objeto" ou a "nenhum meta-objeto". Esta ambiguidade poderia ser eliminada com o uso de constantes pré-definidas pela classe *Guarana*, tornando inválida a utilização do valor nulo na primitiva de reconfiguração.

Outra inovação no desenho do MOP de **Guaraná** é a impossibilidade de se testar a reflexividade de um dado para-objeto qualquer. Ou seja, dado um para-objeto não é possível

determinar se o mesmo está vinculado a uma meta-configuração, se os componentes da meta-configuração assim o desejar.

Ainda neste capítulo, classificamos diversas estruturas presentes no MOP de acordo com a nomenclatura de *padrões de projeto*. Além de clarificar os conceitos presentes em **Guaraná** para aqueles familiares ao jargão de *design patterns*, este esforço serve para comprovar a qualidade técnica do desenho do MOP.

Capítulo 4

Meta-Programação sobre Guaraná: Modelo, Técnicas e Obstáculos

Uma vez familiarizados com a arquitetura do MOP de **Guaraná**, emergem naturalmente as dúvidas: "Para que serve?", "Onde pode ser aplicado?", e "Como utilizá-lo?".

Neste capítulo, iniciamos a busca de respostas a estas perguntas. Aqui propomos um modelo de atuação do meta-programador que serve de substrato à análise de obstáculos e das técnicas para suplantá-los. Apresentamos as ferramentas concebidas a partir desta análise, e exploramos a aplicabilidade da meta-programação através da proposta de outras ferramentas e meta-aplicações.

No capítulo 6 apresentaremos um estudo de caso de alguns dos conceitos discutidos neste capítulo e no capítulo 5. Dependendo da preferência do leitor, a cada conceito teórico discutido, é possível fazer uma leitura adiantada do respectivo exemplo no capítulo 6. Fica a ressalva de que nem todas as técnicas foram ilustradas no capítulo de estudo de caso.

4.1 Termos e Conceitos

Entendemos por **meta-programador** o ator composto por qualquer combinação dos papéis de arquiteto, projetista e programador que se utiliza de reflexão computacional para construir um artefato de software. Em oposição a este personagem, denominamos **para-programador** o ator que possui as mesmas propriedades exceto por não se utilizar de reflexão computacional.

Neste capítulo, vamos restringir ainda mais a definição de meta-programador, re-definindo o termo *meta-programador* como sendo o ator que se utiliza do MOP de **Guaraná** na construção de artefatos de software. Por simetria, *para-programador* é todo ator que não se utiliza do MOP de **Guaraná**.

Essa redefinição é necessária a fim de que não haja dúvidas na identificação de um ator

dada a presente implementação do MOP. Como já foi dito no capítulo 3, o MOP de **Guaraná** está implementado sobre a linguagem Java que já fornece facilidades reflexivas, ainda que extremamente limitadas.

No contexto deste capítulo, vamos definir **meta-programação** como a atuação do meta-programador segundo a definição mais restrita.

4.2 Modelo de Atuação do Meta-Programador

Nesta seção, tentaremos contextualizar o MOP de **Guaraná** no âmbito da Engenharia de Software. Existem três questões importantes que procuramos responder: Qual a motivação por trás do desenvolvimento do MOP no contexto da produção de software? Quais cenários são passíveis de aplicação do MOP ? E, sobretudo, qual a metodologia de aplicação da ferramenta **Guaraná** na prática ?

4.2.1 Motivação

Existem duas necessidades distintas, em Engenharia de Software, que podem ser supridas pela adoção de técnicas reflexivas. A primeira delas é a necessidade de estruturar artefatos de software modularmente, com máxima coesão e mínimo acoplamento. A segunda é a necessidade de evoluir artefatos de software, sem comprometer sua estabilidade e minimizando o custo.

4.2.2 Cenários

Perante tais necessidades surgem dois cenários possíveis. No primeiro cenário, desde a concepção de um artefato de software é previsto o uso de meta-programação. No segundo cenário, já existe um artefato de software que precisa evoluir um subconjunto de sua funcionalidade.

No cenário de concepção, supõe-se que meta-programador esteja familiarizado com o para-nível, possuindo acesso irrestrito a informações: especificação de requisitos, código-fonte, documentação de projeto, etc. A precisão e abundância de informações facilita a atuação do meta-programador quanto aos aspectos: *vinculação*, *execução* e *intervenção*. A disponibilidade de meta-informação para componentes de meta-nível não se sobressai como recurso de programação, uma vez que o meta-programador já possui em seu poder um conjunto mais abrangente de informação.

Até mesmo a utilização de recursos reflexivos torna-se questionável neste primeiro cenário, quando confrontada com requisitos de desempenho. Por outro lado, a modelagem dos aspectos não-funcionais através do MOP desde a concepção resultará num desenho mais flexível. O modelo reflexivo do MOP já assegura um baixo acoplamento (aspecto vinculação) e

alta coesão (aspectos reificação e intervenção). Naturalmente, a desobediência às diretrizes de utilização do MOP podem invalidar essa afirmação, especialmente se houver abuso quanto ao aspecto *execução*.

A título de ilustração, o meta-programador poderia implementar facilidades de *log* diretamente no código-fonte que está disponível. Entretanto, se modelar a atividade de *logging* através de meta-objetos, então possuirá uma ferramenta mais abrangente (um único meta-objeto pode atuar sobre qualquer para-objeto), flexível (o recurso de *logging* poderá ser ativado e desativado sem modificar o código-fonte) e reutilizável.

No cenário de evolução, sem perda de generalidade, vamos supor a ausência de informações relativas à concepção, desenho e implementação do artefato de software que deve evoluir. Essa suposição é válida pois a presença de tais informações propiciaria as mesmas condições existentes no cenário anterior, obscurecendo a distinção entre os dois cenários. Sobretudo, essa suposição é realista. Exceção feita aos modelos **open source** e **free software**, frequentemente o profissional de informática se depara com artefatos de software sobre os quais precisa fazer alguma adaptação, e é tolhido pela falta de documentação ou indisponibilidade do código fonte.

Sem entrar no mérito da ética, neste cenário de evolução, a abordagem reflexiva se revela uma alternativa adequada para implementar as adaptações necessárias sobre o artefato de software. Justamente por este cenário ter recebido menor atenção das comunidades acadêmica e industrial, resolvemos adotá-lo como foco principal para a pesquisa.

4.2.3 Método

Motivado pela necessidade de adaptar um artefato de software mediante o surgimento de novos requisitos, e devido à escassez de informações sobre o próprio artefato, o *meta-programador* só possui os recursos de meta-informação e interceptação providos pelo MOP para atingir seus objetivos.

A título de simplificação, vamos supor que o artefato de software que deve ser adaptado não é reflexivo. Ou seja, não se utiliza de nenhuma facilidade do MOP de **Guaraná** antes de sofrer a adaptação. Sendo assim, doravante o artefato será denominado *para-aplicação*.

A adaptação da para-aplicação aos novos requisitos é atingida pela sua associação a uma meta-aplicação. Desta forma, o meta-programador se depara com quatro tarefas distintas:

- identificação das para-entidades (alvos) cujo comportamento deve ser adaptado;
- construção de meta-entidades encarregadas de impingir novo comportamento às para-entidades alvo;
- definição de um plano de vinculação entre para-entidades alvo e meta-entidades;
- disparar a execução do binômio para-aplicação/meta-aplicação;

Ressalvamos que a ordem com que foram enumeradas as tarefas **não é** a única sequência válida.

Identificação das Para-Entidades

A identificação das para-entidades alvo depende diretamente da quantidade de informação disponível sobre a para-aplicação. Uma para-entidade torna-se um alvo quando ela possui uma parcela de responsabilidade sobre o comportamento que deve ser adaptado. No cenário de concepção, essa tarefa é trivial devido à abundância de informações relativas à arquitetura, ao desenho e a seu mapeamento para a implementação. Esse é o conhecimento chave para a adaptação, não a presença do código-fonte.

Fazendo um paralelo com projetos não-reflexivos, até mesmo projetos *open source* e *free software* podem sofrer da indisponibilidade deste conhecimento chave na hora de agregar funcionalidade. Quando há somente código-fonte disponível o programador tradicional é obrigado a realizar um rastreamento manual a fim de identificar as entidades alvo. Da mesma forma, no domínio de reflexão computacional, nada impede o meta-programador de adotar a mesma técnica se o código-fonte estiver disponível. Entretanto, na ausência de código-fonte, o programador tradicional fica impossibilitado de agir, enquanto o meta-programador ainda tem uma alternativa a sua disposição.

A ausência de código-fonte não é um empecilho para o meta-programador. De posse do artifício da *reificação*, o meta-programador pode conduzir a própria para-aplicação a revelar a identidade das para-entidades alvo. Porém, essa não é uma tarefa trivial. Na seção 4.3 vamos apresentar algumas técnicas de engenharia reversa factíveis sobre **Guaraná**.

Construção das Meta-Entidades

O meta-programador deve designar a cada para-entidade alvo uma meta-configuração (conjunto de meta-entidades) cujo propósito é agregar nova funcionalidade. A quantidade e cardinalidade da meta-configuração varia de acordo com a necessidade de adaptação necessária, a habilidade do meta-programador, e a natureza das para-entidades.

O MOP de **Guaraná** almeja que a construção de uma dada meta-configuração se dê através da composição de meta-objetos **COTS(components "off-the-shelf")**. O que é possível dada a existência dos meta-objetos *Composers*. A meta-configuração resultante será usualmente um grafo hierárquico, onde as folhas do grafo representam os meta-objetos "atuantes", e os demais nós *Composers* – cuja responsabilidade é encaminhar (*routing*) a meta-informação dentro da meta-configuração.

A montagem de uma meta-configuração consistente é responsabilidade do meta-programador, onde um meta-objeto componente não deve interferir no desempenho de outro presente na mesma meta-configuração. Entretanto, se obedecidas as diretrizes de programação do MOP, a não-interferência entre meta-objetos estará garantida por construção.

Tipicamente, o meta-programador dedicado à tarefa de montagem da meta-configuração irá concentrar seus esforços em fatorar o subconjunto de meta-informação relevante a adição da nova funcionalidade. Lembramos que o mecanismo de interceptação/reificação no núcleo do **Guaraná** é insensível às necessidades de uma meta-configuração em particular. Portanto, grande parte da meta-informação entregue à meta-configuração no decorrer da computação pode ser inteiramente irrelevante do ponto de vista da meta-configuração receptora. Esse processo de filtragem pode ser embutido em um *Composer* ou em um meta-objeto folha.

Uma vez resolvido o problema da fatoração da meta-informação relevante, o meta-programador passa a se preocupar com a efetiva manipulação dessa meta-informação através de mecanismos de *intervenção* e *execução*.

Plano de Vinculação

Na tarefa de construção da meta-configuração, o meta-programador precisa fatorar dentre todo a meta-informação reificada um subconjunto relevante. Igualmente, na tarefa de identificação de para-entidades, o meta-programador se depara com a necessidade de fatorar, dentre o universo de entidades da para-aplicação, um subconjunto de alvos.

O planejamento de vinculação está intimamente ligado à identificação das para-entidades, existindo uma relação biunívoca entre as duas. Por um lado, só é possível decidir se um vínculo é apropriado após a identificação da para-entidade como sendo um alvo. Por outro lado, antes de se decidir se uma para-entidade candidata é ou não um alvo de vinculação, pode ser necessária a prévia vinculação de uma meta-configuração ao para-objeto criador ou à para-classe da para-entidade candidata.

Portanto, o meta-programador deve divisar um plano de vinculação que favoreça a identificação das para-entidades alvos, principalmente se a identificação dos alvos for dependente do comportamento ou estado das para-entidades, e não apenas de sua composição estrutural.

A título de ilustração, vamos supor que o meta-programador saiba que seus alvos são todas as instâncias de uma dada para-classe. Neste exemplo, o critério de identificação é estático, dependendo única e exclusivamente da composição estrutural dos para-objetos alvo,

que é determinada pela relação **is-A**. A relação *is-A* é muito utilizada no jargão OO para denotar o relacionamento entre classe e instância.

Portanto, o plano de vinculação pode se resumir à instalação da meta-configuração apropriada na para-classe. Esta meta-configuração deve se propagar para todo para-objeto instanciado a partir da para-classe.

Aproveitamos o mesmo exemplo para ressaltar que MOPs baseados em vinculação em tempo-de-compilação, e que utilizarem técnicas de *wrapping* para implementar o gancho reflexivo, poderiam ser aplicados com sucesso neste caso. Entretanto, estes MOPs seriam ineficazes sobre critérios dinâmicos de identificação de alvos.

A título de ilustração, quando o critério de identificação do alvo é dinâmico, vamos supor a existência de uma meta-aplicação que torne a para-aplicação "supersticiosa", ou seja, que force a substituição de todas as ocorrências do número 13 por outro número, sem que a para-aplicação subjacente tome ciência.

Logo, neste exemplo, os alvos reflexivos são todos os para-objetos que assumam o valor 13 em algum campo ou em algum retorno de método. Neste caso, o meta-programador tem que estabelecer vínculos entre a meta-configuração e diversos para-objetos antes que os últimos tenham sido identificados como alvos efetivos, ou seja, antes que tenham assumido o valor 13. Sendo assim, os vínculos são estabelecidos com "alvos em potencial". No pior caso, todos os para-objetos são "alvos em potencial". Somente MOPs com vinculação em tempo-de-execução são eficazes neste cenário, como é o caso de **Guaraná**.

Disparo da Aplicação

A tarefa de disparar ambas, para-aplicação e meta-aplicação, é comparativamente a mais trivial das quatro atribuições do meta-programador. Entretanto, a fronteira entre para-aplicação e meta-aplicação é tênue. Anteriormente, dissemos que o meta-programa era o somatório de todas as meta-configurações. Entretanto, o meta-programa engloba também o código responsável pela instanciação das meta-configurações, inicialização do plano de vinculação, e pelo subsequente disparo da para-aplicação.

Se a para-aplicação for disparada independentemente do meta-programa, não haverá oportunidade para se iniciar o plano de vinculação, consequentemente a aplicação resultante não será reflexiva.

Na seção 4.3.4 apresentamos técnicas de disparo da meta-aplicação, e na seção 5.2.5 apresentamos uma ferramenta que facilita essa tarefa, independentemente de qual seja a constituição da meta-aplicação ou da para-aplicação.

4.3 Técnicas e Obstáculos

Conhecidas as quatro tarefas que desafiam o meta-programador, nesta seção devemos revisitá-las individualmente, apresentando técnicas para desempenhar tais tarefas sobre o Guaraná, e sobretudo contrastando as facilidades oferecidas pelo MOP com as dificuldades encontradas. O foco desta análise é distinguir limitações existentes no MOP de Guaraná das barreiras intrínsecas à abordagem reflexiva de adaptação.

4.3.1 Identificação das Para-Entidades

Adotando uma postura cartesiana, fracionamos o problema da identificação das para-entidades em dois casos: *estático* e *dinâmico*.

No caso estático, as para-entidades são identificadas por características invariantes durante sua existência na para-aplicação. O que se traduz na capacidade de identificar uma para-entidade por suas características estruturais: quais são os tipos de seus atributos, a partir de que classe foi instanciada, que métodos provê?

Enunciados estes exemplos faz-se necessária uma ressalva. Os exemplos recém apresentados fazem sentido no contexto da linguagem Java, atual hospedeira do MOP, onde informação estrutural é efetivamente invariante. Entretanto existem linguagens no paradigma OO em que informação estrutural não é invariante, como é o exemplo de *Python* apresentada na seção A.3.

No caso dinâmico, o critério de identificação das para-entidades depende de características que variam durante a existência da para-entidade na para-aplicação. Tipicamente, o somatório dessas características é denominado *estado* da para-entidade. Num dado instante, o estado de uma para-entidade se traduz pelo conjunto de valores assumidos por todos seus atributos.

Logo, o problema da identificação das para-entidades se resume a fatoração do espaço de objetos da para-aplicação através de critérios estáticos e dinâmicos resultando num subconjunto de para-objetos alvos de vinculação.

Critérios Estáticos

Pela aplicação de critérios estáticos, o meta-programador procura sempre identificar uma para-classe como alvo de vinculação. No caso trivial, a para-classe é o alvo propriamente dito.

Mesmo quando o alvo final for todas as instâncias de uma dada para-classe, o alvo primário deve ser a própria para-classe. Uma vez que o processo de instanciação é reificado pelo MOP (vide *message NewObject*), é suficiente que a para-classe seja reflexiva, a fim de que suas instâncias sejam identificadas e sofram vinculação.

Existem duas vantagens ao se definir para-classes como alvos primários de vinculação. A primeira é que a vinculação efetiva entre a meta-configuração e a para-classe alvo pode ser feita antes do efetivo disparo da para-aplicação. Ou seja, uma classe utilizada pela para-aplicação é materializada antes da materialização da própria para-aplicação. Isso permite ao meta-programa materializar tais classes, vinculá-las a suas respectivas meta-configurações, para só então materializar a para-aplicação. Essa técnica garante que nenhum evento na para-aplicação transcorra desapercebido pelo meta-programa.

A segunda vantagem é que para-classes reflexivas são excelentes difusoras de meta-configuração pelo mecanismo de propagação. A para-classe reflexiva já determina uma fatoração do espaço de objetos da para-aplicação: suas instâncias.

A título de exemplificação, vamos supor que uma dada meta-aplicação deseje capturar todo e qualquer texto manipulado pela para-aplicação subjacente. A finalidade é irrelevante para o exemplo, mas por completude vamos supor que o objetivo seja fornecer subsídios para a tradução da para-aplicação a um outro idioma. Neste cenário, o meta-programador pode atingir sua meta exclusivamente aplicando critérios estáticos. Por exemplo, seria suficiente vincular a meta-configuração de captura às para-classes responsáveis pela manipulação de cadeias de caracteres (doravante strings). Independentemente de manipulações de mais alto nível, todo e qualquer texto se manifestará através de uma instância da classe string, quando será reificado, apresentado à meta-configuração, traduzido, e só então devolvido ao para-nível. Fica a ressalva de que o problema e solução apresentados como exemplo são simplificações.

Critérios Dinâmicos

Quando critérios estáticos não são suficientes para identificar as para-entidades alvo, devemos recorrer para critérios dinâmicos de identificação. Nestes casos, a qualquer momento uma dada para-entidade pode assumir um estado que a qualifique como um alvo. Se ela já não for reflexiva a transição de estado que interessa ao meta-nível, caracterizando a identificação da respectiva para-entidade como alvo, passará desapercebida. Esse "alvo" terá se perdido no meio da computação. Logo, o problema da utilização de critérios dinâmicos na identificação de para-entidades se traduz por: toda e qualquer para-entidade, *a priori*, é um alvo em potencial; e a decisão se tal para-entidade é um alvo efetivo só pode ser tomada se a mesma for reflexiva (possuir um meta-objeto vinculado).

Logo, no pior caso todas para-entidades devem sofrer vinculação, com uma respectiva meta-configuração pré-programada para propagar-se sempre que possível. Se cada para-entidade estiver vinculada a uma meta-entidade, então nenhuma transição passará indetectada. Qualquer para-objeto que assuma um estado que o qualifique como alvo será identificado pela sua própria meta-configuração, que tanto poderá atuar quanto disparar sua própria substituição por uma outra meta-configuração capaz de atuar.

Denominamos esta técnica de **vinculação plena**, em que toda para-entidade sofre vinculação no momento de sua criação com uma dada meta-configuração que vem se propagando desde o disparo da meta-aplicação. Apesar da hipótese de *vinculação plena* resolver o problema da detecção por critérios dinâmicos em teoria, na prática o consumo de recursos é exorbitante e o desempenho proibitivo. É importante observar que o problema da detecção por critérios dinâmicos não é gerado por uma limitação do MOP. De uma certa forma é semelhante ao célebre problema "O que veio primeiro o ovo ou a galinha?". Sem nos perdermos nos meandros filosóficos do último, a lição que pode ser extraída é a impossibilidade de identificar-se um alvo dinâmico sem que o mesmo já seja reflexivo. Daí a sugestão da aplicação de técnicas de engenharia reversa, por intermédio de mecanismos reflexivos ou não, a fim de analisar a para-aplicação antes do delineamento do plano de vinculação da meta-aplicação.

A técnica que propomos para lidar com o problema da detecção por critérios dinâmicos é a criação de uma única meta-configuração compartilhada pela para-aplicação, cuja atuação é batizada de **meta-hunting**. O processo de *meta-hunting* almeja identificar o momento em que um de seus para-objetos vinculados assume um estado específico, que o qualifica como alvo. Nesse momento, a porção da meta-configuração responsável por *meta-hunting* requisita a ativação da porção da meta-configuração específica para manipular reflexivamente este para-objeto. Essa ativação de uma meta-configuração específica pode implicar ou não na exclusão da participação do porção responsável por *meta-hunting* na composição da meta-configuração resultante no para-objeto alvo. A permanência de meta-entidades responsáveis por *meta-hunting* na meta-configuração final é determinada pelo meta-programador, baseado na possibilidade do para-objeto em questão vir a gerar (instânciação) outros para-objetos, os quais sejam ou possam gerar para-objetos alvos.

A questão da permanência de uma meta-configuração responsável por *meta-hunting* pode ser extrapolada para todo e qualquer meta-objeto, através do seguinte enunciado: "Até quando um meta-objeto deve permanecer em uma dada meta-configuração?" A resposta imediata seria: até que tenha atuado efetivamente sobre o para-objeto. Todavia há dois percalços. O primeiro é que pela natureza dos próprios meta-objetos muitas vezes nenhuma atuação é definitiva, ou seja, a necessidade do meta-objeto atuar pode se manifestar em qualquer momento da existência do para-objeto. Basta tomar como exemplo um meta-objeto de persistência, sua longevidade deve igualar-se à longevidade de seu para-objeto vinculado. Para estes casos a resposta ao questionamento é trivial: o meta-objeto nunca deve deixar a meta-configuração! Entretanto, podem existir casos em que a necessidade de atuação do meta-objeto seja pontual durante a existência do para-objeto.

Suponhamos a existência de um meta-objeto que só precise atuar uma única vez após a construção de seu para-objeto vinculado. Poderia o mesmo ser removido da meta-configuração? A resposta é não.

Eis que surge o segundo percalço. Ainda que o meta-objeto esteja obsoleto reflexivamente quanto ao aspecto *intervenção*, sua presença ainda pode ser imprescindível quanto ao aspecto *vinculação*. O meta-objeto não pode deixar ainda a meta-configuração, pois sua presença é necessária toda vez que o para-objeto criar outros para-objetos. Pelo mecanismo da propagação, todos os meta-objetos pertencentes a uma dada meta-configuração "podem" ser convidados a contribuir na constituição da meta-configuração a ser propagada.

Por essas duas razões, é muito difícil precisar o momento da saída de um meta-objeto de uma meta-configuração, especialmente baseando-se unicamente na meta-informação disponível ao meta-objeto candidato à exclusão. A localidade de atuação e visão do meta-objeto não é suficiente para que o mesmo tome tal decisão. Em contrapartida, colocando-se toda a para-aplicação em perspectiva pode ser possível diagnosticar a obsolecência de um sub-conjunto de meta-objetos culminando em sua exclusão das meta-configurações a que pertencerem. Em suma, informações globais tem maior influência na remoção de meta-objetos do que informações locais. Este fato acarreta uma consequência nociva: o acúmulo de meta-objetos cuja vida útil se exauriu desapercebidamente, e continuam consumindo em vão os preciosos recursos de memória e tempo de processamento.

Propomos uma técnica de **tagging** ou **marcação** para tentar contornar este tipo de problema. A idéia é subordinar cada meta-configuração a um *Composer* marcador (exemplificado na seção 5.3.1). Cada *Composer* marcador registra seu para-objeto e uma chave (*tag*), onde a chave pode ser qualquer objeto que sirva o propósito de delimitar um agrupamento lógico.

Sendo assim, quando a obsolecência de um agrupamento lógico de meta-configurações for detectada num contexto global, a remoção das meta-configurações obsoletas pode ser disparada através de um *broadcast* (vide classes *Guarana* e *Message*) de uma mensagem de remoção sobre todos os para-objetos que tiverem sido agrupados em uma mesma chave. Esta técnica não fere as diretrizes do MOP, pois tomou-se o cuidado de agrupar logicamente meta-objetos, de forma indireta através de seus para-objetos. O sucesso desta abordagem depende da capacidade de se detectar a obsolecência de um agrupamento lógico de meta-objetos, o que por sua vez depende da semântica de ambas para-aplicação e meta-aplicação.

4.3.2 Construção das Meta-Entidades

O desenho e a construção de um meta-objeto no MOP de **Guaraná** ainda é uma tarefa complexa, pois envolve a definição de um curso de ação em cada um dos planos representados pelas 7 primitivas básicas de um meta-objeto qualquer.

initialize e *release*

Com relação aos tratadores (*handlers*) de *initialize* e *release*, o meta-programador está principalmente preocupado com a aquisição e a liberação de recursos, dos quais o meta-objeto depende para exercer sua função. Apesar de qualquer semelhança com os métodos **constructor** e **destructor** (OOP tradicional), as primitivas de *initialize* e *release* não são substitutos dos primeiros, mas complementares. Ou seja, o *constructor* de um meta-objeto é distinto de seu método *initialize*. O primeiro é invocado assim que o meta-objeto é materializado e passa a existir na meta-aplicação. O segundo é invocado sobre meta-objeto para sinalizar que o mesmo acaba de ser vinculado a um para-objeto.

A decisão importante que o meta-programador deve tomar ao codificar a primitiva *initialize* é: Este meta-objeto será **stateful** ou **stateless**? Essa decisão deve ser tomada neste momento considerando que esta é a única oportunidade em que se oferece ao meta-objeto uma referência para a fábrica de operações autorizada a atuar sobre o respectivo para-objeto. Se o meta-objeto não armazenar a referência (fábrica) recebida durante o evento *initialize*, ele não poderá submeter operações ao para-objeto vinculado, ainda que seja possível modificar demais operações de para-nível interceptadas.

Ainda que o meta-programador não precise da capacidade de submissão de para-operações arbitrárias (aspecto execução), ainda pode ser necessário que o meta-objeto seja *stateful*. Isto ocorrerá sempre que o meta-objeto precisar conhecer a lista de para-objetos a que estiver vinculado num dado instante qualquer.

O perfil *stateful* exige do meta-programador um cuidado especial na hora de colecionar as referências de para-objetos. Considerando que o mecanismo de interceptação de **Guaraná** é sensível a *qualquer* manipulação de um para-objeto reflexivo, manipulações de um para-objeto reflexivo por seu próprio meta-objeto disparam também o *gancho reflexivo* reiniciando o processo de reificação.

Essa abrangência do mecanismo reflexivo muitas vezes passa despercebida por usuários incipientes do MOP. A consequência mais comum é a geração de chamadas recursivas infinitas, eventualmente causando o estouro da memória da JVM, e abortando tanto para-aplicação quanto meta-aplicação. Para transpor este obstáculo, foram criadas funções especiais no núcleo do MOP, cuja ativação não sensibiliza o mecanismo de interceptação como, por exemplo, *BR.unicamp.Guarana.getClassName()*. Além das funções especiais da classe *Guarana*, existem também os objetos **HashWrappers** que servem como um "escudo anti-reflexivo", permitindo que referências a para-objetos sejam armazenadas em estruturas de dados (tipicamente *Hashtables*) no meta-nível, sem que haja o disparo do mecanismo de interceptação (gancho reflexivo). Essa discussão revela mais uma diretriz do MOP:

Abrangência Reflexiva:
Todas as outras operações de um para-objeto qualquer estão sujeitas à reificação *e* intervenção, *independentemente do nível de que se originam.*

handleOperation e *handleResult*

Os tratadores de *handleOperation* e *handleResult* são o destino no meta-nível da meta-informação reificada. Especialmente aqueles pertencentes a meta-objetos "folha" (não-*Composers*) de uma dada meta-configuração. O tratador *handleOperation* permite a manipulação de informação pelo meta-objeto, antes que a mesma seja apresentada ao respectivo para-objeto. Particularmente no tratador *handleOperation*, a informação apresentada será sempre um operação de para-nível reificada.

O tratador *handleResult* permite a manipulação da informação de para-nível produzida em resposta a uma prévia e respectiva operação também de para-nível. Entretanto, a ativação deste tratador está condicionada à decisão tomada por ocasião do tratamento da respectiva operação (em *handleOperation*). Essa dependência encadeada entre *handleOperation* e *handleResult* tem uma importante justificativa quanto ao quesito desempenho. Como já foi dito, a somatória dos mecanismos de ativação do gancho reflexivo, reificação e "roteamento" da meta-informação pela árvore da meta-configuração incorre em um alto consumo de ciclos de CPU, aumentando a latência de execução do para-nível. Portanto, toda economia viável de processamento no meta-nível é importante para não penalizar o desempenho do para-nível. Logo, se uma operação reificada não interessa ao meta-objeto, o mesmo pode explicitamente evitar o meta-processamento dos para-resultados. É possível especificar no valor de retorno de *handleOperation* se os para-resultados devem ser ignorados, inspecionados ou modificados. Ainda é possível simular um pseudo-para-resultado, sem que o para-objeto alvo da operação seja ao menos notificado da invocação da operação.

Seja qual for o caminho tomado pelo meta-objeto, o valor produzido em *handleOperation* será encapsulado em um objeto *Result*. Se o meta-objeto em foco não for primário, o objeto *Result* será entregue a um *Composer*, senão será entregue ao núcleo do **Guaraná**. Esse mecanismo foi discutido na seção 3.4.2 e ilustrado na figura 3.4.

Os tratadores *handleOperation* e *handleResult* conferem um caráter reativo ao meta-objeto que os implementa. De maneira geral, esses tratadores são ativados por eventos de para-nível, sendo interpolados transparentemente na cadeia de invocação do para-nível. Na figura 4.1 ilustramos, em um diagrama de interação UML, o ciclo de reificação de uma para-operação e a respectiva ativação do tratador *handleOperation*.

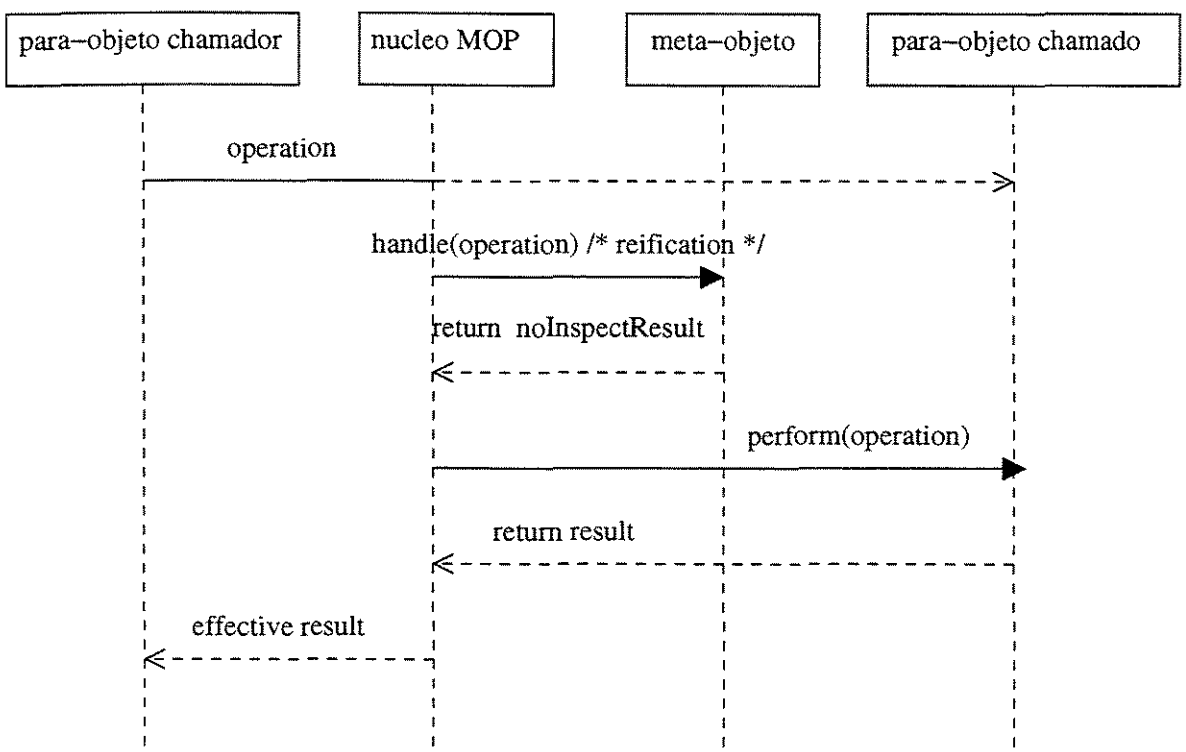


Figura 4.1: Ciclo de reificação de para-operação sem inspeção de resultados

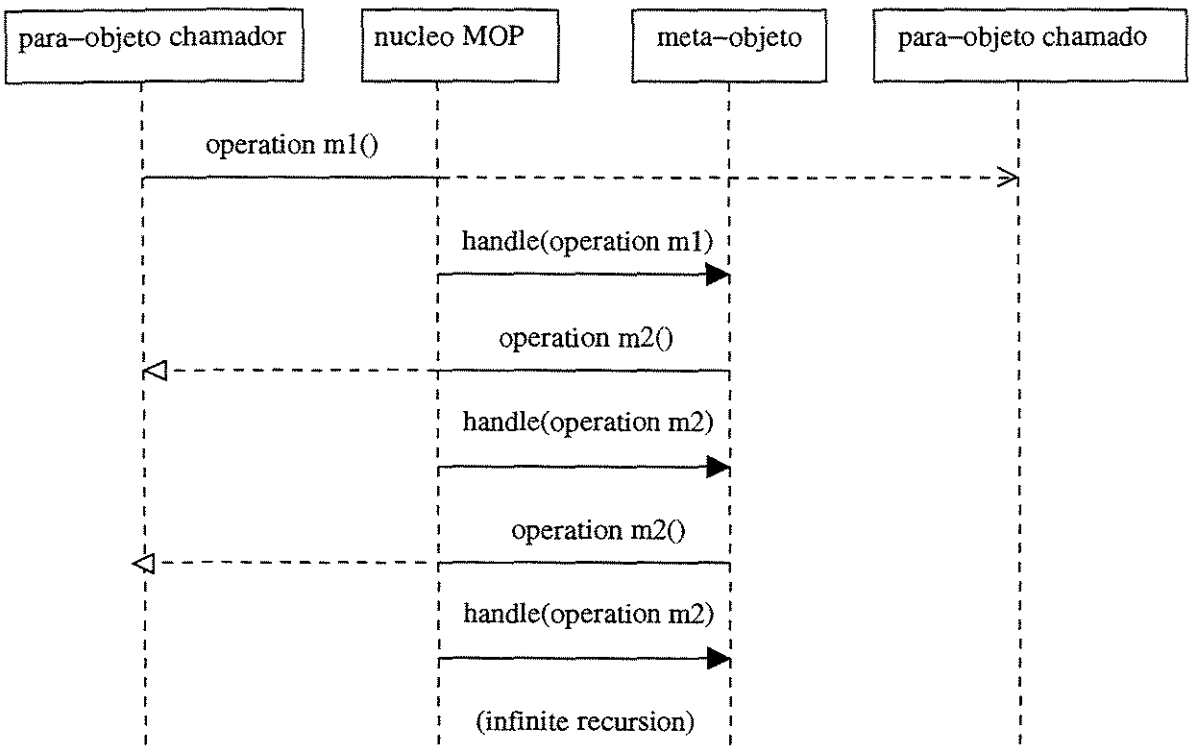


Figura 4.2: Ciclo de reificação de para-operação com recursão infinita

As dificuldades surgem quando dentro de um dos tratadores é necessário atuar (aspectos *intervenção* e *execução*) sobre o para-nível. Pela diretriz *Abrangência Reflexiva*, essa nova cadeia de invocação também deve estar sujeita ao *gancho reflexivo*. Como foi comentado nas seções 4.3.2 e 3.4.2, há o risco da computação se perder num ciclo de recursão infinita se o meta-programador não tomar as devidas precauções.

O risco existe pois é impossível distinguir, dentro dos tratadores em questão, se a operação reificada foi originada pelo para-nível ou pelo próprio meta-nível. E no segundo caso, ainda é impossível saber se foi originada pelo próprio meta-objeto (vinculado ao tratador) ou por outro que possua uma referência ao para-objeto em questão. O ciclo de recursão infinita é ilustrado na figura 4.2.

Uma das técnicas recomendadas para evitar esse problema é a utilização de operações NOP. Digamos que um dado meta-objeto queira invocar o paraObj.m2() de dentro do tratador *handleOperation* na cadeia de tratamento de paraObj.m1(). Ao invés de invocar esse método diretamente através da respectiva fábrica, é invocada uma operação *NOP*. Essa operação *NOP* não tem efeito nenhum no para-nível (como o próprio nome já diz: *no operarion*), mas está sujeita ao *gancho reflexivo* pela diretriz *Abrangência Reflexiva*. Assim,

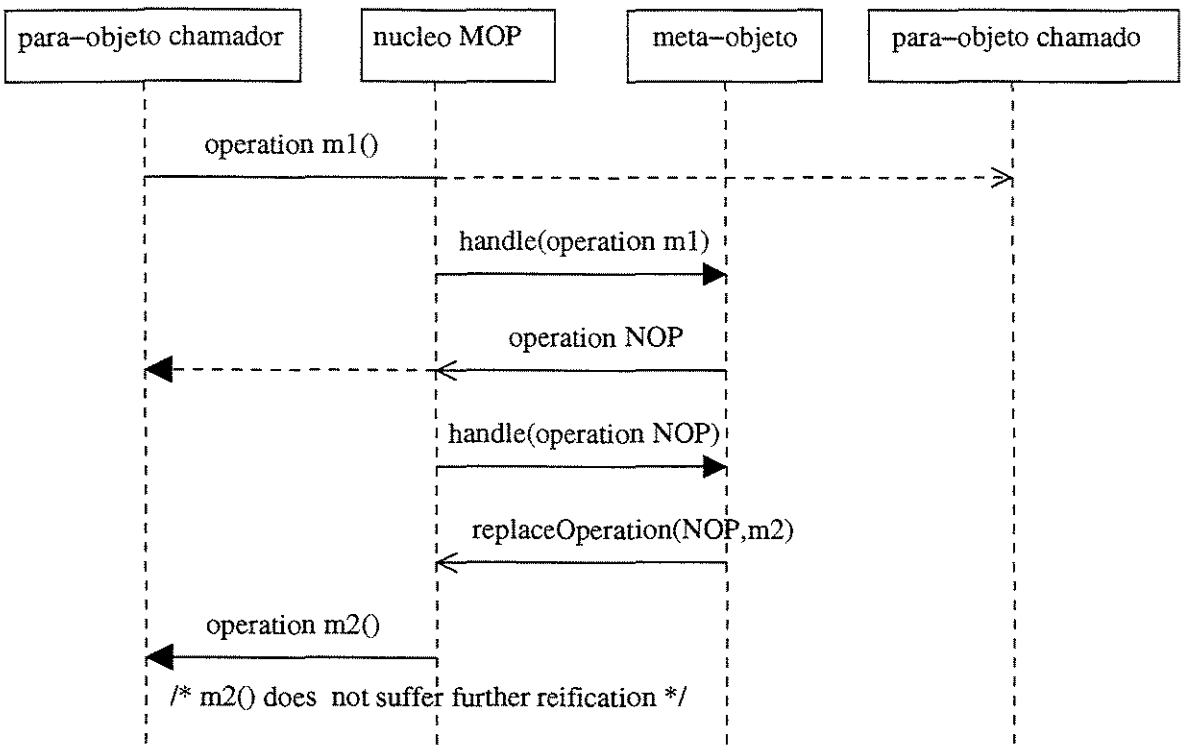


Figura 4.3: Ciclo de reificação de para-operação aplicando técnica NOP

uma segunda cadeia de invocação será iniciada. A operação *NOP* será reificada e entregue ao tratador *handleOperation* que desejava invocar paraObj.m2(). Nesse ponto, o tratador pode gerar um resultado para a operação *NOP*, que é na verdade uma outra operação (*replacement operation*). Portanto, a operação substituta deverá ser paraObj.m2(). Não estará sujeita ao gancho reflexivo por ter sido entregue ao para-nível na forma de resultado, e consequentemente sem o risco de iniciar o ciclo de recursão infinita.

Essa técnica funciona por duas razões. A primeira é que operações *NOP* só podem ser geradas pelo para-nível. Logo, isso resolve o problema da distinção se uma dada operação é oriunda do para-nível ou meta-nível. A segunda razão é que o artifício de embutir uma operação na forma de resultado evita a ativação do gancho reflexivo para tal operação.

Essa técnica é extremamente elegante, pois não viola as diretrizes do MOP. Entretanto, exige um *expertise* por parte do meta-programador. Uma das conclusões importantes desta pesquisa é a necessidade de simplificar o aspecto *Execução* do MOP, dada a dificuldade que o programador incipiente terá em aplicar a técnica *NOP replacement operation*. Esta técnica é ilustrada na figura 4.3.

Na busca de outras técnicas mais simples, seria válido questionar a possibilidade de

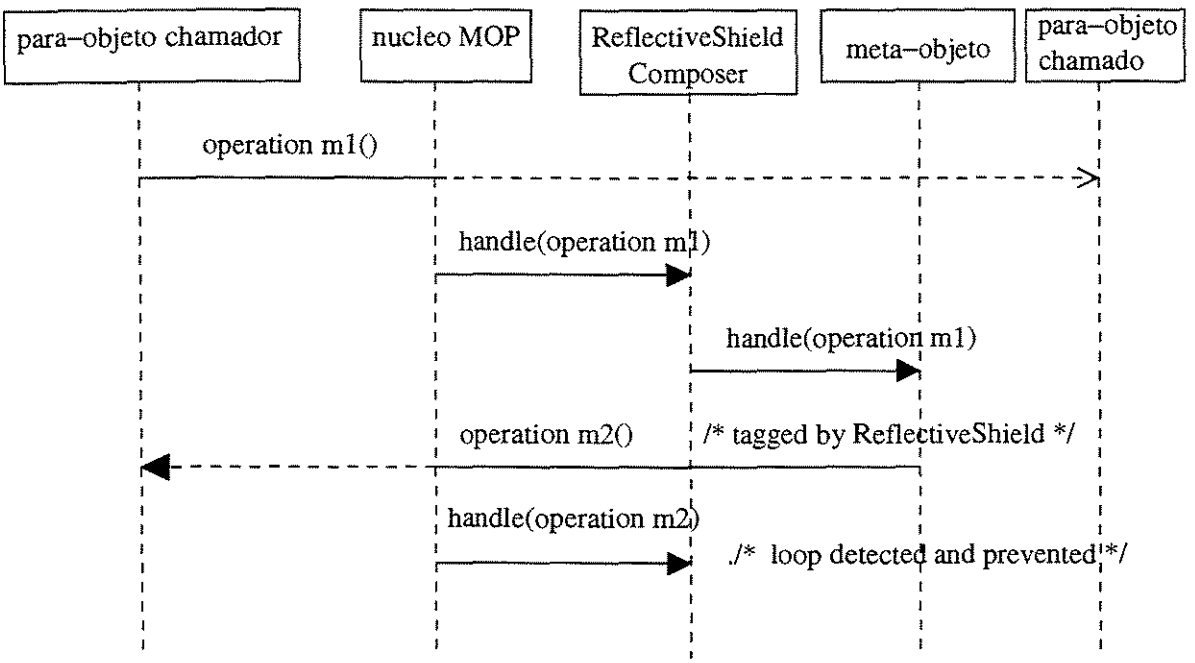


Figura 4.4: Ciclo de reificação de para-operação aplicando técnica *ReflectiveShield*

extrapolar o mecanismo de *HashWrappers*. Ou seja, construir estruturas mais genéricas que blindassem quaisquer que fosse seu conteúdo do gancho reflexivo, como se fossem um **escudo reflexivo**. Apesar do apelo estético, essa opção teve que ser descartada pela quebra das diretrizes *Abrangência Reflexiva* e *Maximização da Transparência*. Ou seja, ainda que um meta-objeto não deseje a reificação das para-operações geradas por si mesmo, os demais meta-objetos na mesma meta-configuração (e hierarquicamente superiores) não podem ser privados dessa meta-informação.

Uma outra técnica, é a utilização de um *Composer* específico na meta-configuração, capaz de proteger sua sub-árvore de meta-objetos subordinados da recursão infinita no momento da intervenção. Este *Composer*, batizado de **ReflectiveShield** (vide seção 5.3.1), geraria uma fábrica especial que marcaria (*tagging*) as para-operações geradas no meta-nível. Quando estas operações "marcadas" fossem reificadas, o próprio composer *ReflectiveShield* seria capaz de reconhecer e então descartar tais operações, evitando assim a recursão. A discussão desta técnica provocou uma alteração no MOP de **Guaraná**, o suporte a *tagged operations*. A aplicação da técnica de *ReflectiveShield* é ilustrada na figura 4.4.

configure

O tratador *configure* é a forma de um dado meta-objeto responder a pergunta: "Como você deseja contribuir na nova meta-configuração de um para-objeto recém-criado?". Essa pergunta será formulada sempre que um objeto de para-nível for instanciado a partir de outro que já seja reflexivo (para-objeto criador). Nessa ocasião, a meta-configuração do para-objeto criador é a primeira a ser consultada pela prioridade do contexto comportamental, como foi visto na seção 3.4.4. A *posteriori*, se a classe do para-objeto criador também for reflexiva, a mesma pergunta será formulada para sua meta-configuração.

Como foi visto na descrição do MOP, *Composers* são também meta-objetos. Por conseguinte, *Composers* possuem o tratador *configure*, onde podem contribuir na composição da meta-configuração a ser instalada no para-objeto recém-criado. Adicionalmente, um dado *Composer* pode encabeçar uma árvore de delegação dos eventos reificados, entre eles o evento *configure*. Sendo assim, um *Composer* é também responsável por ativar (sem obrigatoriedade) os tratadores *configure* dos seus meta-objetos diretamente subordinados dentro da meta-configuração. Uma vez que tais chamadas de *configure* são encadeadas recursivamente através da (árvore) meta-configuração, é até mesmo possível reproduzir na nova meta-configuração, pertencente ao para-objeto recém-criado, a topologia da meta-configuração original. Portanto, o mecanismo de ativação dos tratadores *configure* ao longo dos componentes da meta-configuração responde a pergunta: "Qual é a nova meta-configuração do para-objeto recém-criado?".

Existem dois casos distintos a serem considerados: o receptor de *configure* é um meta-objeto folha ou um *Composer*. No primeiro caso, é preciso decidir uma dentre três opções:

- Não se propagar. Ou seja, não haverá meta-objeto na respectiva posição na nova meta-configuração.

- Propagar-se. Neste caso, o meta-objeto devolve uma referência para si mesmo, significando que esta instância de meta-objeto estará simultaneamente presente em ambas as meta-configurações. A vantagem dessa abordagem é a economia de recursos no meta-nível. A desvantagem é que o meta-objeto precisa estar preparado para gerenciar múltiplos para-objetos, sendo potencialmente mais difícil de ser codificado.

- Criar um clone de si mesmo e propagá-lo. Neste caso, o meta-objeto cria outra instância de si mesmo para ocupar seu respectivo lugar na nova meta-configuração. O clone pode ser inicializado com o estado corrente do objeto matriz.

Se o receptor de *configure* for um *Composer*, além das considerações supra-citadas o Composer pode delegar a primitiva *configure* para seus meta-objetos subordinados. Dependendo das respostas recebidas pelos seus subordinados, é preciso ponderar os seguintes casos:

Tabela 4.1: Combinações de Parâmetros de *reconfigure*

(PO,null,null)	Nenhuma modificação.
(PO,null,MO')	Não se sabe quem deve sair, mas MO' deve ser adicionado a meta-configuração (inserção).
(PO,MO,null)	O meta-objeto MO, se estiver presente, deve sair da meta-configuração (remoção).
(PO,MO,MO')	O meta-objeto MO, se estiver presente, deve se substituído pelo meta-objeto MO' (substituição).

- Não se propagar quando todos os subordinados, por sua vez, decidiram não se propagar. Neste cenário o Composer pode optar por não se propagar para economizar recursos de meta-nível.
- Não se propagar quando somente um subordinado decidiu se propagar. Na presença de apenas um subordinado na nova meta-configuração, o papel do *Composer* como mediador na delegação dos eventos entre meta-compoentes subordinados não é aplicável.

Se o *Composer* em questão agregar qualquer outra funcionalidade, além da mera capacidade de delegação das primitivas do MOP para múltiplos meta-objetos, pode ser importante que ele se propague mesmo que possua um único ou nenhum subordinado.

reconfigure

O tratador *reconfigure* é responsável pela negociação da entrada ou saída de meta-objetos de uma dada meta-configuração. Ele é ativado em um determinado meta-objeto MO em resposta a uma requisição *Guarana.reconfigure(PO,MO',MO")*. Se MO for o meta-objeto primário da meta-configuração vinculada ao para-objeto PO, então seu tratador *reconfigure* certamente será ativado. Se MO pertencer a esta meta-configuração, sem ser o meta-objeto primário, então sua ativação depende das decisões tomadas nos respectivos tratadores *reconfigure* dos *Composers* entre MO e o meta-objeto primário da meta-configuração.

Quanto à semântica dos parâmetros: O primeiro parâmetro indica o para-objeto vinculado ao meta-objeto alvo de reconfiguração. Onde o segundo parâmetro da primitiva indica quem deve sair da meta-configuração, e o terceiro parâmetro indica quem deve entrar. As combinações de parâmetros são descritas na tabela 4.1.

Essas interpretações são sugestões ou *guidelines* do MOP, e seguem a semântica da primitiva *Guarana.reconfigure()*. A obediência, ou não, a essas interpretações fica a critério do meta-programador. Essa liberdade traz tanto benefícios quanto malefícios.

A principal vantagem é permitir a extensão do mecanismo de gerenciamento de meta-configuração. A desvantagem é a potencial geração de bibliotecas de meta-objetos com diferentes semânticas de composição da primitiva *reconfigure*. Mesmo existindo uma única interface para o gerenciamento da meta-configuração, há o risco da co-existência de modelos e semânticas incompatíveis de composição.

A liberdade de estender a semântica do tratador *reconfigure* preenche a lacuna deixada pelo MOP quanto à mesclagem de duas meta-configurações, estando uma delas já vinculada a pelo menos um para-objeto. A mesclagem é necessária quando um determinado objeto é instanciado a partir de uma para-classe reflexiva por intermédio de um outro para-objeto reflexivo pertencente a outra classe. Como foi descrito na seção 3.4.4, durante a propagação de meta-configurações o contexto estrutural possui menor prioridade do que o contexto comportamental. Logo, a meta-configuração pertencente ao contexto estrutural terá que ser "mesclada" com a meta-configuração pertencente ao contexto comportamental.

Por ocasião da reconfiguração de uma meta-configuração, existem três quesitos que precisam ser especificados:

- a semântica da reconfiguração: adição, substituição ou remoção;
- em caso de mesclagem (ocorrendo adição ou substituição), a identidade da nova meta-configuração a ser mesclada, representada por uma referência para o meta-objeto na raiz da meta-configuração;
- a posição alvo de reconfiguração dentro da (árvore) meta-configuração alvo;

A *semântica* é ditada pela combinação de valores do segundo e terceiro parâmetros, como foi visto no início desta seção. A *identidade* dos meta-objetos envolvidos é ditada pelos valores não-nulos do segundo e terceiro parâmetros. Entretanto, não existe na primitiva uma forma de especificar a posição de reconfiguração. A explicação para tal omissão é que no desenho original do MOP cada meta-objeto seria responsável por analisar os parâmetros e decidir se a reconfiguração lhe diz respeito ou não.

A vantagem desse modelo de funcionamento é sua coerência com a diretriz de *Maximização da Transparência*. Ou seja, se a meta-configuração não possuir um ponto de substituição conhecido (referência para outro meta-objeto presente na meta-configuração já vinculada) não é preciso especificar a *posição* alvo. Sua posição efetiva, após a "absorção" pela meta-configuração alvo, será implicitamente definida por uma meta-objeto presente na meta-configuração alvo que assumirá a responsabilidade pela reconfiguração. Enquanto nenhum meta-objeto assumir tal responsabilidade, a requisição de reconfiguração será passada adiante no sentido meta-objeto primário $\rightarrow$ folhas.

Como é impossível que um meta-objeto (presente na meta-configuração vinculada) reconheça todos os possíveis tipos de meta-objetos candidatos à adesão, pode ser necessário que o meta-objeto aceite a adesão incondicionalmente. Fica a cargo do meta-programador elaborar políticas de autenticação entre os meta-objetos candidatos e os já instalados.

Só é preciso tomar cuidado para que um meta-objeto já instalado não revele a sua referência para um meta-objeto externo durante a autenticação. Se isto acontecer, o meta-objeto candidato pode "forçar" a sua entrada na meta-configuração utilizando-se do próprio mecanismo reflexivo. Para tanto, basta que o candidato se instale com o meta-objeto primário do instalado. Através deste segundo plano reflexivo, o meta-objeto candidato poderia manipular o processo eletivo que ocorre no primeiro plano reflexivo. Portanto, o "vazamento" de uma referência para um meta-objeto já instalado em uma meta-configuração qualquer pode representar uma violação à diretriz *Privilégio da Anterioridade*.

Se uma meta-configuração necessita que sua composição seja mutável no decorrer do tempo, podemos aplicar a técnica de adicionar um *Composer* "gregário" no ponto da árvore onde novas meta-configurações possam ser penduradas. Um *Composer* "gregário" simplesmente assume a responsabilidade por qualquer pedido de reconfiguração cujo terceiro parâmetro seja nulo. Ou seja, este *Composer* adiciona toda meta-configuração proveniente de um pedido de reconfiguração entre seus subordinados, desde que o pedido não tenha especificado a identidade de um meta-objeto no terceiro parâmetro.

Essa técnica é limitada por permitir um único ponto de inserção por meta-configuração instalada, e também por não permitir a especificação de uma escala de prioridades de posicionamento (e consequentemente ordenação na delegação) entre as meta-configurações adicionadas.

Para transpor essas limitações, o meta-programador pode criar um protocolo de troca de mensagens entre os meta-objetos candidatos e instalados, a fim de selecionar um dentre múltiplos pontos de inserção (quando existir mais de um *Composer* "gregário" por meta-configuração) ou negociar prioridade de ordenação em um pré-selecionado ponto de inserção.

handleMessage

O tratador *handleMessage* permite a criação de uma interface genérica de comunicação entre meta-objetos, sem violar a diretriz *Maximização da Transparência*.

Esse mecanismo é genérico pois as mensagens trocadas são instâncias de uma hierarquia de classes, cuja raiz é a classe *Message*. Logo, a generalidade é conferida pelo mecanismo de herança. A inviolabilidade da diretriz *Maximização da Transparência* é mantida pelo endereçamento indireto das mensagens. Mensagens, cujos destinatários são meta-objetos, são endereçadas através de seus respectivos para-objetos vinculados. Dessa forma, não há "vazamento" de referências para meta-objetos já instalados em uma meta-configuração (como foi discutido na seção anterior).

Cada mensagem pode definir sua própria interface. Entretanto, existe uma informação comum a qualquer mensagem faltando na interface padrão. Não há como saber se ao emitir uma mensagem a mesma foi recebida ou não. E caso tenha sido recebida, se foi compreendida pelo destinatário. A ausência dessa informação é intencional na interface básica de mensagens, de forma a impedir a determinação da existência (ou não) de uma meta-configuração vinculada a um dado para-objeto. Caso contrário, haveria uma violação da diretriz *Maximização da Transparência*.

Entretanto, existem casos em que a presença deste mecanismo é imprescindível. Por exemplo, para evitar a adesão de meta-objetos em duplicata em uma meta-configuração que possua um *Composer* "gregário". Antes da requisição de reconfiguração, poderia se emitir uma mensagem que se traduzisse por "Esta meta-configuração já está instalada na meta-configuração vinculada?". Cada meta-configuração, ou cada meta-objeto, saberia reconhecer-se e impedir a sua duplicação. Como não existe ainda um padrão no MOP que concretize esta técnica, é preciso cuidado para não se adotar múltiplas convenções distintas que não sejam inter-operantes.

Através da especialização da interface *Message*, de forma a oferecer atributos booleanos (por exemplo "accepted" e "understood"), poderia se testar se a mensagem foi ou não entendida pelo destinatário. Como a primitiva *Guarana.broadcast()* é síncrona, e a mensagem é um objeto, bastaria testar os atributos após o retorno da primitiva *broadcast*. Lembrando que se a mensagem não indicar recebimento, existem três possibilidades: ou o para-objeto destinatário não possuia meta-configuração, ou a meta-configuração do destinatário não entendeu a mensagem, ou a a meta-configuração do destinatário entendeu a mensagem mas preferiu não revelar sua presença. Todas as três possibilidades são válidas.

Brecha de Segurança em campos *final Protected* ou *final Public*

É importante alertar uma possível brecha de segurança durante a construção de para-objetos. Se a linguagem hospedeira do MOP (Java) permitir a declaração de atributos constantes (modificadores *final public* ou *final protected*), existe sempre a possibilidade de se vincular um meta-objeto ao conteúdo de tais campos, e consequentemente violar a premissa de que o atributo seria constante (modificador *final*).

Uma das formas de assegurar a invariância de um atributo agregado constante seria pré-vinculá-lo a um meta-objeto *MetaBlockerReconfigure* (descrito na seção 5.3.1), o qual impediria qualquer reconfiguração do atributo, inclusive as maliciosas.

4.3.3 Plano de Vinculação

O plano de vinculação diz respeito à vinculação entre meta-entidades e para-entidades. Como foi discutido na seção 4.2.3, existe uma estreita interdependência entre o plano de vinculação e a identificação das para-entidades alvo.

No cenário mais simples, todas as para-entidades alvo são conhecidas e identificadas por critérios estáticos antes do disparo da para-aplicação. Portanto, o plano de vinculação resume-se à vinculação direta através da primitiva *Guarana.reconfigure()* aplicada sobre cada para-entidade alvo e sua respectiva meta-configuração. Supondo que todos os alvos sofrem vinculação direta, não há necessidade de utilizar o mecanismo da propagação, representado pelos tratadores *configure* nos meta-objetos instalados.

Na prática, ainda supondo este cenário simplificado, as únicas para-entidades que podem ser identificadas e vinculadas antes do disparo efetivo da para-aplicação são objetos que representam para-classes. Tais objetos são instâncias *singleton*, podendo existir antes da execução da para-aplicação.

Utilizamos o qualificador *singleton* com o significado de "única instância" possuidora das mesmas características estruturais. Na literatura de padrões [17], *singleton* é o padrão que garante a existência de uma única instância por classe, provendo um ponto de acesso global a mesma. Gostaríamos de ressaltar que na linguagem Java todos os objetos que representam classes são "instâncias" de uma mesma classe chamada java.lang.Class. Portanto, estamos cientes de que não poderiam ser qualificados como *singleton* pela definição literal.

Objetos que representam instâncias de para-classes só serão materializados após o disparo da para-aplicação e, por conseguinte, não podem sofrer vinculação direta antes do disparo da para-aplicação.

Num cenário um pouco mais complexo, as para-entidades alvo seriam todas as instâncias de um conjunto finito e conhecido de para-classes. Assim como no cenário anterior, o critério de identificação de para-entidades alvo é puramente estático. Diferentemente do cenário anterior, vinculação direta não é suficiente pois os para-objetos alvo não estão disponíveis antes do disparo da para-aplicação, apenas suas respectivas para-classes. Sendo assim, algumas para-classes se tornam alvos também. Então, o plano de vinculação deve possuir dois estágios:

1. vinculação das meta-configurações às respectivas para-classes alvos.

2. propagação de uma dada meta-configuração, a partir da para-classe em que está instalada, para cada instância da própria para-classe.

O primeiro estágio deve ser efetuado antes do disparo da para-aplicação, da mesma forma como foi feito no cenário anterior. Utilizando a primitiva *Guarana.reconfigure()* aplicada sobre cada para-classe alvo e sua respectiva meta-configuração. O segundo estágio deve

ser codificado nos tratadores *handleMessage* das meta-configurações instaladas. Quando em um desses tratadores for sinalizada a mensagem *NewObject*, a meta-configuração pode utilizar a primitiva *Guarana.reconfigure()* para se propagar para seu alvo final: a instância recém-criada.

O cenário mais complexo seria aquele em que as para-entidades alvo só serão conhecidas no decorrer da execução da para-aplicação pela aplicação de critérios dinâmicos. Neste cenário, o meta-programa possui três tarefas (em ordem de dependência):

1. Tarefa intrínseca do meta-programa, que só é aplicável após uma meta-configuração qualquer estar vinculada a uma para-entidade que seja alvo final.

2. Identificar se uma dada para-entidade é ou não alvo final, e em caso positivo vincular a mesma a meta-configuração responsável pela execução da tarefa 1.

3. Identificar se uma dada para-entidade é ou não alvo intermediário. Uma para-entidade é alvo intermediário se ela pertencer à cadeia de construção de objetos que culmina em um alvo final.

Essas três tarefas podem ser desempenhadas por um mesmo meta-objeto, por diferentes meta-objetos em uma mesma meta-configuração, ou até mesmo por meta-configurações distintas. A escolha fica inteiramente a critério do meta-programador.

À primeira vista, esta abordagem de programação pode parecer infactível ou simplesmente contra-producente dada a complexidade da tarefa. Contudo, mediante uma penalidade no desempenho e flexibilidade de manutenção e reuso, um único meta-objeto que agregasse as três tarefas seria capaz de desempenhar o mesmo papel que meta-configurações mais complexas. Para tanto, bastaria que este meta-objeto fosse vinculado a toda e qualquer instância criada pela para-aplicação.

O MOP é suficientemente flexível de forma a oferecer ao meta-programador um amplo espectro de escolhas acerca de onde concentrar a complexidade da meta-aplicação. Numa ponta do espectro, é possível criar um meta-programa mais simples e com um plano de vinculação claro em detrimento de flexibilidade e desempenho. No outro extremo, é possível criar um meta-programa complexo otimizado para menor consumo de recursos e maior flexibilidade.

Qualquer que seja a abordagem adotada pelo meta-programador, é preciso garantir que meta-objetos responsáveis pela tarefa 1 permaneçam inertes enquanto participarem de meta-configurações vinculadas a para-objetos alvo intermediários. Quando a meta-configuração é composta por um único meta-objeto, é necessário que este acumule as três tarefas. Quando a meta-configuração é uma árvore de meta-objetos, a sub-árvore responsável pela tarefa 1 pode ser mantida inerte através de um *Composer* que a isole do resto enquanto vinculada a alvos intermediários. Outra técnica viável é tornar a meta-configuração, encarregada das tarefas 2

e 3, responsável por criar e vincular ao alvo uma segunda meta-configuração exclusivamente responsável pela tarefa 1.

Quando o meta-programador desejar causar o menor impacto possível sobre a para-aplicação, no que tange ao desempenho e ao consumo de recursos, será necessário adicionar o mínimo de elementos ao meta-nível. Quanto menos elementos presentes no meta-nível, menor será o número de reificações, maior será o tempo de execução dedicado ao para-nível.

Para otimizar o meta-nível, os candidatos imediatos à eliminação são os meta-objetos exclusivamente encarregados da tarefa 3: identificação de para-entidades que são alvos intermediários. Isso se traduz pela criação de um plano de vinculação mais elaborado e voltado especificamente para uma para-aplicação em particular. O plano de vinculação deve permitir que as meta-configurações responsáveis pela tarefa 1 sejam vinculadas ao respectivos para-objetos alvo, minimizando o número de para-objetos alvo intermediários. Sobretudo garantindo que nenhum para-objeto alvo deixará de ser vinculado. Para todos os alvos finais que sejam para-classes (instâncias de *java.lang.Class*), a solução é trivial através da vinculação direta. Quando os alvos finais forem para-objetos, a minimização exige a identificação da *cadeia de construção* do para-objeto alvo final.

Na definição de planos de vinculação mais elaborados, existe ainda o problema em aberto da combinação entre meta-configurações provenientes de ambos contextos: dinâmico (para-objeto criador) e estático (para-classe do para-objeto alvo). Apesar do *Composer* e seus derivados permitirem a co-existência de ambas meta-configurações vinculadas ao para-objeto alvo, não existe ainda um mecanismo de negociação de "alto nível" no MOP que permita ao meta-programador especificar regras de reconfiguração nesta situação. Ainda que rudimentar, propomos um mecanismo que contempla este problema na seção 5.3.1.

Identificação da Cadeia de Construção

A **cadeia de construção** de uma para-entidade consiste na enumeração ordenada de todas as outras para-entidades, desde o disparo da para-aplicação, que culminam na construção da primeira.

Representemos o para-programa através de um grafo orientado. Os nós do grafo representam as para-entidades do para-programa. As arestas orientadas do grafo representam um relacionamento de construção (alocação e inicialização), onde o destino é o para-objeto recém-construído. Neste modelo, um caminho no grafo entre o nó origem e outro nó qualquer representa a **cadeia de construção** deste nó.

Em particular no caso de Java, o nó origem é representado pelo objeto para-classe cujo método *main* foi invocado para disparar a para-aplicação. Através da *cadeia de construção*

de um para-objeto alvo é possível propagar (vide seção 3.4.4) uma meta-configuração desde o primeiro para-objeto intermediário, até vinculá-la ao alvo final. A propagação de uma meta-configuração através da *cadeia de construção* de uma para-entidade é facilmente obtida pelo uso dos tratadores *configure* dos meta-objetos.

Um problema típico na aplicação desta técnica surge quando existe um método de classe intercalado na *cadeia de construção*. Ou seja, um dos para-objetos intermediários na cadeia de construção invoca um método de alguma classe para construir o próximo elemento da cadeia. Esta estratégia é frequente quando se utilizam padrões de projeto semelhantes a *Factory Methods* [17].

O método de classe, por definição, pertence à classe, que por sua vez não é criada por nenhum para-objeto. Particularmente na linguagem Java, classes são criadas por uma classe em especial denominada **ClassLoader**. Logo, métodos de classe introduzem rupturas em uma dada cadeia de construção qualquer.

A para-classe cujo método foi invocado não foi necessariamente vinculada à meta-configuração que vinha sendo propagada através da cadeia. A ruptura introduzida pelo método de classe pode encerrar a propagação da meta-configuração. Esse fenômeno é ilustrado na figura 4.5.

Para solucionar este problema, seria necessário que o mecanismo de propagação atuasse tanto sobre a *cadeia de construção* de objetos, quanto sobre as classes cujos métodos participam da cadeia de construção. A idéia é automatizar a propagação da meta-configuração também para as para-classes dos para-objetos na *cadeia de construção*. Entretanto, essa técnica introduz o risco de dupla propagação ilustrado na seção 4.3.3. Para facilitar a segura aplicação desta técnica foi elaborado o **ClimberFilterComposer** que será descrito na seção 5.1.5.

Identificação da Cadeia de Invocação

Generalizar a *identificação da cadeia de construção* seria identificar uma **cadeia de invocação** qualquer. Enquanto a primeira diz respeito a métodos especiais chamados *construtores* de objetos, a segunda se refere a qualquer método de um objeto.

Como foi visto na seção anterior, identificar a *cadeia de construção* nada mais é do que propagar uma meta-configuração sempre que a mesma for sinalizada por seu tratador *configure*, notificada por ocasião da criação de um novo para-objeto. O tratador *configure* é o único dispositivo do MOP com reificação *ativa* ou de saída. Ou seja, que sinaliza ao meta-objeto que seu respectivo para-objeto invocou um método sobre outro para-objeto.

A *identificação da cadeia de construção* é facilidada pelo respaldo de *reificação ativa* do MOP. Já a *identificação da cadeia de invocação* não é tão simples, uma vez que não existe respaldo direto nos mecanismos oferecidos pelo MOP.

Ainda na tentativa de determinar a *cadeia de invocação* apenas com *reificação passiva*,

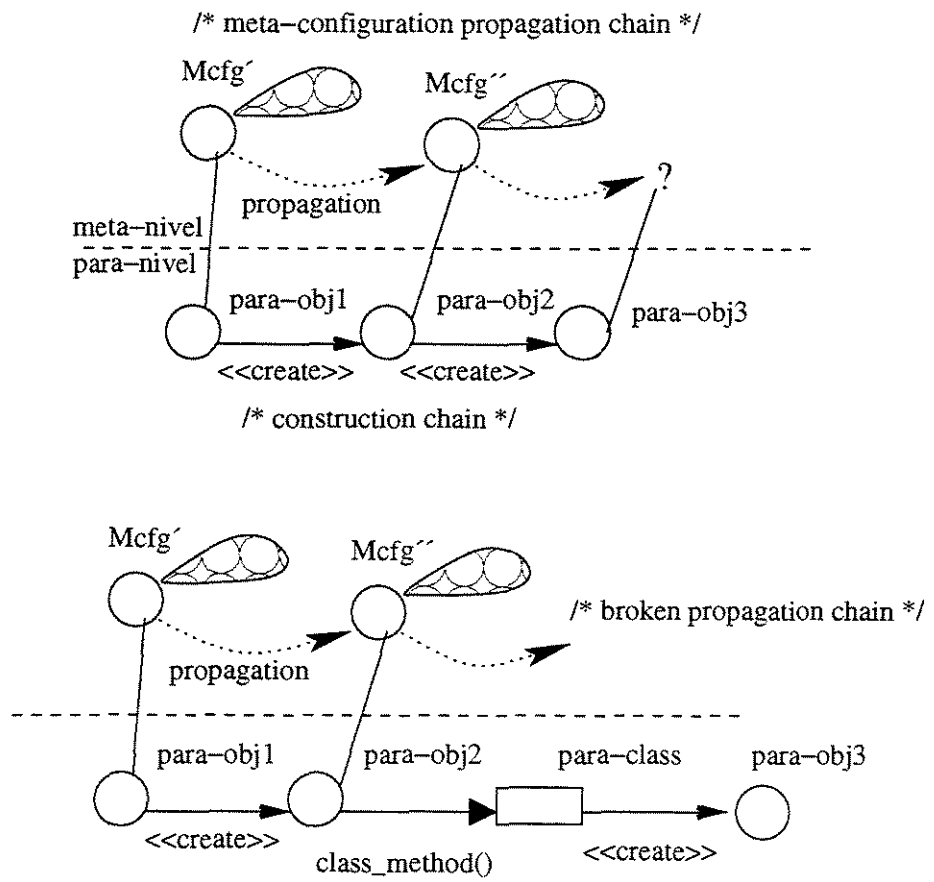


Figura 4.5: Problema da quebra da cadeia de propagação por métodos de classe

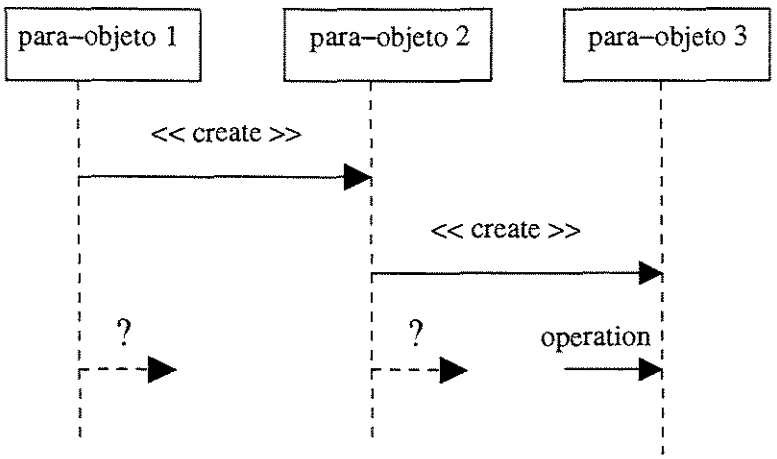


Figura 4.6: Problema da identificação da cadeia de invocação

vamos supor que todos os para-objetos na cadeia já fossem reflexivos antes da primeira invocação pertencente a cadeia ser disparada. Sem perda de generalidade, vamos supor que a meta-configuração de todos os para-objetos na cadeia seja composta por uma mesma instância de um único meta-objeto. Este meta-objeto seria sinalizado a cada invocação de método da cadeia. Mas não possuiria informação sobre a identidade do chamador (*caller id*). Sem essa informação, para qualquer cadeia de invocação de tamanho n (n>2) é impossível determinar qual dos n-1 componentes da cadeia é responsável pela n-ésima invocação de método. Ou seja, o obstáculo é a identificação do chamador, tal como é ilustrado na figura 4.6.

Existem linguagens que geram informação sobre a *cadeia de invocação* na pilha de execução, sobre estas linguagens seria possível reconstruir a cadeia de invocação pelo exame da pilha de execução, seguindo cada quadro da pilha (stack frame) na ordem reversa. A título de exemplificação, a linguagem Python fornece meta-informação sobre a cadeia de invocação, a linguagem Java não.

Se o MOP de **Guaraná** fosse modificado para oferecer informação de *caller id* no tratador de *handleOperation* se tornaria possível identificar a *cadeia de invocação*, desde que todo para-objeto fosse reflexivo.

O obstáculo à incorporação de *caller id* é uma potencial violação de segurança. Na implementação corrente do MOP o único *id* de para-objeto é uma referência para o prórpio objeto. Consequentemente, se um meta-objeto fosse informado do *caller id* em uma reificação passiva, ele poderia não só identificar o chamador como também tornar-se o meta-objeto do para-objeto chamador. No modelo atual de OO isso não pode ocorrer, pois um objeto não expõe necessariamente a sua interface pública através da invocação de métodos, a não ser que exporte explicitamente uma referência para si prórpio em um dos parâmetros do método.

Se o MOP suportasse *reificação ativa* para invocação de métodos quaisquer, e não só para construtores de para-objetos,a identificação da *cadeia de invocação* seria facilitada. Entretanto, as implicações da extensão do MOP para incorporação de *reificação ativa* não foram contempladas neste trabalho de pesquisa.

Em suma, no estado atual do MOP, *a identificação da cadeia de construção* só exige que cada um dos elementos da cadeia seja reflexivo, possuindo ao menos um meta-objeto vinculado. Este estado pode ser obtido pelo mecanismo da propagação de meta-configuração, desde que as mesmas utilizem um *ClimberFilterComposer* ou recurso equivalente.

Enquanto que a *identificação da cadeia de invocação* de uma método qualquer exige a perícia do meta-programador. Em teoria, existe uma técnica viável para a *identificação da cadeia de invocação* que utiliza exclusivamente os recursos já presentes no MOP. Esta técnica permite a identificação do para-objeto chamador através do uso exclusivo de *reificação passiva*. O segredo reside em se criar distintos *para-objetos proxy* do para-objeto chamado, havendo um para cada possível para-objeto chamador. O *proxy* ativado na invocação implicitamente identifica o chamador. Entretanto, a construção deste cenário exige uma preparação prévia ao longo da *cadeia de invocação* que se quer identificar. Cada para-objeto da cadeia precisará possuir uma meta-configuração responsável por identificar todas as referências externas deste para-objeto. Cada uma destas referências é "elegível" como um possível alvo de invocação futuro. Por conseguinte, precisará ser substituída por um *para-objeto proxy*. A descoberta e reconfiguração reflexiva destas referências não é trivial e cara quanto ao consumo de CPU e memória.

Em suma, os fatores que dificultam a *identificação da cadeia de invocação* são a ausência de suporte a reificação *ativa*, suporte a meta-informação de *caller id*, ou suporte nativo da linguagem hospedeira através da pilha de execução.

O Fenômeno da Dupla Propagação

A seção 3.4.4 introduz a dinâmica de propagação de uma meta-configuração. Como foi visto, a composição da nova meta-configuração do para-objeto alvo pode ser determinada por vinculação direta, por propagação da meta-configuração no contexto comportamental (a partir do para-objeto criador), por propagação da meta-configuração no contexto estrutural (a partir da para-classe do para-objeto alvo), ou por uma combinação dos dois últimos casos. Esta última possibilidade ilustra o problema da combinação entre meta-configurações provenientes do contexto comportamental e estrutural, de cujo problema o fenômeno da **dupla propagação** é uma instância.

O fenômeno da *dupla propagação* ocorre quando uma mesma meta-configuração (sejam duas instâncias de uma mesma meta-classe ou uma única instância) tenta se propagar para um mesmo para-objeto alvo, sendo proveniente tanto do contexto comportamental quanto do contexto estrutural, sendo ilustrado pela figura 4.7.

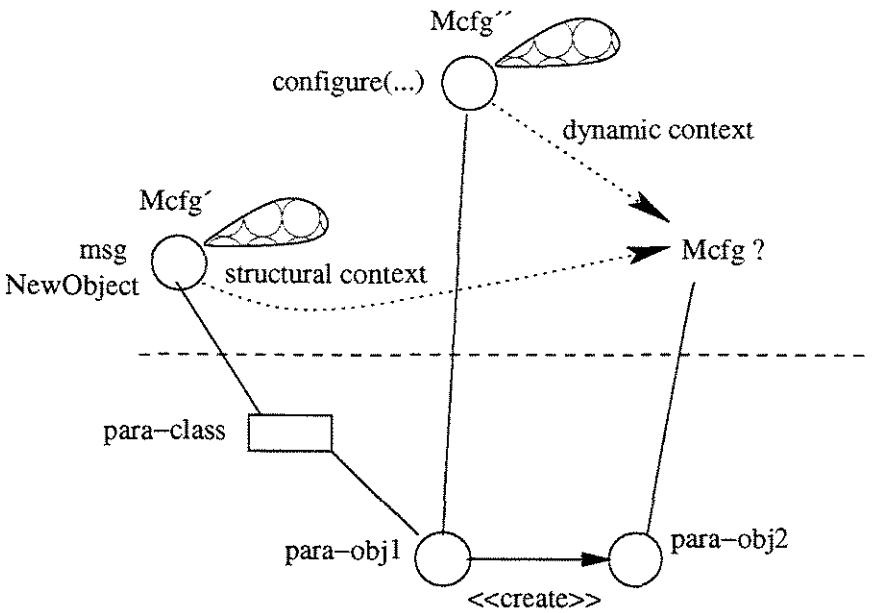


Figura 4.7: Problema da dupla propagação

Enquanto explorávamos o problema da identificação da cadeia de construção (seção 4.3.3), sugerimos a técnica de propagar uma meta-configuração a partir do para-objeto criador para ambos, para-objeto criado (alvo) e sua para-classe, através de um *ClimberFilterComposer* ou mecanismo equivalente.

Esta técnica permite a identificação da *cadeia de construção*, mesmo na presença de chamadas de métodos de classe. Também chamados *estáticos* ou *static* no jargão Java. Entretanto, se a meta-configuração instalada na para-classe do para-objeto alvo estiver configurada para se propagar para as instâncias desta classe, o contexto estrutural tentará contribuir na composição da meta-configuração do para-objeto alvo. Note que o para-objeto alvo já está vinculado a uma meta-configuração oriunda do contexto comportamental (para-objeto criador).

Em particular, é possível que as meta-configurações já instaladas no para-objeto alvo e em sua para-classe sejam instâncias de uma mesma meta-classe (classe de meta-nível) ou até mesmo sejam ambas a mesma instância. Neste cenário de dupla propagação, a vinculação da meta-configuração oriunda da para-classe é obviamente desnecessária e indesejada.

Como evitar a dupla propagação? O caso trivial é a dupla propagação de uma mesma instância. Ao mencionarmos "instância" aqui não nos restringimos a meta-configurações de composição unitária (instância de meta-objeto), mas sim ao conjunto de meta-objetos como um todo (instância de meta-configuração). Portanto, no caso de uma mesma instância de meta-configuração, o meta-objeto primário pode ser *stateful* e "lembrar" dos para-objetos

aos quais já está vinculado. Neste caso, a meta-configuração pode optar por não se propagar ainda quando vinculada a para-classe, evitando assim a dupla propagação.

Quando existem duas instâncias idênticas (mesma composição de meta-configuração) vinculadas a para-classe e ao para-objeto alvo, a solução anterior não funciona pois a instância presente na para-classe "não sabe" que possui um clone já vinculado ao para-objeto recém criado. O pior é que a meta-configuração da para-classe não pode "perguntar" ao para-objeto alvo se já possui algo equivalente a si já vinculado, impedida pela diretriz da *Maximização da Transparência*. Ou seja, a meta-configuração instalada na para-classe tentará se propagar em direção ao para-objeto alvo sem saber que é redundante. Cabe à meta-configuração clone já instalada no para-objeto alvo impedir a entrada de seu clone redundante proveniente da para-classe, ou aceitar ser inteiramente substituída por ele.

Problemas em Aberto

O fenômeno da dupla propagação fomenta a discussão de algumas questões ainda em aberto no MOP.

Como mesclar a meta-configuração proveniente do contexto estrutural com a meta-configuração (já instalada) proveniente do contexto comportamental? Até o presente estado do MOP, cabe ao meta-programador criar explicitamente o código para a absorção da meta-configuração oriunda da para-classe. Não há como a meta-configuração entrante especificar em que ponto da meta-configuração já instalada onde ela deveria ser enxertada.

Como testar a equivalência, intersecção ou diferença entre meta-configurações? Estes mecanismos nunca poderiam ser aplicados a partir da meta-configuração entrante, para não violar a diretriz da *Maximização da Transparência*. Em contrapartida, são imprescindíveis para as meta-configurações já instaladas poderem absorver apenas os meta-objetos essenciais a partir de meta-configurações entrantes ou candidatas (qualquer meta-configuração que tenha feito um pedido de reconfiguração sobre um para-objeto já reflexivo).

4.3.4 Disparo da Aplicação

Na seção 4.2.3 já apresentamos a tarefa de disparo da meta-aplicação, que é uma das quatro atribuições básicas do meta-programador. Nesta seção vamos explorar quais alternativas estão disponíveis para efetuar esta tarefa.

A primeira alternativa é misturar código de meta-nível com código de para-nível. Ou seja, no próprio fonte da para-aplicação criar uma meta-configuração, vinculando-a a um para-objeto qualquer. Apesar de trivial, essa técnica criar um acoplamento forte entre para-nível e meta-nível, e portanto desaconselhável do ponto de vista da Engenharia de Software.

Quaisquer mudanças no meta-nível, implicariam também em mudanças na para-aplicação, o que anula a maior vantagem do MOP: a independência entre para-nível e meta-nível.

A segunda alternativa é criar um meta-programa que faz a instanciação das meta-configurações iniciais, faz a carga da para-aplicação em memória, aplica o plano de vinculação associando as meta-configurações criadas a para-classes chaves da para-aplicação, e finalmente invoca a função estática *main()* da para-aplicação. Essa técnica é melhor que a anterior pois diminui a dependência entre para-nível e meta-nível. Porém é trabalhosa e contraproducente, uma vez que a aplicação de um mesmo conjunto de meta-configurações a diferentes para-aplicações exigiria sempre modificação no meta-programa.

A terceira alternativa é a utilização de parâmetros de linha de comando ou arquivos de configuração, associados a capacidade reflexiva da linguagem, para montar dinamicamente o vínculo entre meta-aplicação e para-aplicação. Esta técnica é concretizada na ferramenta *Launcher* descrita na seção 5.2.5, permitindo total independência entre para-aplicação e meta-aplicação.

4.4 Sumário

Iniciamos o capítulo complementando a terminologia introduzida no capítulo 2, cunhando os termos *meta-programador* e *para-programador*. Logo em seguida, sugerimos um modelo de atuação a ser seguido pelo meta-programador, no exercício de sua atividade: o projeto e implementação de meta-aplicações sobre o MOP de **Guaraná**.

As quatro atribuições básicas do meta-programador são: a identificação das para-entidades alvos, a construção de meta-configurações, a definição de um plano de vinculação entre para-entidades alvo e meta-configurações, e o disparo do binômio para-aplicação/meta-aplicação. Onde a sequência de atividades não está pré-definida, e depende do contexto e finalidade da meta-aplicação.

Em cada uma das primitivas oferecidas por *MetaObject*, o elemento central de programação do MOP, analisamos a atuação do meta-programador.

Quando a identificação de para-objetos alvo é determinada por critérios dinâmicos, potencialmente todas as para-entidades são alvos de vinculação efetivos: *vinculação plena.* Estes casos exigem um plano de vinculação bem elaborado, que pode fazer uso de propagação dinâmica de meta-objetos em tempo-de-execução. Muitas vezes é preciso analisar previamente a para-aplicação através de técnicas de engenharia reversa, antes que se possa delinear o plano de vinculação.

Contrastando MOPs com vinculação em tempo-de-compilação e MOPs com vinculação em tempo-de-execução, observamos que em ambos os casos a abordagem típica é vincular meta-entidades a uma para-classe. A diferença entre ambos é que MOPs do primeiro caso estão restritos à essa abordagem, enquanto que nos MOPs do segundo caso a abordagem é facultativa, como é o caso de **Guaraná**.

No *métier* da meta-programação surgem diversas omissões e obstáculos a serem transpostos, cujos principais representantes elencados foram:

- a fatoração do espaço de para-objetos através de critérios estáticos e dinâmicos (identificação de alvos);
- o tempo de permanência de um meta-objeto em uma meta-configuração qualquer;
- a questão da escolha dos tratadores de *MetaObject* para intervenção (onde manipular para-nível?);
- o problema da recursão infinita reflexiva;
- o problema da vulnerabilidade ao se publicar uma referência a um meta-objeto qualquer;
- a ausência dos campos "accepted" e "understood" na interface de *Message*;

- a brecha de segurança nos campos *final public* e *final protected*;
- a ausência de mecanismo de negociação para mesclar meta-configuração oriunda do contexto comportamental, com meta-configuração oriunda do contexto estrutural;
- quebra da cadeia de propagação quando existem métodos de classe, prejudicando a *identificação da cadeia de construção*;
- a dificuldade na *identificação da cadeia de invocação* somente com reificação *passiva* ou *de chegada*;
- a dupla propagação de uma meta-configuração oriunda de ambos contextos: comportamental e estrutural;

Para cada uma das atribuições básicas do meta-programador, incluindo os obstáculos apresentados, expusemos uma ou mais técnicas:

- *marcação* ou *tagging*;
- *NOP replacement operations* para intervenção imune à recursão infinita;
- *ReflectiveShield Composer* também para imunizar operações de meta-nível da recursão infinita;
- criação de *composer* "gregário" para atuar como ponto de mesclagem entre meta-configuração instalada e candidata;
- uso do *ClimberFilterComposer* para permitir a *identificação da cadeia de construção*;
- a técnica de *para-objetos proxy* para a *identificação da cadeia de invocação*;
- extensão do MOP para suportar reificação *ativa* de forma a facilitar a *identificação da cadeia de invocação*;
- utilizar meta-configuração *stateful* ou introduzir um protocolo de negociação entre a meta-configuração vinculada e a redundante para evitar a *dupla propagação*;

Levantamos problemas em aberto relacionados à atuação do meta-programador e aos mecanismos disponíveis no MOP, cujos enunciados são:

- Como mesclar a meta-configuração proveniente do contexto estrutural com a meta-configuração (já instalada) proveniente do contexto comportamental?
- Como testar a equivalência, intersecção ou diferença entre meta-configurações?

Complementando o capítulo 3, enunciamos aqui mais uma diretiva do mop: *Abrangência Reflexiva.*

Encerramos o capítulo analisando o problema do disparo da meta-aplicação, para o qual foi concebida a ferramenta *Launcher*. Tal ferramenta permite total independência entre para-aplicação e meta-aplicação, sendo assim coerente com o anseio de baixo acoplamento entre para-nível e meta-nível.

Capítulo 5

Ferramentas de Suporte à Meta-Programação

Neste capítulo, ilustraremos os principais aspectos do MOP através da apresentação de um conjunto de ferramentas construídas com o intuito de facilitar a meta-programação. Também vamos elencar uma série de extensões, muitas delas ainda não implementadas, que complementam o conjunto já existente e servem de ponto de partida para pesquisa e trabalhos futuros.

5.1 GDK – Guaraná Development Kit

O **Guaraná Development Kit**, doravante **GDK**, pretende ser uma coletânea de módulos (classes em Java – atual linguagem hospedeira do MOP), cujo propósito é facilitar a atividade de meta-programação. Como foi visto na seção 4.2.3, o meta-programador se depara com 4 tarefas básicas: identificação das para-entidades alvo, construção das meta-entidades, definição do plano de vinculação e disparo da meta-aplicação. Nesta seção vamos examinar cada um dos componentes do *GDK* individualmente, contextualizando-o em uma determinada tarefa de meta-programação.

Note que a divisão entre os componentes "nativos" do MOP e aqueles introduzidos pelo GDK serve ao propósito de manter o MOP o mais simples possível, facilitando sua compreensão e porte para outras linguagens hospedeiras. A figura 5.1 ilustra os componentes já implementados no *GDK*.

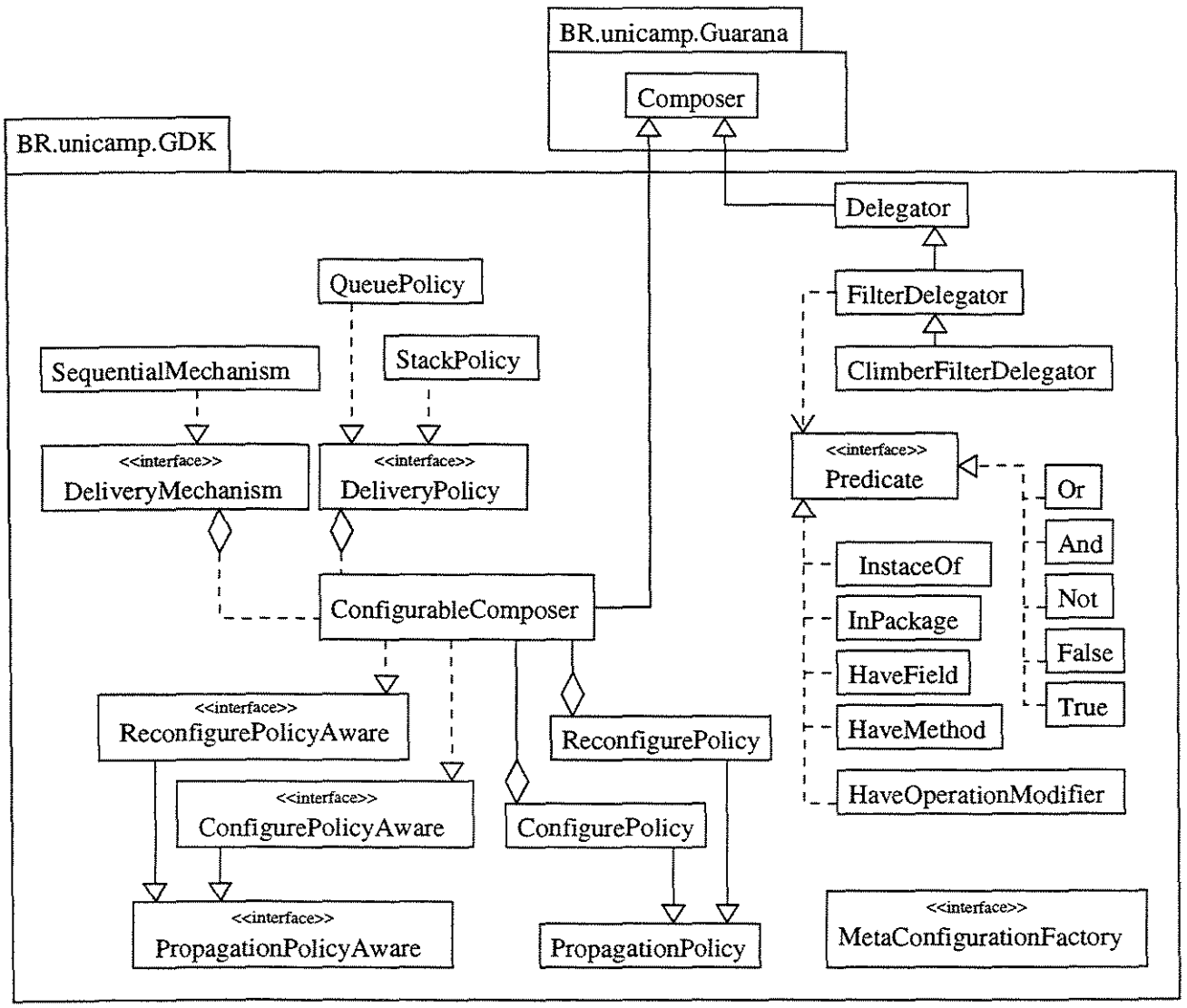


Figura 5.1: **Guaraná** Development Kit Framework

No escopo do GDK, existe ainda um sub-agrupamento de componentes denominado **dejavu**, que é uma abreviação para **Debugging Java Utility**. O *dejavu* foi originalmente concebido para oferecer facilidades de depuração sobre uma para-aplicação qualquer minimizando a intervenção sobre mesma, e sem incorrer na necessidade de recompilação de quaisquer módulos.

5.1.1 ConfigurableComposer

Os *Composers* são meta-objetos que possuem um papel fundamental: a distribuição do fluxo de meta-informação ao longo da árvore de meta-objetos que constituem a meta-configuração. Idealmente, pode-se dizer que todo elemento não-folha de uma meta-configuração assume o papel de *Composer*.

O MOP de **Guaraná** define a interface *Composer* (atualmente na forma de classe abstrata) permitindo a identificação de meta-objetos que assumam o papel de *Composers*, e padronizando a forma de percorrer a árvore de meta-objetos "pendurada" em um *Composer*. Ou seja, todo meta-objeto derivado (sub-classe) de *Composer* é um nó não-folha de uma meta-configuração cujo acesso aos nós filhos se dá através do método *getMetaObjects()* ou *getMetaObjectsArray()*.

Existem três pontos passíveis de flexibilização na construção de um *Composer*, dois deles relacionados à capacidade do *Composer* de replicar-se (propagação descrita na seção 3.4).

O primeiro ponto diz respeito a que tipo de *Composer* deve ser gerado na replicação (propagação do *Composer*), uma instância da mesma classe ou de outra classe. Decidindo-se em favor da primeira, é melhor propagar a própria instância ou um clone?

O segundo ponto diz respeito à constituição dos meta-objetos subordinados ao *Composer*. Se restar somente um meta-objeto subordinado após uma requisição configuração, deve-se propagar o *Composer* ou apenas o meta-objeto subordinado restante? E se não restar nenhum, ainda é necessário propagar o *Composer*, mesmo na ausência de meta-objetos subordinados?

Estes dois primeiros aspectos dependem muito mais da composição de uma meta-configuração em particular do que de um *Composer* propriamente dito. Logo, é desejável que tais aspectos sejam parâmetros da construção de um *Composer*, ao invés de estarem embutidos na estrutura de cada tipo específico de *Composer*.

O terceiro aspecto diz respeito à ordem de delegação de meta-informação definida por um *Composer* sequencial. Já antecipamos ao leitor mais ávido que *Composers* paralelos, para os quais o conceito de ordem de delegação não faz tanto sentido, serão discutidos ainda neste capítulo na seção 5.3.1.

Voltando aos *Composers* sequenciais, existem duas semânticas de ordenação reincidentes na Ciência da Computação, são elas: *Last In First Out (LIFO)* e *First In First Out(FIFO)*. Usualmente apelidadas de pilhas e filas, respectivamente. É concebível que a semântica

de ordenação seja ortogonal a grande parte dos diferentes tipos de *Composers* sequenciais imagináveis, justificando assim que a ordenação seja também um parâmetro na construção do *Composer*.

Até a versão 1.6 de **Guaraná**, além da classe abstrata *Composer*, o MOP fornecia apenas a classe *SequentialComposer* como ponto de partida para o meta-programador gerar meta-configurações. Os três pontos passíveis de flexibilização estavam imiscuídos na hierarquia de generalização/especialização que levava da classe *Composer* a classe SequentialComposer.

Essa opção de implementação oferece uma implementação econômica quanto ao consumo de memória. Porém, nossa experiência indicou esta estruturação era um pouco obtusa. Por um lado dificultava o meta-programador inexperiente em perceber como flexibilizar um *Composer* de acordo com suas necessidades no contexto de uma meta-configuração em particular. Por outro lado, essa abordagem era onerosa ao meta-programador inexperiente que se via forçado a fazer *subclassing* toda vez que precisava gerar um *Composer* com parâmetros diferentes.

Essas observações estimularam a criação do **ConfigurableComposer**, que oferece em seu construtor parâmetros com os quais calibrar seu comportamento no âmbito dos três pontos recém discutidos. Algumas das características do *ConfigurableComposer* foram embutidas na reformulação do *SequentialComposer* no release 1.7 de **Guaraná**. No entanto, ainda estamos convencidos que o *ConfigurableComposer* é mais intuitivo e tem maior apelo ao meta-programador.

Quando um *ConfigurableComposer* é criado, além de especificar o conjunto de meta-objetos subordinados, especificam-se três políticas (formas pré-definidas de comportamento) através dos parâmetros: ConfigurePolicy, ReconfigurePolicy e DeliveryPolicy.

As duas primeiras versam sobre a permanência ou não do *Composer* no contextos de reconfiguração e configuração (propagação) respectivamente. Tanto ConfigurePolicy como ReconfigurePolicy oferecem o mesmo espectro de parâmetros, sendo eles descritos na tabela 5.1.

A terceira política, DeliveryPolicy, define a semântica de ordenação sequencial, tipicamente de pilha (*StackPolicy*) ou fila (*QueuePolicy*). No primeiro caso (pilha), cada operação de para-nível será entregue a todos meta-objetos subordinados antes que seja iniciada a coleta de resultados na ordem inversa de entrega. No segundo caso (fila), a ordem de coleta de resultados é a mesma utilizada na entrega. Ambas subclasses de *DeliveryPolicy* se utilizam de um *DeliveryMechanism* para efetuar a entrega de operações e coleta de resultados, que por sua vez possui a subclasse *SequentialMechanism* que implementa trocas síncronas no âmbito dos diversos subordinados de um mesmo *ConfigurableComposer*.

Tabela 5.1: Parâmetros de configuração de *ConfigurableComposer*

ALWAYS_INVOKE_NEW_COMPOSER	Independentemente da composição dos meta-objetos subordinados após uma configuração ou reconfiguração, sempre será invocado o método NewComposer() que especifica como construir o *Composer* a ser instalado na meta-configuração destino com os meta-objetos subordinados remanescentes.
NO_COMPOSER_IF_EMPTY	Não introduz um *Composer* na meta-configuração resultante se o mesmo não possuir subordinados.
NO_COMPOSER_IF_UNITARY	Não introduz um *Composer* na meta-configuração resultante se existir um único meta-objeto subordinado.
SAME_COMPOSER_IF_IDENTICAL	Não produzir a clonagem de nenhum elemento da meta-configuração corrente, mantendo as mesmas instâncias de meta-objetos na meta-configuração resultante.
ECONOMICAL	Somatório das três opções acima, minimizando o consumo de memória no decorrer da propagação e meta-objetos.

5.1.2 Predicates

Predicate é uma interface comum à objetos de meta-nível que oferecem um método cuja avaliação produz um resultado lógico (booleano): verdadeiro ou falso. Objetos predicados servem ao propósito de encapsularem expressões de teste tornando-as *first class values*, e por conseguinte permitindo que tais expressões sejam passadas como parâmetro e armazenadas por meta-objetos.

Os predicados construídos para o GDK se dividem em: constantes booleanas (True,False), operações booleanas (And,Or,Not) e operações reflexivas (HaveMethod, HaveField, InstanceOf, InPackage, HaveOperationModifier).

Um caso típico do uso de predicados é ilustrado pelas construções derivadas de *Delegators* (vide seção 5.1.3), nos quais os predicados determinam se uma determinada primitiva do MOP deve ser delegada ou não para um meta-objeto subordinado ao delegador.

Predicados também possuem um papel importante na manutenção da segurança do MOP.

Como já foi discutido na diretriz de *Maximização da Transparência*, é indesejável que um meta-objeto exponha uma referência para si mesmo. Pois, dessa forma, o meta-objeto permitiria que um elemento externo vinculasse sobre si um outro meta-objeto, adicionando um novo nível reflexivo. Nestas circunstâncias, o meta-objeto que revelou sua referência torna-se um para-objeto em relação ao novo nível reflexivo, e como tal sujeito a qualquer tipo de manipulação. Em suma, a publicação da referência de um meta-objeto pode significar uma quebra de segurança. Em contrapartida, existem momentos em que é desejável fazer testes sobre um determinado meta-objeto, por exemplo em uma negociação de reconfiguração. É nessas ocasiões que os predicados revelam seu valor.

Um meta-objeto pode publicar um predicado cuja avaliação seja equivalente em resultado a qualquer teste feito diretamente através da referência do meta-objeto. É importante ressaltar que tais predicados não devem armazenar internamente a referência do meta-objeto que os publicou, caso contrário estariam expondo indiretamente esta referência que poderia ser obtida reflexivamente.

A título de completude, existe uma técnica que previne a quebra de segurança por manipulação reflexiva. Basta que a entidade exposta seja pré-vinculada a um *MetaBlockerReconfigure*, o que impedirá quaisquer tentativas de reconfiguração reflexiva. Esta técnica é aplicável diretamente ao meta-objeto do exemplo anterior, bem como a qualquer predicado por ele exportado. O *MetaBlockerReconfigure* é descrito na seção 5.3.1.

5.1.3 Delegator

O **Delegator** é um *Composer* que aceita um único meta-objeto subordinado, para o qual delega incondicionalmente todas suas primitivas. A utilidade do *Delegator* é servir como classe base para a criação de filtros ou delegadores condicionais.

5.1.4 FilterDelegator

O **FilterDelegator** é uma subclasse de *Delegator*, logo é também um *Composer*. Porém, a delegação de suas primitivas está condicionada a avaliação de predicados. Ou seja, cada uma das primitivas básicas, exceto *initialize* e *release*, está sujeita avaliação de uma instância de independente de *Predicate*. Se a avaliação do respectivo predicado resultar verdadeira a primitiva será delegada, do contrário não será repassada para o meta-objeto subordinado ao *FilterDelegator*.

FilterDelegators, consoantes a criação de meta-objetos genéricos, permitem a utilização focalizada de componentes de meta-nível, ainda que os últimos tenham sido construídos sem um foco específico de atuação. Em outras palavras, é possível restringir a atuação indesejada de um meta-objeto genérico, se beneficiando apenas da funcionalidade desejada que oferecida pelo meta-objeto filtrado.

A título de exemplificação, suponhamos a existência de um meta-objeto que monitora as atividades de atributos e métodos de seus para-objetos. Para monitorar exclusivamente a atividade de métodos de para-objetos podemos reutilizar tal componente sem modificá-lo, para tanto basta subordiná-lo em uma meta-configuração a um *FilterDelegator* que não delegue a reificação de atributos.

5.1.5 ClimberFilterDelegator

O **ClimberFilterDelegator** deriva de um FilterDelegator, diferenciando-se pela capacidade de propagar-se para classes, além de objetos. Na ativação de sua primitiva *initialize*, se a para-classe do para-objeto a que está vinculado for diferente de java.lang.Class, então toda a meta-configuração encabeçada por este componente tentará se instalar nesta para-classe.

A necessidade de propagar uma meta-configuração do para-objeto para sua para-classe foi discutida na seção 4.3.3. Sem essa propagação, a invocação de métodos de classe numa cadeia de invocação representaria uma barreira impedindo a meta-configuração que tenta se propagar ao longo da cadeia.

É importante notar que nenhum meta-objeto pertencente a meta-configuração que migra em direção a para-classe deve tratar a mensagem *NewObject*. Caso algum meta-objeto o faça inadvertidamente, pode-se incorrer no fenômeno da dupla propagação discutido na seção 4.3.3.

5.1.6 MetaConfigurationFactory

A **MetaConfigurationFactory** é uma interface cujo propósito é encapsular meta-configurações segundo o conhecido padrão *AbstractFactory* [17] . Em particular, instâncias concretas de *MetaConfigurationFactory* são utilizadas pelo utilitário *Launcher* (seção 5.2.5).

5.2 dejavu – Debugging Java Utility

O **dejavu**, abreviação para *Debugging Java Utility*, é um subconjunto do GDK voltado para depuração de para-aplicações e meta-aplicações. Houve duas motivações por trás da concepção do *dejavu*.

A primeira motivação é de natureza prática, a máquina virtual Java[1] sobre a qual o MOP de **Guaraná** foi implementado não possuia recursos elementares de depuração. Logo, tornou-se premente o desenvolvimento de algum tipo de ferramenta de auxílio à depuração.

E por que não no meta-nível? Consequentemente, o desenvolvimento de *dejavu* revelou que o MOP é uma ferramenta extremamente adequada à criação de recursos de depuração abrangentes e transparentes.

O segundo motivo que impulsionou a concepção do *dejavu* foi a necessidade de testar a flexibilidade do MOP em algum nicho de aplicação. Desta forma, encaramos o *dejavu* como uma implementação *proof-of-concept*, enquanto os demais componentes do *GDK* são encarados como extensões do MOP.

5.2.1 XMLogger

O primeiro elemento criado foi o **XMLogger**, cuja escolha foi feita por sua proximidade funcional e estrutural a um elemento já presente no MOP, o *MetaLogger*. Em função dessa proximidade, foi possível experimentar com sucesso a manipulação do meta-nível mesmo possuindo-se pouco *know-how* sobre o MOP.

O *XMLogger* introduz uma sintaxe no formato XML [77], oferecendo mais meta-informação do que seu precursor. Sobretudo, a escolha da sintaxe XML facilita o *parsing* e o intercâmbio com outros aplicativos.

Cada instância de *XMLogger* pode ser "batizada" com um prefixo *string* para auxiliar sua identificação. Uma técnica de depuração muito eficaz,que utiliza este recurso, é intercalar o meta-objeto alvo de depuração entre duas instâncias de *XMLogger* com diferentes prefixos. A análise subsequente do *log* gerado permite analisar o comportamento do componente alvo e do *Composer* de quem o alvo está subordinado.

5.2.2 InspectClass

O objeto de meta-nível **InspectClass** não é um meta-objeto, ou seja não pode ser vinculado a nenhum para-objeto. *InspectClass* é um objeto visual de meta-nível para inspeção estrutural de classes.

[1] Inicialmente o MOP de **Guaraná** foi implementado sobre o Kaffe 1.0.5, e posteriormente portado para a versão 1.0.6

Em particular, este componente não utiliza nenhuma característica especial do MOP de **Guaraná**, apenas faz uso das facilidades reflexivas já presentes na linguagem Java – *Java Core Reflection (JCR)*. Logo, pode ser utilizado em qualquer JVM sem restrição.

A motivação por trás da criação deste componente foi explorar os recursos disponíveis em JCR. A utilidade deste componente é permitir a documentação estrutural de qualquer classe, seja ela para-classe ou meta-classe, no formato XML.

5.2.3 InspectObject

O **InspectObject** cria uma interface gráfica para manipulação interativa dos atributos do para-objeto vinculado. Para cada atributo é gerado um rótulo com o nome do atributo e uma caixa-de-edição com o valor do atributo. Dessa forma, o para-objeto vinculado poderia ser inspecionado e modificado pelo meta-programador interativamente.

Entretanto, o *InspectObject* sofre três limitações sérias na sua versão atual.

A primeira delas é a falta de otimização em relação à ativação do gancho reflexivo. Onde qualquer intervenção sobre o para-objeto vinculado para alteração do valor de um atributo gera uma re-ativação do próprio meta-objeto. Uma solução para este problema é proposta na seção 5.3.1 que discute o *ReflectiveShield*.

A segunda limitação é o tratamento exclusivo a atributos primitivos. Ou seja, se o para-objeto agrega outros para-objetos que não sejam tipos primitivos da linguagem, então os últimos não estão sujeitos a manipulação pelo *InspectObject*. Do ponto de vista teórico, bastaria que ao *InspectObject* propagar-se para tais atributos agregados. Entretanto, como não foi resolvido ainda o problema da combinação dinâmica de meta-configurações (vide seção 5.3.1) optou-se por uma implementação *proof-of-concept* mais simples.

A última limitação se relaciona à propagação deste meta-objeto. Uma vez que cada instância de *InspectObject* cria uma janela independente no ambiente gráfico para visualização e manipulação, a propagação indiscriminada deste componente pode resultar numa explosão de janelas impossível de se manipular. Portanto, a lógica de propagação do *InspectObject* deve ser elaborada com cuidado, e de preferência minimizando a propagação. Desta forma, a inclusão deste componente no meta-programa exige num plano de vinculação extremamente focado em para-objetos específicos.

5.2.4 BreakPoint e MetaBreak

Tradicionalmente, ambientes de depuração para linguagens compiladas faziam (e ainda fazem) uso de instruções privilegiadas (TRAP) dos processadores para permitir o chaveamento entre a aplicação depurada e depurador.

Muitas vezes é preciso re-compilar a aplicação para inclusão da tabela de símbolos, a qual permitirá a legibilidade das estruturas sendo depuradas, traduzindo endereços para os respectivos identificadores no código-fonte original.

Já os ambientes de depuração para linguagens interpretadas são muito mais ricos em meta-informação e facilidades de depuração, uma vez que a integração entre ambiente de desenvolvimento e interpretador é mais fácil do que entre ambiente de desenvolvimento e processador.

A maioria das linguagens interpretadas modernas adota uma modelo misto, onde existe um passo de pré-compilação para agilizar a interpretação. O código resultante desta pré-compilação é usualmente denominado *bytecode*, que é efetivamente submetido ao interpretador ao invés do código-fonte original.

Apesar da atual linguagem hospedeira do MOP (Java) possuir diversos recursos e ambientes de depuração, a JVM sobre a qual **Guaraná** foi implementado não oferecia nenhum recurso de depuração no âmbito do código Java. Só era possível depurar a própria JVM, que foi implementada em C.

Logo, a despeito do uso extensivo de *logs*, a única alternativa de depurar uma aplicação sobre **Guaraná** era no nível de abstração da máquina virtual. Essa limitação fomentou a criação de mecanismos, ainda que rudimentares, para depuração de meta-aplicações em **Guaraná**.

As classes **BreakPoint** e **MetaBreak** cooperam para o oferecimento da facilidade de depuração passo a passo.

BreakPoint é um *singleton* [17] que gerencia um evento de espera para todas *Threads* pertencentes a um mesmo grupo, bem como provê uma interface gráfica para o meta-programador controlar o fluxo de execução.

Por sua vez, instâncias de *MetaBreak* são meta-objetos que ativam a situação de parada em *BreakPoint*. Um meta-objeto *MetaBreak* pode ser inserido em qualquer posição dentro de uma meta-configuração. Inclusive uma mesma meta-configuração pode conter diversas instâncias de *MetaBreak*, o que é extremamente útil quando o alvo da depuração é a própria meta-configuração e não seu para-objeto vinculado. Em particular, na construção de uma instância de *MetaBreak* é possível parametrizar um *Predicate* por primitiva do MOP gerando pontos-de-parada condicionais.

5.2.5 Launcher

O **Launcher** é uma ferramenta para disparar uma para-aplicação que esteja vinculada a uma meta-aplicação, com a restrição de que o plano de vinculação sempre se inicie pela vinculação de meta-configurações a para-classes.

Apesar do *Launcher* não ser essencialmente uma ferramenta de depuração, sua construção foi motivada pela necessidade de disparar vários meta-programas distintos sobre uma mesma para-aplicação. Entre esses meta-programas, vários destinavam-se à depuração.

Outra razão para o *Launcher* pertencer ao *dejavu* ao invés de diretamente ao *GDK*, é que o primeiro contém ferramentas prontas enquanto o segundo contém bibliotecas. Mas esta distinção é tênue, podendo desaparecer em versões futuras.

O *Launcher* recebe como parâmetros a classe da para-aplicação a ser executada, seus argumentos de execução, e uma lista de vínculos entre para-classes e suas respectivas meta-configurações (primeira etapa do plano de vinculação). Estes parâmetros podem ser passados através da linha de comando da *shell* ou através de um arquivo de configuração em XML. A vantagem do último é documentar e tornar persistente o plano de vinculação.

Inicialmente, o *Launcher* inspeciona reflexivamente a classe que representa a para-aplicação, verificando se a mesma possui o convencional ponto de partida representado pelo método *main()*. Se este método estiver presente a para-aplicação é válida, por conseguinte, o *Launcher* efetuará as vinculações especificadas através de chamadas a *Guarana.reconfigure()*. Só então o método *main()* será invocado com os respectivos argumentos, iniciando assim a execução da para-aplicação.

Note que o meta-programa já está em execução mesmo antes do início da para-aplicação, pois as para-classes reflexivas são carregadas antes da para-aplicação requisitar sua carga.

5.2.6 GuaraLAB

O **GuaraLAB** é um rudimento de ambiente de desenvolvimento de meta-aplicações. Existem duas motivações por trás da construção do *GuaraLAB*. A primeira delas é ilustrar uma aplicação da trinca **Guaraná**, *GDK* e *dejavu* na geração automática de diagramas de interação, que são representações gráficas da troca de mensagens entre objetos de acordo com o padrão UML [6, 48].

A segunda motivação é a criação de um ambiente *user friendly* que facilite a interação entre meta-programadores e o MOP de **Guaraná**. A primeira motivação é notadamente a mais relevante das duas no escopo desta dissertação.

GuaraLAB oferece três funcionalidades básicas.

1. Uma interface gráfica para o *Launcher*, que facilita a escolha da para-aplicação, especificação de parâmetros, e definição do plano de vinculação.

2. Um interpretador dos *logs* produzidos por instâncias de *XMLogger*, que oferece sintaxe colorida (syntax highlight) e tradução do formato XML para outro formato textual com maior legibilidade ou favorável a intercâmbio.

3. Um diagramador em UML que traduz graficamente os *logs* de *XMLogger* em diagramas de interação.

No futuro, planeja-se que o *GuaraLAB* também suporte a construção visual de meta-configurações.

5.3 Aplicabilidade e Extensões

Além dos componentes já apresentados no *GDK* e *dejavu*, existe uma série de outros componentes de meta-nível cuja concepção é uma consequência natural dos temas abordados nesta dissertação.

Alguns dos componentes a serem apresentados nesta seção começaram a ser pesquisados, porém não foram finalizados em tempo hábil ou simplesmente ainda não ofereceram resultados conclusivos. Outros componentes foram apenas idealizados ou representam nichos potenciais de aplicação do MOP.

Ao mesmo tempo que prenunciamos o prosseguimento de nossa linha de pesquisa no cenário pós-dissertação, tentamos despertar o interesse de outros pesquisadores para estes novos nichos. Sobretudo, a enumeração desta lista de componentes pode servir de desafio à comunidade *open source* em contribuir no projeto **Guaraná**.

5.3.1 Expandindo GDK

Existe uma série de meta-componentes que já foi projetada, contudo ainda não foi implementada no GDK. Esta seção enumera e discute diversos destes meta-componentes, que representam a materialização das técnicas discutidas no capítulo 4 na forma de artefatos de software presentes no meta-nível.

MetaBlockerReconfigure

O **MetaBlockerReconfigure** é o meta-objeto cuja finalidade é impedir a reconfiguração reflexiva de seu para-objeto. Ou seja, uma vez que um para-objeto seja vinculado a uma meta-configuração que possua um MetaBlockerReconfigure, então essa meta-configuração não poderá mais ser trocada por outra, por conseguinte o para-objeto ficará permanentemente vinculado a tal meta-configuração até a extinção do próprio para-objeto.

Este dispositivo é extremamente útil para conferir segurança a para-entidades que habitem um cenário reflexivo. Sobretudo, este dispositivo respeita o princípio da *conexão causal* e não fere a diretriz da *Abrangência Reflexiva*.

Utilizando o MOP de **Guaraná** é extremamente fácil implementar o *MetaBlockerReconfigure*. Basta que o meta-objeto primário sempre retorne o valor nulo em seu tratador *reconfigure*, rejeitando assim toda e qualquer reconfiguração. Utilizando-se uma especialização de *FilterDelegator* seria possível compor este comportamento de forma ortogonal ao resto da meta-configuração.

ReflectiveShield

O **ReflectiveShield** visa facilitar o aspecto *intervenção* do MOP, evitando a ativação do *gancho reflexivo* quando um meta-objeto intervém sobre seu próprio para-objeto e não deseja tratar reflexivamente esta operação.

Pela diretriz da *Abrangência Reflexiva*, o *gancho reflexivo* está sempre ativo, resultando na reificação de qualquer operação sobre um para-objeto reflexivo, ainda que oriunda de um meta-objeto presente na própria meta-configuração do para-objeto alvo.

Como foi discutido na seção 4.3.2, a reificação de operações originadas do meta-nível dificulta a tarefa do meta-programador, que precisa criar explicitamente um mecanismo para ignorar o tratamento destas operações que ele próprio originou.

Sem violar a diretriz da *Abrangência Reflexiva*, o *ReflectiveShield* é um *Composer* que repassa a seus meta-objetos subordinados uma fábrica de operações modificada. Tal fábrica produz operações "marcadas" com o selo de que foram produzidas no meta-nível. Então as operações são submetidas ao para-nível e consequentemente reificadas. Porém, o *ReflectiveShield* consegue distinguir estas operações daquelas oriundas do próprio para-nível, filtrando as primeiras e delegando as últimas. Logo, os meta-objetos subordinados ao *ReflectiveShield Composer* estão protegidos da reificação de suas próprias operações sobre o para-nível.

GatherComposer

Um dos problemas ainda em aberto no MOP é a combinação dinâmica entre meta-configurações. Apesar de já existirem mecanismos para a troca de componentes da meta-configuração, não estão claras as políticas de reconfiguração. Em particular, um desses problemas que se

manifesta com frequência é quando o contexto estrutural tenta instalar uma meta-configuração sobre um para-objeto já reflexivo a partir do contexto comportamental.

Nessa situação sentimos falta de um mecanismo prático para combinar as duas meta-configurações em uma única, sem que haja meta-componentes de mesma funcionalidade replicados na meta-configuração resultante.

Em teoria, a meta-configuração já instalada precisaria fazer o levantamento do grafo de seus meta-componentes, contrastando-o com o grafo de meta-componentes da meta-configuração entrante. A diferença entre os dois grafos poderia ser adicionada à meta-configuração já instalada. Contudo, dependendo da composição de ambas meta-configurações pode não estar claro em que nó da árvore instalada os novos meta-componentes podem ser pendurados.

Na falta de um protocolo para negociação posicional entre a meta-configuração entrante e a já instalada, propomos um mecanismo mais simplório ainda que extremamente útil: o **GatherComposer**. Se uma meta-configuração pressupõe a aceitação dinâmica de outros meta-componentes ela incluirá um *GatherComposer* em sua constituição. O *GatherComposer* assinala o ponto de inserção para a meta-configuração entrante, que pode ter sido simplificada pela remoção dos componentes em duplicata.

SmartDelegation

SmartDelegation não é um novo componente de meta-nível, mas sim uma otimização aplicável sobre quaisquer um dos *Composers* já discutidos anteriormente.

As duas implementações concretas de *Composers*, *SequentialComposer* (no MOP) e *ConfigurableComposer* (no GDK), sempre delegam os resultados de operações a todos meta-objetos subordinados, desconsiderando se tais meta-objetos requisitaram explicitamente a inspeção de tais resultados ou não. Levar em consideração a requisição de inspeção de resultados pode diminuir drasticamente o processamento consumido no meta-nível por meta-configurações extensas.

MetaTAG

MetaTAG é um meta-objeto que serve para definir um agrupamento lógico de meta-configurações, e consequentemente de seus para-objetos associados. Cada *MetaTAG* é parametrizado por uma "chave", que pode ser representada por qualquer objeto imutável, como um inteiro ou uma cadeia de caracteres (string).

Uma mesma "chave" identifica um agrupamento lógico. Uma vez definido um agrupamento lógico, torna-se possível manipular conjuntos de meta-configurações atomicamente. Por exemplo, o *MetaTAG* permitiria reconfigurar simultaneamente diversas meta-configurações.

MetaHistory

MetaHistory é um meta-objeto que armazena sequencialmente todas as operações sofridas por seu para-objeto vinculado, possibilitando assim a implementação das facilidades de "undo" e "redo" ortogonalmente a estrutura do para-objeto associado.

Se os parâmetros das operações armazenadas forem serializáveis, é ainda possível criar o histórico de operações de forma persistente.

ConcurrentComposer

Através de um **ConcurrentComposer** seria possível tratar reflexivamente um para-objeto através de vários meta-objetos simultaneamente. Para tanto, o *ConcurrentComposer* delegaria as primitivas básicas do MOP a meta-objetos subordinados que executassem em *threads* independentes.

Uma forma elegante de implementar tal funcionalidade seria através da criação de um *ConcurrentMechanism* derivado de *DeliveryMechanism*, que se encaixaria no arcabouço proposto na seção 5.1.1 para o *ConfigurableComposer*.

Ainda seria necessária a introdução de um terceiro elemento, *ResponseCombinatorPolicy*, cuja responsabilidade seria definir uma semântica para a coleta e combinação das diversas respostas dos meta-objetos paralelos. Entre as semânticas mais óbvias podemos citar: preferência pela primeira resposta válida, preferência pela resposta mais frequente, entre outras.

Percebe-se uma aplicação direta do *ConcurrentMechanism* no suporte a tolerância a falhas no meta-nível.

5.3.2 Expandindo dejavu

Nesta seção, apresentaremos uma série de sugestões que ilustram o potencial de aplicação do MOP de **Guaraná** no contexto de Engenharia de Software. Enquanto os meta-componentes apresentados na seção 5.3.1 auxiliavam a construção de meta-configuração; as extensões propostas nesta seção são sugestões de ferramentas, cujo propósito é prestar assistência ao meta-programador durante as fases de: desenvolvimento, depuração, otimização, testes e manutenção do binômio para/meta-aplicação.

Nenhuma das ferramentas apresentadas nesta seção foi efetivamente implementada, consistem apenas em sugestões de áreas que podem ser exploradas reflexivamente através do MOP de **Guaraná**.

MetaStress

A idéia do *MetaStress* é simular a baixa disponibilidade de recursos de forma ortogonal a qualquer para-aplicação. A título de exemplificação, podemos pensar em instâncias de *MetaStress* responsáveis pelo consumo de memória ou ciclos de CPU.

MetaProfiling

Outro nicho muito próximo ao de depuração é a análise de desempenho de para-aplicações, que também pode se beneficiar das facilidades reflexivas do MOP. Entre as atribuições de um componente de **MetaProfiling**, podemos citar a contagem do número de acessos a atributos e métodos do para-objeto vinculado, bem como a medição dos tempos de processamento envolvidos.

É importante ressaltar que na medição de tempos tal componente precisaria ser calibrado para descontar os tempos gastos no processo de reificação.

MetaTesting

MetaTesting não é um componente mas toda uma área a ser explorada através do MOP.

Devido a diversidade de mecanismos reflexivos oferecidos, o MOP de **Guaraná** aparenta ser uma plataforma atraente para a implementação de ferramentas automáticas de testes.

Para citar um exemplo, a análise de mutantes por injeção de falhas pode ser abordada através do recurso de *replacement operations*, que permite a troca dos argumentos de um método antes que este seja efetivamente submetido ao para-objeto alvo.

Simultaneamente a redação desta dissertação já existem outros trabalhos [67, 68] voltados a explorar *MetaTesting* através dos recursos de **Guaraná**.

MetaAssert

Um recurso clássico em depuração de programas é a utilização de cláusulas *assert* em pontos estratégicos do programa. Tais cláusulas avaliam uma expressão booleana, que ao assumir um valor falso produzem uma exceção ou instrução de *trap* para depuração. Tipicamente, cláusulas *assert* podem ser desabilitadas através de diretivas de compilação. Sendo assim, a intrusão das cláusulas *assert* no código-fonte é amenizada pelo recurso de compilação condicional.

Através do MOP é possível criar um componente semelhante, em funcionalidade, às cláusulas *assert*, porém sem incorrer em quaisquer alteração no código-fonte da para-aplicação. O **MetaAssert** é um meta-objeto parametrizado por *Predicates* (seção 5.1.2), que permite a validação do estado do para-objeto vinculado antes ou após a ativação de seus métodos

ou do acesso a seus atributos. O *MetaAssert* pode até ser usado em combinação com outras meta-entidades de depuração como *MetaBreak* e *BreakPoint*.

MetaConsole

O **MetaConsole** consiste em um meta-componente que oferece um console para entrada e saída de comandos, criando um diálogo interativo entre o meta-programador e as demais meta-entidades. Este tipo de ferramenta traria benefícios para o aprendizado, a experimentação com o MOP, e o teste de meta-entidades.

Existem hoje diversas linguagens interpretadas que foram implementadas sobre uma JVM, e já oferecem um console de comandos. Qualquer uma dessas linguagens serviria como um bom ponto de partida para a implementação do *MetaConsole*. Em particular, fizemos experiências com a implementação de Python (descrita na seção) sobre a JVM instrumentada por **Guaraná**. Os resultados foram promissores, como pode ser conferido no trecho a seguir:

```
JPython 1.1 on java1.0.6-20010401 (JIT: kaffe.jit)
Copyright (C) 1997-1999 Corporation for National Research Initiatives
>>> import BR.unicamp.Guarana.MetaLogger
>>> import java.lang.String
>>> x = java.lang.String('Rodrigo')
>>> import BR.unicamp.Guarana.MetaLogger
>>> y = BR.unicamp.Guarana.MetaLogger()
>>> BR.unicamp.Guarana.Guarana.reconfigure(x,None,y)
Initialize: java.lang.String@8205760
>>> x
Operation: java.lang.String@8205760.java.lang.String.toString()
Result: return java.lang.String@8205760
Operation: java.lang.String@8205760.java.lang.String.value
Result: return [C@83700f8
Operation: java.lang.String@8205760.java.lang.String.offset
Result: return 0
Operation: java.lang.String@8205760.java.lang.String.count
Result: return 7
Rodrigo
>>>
```

Neste exemplo, numa sessão de execução do console de JPython[2] criamos um para-objeto

[2]Na ocasião do teste a linguagem se chamava JPython, porém tal projeto agora denomina-se Jython. maiores informações estão disponíveis no site http://www.jython.org .

em Java instanciando a classe java.lang.String. Após a vinculação entre o para-objeto 'x' e o meta-objeto *MetaLogger*, podemos monitorar todas as operações subjacentes à um pedido de avaliação do valor de 'x'.

5.4 Sumário

Neste capítulo, apresentamos extensões reflexivas ao MOP de **Guaraná** agrupadas no toolkit GDK. Tais extensões ilustram muitas das diversas técnicas propostas para contornar as armadilhas da meta-programação.

A primeira sugestão de extensão é a incorporação do recurso de reificação ativa diretamente no núcleo do MOP, onde detalhamos o formato do tratador e seu comportamente esperado para uma integração harmoniosa com as demais estruturas do MOP.

Foi dado um destaque especial para os meta-componentes destinados à tarefa de depuração, os quais foram agrupados em uma sub-divisão do GDK batizada de *dejavu* ou *DEbugging JAVa Utility*.

Dentre os componentes do GDK, exploramos:

ConfigurableComposer que possui propagação e semântica de delegação configuráveis;

Predicates que encapsulam expressões booleanas de teste em *first class values*;

Delegator que serve de base para criação de delegadores específicos por herança;

FilterDelegator que faz delegação condiciona à avaliação de predicados;

MetaConfigurationFactory que encapsula meta-configurações;

Dentre os componentes do dejavu, exploramos:

XMLogger que é uma extensão do *MetaLogger* utilizando a sintaxe XML;

InspectClass que é um objeto de meta-nível visual para inspeção estrutural de classes;

InspectObject que é um meta-objeto visual para inspeção comportamental de para-objetos;

Launcher que é uma ferramenta para o disparo de meta-aplicações sobre para-aplicações parametrizados por um plano de vinculação;

GuaraLAB que é um rudimento de ambiente de desenvolvimento de meta-aplicações;

Adicionamos ao conjunto de técnicas já apresentado, a utilização de um conjunto de *MetaBlockerReconfigure* num segundo nível reflexivo, cujos para-objetos vinculados são os meta-objetos do primeiro nível reflexivo. Esta associação visa proteger os meta-objetos vulneráveis de manipulações reflexivas maliciosas. Entretanto, tais meta-objetos não poderão mais ser reconfigurados reflexivamente, ainda que seus para-objetos possam.

Ao final do capítulo, apresentamos diversas extensões sobre *GDK* e *dejavu* que são candidatas à exploração no prosseguimento desta linha de pesquisa. Entre elas: MetaBlockerReconfigure, ReflectiveShield, GatherComposer, SmartDelegation, MetaTAG, MetaHistory, ConcurrentComposer, MetaStress, MetaProfiling, MetaTesting, MetaAssert, MetaConsole.

Capítulo 6

Exemplo de Programação Reflexiva com GDK/Dejavu

6.1 Definindo o Exemplo

Neste capítulo, ilustraremos um sub-conjunto das técnicas e ferramentas propostas através da construção de um exemplo simples. Tal exemplo consiste em uma meta-aplicação capaz de tornar qualquer para-aplicação "supersticiosa" em relação ao número 13. Para tanto vamos construir um meta-objeto capaz de monitorar escritas do valor 13, substituindo-as por escritas do valor 7.

A título de simplificação vamos restringir a abrangência reflexiva sobre campos do tipo *integer*, deixando outros tipos inteiros primitivos de lado. Essa omissão se justifica no anseio de manter a clareza e objetividade do exemplo.

Através deste exemplo seremos capazes de ilustrar:

- como um meta-objeto é capaz de tratar meta-informação reificada e intervir sobre seu respectivo para-objeto;
- como combinar a funcionalidade de meta-objetos ortogonais entre si em uma única meta-configuração;
- como disparar o binômio meta-aplicação/para-aplicação;
- como utilizar componentes do GDK/dejavu para analisar a execução da aplicação.

Uma das motivações à construção deste exemplo em particular é ilustrar a capacidade de **Guaraná** de utilizar critérios dinâmicos ou comportamentais como fatores de controle sobre a meta-computação. Ou seja, neste exemplo o que determina a atuação do meta-nível é o comportamento dinâmico da para-aplicação ao invés de sua estrutura estática.

6.2 Identificação das Para-Entidades

No contexto do exemplo, as para-entidades alvo são todos os para-objetos (ou para-classes) capazes de assumir o valor 13 em um campo inteiro.

6.3 Plano de Vinculação

Supondo total desconhecimento sobre a para-aplicação, é razoável supor que qualquer para-entidade seja elegível como um alvo em potencial. Logo, projetaremos uma meta-configuração capaz de *vinculação plena*. Através de critérios dinâmicos será possível identificar se uma para-entidade qualquer é ou não um alvo efetivo. Em particular basta reificar todo e qualquer acesso de escrita sobre campos inteiros, verificando se o mesmo assume o valor 13. Em caso positivo, a respectiva para-entidade que originou a reificação é efetivamente identificada com alvo.

6.4 Construção das Meta-Entidades

A seguir descreveremos o meta-objeto *Superstitious*, responsável pela inspeção e interveção sobre o para-nível em qualquer manifestação de escrita do valor 13. Este meta-objeto pode ser vinculado a múltiplos para-objetos, o que é coerente com a decisão de vinculação plena. Em função desta capacidade, tal meta-objeto foi projetado para propagar a si mesmo ao invés de clones ao longo da cadeia de criação de novos para-objetos. Esta decisão almeja economizar o consumo de memória, uma vez que não se sabe *a priori* quão populoso é o para-nível.

Source: Superstitious.java

```
1 import java.util.HashMap;
2 import java.lang.reflect.*;
3 import BR.unicamp.Guarana.*;
4 import java.lang.IllegalAccessException;

5 public class Superstitious extends MetaObject {

6   private HashMap __pob2factory ;
7   public Superstitious() { __pob2factory = new HashMap(); }

8   public void initialize(OperationFactory factory, Object obj) {
9      final HashWrapper obw = new HashWrapper(obj);
10     __pob2factory.put(obw,factory);
11   }

12  public Result handle(Operation op,Object obj) {
13     HashWrapper obw = new HashWrapper(obj);
14     OperationFactory factory =  (OperationFactory)__pob2factory.get(obw);
15     try {
16          if (op.isWriteOperation()) { // intervenção sobre escrita
17             final Field f = op.getField();
18             final Class type = f.getType();
19             if ((type.isPrimitive()) && (type.equals(Integer.TYPE))) {
20                final Object value = op.getValue();
21                if (value.equals(new Integer(13)) && (factory!=null))
22                   return Result.operation(factory.write(f,new Integer(7),op),
23                                           Result.inspectResultMode);
24             }
25          }
26     } catch (IllegalAccessException ex) {}
27     return null; // não desejo inspecionar o resultado
28   }

29   public MetaObject configure(final Object newObject,final Object object) {
30       return this; // este meta-objeto propaga a si mesmo quando requisitado
31   }
32 }
```

Source: SuperstitiousMcfg.java

```
1  import BR.unicamp.Guarana.*;
2  import BR.unicamp.GDK.*;
3  import BR.unicamp.dejavu.*;

4  public class SuperstitiousMcfg
5      implements MetaConfigurationFactory {

6      public MetaObject getMetaObject() {

7          Predicate pConfigure = null;
8          String[] avoidClasses = {"java.io.ByteArrayOutputStream",
9                                   "java.lang.String",
10                                  "java.lang.StringBuffer",
11                                  "java.lang.Integer"};

12         try {
13             pConfigure = new Not(new InstanceOf(avoidClasses,"new-para-object"));
14         } catch (ClassNotFoundException e) {}

15         MetaObject md1       = new XMLogger("Log"),
16                    md2       = new MetaBreak(true),
17                    md3       = new Superstitious(),
18                    mos[]     = {md1,md2,md3},
19                    composer  = new ConfigurableComposer(mos),
20                    primary   = new ClimberFilterDelegator(composer,
21                                                           pConfigure,
22                                                           null,
23                                                           null,
24                                                           pConfigure
25                                                             ,null);
26         return primary;
27     }
28 }
```

Neste exemplo, o meta-objeto *Superstitious* será acomponhando por outros meta-objetos na composição de meta-configuração. Incluiremos uma instância de *XMLogger* na composição meta-configuração, cuja finalidade será retratar detalhadamente os eventos ocorridos no para-nível durante a computação. Também estará presente uma instância de *MetaBreak* que permite a execução passo-a-passo do para-nível.

Os três meta-objetos apresentados serão combinados através de um ConfigurableComposer, que utiliza por default um mecanismo sequencial de delegação com política de fila. A menos que especificado o contrário, o *ConfigurableComposer* utilizará uma política econômica de propagação. Ou seja, se a meta-configuração mudar de composição e restarem um ou menos componentes, o próprio *ConfigurableComposer* deixará a meta-configuração. Todo este comportamento está definido implicitamente pela escolha do construtor utilizado. Entretanto, todos estes parâmetros são configuráveis atrvés do uso de outros construtores.

Ainda na composição da meta-configuração é utilizado um *ClimberFilterDelegator* que serve a dois propósitos. O primeiro propósito é criar um filtro de propagação condicional através do uso de predicados, impedindo a propagação da meta-configuração para um conjunto pré-especificado de classes. O segundo propósito é conferir a meta-configuração a capacidade de se propagar tanto para objetos quanto para suas classes. Ao se optar por *vinculação plena* em razão de desconhecimento sobre a para-aplicação, essa capacidade torna-se indispensável.

6.5 Disparo do Binômio Para-Aplicação/Meta-Aplicação

Para ilustrar a atuação da meta-configuração escolhemos uma para-aplicação simples, descrita na classe *Hello* a seguir.

Source: **Hello.java**

```
1  public class Hello {
2      public static void main(String[] args) {
3          for(int i=12;i<15;i++)
4              System.out.println("before = "+i+" after= "+ new Number(i).toString());
5      }
6  }

7  class Number {
8      private int luckyNumber;
9      Number(int luckyNumber) {
10         this.luckyNumber = luckyNumber;
11     }
```

```
12      public String toString() {
13          return new Integer(luckyNumber).toString();
14      }
15  }
```

Ao se executar a para-aplicação isoladamente, sem qualquer meta-nível vinculado, obtém-se o seguinte resultado:

```
[rodrigo@Max]$ java Hello
parameter = 12 field = 12
parameter = 13 field = 13
parameter = 14 field = 14
```

Para disparar o binômio para-aplicação/meta-aplicação utilizaremos a ferramenta *Launcher* do *dejavu*. Através do *Launcher* é possível vincular para-nível e meta-nível através de um arquivo de configuração. Este arquivo pressupõe uma estratégia simples de vinculação, onde cada para-classe especificada será vinculada a uma respectiva meta-configuração.

```
<MetaApplication>
 <ParaApplication main="Hello" arguments="" />
 <Reconfigure  class="Hello" mcfg="SuperstitiousMcfg"/>
</MetaApplication>
```

O disparo da meta-aplicação e o produto de sua execução são representados abaixo nas formas textual e gráfica:

```
[rodrigo@Max]$ guarana BR.unicamp.dejavu.Launcher --file=superstitious.xml
before = 12 after= 12
before = 13 after= 7
before = 14 after= 14
```

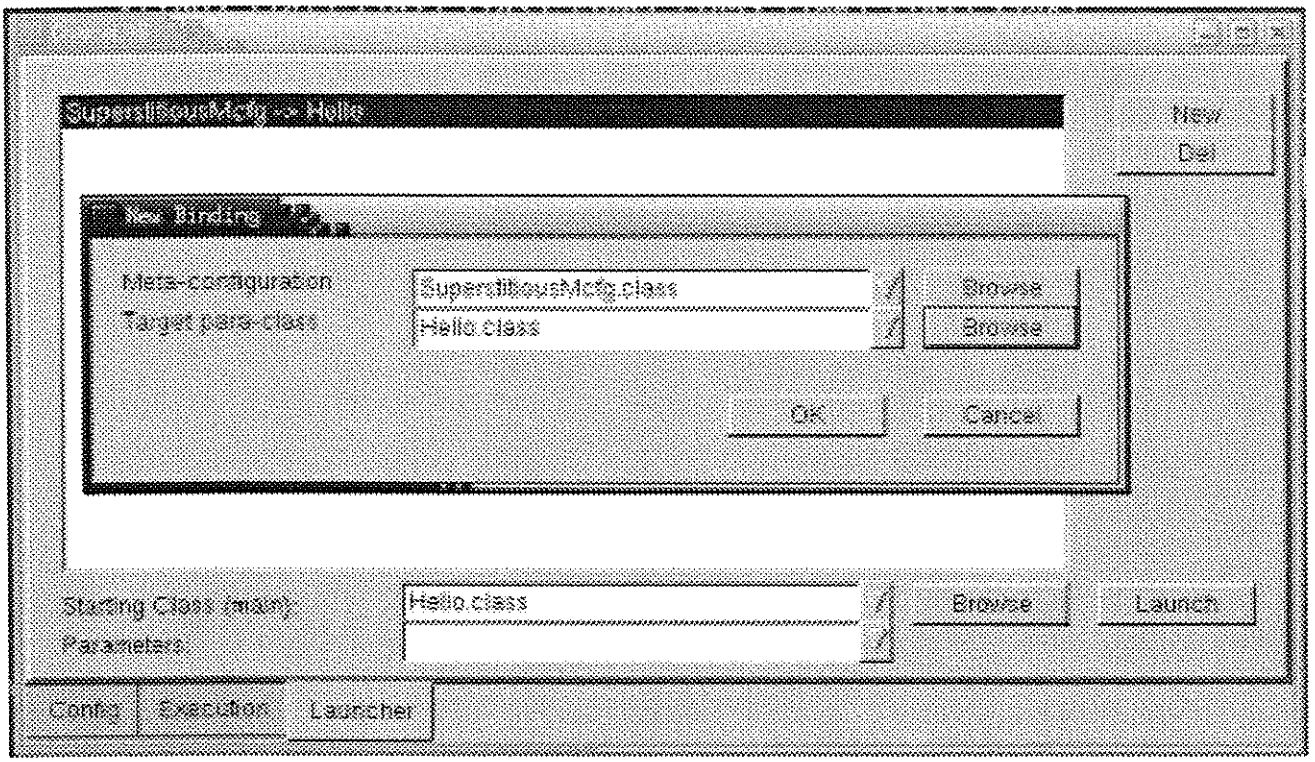


Figura 6.1: Interface gráfica do Launcher no contexto do GuaraLAB

Figura 6.2: Interface gráfica do MetaBreak

6.6 Análise dos Registros de *XMLogger*

Segue uma transcrição dos registros produzidos pelo XMLogger durante a computação:

```
<?xml version='1.0' encoding='ISO-8859-1' standalone='no'?>
<XMLogger>
<log prefix="Log" time="60ms" para-oid="Hello" para-class="java.lang.Class" primitive="initialize"
factory="8339a78"/>
<log prefix="Log" time="99ms" para-oid="Hello" para-class="java.lang.Class" primitive="operation"
opid="8322df0"  thread="java.lang.Thread@81731a0" method="getMethod(...)" />
<log prefix="Log" time="109ms" para-oid="Hello" para-class="java.lang.Class" primitive="operation"
opid="8351338"  thread="java.lang.Thread@81731a0" method="getMethod0(...)" />
<log prefix="Log" time="117ms" para-oid="Hello" para-class="java.lang.Class" primitive="result"
return="java.lang.reflect.Method@834c710" opid="8351338" thread="java.lang.Thread@81731a0"
method="getMethod0(...)"/>
<log prefix="Log" time="118ms" para-oid="Hello" para-class="java.lang.Class" primitive="result"
return="java.lang.reflect.Method@834c710" opid="8322df0" thread="java.lang.Thread@81731a0"
method="getMethod(...)"/>
<log prefix="Log" time="119ms" para-oid="Hello" para-class="java.lang.Class" primitive="operation"
opid="8351db8"  type="static" thread="java.lang.Thread@81731a0" method="main(...)" />
<log prefix="Log" time="131ms" para-oid="Hello" para-class="java.lang.Class" primitive="configure"
new-para-oid="832d8e0" new-para-class="Number" />
<log prefix="Log" time="135ms" para-oid="832d8e0" para-class="Number" primitive="initialize"
factory="82e9438" />
<log prefix="Log" time="136ms" para-oid="832d8e0" para-class="Number" primitive="configure"
new-para-oid="Number" new-para-class="java.lang.Class" />
<log prefix="Log" time="136ms" para-oid="Number" para-class="java.lang.Class" primitive="initialize"
factory="835d498"/>
<log prefix="Log" time="145ms" para-oid="Number" para-class="java.lang.Class" primitive="message"
msgclass="BR.unicamp.Guarana.NewObject" new-para-oid="832d8e0" new-para-class="Number" />
<log prefix="Log" time="147ms" para-oid="832d8e0" para-class="Number" primitive="operation"
opid="836abf8"  thread="java.lang.Thread@81731a0" ctor="Number(int 12)" />
<log prefix="Log" time="148ms" para-oid="832d8e0" para-class="Number" primitive="operation"
opid="836af40"  thread="java.lang.Thread@81731a0" ctor="java.lang.Object()" />
<log prefix="Log" time="148ms" para-oid="832d8e0" para-class="Number" primitive="result"
return="null" opid="836af40"  thread="java.lang.Thread@81731a0" ctor="java.lang.Object()" />
<log prefix="Log" time="149ms" para-oid="832d8e0" para-class="Number" primitive="operation"
opid="837a5d8"  thread="java.lang.Thread@81731a0" write="luckyNumber" value="12" />
<log prefix="Log" time="150ms" para-oid="832d8e0" para-class="Number" primitive="result"
return="null"  opid="837a5d8" thread="java.lang.Thread@81731a0" write="luckyNumber" value="12" />
<log prefix="Log" time="150ms" para-oid="832d8e0" para-class="Number" primitive="result"
return="null"  opid="836abf8" thread="java.lang.Thread@81731a0" ctor="Number(int 12)" />
<log prefix="Log" time="151ms" para-oid="832d8e0" para-class="Number" primitive="operation"
opid="837afb0"  thread="java.lang.Thread@81731a0" method="toString()" />
<log prefix="Log" time="152ms" para-oid="832d8e0" para-class="Number" primitive="operation"
opid="8383338"  thread="java.lang.Thread@81731a0" read="luckyNumber" />
<log prefix="Log" time="152ms" para-oid="832d8e0" para-class="Number" primitive="result"
return="12" opid="8383338"  thread="java.lang.Thread@81731a0" read="luckyNumber" />
<log prefix="Log" time="152ms" para-oid="832d8e0" para-class="Number" primitive="result"
return="12" opid="837afb0"  thread="java.lang.Thread@81731a0" method="toString()" />
<log prefix="Log" time="157ms" para-oid="Hello" para-class="java.lang.Class" primitive="configure"
new-para-oid="8189940" new-para-class="Number" />
<log prefix="Log" time="157ms" para-oid="Number" para-class="java.lang.Class" primitive="operation"
opid="8383f78" thread="java.lang.Thread@81731a0" method="getName()" />
<log prefix="Log" time="158ms" para-oid="Number" para-class="java.lang.Class" primitive="result"
return="Number"  opid="8383f78" thread="java.lang.Thread@81731a0" method="getName()" />
<log prefix="Log" time="158ms" para-oid="8189940" para-class="Number" primitive="initialize"
factory="835d818" />
<log prefix="Log" time="158ms" para-oid="8189940" para-class="Number" primitive="configure"
new-para-oid="Number" new-para-class="java.lang.Class" />
<log prefix="Log" time="168ms" para-oid="Number" para-class="java.lang.Class" primitive="reconfigure"
old-meta-oid="0" old-meta-class="null" new-meta-oid="8391110"
new-meta-class="BR.unicamp.GDK.ClimberFilterDelegator" />
<log prefix="Log" time="170ms" para-oid="Number" para-class="java.lang.Class" primitive="initialize"
factory="835dd18" />
<log prefix="Log" time="173ms" para-oid="Number" para-class="java.lang.Class" primitive="release" />
<log prefix="Log" time="173ms" para-oid="Number" para-class="java.lang.Class" primitive="message"
msgclass="BR.unicamp.Guarana.NewObject" new-para-oid="8189940" new-para-class="Number" />
<log prefix="Log" time="174ms" para-oid="8189940" para-class="Number" primitive="operation"
opid="839c258" thread="java.lang.Thread@81731a0" ctor="Number(int 13)" />
<log prefix="Log" time="174ms" para-oid="8189940" para-class="Number" primitive="operation"
opid="839c5a0"  thread="java.lang.Thread@81731a0" ctor="java.lang.Object()" />
<log prefix="Log" time="174ms" para-oid="8189940" para-class="Number" primitive="result"
return="null" opid="839c5a0"  thread="java.lang.Thread@81731a0" ctor="java.lang.Object()" />
<log prefix="Log" time="175ms" para-oid="8189940" para-class="Number" primitive="operation"
```

```
opid="839cb88"  thread="java.lang.Thread@81731a0" write="luckyNumber" value="13" />
<log prefix="Log" time="176ms" para-oid="8189940" para-class="Number" primitive="result"
return="null" opid="839ced0"  thread="java.lang.Thread@81731a0" write="luckyNumber" value="7" />
<log prefix="Log" time="176ms" para-oid="8189940" para-class="Number" primitive="result"
return="null" opid="839c258"  thread="java.lang.Thread@81731a0" ctor="Number(int 13)" />
<log prefix="Log" time="177ms" para-oid="8189940" para-class="Number" primitive="operation"
opid="83a6680"  thread="java.lang.Thread@81731a0" method="toString()" />
<log prefix="Log" time="177ms" para-oid="8189940" para-class="Number" primitive="operation"
opid="83a6958" thread="java.lang.Thread@81731a0" read="luckyNumber" />
<log prefix="Log" time="178ms" para-oid="8189940" para-class="Number" primitive="result"
return="7" opid="83a6958"  thread="java.lang.Thread@81731a0" read="luckyNumber" />
<log prefix="Log" time="178ms" para-oid="8189940" para-class="Number" primitive="result"
return="7" opid="83a6680"  thread="java.lang.Thread@81731a0" method="toString()" />
<log prefix="Log" time="179ms" para-oid="Hello" para-class="java.lang.Class" primitive="configure"
new-para-oid="836b9e8" new-para-class="Number" />
<log prefix="Log" time="179ms" para-oid="Number" para-class="java.lang.Class" primitive="operation"
opid="83af648" thread="java.lang.Thread@81731a0" method="getName()" />
<log prefix="Log" time="180ms" para-oid="Number" para-class="java.lang.Class" primitive="result"
return="Number"  opid="83af648" thread="java.lang.Thread@81731a0" method="getName()" />
<log prefix="Log" time="180ms" para-oid="836b9e8" para-class="Number" primitive="initialize"
factory="835dab8" />
<log prefix="Log" time="180ms" para-oid="836b9e8" para-class="Number" primitive="configure"
new-para-oid="Number" new-para-class="java.lang.Class" />
<log prefix="Log" time="180ms" para-oid="Number" para-class="java.lang.Class" primitive="reconfigure"
old-meta-oid="0" old-meta-class="null" new-meta-oid="83915f0"
new-meta-class="BR.unicamp.GDK.ClimberFilterDelegator"/>
<log prefix="Log" time="181ms" para-oid="Number" para-class="java.lang.Class" primitive="initialize"
factory="835d678"/>
<log prefix="Log" time="181ms" para-oid="Number" para-class="java.lang.Class" primitive="release" />
<log prefix="Log" time="181ms" para-oid="Number" para-class="java.lang.Class" primitive="message"
msgclass="BR.unicamp.Guarana.NewObject" new-para-oid="836b9e8" new-para-class="Number" />
<log prefix="Log" time="182ms" para-oid="836b9e8" para-class="Number" primitive="operation"
opid="83b67d0"  thread="java.lang.Thread@81731a0" ctor="Number(int 14)" />
<log prefix="Log" time="182ms" para-oid="836b9e8" para-class="Number" primitive="operation"
opid="83b6b18"  thread="java.lang.Thread@81731a0" ctor="java.lang.Object()" />
<log prefix="Log" time="183ms" para-oid="836b9e8" para-class="Number" primitive="result" return="null"
opid="83b6b18" thread="java.lang.Thread@81731a0" ctor="java.lang.Object()" />
<log prefix="Log" time="183ms" para-oid="836b9e8" para-class="Number" primitive="operation"
opid="83bf1b0"  thread="java.lang.Thread@81731a0" write="luckyNumber" value="14" />
<log prefix="Log" time="184ms" para-oid="836b9e8" para-class="Number" primitive="result"
return="null" opid="83bf1b0"  thread="java.lang.Thread@81731a0" write="luckyNumber" value="14" />
<log prefix="Log" time="184ms" para-oid="836b9e8" para-class="Number" primitive="result"
return="null" opid="83b67d0"  thread="java.lang.Thread@81731a0" ctor="Number(int 14)" />
<log prefix="Log" time="185ms" para-oid="836b9e8" para-class="Number" primitive="operation"
opid="83bfb88"  thread="java.lang.Thread@81731a0" method="toString()" />
<log prefix="Log" time="185ms" para-oid="836b9e8" para-class="Number" primitive="operation"
opid="83bfe60"  thread="java.lang.Thread@81731a0" read="luckyNumber" />
<log prefix="Log" time="185ms" para-oid="836b9e8" para-class="Number" primitive="result" return="14"
opid="83bfe60"  thread="java.lang.Thread@81731a0" read="luckyNumber" />
<log prefix="Log" time="186ms" para-oid="836b9e8" para-class="Number" primitive="result" return="14"
opid="83bfb88"  thread="java.lang.Thread@81731a0" method="toString()" />
<log prefix="Log" time="186ms" para-oid="Hello" para-class="java.lang.Class" primitive="result"
return="null" opid="8351db8" type="static" thread="java.lang.Thread@81731a0" method="main(...)"/>
</XMLogger>
```

Nitidamente, os registros produzidos pelo *XMLogger* não são adequados à análise direta devido a sua complexidade de detalhes. Daí se justifica a criação do GuaraLAB, em cujo ambiente há facilidades para filtragem e visualização da meta-informação.

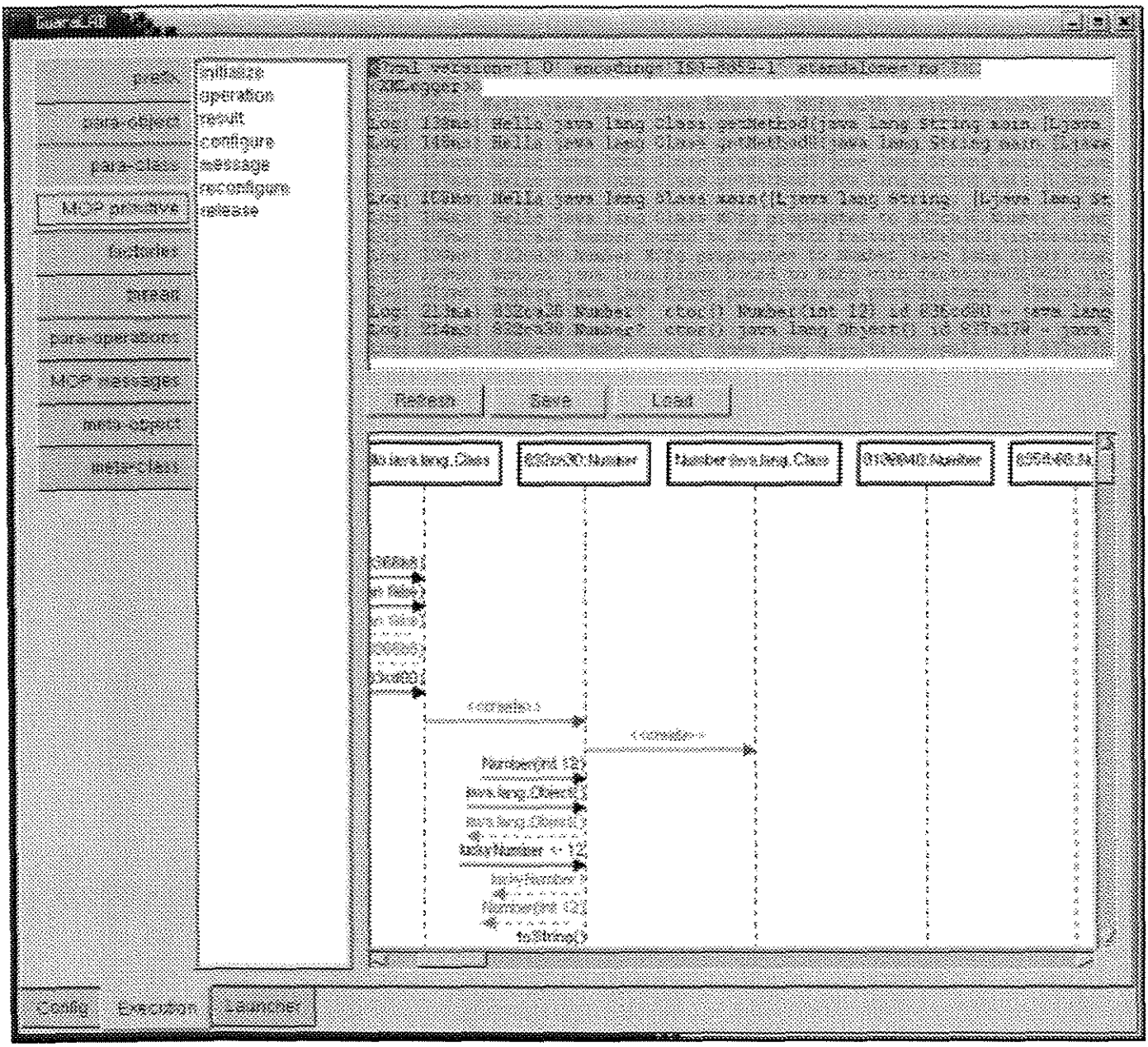


Figura 6.3: **Guaraná** GuaraLAB

Capítulo 7

Conclusão

Procuramos neste capítulo revisar os tópicos mais importantes apresentados nesta dissertação, dando ênfase às contribuções do autor. Discutimos a relação entre *conexão causal* e MOPs seguros. Enunciamos as questões deixadas em aberto, as quais ainda não foram respondidas previamente tampouco nesta dissertação. Finalmente, apresentamos nossa visão sobre os caminhos mais promissores a serem trilhados na continuidade deste trabalho de pesquisa.

7.1 Contribuições

Nesta pesquisa, as contribuições do autor se subdividem em sete grupos de igual importância: introdução de terminologia complementar ao jargão de reflexão computacional, unificação dos critérios de comparação entre MOPs, revisão bibliográfica abrangente e detalhada com ênfase sobre MOPs implementados sobre Java, descrição do MOP de **Guaraná** sob a perspectiva do usuário, proposta de um modelo de atuação para o meta-programador, análise das técnicas e obstáculos a programação reflexiva e, finalmente, proposta e construção de ferramental à meta-programação.

7.1.1 Terminologia

No capítulo 2 introduzimos o termo *para-objeto*, que designa um objeto qualquer de um nível de abstração o qual está vinculado reflexivamente a um conjunto de meta-objetos em um nível de abstração superior. Extrapolando a semântica e uso deste prefixo podemos aplicá-lo para designar quaisquer estruturas do nível subjacente ao reflexivo.

Esta inovação terminológica se justifica por duas razões. A primeira é que o termo "base", amplamente utilizado na literatura sobre reflexão, não possui a mesma carga semântica que o prefixo "para". O primeiro designa uma entidade *não reflexiva* que *reside no primeiro*

nível da pilha reflexiva, enquanto o último designa apenas uma entidade que *sofre reflexão*, podendo ela residir em qualquer nível da pilha reflexiva, e por conseguinte podendo ela *exercer reflexão* sobre uma entidade em um nível subjacente. Logo, o prefixo "para" não intenciona substituir o adjetivo "base" ou a locução "do nível base", não obstante almeja complementar uma lacuna semântica.

O texto dessa dissertação é um exemplo concreto da aplicabilidade do termo, facilitando ambos: a redação e a compreensão.

7.1.2 Critérios de Classificação de um MOP

No decorrer do capítulo 2 também formalizamos a análise de propostas de protocolos de meta-objetos, através da compilação de quatro aspectos (Vinculação,Reificação,Execução e Intervenção) e cinco critérios (Temporalidade, Associatividade, Transparência, Abrangência e Reflexividade).

Esta compilação procurou ser o mais abrangente possível, almejando fornecer arcabouço teórico para a descrição de um MOP em particular ou uma análise comparativa entre MOPs.

7.1.3 Análise Comparativa entre MOPs

Ainda no capítulo 2 apresentamos uma revisão histórica de diversos projetos que incorreram no campo da Reflexão Computacional. Em particular, apresentamos uma análise comparativa entre **Guaraná** e os demais MOPs implementados sobre uma Máquina Virtual Java. Essa análise comparada serve ao propósito de contextualizar **Guaraná** no seu nicho de pesquisa, e sobretudo testa a adequação dos critérios propostos para a classificação de MOPs.

O resultado da avaliação quantitativa indicou que o MOP de **Guaraná** é mais abrangente que os demais no oferecimento de recursos. Dentre as omissões mais relevantes no MOP de **Guaraná**, ressaltamos:

- ausência de *reificação ativa* ou *interceptação de saída*, presente em metaXa;
- ausência de objetos ativos, presente em Correlate;
- independência de JVM, nos moldes de Dalang/Kava.

Entre as vantagens oferecidas por **Guaraná** ressaltamos:

- transparência implícita;
- composição hierárquica arbitrária de meta-objetos;
- temporalidade durante-execução para os 4 aspectos do MOP;
- vinculação baseada em instância e classe.

Observamos também que a extensão reflexiva através de modificação da JVM é extremamente transparente (transparência implícita), contudo não permite modificações sintáticas na linguagem. Esse fato pode ser interpretado como uma vantagem ou desvantagem dependendo do contexto.

A extensão do MOP de Guaraná para suportar a aplicação de técnica de *wrapping* ou *shadowing* poderia ser uma alternativa para contornar o problema de desempenho de uma meta-aplicação específica, ou permitir a introdução de novos recursos sintáticos na linguagem hospedeira. Entretanto, nestes modo de operação pagar-se-ia o preço de menor flexibilidade e menor transparência.

7.1.4 Descrição de Guaraná sob Perspectiva do Meta-Programador

No capítulo 3 descrevemos detalhadamente o MOP proposto por Alexandre Oliva, entretanto com um enfoque diferente da documentação original.

Além de utilizarmos os termos e critérios convencionados previamente, elencamos as 5 *diretrizes* do MOP:

- Isolamento entre meta-objeto primário e para-objeto.
- Privilégio da Anterioridade
- Minimalidade e Completude
- Maximização da Transparência
- Abrangência Reflexiva

As diretrizes do MOP representam *guidelines* que orientam a utilização do MOP, na medida que ilustram os princípios que guiaram sua própria construção (projeto e implementação).

A título de complementação à documentação original, apresentamos o mapeamento de algumas estruturas disponíveis no MOP em padrões de projeto amplamente conhecidos.

7.1.5 Modelo do Meta-Programador

No capítulo 4 construímos um modelo de atuação para o meta-programador, constituído por diversos cenários de atuação e um método de quatro fases:

1. identificação das para-entidades;
2. construção das meta-entidades;
3. elaboração do plano de vinculação;
4. disparo da (para/meta) aplicação.

Existem preocupações que foram deixadas de fora do modelo de meta-programação supracitado. São elas:

- Quando um meta-objeto deve escolher propagar um clone ou a si mesmo?
- Como determinar o tempo útil de permanência de um meta-objeto em uma dada meta-configuração qualquer?

Tais preocupações são dependentes do contexto, ou seja da semântica dos meta-objetos envolvidos, cujas respostas ficam a critério do meta-programador.

7.1.6 Obstáculos x Técnicas

Outra contribuição importante desta dissertação foi determinar os principais obstáculos e armadilhas existentes na meta-programação através de **Guaraná**, qual sua origem, e quais são as técnicas para contorná-los.

Este contraste é resumido na tabela 7.1

Tabela 7.1: Quadro comparativo entre obstáculos e técnicas na meta-programação

Obstáculo	**Técnica**
a fatoração do espaço de para-objetos através de critérios estáticos e dinâmicos (identificação de alvos)	*marcação* ou *tagging* através de *MetaTAG*
a brecha de segurança nos campos *final public* e *final protected*	utilização de *MetaBlockerReconfigure* ou substituição por campos *private*
dificuldade de mesclar meta-configuração oriunda do contexto comportamental com meta-configuração oriunda do contexto estrutural	uso de *GatherComposer* para atuar como ponto de mesclagem entre meta-configuração instalada e candidata
quebra da cadeia de propagação quando existem métodos de classe, prejudicando a *identificação da cadeia de construção*	uso do *ClimberFilterComposer* encabeçando a meta-configuração
a dificuldade da *identificação da cadeia de invocação* somente com reificação *passiva* ou *de chegada*, ou se já houvesse na linguagem informação contextual na pilha de execução	aplicação da técnica de *para-objetos proxy*, ou a introdução do suporte a reificação *ativa(de chegada)*, ou introdução de identificação única de objetos
a dupla propagação de uma meta-configuração oriunda de ambos contextos: comportamental e estrutural	tornar a meta-configuração *stateful*, ou introduzir um protocolo de negociação entre a meta-configuração vinculada e a redundante
o problema da recursão infinita reflexiva	*NOP replacement operations* ou *ReflectiveShield Composer*
o problema da vulnerabilidade ao se publicar uma referência a um meta-objeto qualquer	utilização de *MetaBlockerReconfigure* cujos para-objetos são aos próprios meta-objetos vulneráveis
incapacidade de saber se uma mensagem foi entregue e compreendida	adicionar campos "accepted" e "understood" na interface de *Message*
ambiguidade semântica da valor nulo (null) no uso do método *Guarana.reconfigure()*	introduzir constantes para distinguir referência a "qualquer meta-objeto" de referência a "nenhum meta-objeto"

7.1.7 Ferramental para Meta-Programação

Uma contribuição concreta desta pesquisa foi a disponibilização de ferramental para meta-programação, que se manifesta através do componentes do *GDK* (ConfigurableComposer, Predicates, Delegator, FilterDelegator e MetaConfigurationFactory) e do *dejavu* (XMLogger, InspectClass, InspectObject, Launcher e GuaraLAB).

Para se ter uma noção do esforço de codificação, o *GDK* é composto por 27 classes, totalizando 2.85 KLoc. O *dejavu* é composto por 7 classes, totalizando 1.33 KLoc. Totalizando 4.18 KLoc codificados em Java. O GuaraLAB se destaca dos demais por ter sido codificado em Python, incluindo 6 módulos que acrescem 1KLoc ao montante total.

Considerando toda a funcionalidade agregada por este conjunto de software, 5.18KLoc representa um esforço de codificação bem pequeno se comparado a qualquer projeto comercial mesmo de pequeno porte.

7.2 Conexão Causal

Uma questão ainda mais ampla, que extrapola o contexto do MOP de **Guaraná**: **Do ponto de vista puramente teórico, seria possível construir um MOP simultaneamente fiel à conexão causal, seguro e pertencente ao domínio reflexivo?** Se o MOP é fiel à conexão causal, sua representação subjacente está biunivocamente ligada com sua implementação efetiva. Se o MOP é seguro deveria ser possível garantir a inviolabilidade de suas estruturas fundamentais e assegurar um conjunto minimal de propriedades invariantes. Se o MOP pertence ao domínio reflexivo, toda e qualquer estrutura do MOP deve ser passível de reflexão.

Por conseguinte, se é possível manipular a representação do MOP reflexivamente como é possível afirmar que tal MOP é seguro? Como é possível garantir a inviolabilidade de suas estruturas, ou afirmar enumerar ao menos uma propriedade invariante? Se o núcleo é reflexivo, a reificação de qualquer operação do para-nível é também uma operação a ser reificada, e sua reificação também e assim sucessivamente. Por construção, esse modelo teórico é infactível sem que ao menos uma estrutura não seja excluída do domínio reflexivo para servir de base à recursão no processo de reificação. Logo, concluímos que não é possível obter um MOP totalmente fiel a conexão causal e inteiramente construído sobre o domínio reflexivo.

Lembramos apenas que o MOP de **Guaraná** é seguro, pois seu núcleo não pertence ao domínio reflexivo, como foi visto na seção 3.2.1.

7.3 Questões em Aberto

Existem obstáculos a meta-programação que ainda não foram satisfatoriamente contornados. Entre eles podemos citar os questionamentos:

- Como mesclar a meta-configuração proveniente do contexto estrutural com a meta-configuração (já instalada) proveniente do contexto comportamental?
- Como testar a equivalência, intersecção ou diferença entre meta-configurações?

Estes dois questionamentos estão interligados. O componente *GatherComposer* do GDK, descrito na seção 5.3.1, é um primeiro esforço de atender ao primeiro questionamento sem entrar no mérito do segundo. Entretanto, encontrar uma resposta ao segundo questionamento permitiria a criação de mecanismos mais flexíveis que *GatherComposer*.

Uma outra lacuna a ser preenchida é a definição de uma linguagem visual para representação padronizada da estrutura e comportamento de um MOP. A proposta que mais se aproxima seria a UML, que é adequada a representação do paradigma OO em geral, contudo não apresenta simbologia para expressar interações inter-nível, tampouco apresenta simbologia para representar meta-entidades. A medida que recursos reflexivos vem sendo incorporados às mais diversas linguagens de programação essa lacuna torna-se cada vez mais relevante, limitando o intercâmbio de material científico sobre o assunto. Existe uma proposta de extensão a UML chamada j-UML, que resolve as ambiguidades de UML quando mapeada para o contexto de implementações em Java, porém essa proposta também não contempla o âmbito reflexivo.

Outra questão em aberto são as implicações de facilidades reflexivas no campo da Engenharia de Software, e da Engenharia Reversa em particular, sem mencionar o mercado de software como um todo. Recentemente os movimentos *Free Software* e *Open Source* vem fomentando uma revolução na produção, distribuição e comercialização de software [59]. No epicentro deste turbilhão está a capacidade de se modificar software dada a disponibilidade do código fonte. Levando isto em consideração, **não seria razoável prever que se as técnicas de Reflexão Computacional forem amplamente difundidas atingiremos um cenário equivalente ao triunfo desses movimentos sobre o paradigma do "código proprietário"?**

Clarificando, dada premissa reflexiva de que a estrutura de software pode ser inspecionada e seu comportamento modificado, a presença ou ausência do código-fonte torna-se irrelevante, uma vez que os mesmos benefícios são atingidos por outro mecanismo: a manipulação reflexiva do binário em execução sem os custos de recompilação. Ainda nessa linha de raciocínio, **Quais são as implicações do uso de reflexão computacional sobre "código fechado" com copyright contra Engenharia Reversa?** A discussão deste tema deveria ser levada num contexto mais amplo, envolvendo membros da academia e da indústria. Um esforço pioneiro neste sentido foi documentado por François Rideau [15].

7.4 Próximos Passos

A continuação natural deste trabalho de pesquisa seria explorar as questões em aberto supracitadas e implementar as extensões propostas para GDK e dejavu nas seções 5.3.1 e 5.3.2.

Outro caminho a ser seguido é a implementação do MOP de **Guaraná** sobre uma outra linguagem hospedeira, diferente de Java. Tal esforço seria justificado para validar a independência do MOP quanto a linguagem de programação, bem como testar a universalidade das diretrizes delineadas. Nessa linha de trabalho, a linguagem Python é até o momento a escolha preferencial como linguagem hospedeira.

Outro nicho a ser explorado seria a implementação de outros MOPs sobre a plataforma reflexiva oferecida por **Guaraná**. Tal esforço seria justificado para validar a abrangência e flexibilidade oferecidas pelo MOP de **Guaraná**. Nessa linha de trabalho, os MOPs de Correlate [3, 29, 63, 65] e AspectJ [78] seriam as escolhas preferenciais.

O último em particular, AspectJ, abre um horizonte interessante, que seria validar **Guaraná** como MOP de baixo nível para oferecer o suporte a Aspect Oriented Programming [2, 34, 72].

Apêndice A

Suporte Reflexivo em Linguagens de *Scripting* Modernas

Uma das ramificações do universo de linguagens de programação é o das linguagens de *Scripting*, as quais se caracterizam por serem interpretadas e possuírem tipagem fraca. Além do vínculo histórico entre Reflexão Computacional e linguagens de *Scripting*[1], o crescente suporte a facilidades reflexivas nessas linguagens torna sua análise relevante no contexto desta dissertação.

Entre a miríade de linguagens de *scripting* presentes nos últimos dez anos, selecionamos apenas algumas que satisfazem os seguintes critérios: suporte a características reflexivas, penetração tanto na comunidade acadêmica quanto no âmbito comercial, e sobretudo o fato de continuarem em evolução concomitantemente a realização deste trabalho de pesquisa.

A.1 Tcl

Tcl [57] ou **Tool Command Language**, criada por John Ousterhout em 1988, se diz uma linguagem de *scripting* que almeja controlar e estender aplicações seguindo o modelo imperativo procedimental.

Do ponto de vista reflexivo, Tcl oferece suporte aos quatro aspectos reflexivos ainda que não esteja baseada no paradigma OO.

Quanto ao aspecto reificação, Tcl provê o comando **info** que multiplexa diversas sub-funcionalidades relacionadas a disponibilização de meta-informação estrutural. A descrição completa das sub-funções de reificação do mecanismo *info* é dada pela Tabela A.1; pela qual se percebe, a despeito de sua abrangência, sua natureza *ad hoc*.

[1] A primeira implementação formal de Reflexão baseou-se em Lisp (vide seção 2.3.1) que é uma das linguagens de *scripting* pioneiras.

[**] Estas sub-funções são as únicas que reificam meta-informação comportamental.

A ausência de suporte a reflexão comportamental foi sanada *a posteriori* pela extensão **XOTcl** [47], que adere ao paradigma OO e possui um MOP oferecendo facilidades para interceptação de mensagens entre objetos.

Tcl oferece também o comando **trace**, que contempla os três aspectos reificação, vinculação e intervenção. Através deste comando é possível vincular variáveis a procedimentos. A *abrangência* reflexiva de Tcl permite apenas que variáveis assumam o papel de para-entidades. Qualquer acesso às para-variáveis será reificado e entregue ao respectivo meta-procedimento.

Além da capacidade de monitoração, é permitido ao meta-procedimento intervir na computação corrente de duas formas possíveis: alterando o valor da para-variável ou negando acesso a mesma.

Por último, Tcl suporta o aspecto execução ao permitir que os meta-procedimentos de *trace*, que representam as meta-entidades, executem qualquer outro procedimento e tenham acesso a qualquer variável, desde que estejam em um contexto (espaço de nomes acessível) pelo meta-procedimento de *trace*. É ainda oferecido ainda um mecanismo de reificação que informa todas as associações de *trace* em um dado instante da computação.

Tabela A.1: Descrição das sub-funções de reificação do comando Tcl *info*

info args	Lista de nomes dos argumentos formais de um dado procedimento.
info default	Determina se um dado argumento de um determinado procedimento possui valor default especificado.
info body	Lista o corpo de um dado procedimento.
info locals	Lista variáveis locais no escopo de execução corrente.
info globals	Lista todas as variáveis globais.
info vars	Lista todas variáveis acessíveis dado o escopo corrente de execução.
info commands	Lista todos os procedimentos definidos e embutidos existentes no interpretador por ocasião da invocação.
info procs	Lista todos os comandos definidos no interpretador por ocasião da invocação.
info exists	Determina se determinada variável é acessível no contexto corrente.
info script	Informa o nome do *script* que está sendo interpretado.
info tclversion	Informa versão do interpretador.
info library	Informa a localização do diretório onde residem as bibliotecas Tcl.
info cmdcount**	Informa a contagem do número de comandos executados pelo interpretador na sessão corrente.
info level**	Determina o nível de aninhamento do escopo corrente de execução.

A.2 Perl

Criada por Larry Wall, *Perl* [79, 71] é a abreviação de *Practical Extraction and Report Language*. A linguagem possui um caráter eclético, agregando mecanismos derivados de outras linguagens como *C*, *sed* e *awk*. Em coerência com seu ecletismo, *Perl* oferece suporte a reflexão computacional, ainda que limitado.

Assim como *Tcl*, *Perl* contempla os quatro aspectos reflexivos, nominalmente: vinculação, reificação, execução e intervenção de forma *ad hoc*. Entretanto, quanto a importância os dois primeiros aspectos se destacam em relação ao dois últimos. O conjunto de facilidades reflexivas disponíveis na linguagem *Perl* abrange tanto reflexão estrutural quanto comportamental, sendo composto pelos mecanismos descritos na tabela A.2.

Além dos mecanismos descritos acima, ainda é possível consultar informações sobre as tabelas de símbolos de cada módulo (**package** na terminologia *Perl*), e consequentemente "reificar" todo o conjunto de para-entidades existente num dado instante da computação.

Portanto, *Perl* apresenta associatividade baseada em instância, temporalidade durante-execução, abrangência estrutural e comportamental, e transparência implícita somente para o mecanismo *tie*.

Tabela A.2: Descrição das facilidades reflexivas em Perl

info args	Lista de nomes dos argumentos formais de um dado procedimento.
função *tie()*	Este mecanismo permite a reificação dos acessos de leitura e escrita a uma dada variável(para-entidade) alvo de vinculação com uma dada meta-entidade. As para-entidades passíveis de vinculação em *Perl* são *scalar*, *array* [a], *hash* e *filehandle*. As meta-entidades são objetos, que por sua vez são implementados como dicionários do tipo*hash*. Dessa forma, as meta-entidades também podem assumir o papel de para-entidades. [a]No caso particular de variáveis do tipo *array*(ou vetor) só há reificação de seus elementos, e não do vetor como um todo.
função *tied()*	Este mecanismo informa a meta-entidade que está associada a uma para-entidade fornecida como parâmetro.
função *caller()*	Este mecanismo permite reificar estrutura e comportamento, rotulando as rotinas na pilha de execução com o módulo, arquivo e linha a que pertencem.
função *ref()*	Este mecanismo é uma manifestação de **run-time type information(rtti)** ou informação sobre tipos durante a execução. Rtti pode ser considerado um dos embriões de reflexão em linguagens de programação. Como se pode intuir, o mecanismo retorna o tipo da variável fornecida como parâmetro.
método *isa()*	Este mecanismo está presente em todo e qualquer objeto, através do qual é possível testar se o objeto pertence a hierarquia de especialização/generalização fornecida como parâmetro.
método *can()*	Assim como o mecanismo *isa*, *can* existe em todos os objetos, para os quais permite a uma entidade externa testar se um dado método é suportado pelo objeto.

A.3 Python

A linguagem de scripting Python [43, 58], idealizada por Guido Van Rossum em 1989, possui grande variedade de mecanismos de reificação de meta-informação. Assim como Perl, Tcl e Java, Python é uma linguagem interpretada, cujo interpretador é escrito em outra linguagem (tipicamente na linguagem C). Diferentemente de Tcl e Perl, Python permite intrinsecamente a construção de software segundo o paradigma OO, não havendo a necessidade de estender a linguagem através de mecanismos artificiais para simular orientação a objetos.

Enquanto nas três primeiras linguagens o mecanismo de reificação é "reflexivamente míope" em relação as entidades internas ao interpretador, em Python a jurisdição dos mecanismos de reificação abrange tanto estruturas internas ao interpretador, quanto estruturas definidas na aplicação sujeita a interpretação. Tal abrangência só é possível devido a existência de um protocolo de objetos que contempla reificação.

Apresentamos alguns dos principais mecanismos reflexivos disponíveis em Python na tabela A.3.

Os mecanismos **__getattr__()** e **__setattr__()** são manifestações dos aspectos reificação e intervenção misturados no próprio protocolo de objetos do para-nível. Quando declarados como membros de um objeto, estes métodos permitem a implementação de um conjunto arbitrário de atributos ou métodos virtuais, onde "virtual" não significa um processo polimórfico de vinculação tardia. Tais métodos são invocados a cada tentativa de acesso (leitura ou escrita respectivamente) a atributos ou métodos do mesmo para-objeto que não tenham sido definidos previamente. É importante ressaltar que estes mecanismos só são ativados para métodos ou atributos que não estejam efetivamente definidos, portanto não podem ser utilizados como interceptadores para métodos ou atributos já existentes. Para obter a funcionalidade de interceptação seria necessário aplicar uma técnica de *wrapping*. O MOP de **Guaraná** permite a implementação desses mecanismos transparentemente, sem que haja necessidade de sobrecarregar a sintaxe da linguagem hospedeira.

Existe ainda o mecanismo **__call__()**, que é uma função membro, cuja ativação depende de que a para-entidade que a contém seja utilizada como se fosse uma classe sendo instanciada. Este é um mecanismo *sui generis* cuja finalidade é propiciar a criação de um MOP em Python cuja associatividade é baseada no modelo de meta-classe. Portanto, consideramos que o mecanismo *__call__()*, ao permitir a construção de meta-classes, é uma manifestação genuína do aspecto vinculação. Uma instância de classe que suporte este método, pode ser utilizada como se a própria instância fosse uma classe. Por conseguinte, sua classe é promovida a meta-classe (classe de "entidade que se comporta como classe").

No que tange ao aspecto execução, Python disponibiliza o mecanismo **apply()** que recebe dois parâmetros: uma rotina e uma lista de argumentos, devolvendo como resultado o produto da invocação da rotina sobre a lista de parâmetros. Este recurso é aplicável tanto do para-nível quanto do meta-nível (método de meta-classe).

O MOP fundamentado em meta-classes é construído através da combinação dos quatro últimos mecanismos, nominalmente: *__getattr__()*, *__setattr__()*, *__call__()* e *apply()*. Toda meta-classe, que é do mesmo tipo que uma para-classe, deve possuir como membro o método *__call__()*. Por ocasião da construção de uma meta-instância, o método *__call__()* da meta-classe é ativado construindo o respectivo meta-objeto (ou meta-instância) de forma que já possua como membros os métodos *__setattr__()* e *__getattr__()*. Meta-objetos serão utilizadas como classes, ocupando a base da hierarquia de especialização de para-classes. Dado que uma para-classe derivada de um meta-objeto instancie para-objetos, o acesso aos atributos e membros destes para-objetos será reificado e entregue aos métodos *__setattr__()* e *__getattr__()* do respectivo meta-objeto. Este processo ocorre naturalmente por herança, uma vez que o meta-objeto é superclasse da para-classe do para-objeto. Ainda sob controle do meta-objeto após a reificação, é realizado um encapsulamento da mensagem de para-nível (seja ela acesso a um atributo ou método) em um objeto de meta-nível possuidor do método *__call__*. Este último será responsável pelo efetivo processamento de meta-nível e pela execução efetiva do correspondente em para-nível através do mecanismo *apply()*.

Existem ainda em Python mecanismos que permitem reificar a pilha de execução da computação, permitindo rastrear o caminho de execução da computação sem a utilização do MOP baseado em meta-classes. O ferramental para reificação é tão poderoso que o depurador da linguagem foi escrito na própria linguagem, sem a necessidade de utilização de módulos externos ou manipulação do interpretador. Isso foi possível através da disponibilização da API *sys.settrace*, que permite o registro de uma *callback* que sirva de **gancho reflexivo** [2]. Toda vez que um conjunto determinado de *bytecodes* é interpretado, os ganchos reflexivos são ativados. No tratamento da *callback* são reificados por default o tipo de evento (call,line,return ou exception), parâmetros específicos dependentes do evento, e o frame corrente de execução. Pela aplicação dos mecanismos reflexivos já citados sobre estes três parâmetros é possível inspecionar todo tipo de meta-informação relevante, inclusive a própria sequência de *bytecodes* que será avaliada.

Python é uma linguagem extremamente poderosa dada sua dinamicidade. Sua sintaxe clara e simplificada associada à abundância de mecanismos reflexivos conferem a linguagem um caráter convidativo a implementação e experimentação de MOPs.

[2]Na literatura de reflexão é recorrente o termo **reflective hook**, neste texto traduzido para *gancho reflexivo*. Como se pode inferir pelo contexto acima, o *gancho reflexivo* denota a capacidade do interpretador em interromper a computação de para-nível e redirecionar o fluxo de execução para o meta-nível, iniciando o processo de reificação.

Tabela A.3: Descrição das facilidades reflexivas em Python

str()	rotina que retorna uma cadeia de caracteres que descreve a semântica do para-objeto parâmetro.
repr()	retorna uma cadeia de caracteres que descreve a representação interna do objeto parâmetro, incluindo sua tipologia e identificação.
dir()	rotina que lista "alguns" [a] dos descritores das entidades contidas no espaço de nomes da para-entidade parâmetro. [a]A rotina dir() não faz uma enumeração exaustiva de todas as entidades contidas no espaço de nomes da para-entidade alvo. Lista apenas as entidades definidas explicitamente, em oposição as entidades implícitas inseridas virtualmente no mesmo espaço de nomes por ação do interpretador que não são listadas. Entretanto, estas últimas são também passíveis de reificação pelo exame do dicionário __builtin__
type()	rotina que determina a tipologia da entidade parâmetro.
__doc__	atributo de para-entidade [a] que contém documentação em tempo de execução sobre a respectiva para-entidade. [a]Em Python, classes, instâncias, métodos, módulos, código, exceções, frames de execução, todas as para-entidades são objetos. Portanto, oferecem todos os mecanismos de reificação presentes no protocolo de objetos.
__name__	atributo com descritor da para-entidade.
__class__	atributo que descreve e vincula um para-objeto a uma para-classe. Alteração no valor deste atributo permite mudar a hierarquia de generalização do para-objeto alvo.
__bases__	atributo de classe que contém os elementos da hierarquia de generalização.
__dict__	atributo de para-entidade que contém o mapeamento entre entidades e valores definidos diretamente(desconsiderando herança) no espaço de nomes da para-entidade alvo . Alterações neste atributo permite variar dinamicamente o conjunto de atributos ou métodos contidos na para-entidade alvo.
__module__	atributo que contém o descritor do módulo no qual a para-entidade alvo foi definida.
sys.modules	atributo do módulo de sistema que lista o conjunto de módulos externos importados dinamicamente no espaço de nomes da computação corrente.
sys.builtin_module_names	atributo do módulo de sistema que lista o conjunto de módulos nativos importados estaticamente independentemente da computação corrente ou de uma computação qualquer.

A.4 Sumário

Analisamos características reflexivas entre linguagens de *Scripting*.

Tcl é uma linguagem limitada, suportando somente tipos numéricos e cadeias de caracteres, que reflete essa exiguidade nas facilidades reflexivas oferecidas.

Perl é mais rica reflexivamente que Tcl, porém a ausência de suporte nativo a OO e sua sintaxe obtusa lhe tornam desinteressante como "playground" para a experimentação de MOPs.

Finalmente, Python se destaca positivamente das outras duas por quatro fatores: ser intinsecamente OO, suportar a reificação de meta-informação, suportar o registro de ganchos reflexivos, e suportar a construção de meta-classes. Dessa forma, Python torna-se extremamente convidativa a implementação e experimentação de MOPs.

Bibliografia

[1] M. Ancona, W. Cazzola, G. Dodero, and V. Gianuzzi. Channel Reification: a Reflective Approach to Fault-Tolerant Software Development. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications. OOPSLA'95 (poster section)*, page 137, Austin, Texas, USA, on 15th-19th Oct. 1995. ACM. Available at `http://www.disi.unige.it/person/CazzolaW/references.html`.

[2] L. Bergmans and M. Aksit. Composing crosscutting concerns using composition filters. *Communications of the ACM*, 44(10), October 2001.

[3] Bert Robben and Wouter Joosen and Frank Matthijs and Bart Vanhaute and Pierre Verbaeten. Building a meta-level architecture for distributed applications. Technical Report CW265, Department of Computer Science, K.U.Leuven, May 1998.

[4] S. Bijnens, W. Joosen, and P. Verbaeten. A reflective invocation scheme to realise advanced object management. In *ECOOP'93 Workshop on Reflective Object-Oriented Programming Systems*, volume 791, pages 142–154, 1994.

[5] D. Bobrow and M. Stefik. The loops manual. Technical report, Xerox Palo Alto Center, 1981. Technical Report KB-VLSI-81-13.

[6] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide.* Addison-Wesley, 1999.

[7] K. Bowen. Meta-level techniques in logic programming. In *Proceedings of the International Conference on Artificial Intelligence and its Applications*, 1986.

[8] J.-P. Briot. Actalk: A testbed for classifying and designing actor languages in the smalltalk-80 environment. In *Proceedings European Conference on Object Oriented Programming. ECOOP'89*, pages 109–129, Nottingham, July 1989.

[9] J. P. Briot and P. Cointe. The objvlisp model: Definition of a uniform reflexive and extensible object-oriented language. In *Proceedings of the European Conference on Artificial Intelligence*, 1986.

[10] W. Cazzola. Evaluation of Object-Oriented Reflective Models. In *proceedings of ECOOP Workshop on Reflective Object-Oriented Programming and Systems (EWROOPS'98)*, in 12th European Conference on Object-Oriented Programming (ECOOP'98), Brussels, Belgium, July 1998. Extended Abstract also published on ECOOP'98 Workshop Readers, S. Demeyer and J. Bosch editors, LNCS 1543, ISBN 3-540-65460-7pages 386-387.

[11] S. Chiba. A metaobject protocol for c++. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications. OOPSLA'95.*, October 1995.

[12] S. Chiba. Load-time structural reflection in Java. In *ECOOP'2000 Workshop on Reflective Object-Oriented Programming Systems*, pages 313–336, 2000.

[13] S. Chiba, G. Kiczales, and J. Lamping. Avoiding confusion in metacircularity: The meta-helix. *Lecture Notes in Computer Science*, 1049, 1996.

[14] R. Davis and D. Lenat. *Knowledge-Based Systems in Artificial Intelligence.* PhD thesis, McGraw-Hill, New York, 1982.

[15] D.-V. B. François-René Rideau. Metaprogramming and free availability of sources: Two challenges for computing today. http://www.tunes.org/ fare.

[16] Y. Futamura. Partial computation of programs. In *Proceedings of RIMS Symposia on Software, Science and Engineering*, pages 1–35, 1982. LNCS 247.

[17] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1994.

[18] M. Genesereth. Prescriptive introspection. In *Meta-Level Architectures and Reflection*, June 1987.

[19] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, Reading, Massachusetts, 1983.

[20] M. Golm. Design and implementation of a meta architecture for Java. Master's thesis, Universität Erlangen-Nürnberg, 1997.

[21] M. Golm. metaxa and the future of reflection. In *OOPSLA Workshop on Reflective Programming in C++ and Java*, Vancouver, British Columbia, October 1998.

[22] R. Greiner. Rll-1: A representation language language. Technical report, Stanford University, 1980. Heuristic Programming Project HPP-80-9.

[23] H. Ogawa, K. Shimura, S. Matsuoka, F. Maruyama, Y. Sohda and Y. Kimura. Openjit: An open-ended, reflective jit compile framework for Java. Submitted to ECOOP '2000.

[24] H. Ogawa, K. Shimura, S. Matsuoka, F. Maruyama, Y. Sohda and Y. Kimura. Openjit frontend system: an implementation of the reflective jit compiler frontend. To appear LNCS 1826.

[25] P. Hayes. *The Language GOLUX.* PhD thesis, University of Essex, 1974.

[26] Y. I. Hideaki Okamura and M. Tokoro. Al-1/d: A distributed programming system with multi-model reflection framework. In *Proceedings of the International Workshop on New Models for Software Architecture '92*, pages 36–47, 1992.

[27] U. Holzle. Integrating independently developed components in object-oriented languages. In *ECOOP'93 Workshop on Reflective Object-Oriented Programming Systems*, 1993.

[28] J.Ferber. Computational reflection in class based object-oriented languages. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications. OOPSLA'89*, pages 317–326, October 1989.

[29] W. Joosen, B. Robben, H. V. Wulpen, and P. Verbaeten. Experiences with an object-oriented parallel language: The correlate project. In *Proceedings of International Scientific Computing in Object-Oriented Parallel Environments Conference (ISCOPE'97)*, 1997.

[30] J.P.Briot and P.Cointe. Programming with explicit metaclasses in Smalltalk. In *Proceedings of ACM Object Oritented Programming, System Languages, Applications*, pages 419–431, 1989.

[31] L. H. R. Jr. Coarse-grained parallelism using metaobject protocols. Technical Report Technical Report SSL-91-61, XEROX PARC, Palo Alto, CA, 1991.

[32] G. Kickzales, J. des Rivières, and D. G. Bobrow. The art of the metaobject protocol. Technical report, Massachusetts Institute of Technology, 1991. chapter 5,6.

[33] G. Kiczales. Towards a new model of abstraction in the engineering of software. *IMSA '92 Proceedings*, 1992.

[34] G. Kiczales, T. Elrad, et al. Discussing aspects of AOP. *Communications of the ACM*, 44(10), October 2001.

[35] J. Kleinöder and M. Golm. MetaJava: An efficient run-time meta architecture for Java. In *International Workshop on Object Orientation in Operating Systems - IWOOOS'96*, Seattle, Washington, October 1996. IEEE.

[36] J. Kleinöder and M. Golm. Transparent and adaptable object replication using a reflective Java. Technical report, Universität Erlangen-Nürnberg, September 1996. IMMD IV.

[37] J. Kleinöder and M. Golm. MetaJava - a platform for adaptable operating system mechanisms. In *ECOOP 97 Workshop on Object-Orientation and Operating Systems*, Jyväskylä,Finland, June 1997. LNCS 1357.

[38] J. Laird, P. Rosenbloom, and A. Newell. Chunking in soar: The anatomy of a general learning mechanism. In *Machine Intelligence*, volume 1. Kluwer Academic Publishers, 1986.

[39] J. Lamping, G. Kiczales, L. Rodriguez, and E. Ruf. An architecture for an open compiler. In A. Yonezawa and B.C.Smith, editors, *Proc. of the Int'l Workshop on Reflection and Meta-Level Architectures*, pages 95–106, 1992.

[40] H. Lieberman. *A Preview of ACT1.* PhD thesis, Massachusetts Institute of Technology, 1981. MIT AI-Memo 625.

[41] M. L. B. Lisbôa. A new trend on the development of fault-tolerant applications: Software meta-level architectures. *Journal of the Brazilian Computer Society*, 4(2), November 1997.

[42] B. Liskov and L. Shrira. Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. In *Conference on Programming Language Design and Implementation. SIGPLAN 88.*, Atlanta, Georgia, June 1988.

[43] M. Lutz. *Programming Python.* O'Reilly and Associates Inc, 1996.

[44] P. Maes. Concepts and experiments in computational reflection. In *ACM SIGPLAN Notices*, December 1989.

[45] H. Masuhara. Efficient implementation techniques of distributed/mobile reflective objects. In *ECOOP'98 Workshop on Reflective Object-Oriented Programming and Systems*, 1998. (position paper).

[46] J. McAffer. Meta-level programming with coda. In *Proceedings of the 9th European Conference on Object-Oriented Programming ECOOP'95*, August 1995.

[47] G. Neumann and U. Zdun. Filters as a language support for design patterns in object-oritented scripting languages. In *COOTS'99. 5th Conference on Object-Oriented Technologies and Systems*, May 1999.

[48] Object Management Group, Inc. *OMG Unified Modeling Language Specification*, June 1999. version 1.3.

[49] A. Oliva. Uma arquitetura reflexiva de software para reflexão computacional. Master's thesis, Instituto de Computação, Universidade Estadual de Campinas, Setembro 1998.

[50] A. Oliva and L. E. Buzato. Composition of meta-objects in Guaraná. *Workshop on Reflective Programming in C++ and Java,OOPSLA'98*, pages 86–90, October 1998.

[51] A. Oliva and L. E. Buzato. Guaraná: A tutorial. Technical report, Instituto de Computação, Universidade Estadual de Campinas, Setembro 1998. Relatório Técnico IC-98-31.

[52] A. Oliva and L. E. Buzato. The implementation of Guaraná on Java. Technical report, Instituto de Computação, Universidade Estadual de Campinas, Setembro 1998. Relatório Técnico IC-98-32.

[53] A. Oliva and L. E. Buzato. An overview of molds: A meta-object library for distributed systems. Technical report, Instituto de Computação, Universidade Estadual de Campinas, Abril 1998. Relatório Técnico IC-98-15.

[54] A. Oliva and L. E. Buzato. The reflective architecture of Guaraná. Technical report, Instituto de Computação, Universidade Estadual de Campinas, Abril 1998. Relatório Técnico IC-98-14.

[55] A. Oliva and L. E. Buzato. The design and implementation of Guaraná. *5th Conference on Object-Oriented Technologies and Systems (USENIX)*, May 1999.

[56] A. Oliva and L. E. Buzato. Designing a secure and reconfigurable meta-object protocol. Technical report, Instituto de Computação, Universidade Estadual de Campinas, Fevereiro 1999. Relatório Técnico IC-99-08.

[57] J. K. Ousterhout. *Tcl and the Tk Toolkit.* Addison-Wesley Publishing Company, 1994.

[58] L. Prechelt. An empirical comparison of c,c++,Java,perl,rexx and tcl. Submission to IEEE Computer 2000, March 2000.

[59] E. S. Raymond. *The Cathedral and The Bazaar: Musings on Linux and Open Source by an accidental revolutionary.* O'Reilly, revised edition edition, 2001.

[60] Reflection'96. *Meta-Object Protocols for C++: The Iguana Approach*, San Frnacisco,California, April 1996.

[61] F.-R. Rideau. Métaprogrammation et libre disponibilité des sources. In *Actes de la conférence «Autour du Libre 1999»*, January 1999. http://tunes.org/~fare/articles/ll99/index.fr.html.

[62] F. Rivard. *Evolution du comportment des objects dans les langages à classes refléxifs.* PhD thesis, Université de Nantes, 1996.

[63] B. Robben. *Language Technology and Metalevel Architectures for Distributed Objects.* PhD thesis, Katholieke Universiteit Leuven, 1999.

[64] B. Robben. *Language Technology and Metalevel Architectures for Distributes Objects.* PhD thesis, Katholieke Universiteit Leuven, 1999.

[65] B. Robben, B. Vanhaute, W. Joosen, and P. Verbaeten. Non-functional policies. In *Proceedings of the Second International Conference on Metalevel Architectures and Reflection*, Saint-Malo, France, July 1999.

[66] T. S. Chiba. Designing an extensible distributed language with a meta-level architecture. In *ECOOP '93 - Proceedings of the 7th European Conference on Object-Oriented Programming*, pages 482–501, 1993.

[67] F. F. Silveira and A. M. de Alencar Price. Ferramenta de apoio ao teste de aplicações Java. Evento VI SEMANA ACADÊMICA DO PPGC, Setembro 2000.

[68] F. F. Silveira and A. M. de Alencar Price. Utilização de reflexão computacional no teste de softwares desenvolvidos em Java. Trabalho Individual I no Curso de Mestrado em Ciência da Computação, Novembro 2000.

[69] B. Smith and C. Hewitt. *A PLASMA Primer.* PhD thesis, Massachusetts Institute of Technology, 1975. Technical Report 272.

[70] B. C. Smith. *Reflection and Semantics in a Procedural Language.* PhD thesis, M.I.T, 1982. Tech. Report MIT/LCS/TR-272.

[71] S. Srinivasan. *Advanced Perl Programming.* O'Reilly and Associates Inc, 1997.

[72] G. T. Sullivan. Aspect-oriented programming using reflection and metaobject protocols. *Communications of the ACM*, 44(10), October 2001.

[73] Sun Microsystems, Java Soft, Mountain View, CA, USA. *Java Core Reflection: API and Specification*, October 1996.

[74] Sun Microsystems. *Javasoft: Java(TM) Core Reflection API and Specification*, 1997.

[75] M. Tatsubori. An extension mechanism for the Java language. Master's thesis, University of Tsukuba - Graduate School of Engineering, February 1999.

[76] M. Tatsubori. *An Extension Mechanism for the Java Language.* PhD thesis, University of Tsukuba, 1999.

[77] C. M. S. M. Tim Bray. *XML 1.0 Specification.* W3C Consortium.

[78] M. Voelter. Aspectj-oriented programming in Java. *Java Report*, January 2000.

[79] L. Wall, T. Christiansen, and R. L. Schwartz. *Programming Perl.* O'Reilly and Associates Inc, 1991.

[80] T. Watanabe and A. Yonezawa. Reflection in an object-oriented concurrent language. In *Proceedings OOPSLA '92*, pages 414–434, October 1992.

[81] I. S. Welch and R. J. Stroud. Dalang - a reflective Java extension. In *OOPSLA'98 Workshop on Reflective Programming in C++ and Java*, October 1998.

[82] I. S. Welch and R. J. Stroud. A reflective Java class loader. In *ECOOP'98 Workshop on Reflective Object-Oriented Programming Systems*, July 1998.

[83] R. Weyhrauch. Prolegomena to a theory of mechanized formal reasoning. In *Artificial Intelligence*, volume 13. North Holland, 1980.

[84] Z. Wu. Reflective Java and its applications. Technical Report APM 1971.01, ANSA Phase III, March 1997.

[85] Z. Wu and S. Schwiderski. Reflective Java. Technical Report APM 1931.01, ANSA Phase III, January 1997.

[86] Z. Wu and S. Schwiderski. Reflective Java:making Java even more flexible. Technical Report APM 1936.02, ANSA Phase III, February 1997.