Carla de Carvalho Marchioro

Persistência de Objetos baseada em Reflexão Computacional

Curitiba
2002

Carla de Carvalho Marchioro

Persistência de Objetos baseada em Reflexão Computacional

Dissertação apresentada ao Programa de Pós-Graduação em Informática Aplicada da Pontifícia Universidade Católica do Paraná, como requisito parcial para obtenção do título de Mestre em Informática Aplicada

Orientador:

Prof. Dr Alcides Calsavara

Curitiba

2002

Persistência de Objetos baseada em Reflexão Computacional

Carla de Carvalho Marchioro

Esta dissertação foi julgada adequada para a obtenção do título de **Mestre em Informática Aplicada**, área de concentração **Sistemas Distribuídos, e aprovada em sua forma final pelo Programa de Pós-Graduação em Informática Aplicada.**

Curitiba, 15 de Janeiro de 2002.

Prof. Dr. Alcides Calsavara, orientador

Prof. Dr. Ciclano de tal
coordenador do curso de Pós-Graduação em Engenharia Elétrica
da Universidade Federal de Santa Catarina.

Banca Examinadora

Prof. Dr. Fulano de tal, orientador

Prof. Dr. Beltrano de tal, co-orientador

Prof. Dr. Propano de tal

Prof. Dr. Butano de tal

Marchioro, Carla de Carvalho

Persistência de Objetos baseada em Reflexão Computacional
Curitiba, 2002. 129p.
Dissertação - Pontifícia Universidade Católica do Paraná. Programa de Pós-Graduação em Informática Aplicada.

1. Persistência 2. Reflexão Computacional 3. Máquinas Virtuais
4. Orientação a Objetos.

I.Pontifícia Universidade Católica do Paraná. Centro de Ciências Exatas e de Tecnologia. Programa de Pós-Graduação em Informática Aplicada II-t

À minha querida filha Thais...

Agradecimentos

Ao professor Alcides Calsavara pela orientação, estímulo e participação em todas as fases deste trabalho.

Ao meu marido Carlos e à minha filha Thais pelo carinho, paciência e compreensão pelas horas dedicadas a esta dissertação.

Aos meus filhos André e Gustavo que ainda não nasceram, mas já são muito amados.

Aos meus pais pela vida, formação e incentivo ao crescimento pessoal e profissional.

À Maria Rosa e Arnaldo Marchioro pelo carinho e apoio.

Aos colegas Gildo Medeiros, Carlos Kolb e Leonardo pelo companheirismo.

Aos representantes da Polo de Software e Visionnaire pela oportunidade de participar de seu programa de incentivo à educação.

Resumo

O propósito deste trabalho é definir um mecanismo para persistência de objetos em ambiente distribuído, baseado em reflexão computacional. Sua principal função é controlar o armazenamento e restauração do estado dos objetos de forma transparente.

Armazenamento de dados pode ser implementado por diversas plataformas, tais como OODBMS, ORDBMS, RDBMS e sistemas de arquivos. Devido à heterogenidade destes sistemas e à necessidade de troca de dados, mecanismos de conversão para plataformas específicas podem ser implementados.

O mecanismo de persistência será incorporado a um ambiente de execução distribuído orientado a objetos. Este ambiente é composto de máquinas virtuais cooperantes que executam em computadores conectados por uma rede sobre sistemas operacionais padrão. Ele dá suporte diretamente aos princípios de orientação a objetos, comunicação entre objetos distribuídos e reflexão computacional.

O mecanismo propõe o uso de reflexão computacional de forma diferenciada, evitando desvios entre os níveis meta e base, que são comuns em abordagens reflexivas e em geral reduzem o desempenho do sistema. Nossa proposta é entrelaçar o código objeto de nível meta ao da aplicação de forma coerente, obtendo-se um código resultante que execute sem desvios. O entrelaçamento (weave) é feito entre as árvores de programas (Abstract Syntax Tree) que representam os códigos objeto de negócio e de persistência.

Esta solução de entrelaçamento permite a construção de aplicações modulares através da separação entre atividades de sistemas e atividades funcionais. Em outras palavras, evita que a implementação de persistência interfira no código fonte do negócio. Desta forma, a persistência é introduzida somente em tempo de execução, facilitando o re-uso de funcionalidades e futuras manutenções no código.

Para validar o mecanismo proposto nós construímos algumas árvores de programa que mostram que o mecanismo está de acordo com nossos objetivos.

Abstract

The purpose of this work is to define an object persistence mechanism on a distributed environment, based on computational reflection. Its main function is to store and retrieve object states in a transparent and controlled way.

Data storage can be implemented on several platforms, such as OODBMS, ORDBMS, RDBMS and file systems. Due to the heterogeneity of these systems and the need for data exchange, platform specific data conversion mechanisms can be implemented.

The persistence mechanism is part of a distributed object-oriented execution environment composed of co-operating virtual machines that run in network computers on ordinary operating systems. It supports the principles of object-oriented programming, communication between distributed objects and computational reflection.

Differently from traditional reflective systems, the proposed mechanism applies computational reflection by avoiding interception between meta-level code and base-level code. We propose to weave the two different levels of code in the coherent way, so that the resulting code can be executed without deflection. As consequence better performance can be obtained. Weaving is applied to program trees (Abstract Syntax Trees) that represent business and persistence object codes.

The weaving approach permits to build modular applications by means of separating system and functional activities. In other words, it avoids mixing persistence and business code. Thus, persistence is only introduced at runtime, making easy to reuse functionalities and to maintain code.

In order to validate the proposed mechanism, we built several program trees that show that the mechanism accomplishes our objectives.

Sumário

Lista de Figuras

Capítulo 1

Introdução

1.1 Motivação

Muitas pesquisas têm sido realizadas com o intuito de melhorar o desenvolvimento de sistemas em ambiente distribuído. Busca-se principalmente formas de desenvolvimento com alto grau de produtividade e confiabilidade. Estas características tornaram-se fundamentais, principalmente devido ao crescimento de aplicações voltadas para Internet. Muitas das propostas apresentadas têm empregado orientação a objetos. Atualmente, diversas soluções que objetivam auxiliar na construção de sistemas distribuídos empregam orientação a objetos usando linguagens como C++ e Java. Portabilidade é obtida pela execução em ambiente baseado em máquinas virtuais e tem sido considerada um dos motivos para o sucesso de Java.

Alguns grupos de pesquisa têm trabalhado no desenvolvimento de máquinas virtuais que dão apoio a requisitos básicos de sistemas. Assim, o conceito de reflexão computacional também tem sido usado no sentido de separar as atividades de sistemas das atividades funcionais propriamente ditas. Essas pesquisas têm contribuído não só com a criação de ambientes, mas principalmente com a verificação da viabilidade de uso de reflexão computacional em sistemas distribuídos. A solução baseada em biblioteca [BRL98], oferece um alto grau de flexibilidade, ao decompor sistemas em vários componentes com interfaces claras, mas requer programadores altamente especializados e qualificados; a solução integrativa, permite que os programadores fiquem isentos dos aspectos de sistema mas tornam as aplicações dependentes do comportamento embutido na plataforma.

Uma nova tendência em pesquisas pode ser notada com o aparecimento de propostas que unem conceitos de orientação a objetos, máquinas virtuais e re-

flexão computacional. Essa abordagem tem como meta prover transparência, assim como a abordagem integrativa, e flexibilidade, tal como a abordagem baseada em bibliotecas, sem requerer programadores com conhecimentos sobre aspectos básicos do sistema de execução. Várias pesquisas têm sido realizadas buscando transparência e/ou flexibilidade. Além de outras vantagens o código referente a tarefas básicas do sistema não se mistura ao código da aplicação propriamente dita. Isso melhora a legibilidade do código, além de melhorar a produtividade durante o desenvolvimento de aplicações, visto que o programador fica isento da codificação de atividades básicas do sistema e evita erros de programação.

Essas novas propostas, necessitam aprimoramento através de trabalhos práticos e refinamentos para diminuir o grau de complexidade e viabilizar sua ampla utilização.

Como citado anteriormente, grupos de pesquisa têm trabalhado no desenvolvimento de máquinas virtuais que suportam requisitos básicos de sistemas. Com o intuito de verificar a adequação dessa tendência, selecionamos um dos requisitos considerados fundamentais a qualquer sistema: a capacidade de persistir dados. A importância da persistência de dados para aplicações, distribuídas ou não, é inquestionável. "Aplicações são construídas para servir a necessidades humanas e a maioria das atividades humanas que merecem tal investimento envolvem uso prolongado. Mesmo para jogos e entretenimento, pessoas esperam que suas preferências sejam lembradas. Em aplicações suportando planejamento, projeto ou gerenciamento, os processos humanos são prolongados por causa da dificuldade das tarefas, a necessidade de troca entre atividades, a necessidade de consultar e coletar informações"[ADJ+96]. Embora, muitos trabalhos tenham sido realizados no sentido de se obter um mecanismo de persistência, não há ainda, uma solução aplicada a um ambiente distribuído como *Virtuosi* [Cal00], que faça uso de orientação a objetos, máquinas virtuais e reflexão computacional e que ofereça este serviço de modo simples, flexível e transparente.

Persistência representa alto custo no desenvolvimento de aplicações orientadas a objetos. Atualmente o desenvolvedor ainda gasta muito tempo com essas implementações, quando deveria se preocupar apenas com definições e codificações a respeito do negócio da aplicação.

Modularidade é uma característica desejável no desenvolvimento de aplicações, porque facilita o re-uso, clareza do código fonte e torna o sistema mais flexível. As propostas atuais para persistência de objetos dificultam o desenvolvimento de aplicações modulares porque requerem chamadas a bibliotecas

ou APIs que ficam espalhadas por todo o código de negócio. Assim, o código da aplicação torna-se uma mistura de código de negócio e persistência, inviabilizando a localização de funcionalidades que possam ser reutilizadas na implementação de outras aplicações. A ilegibilidade do código fonte e falta de flexibilidade prejudicam principalmente a fase de manutenção da aplicação. Por exemplo, a troca de um banco de dados poderia requerer grande esforço do programador, que teria que revisar toda a aplicação em busca do código que interage com o meio de armazenamento.

Reflexão computacional possibilita executar alterações na persistência de objetos de forma dinâmica. As vantagens de se ter reflexão dinâmica no caso de persistência de objetos é que ela permite:

- alterar a política de persistência de um certo objeto (ou classe) em tempo de execução
- tornar persistente um objeto transiente, e vice-versa

A maior parte das abordagens reflexivas, utilizam uma forma de implementação que realiza desvios entre os níveis base e meta durante a execução. Isso tende a reduzir o desempenho do sistema. Javassist [Chi00] propõe o uso bytecodes para obter reflexão computacional sem desvio na execução. *Bytecodes* são seqüências de instruções da máquina virtual Java independentes da arquitetura computacional onde o programa executará. Porém, a independência de plataforma provida pelos *bytecodes* pode acarretar problemas de desempenho nas aplicações. O problema de desempenho pode ser contornado com o uso de compiladores *Just-in-time*, que traduzem os *bytecodes* em seqüências de instruções na linguagem de máquina nativa. Esses compiladores melhoram a execução interpretada, mas não atingem o grau de otimização oferecido por compiladores convencionais como C++, porque consomem muito tempo na compilação dos *bytecodes*. Além disso, as informações estruturais de alto nível do programa em execução são eliminadas ao traduzir o código fonte para *bytecodes*, limitando a capacidade de otimização do compilador *Just-in-time*. É possível reconstruir o código a partir dos *bytecodes*, entretanto este processo é muito lento em relação ao tempo de execução.

1.2 Proposta

Este trabalho propõe a definição de um mecanismo para persistência de objetos em ambiente distribuído, baseado em reflexão computacional. Este mecanismo é visto como um componente, no ambiente da Virtuosi [Cal00], e possui propriedades reflexivas. Por esta razão é denominado *metacomponente de persistência*.

A provisão de persistência através de um componente contribui para viabilizar a implementação de aplicações modulares, facilitando o re-uso de funcionalidades, a escrita de código legível e a flexibilidade da aplicação.

O metacomponente de persistência tem por objetivo controlar o armazenamento e restauração do estado dos objetos, de forma transparente considerando as diversas maneiras de implementar memória estável. O mecanismo de persistência será aplicado a um ambiente de execução distribuída de sistemas de software orientados a objetos. A arquitetura deste ambiente, denominada *Virtuosi* [Cal00], é composta de máquinas virtuais cooperantes que executam em computadores conectados por uma rede sobre sistemas operacionais padrão. A Virtuosi tem como características principais o suporte direto aos princípios de orientação a objetos, a comunicação entre objetos distribuídos e o uso de reflexão computacional.

O mecanismo propõe o uso de reflexão computacional de forma diferenciada, evitando desvios entre os níveis meta e base, que são comuns em abordagens reflexivas e em geral reduzem o desempenho do sistema. Nossa proposta é entrelaçar o código objeto de nível meta ao da aplicação de forma coerente, obtendo-se um código resultante que execute sem desvios. O entrelaçamento (weaving) é feito entre as árvores de programas (Abstract Syntax Tree) que representam os códigos objeto de negócio e de persistência.

A solução de entrelaçamento permite a construção de aplicações modulares através da separação entre atividades de sistemas e atividades funcionais. Em outras palavras, evita que a implementação de persistência interfira no código fonte do negócio. Desta forma, a persistência é introduzida somente em tempo de execução, facilitando o re-uso de funcionalidades e futuras manutenções no código.

O uso de reflexão sem desvios na execução também é proposta em Javassist [Chi00], que modifica *bytecodes* para obter reflexão computacional. Porém, como *bytecodes* são seqüências de instruções da máquina virtual Java independentes da arquitetura computacional onde o programa executará, podem acarretar problemas de desempenho nas aplicações. Este problema pode ser

contornado com o uso de compiladores *Just-in-time*, mas possui limitações já discutidas na seção anterior. Outra forma de solucionar estes problemas é compilar o código fonte em uma forma de representação intermediária, como proposto em *Slim Binaries* [FK97]. Esta forma de representação intermediária preserva todas as informações de alto nível do programa e viabiliza as otimizações. Este princípio é utilizado na Virtuosi para facilitar o entrelaçamento (weaving) entre atividades de sistema e atividades funcionais. A representação intermediária usada por Virtuosi e consequentemente pelo mecanismo de persistência, é denominada árvore de programa. Sua principal vantagem é ser um programa compilado que contém todas as informações semânticas e está pronto para ser executado.

O metacomponente de persistência levará em consideração um modo simples de indicar objetos e classes que deverão ter seus estados guardados em memória estável. A partir desta indicação, o estado do objeto será persistido no momento e condições apropriadas que podem ser determinados verificando-se as preferências configuradas pelo desenvolvedor.

Armazenamento de dados pode ser implementado por diversas plataformas, tais como OODBMS, ORDBMS, RDBMS e sistemas de arquivos. Muitas organizações vem adotando a tecnologia de desenvolvimento de software orientada a objetos, entretanto, algumas delas optam pelo uso de base de dados relacionais, objeto-relacional ou outra forma não orientada a objetos. Frente a esta necessidade nossa solução será flexível, permitindo a conexão de adaptadores responsáveis por converter as informações contidas nos objetos para o formato particular da plataforma adotada.

A principal contribuição do trabalho é propor um mecanismo de persistência que utiliza técnicas para aumentar o grau de modularidade das aplicações, interferindo o mínimo possível no código de negócio para facilitar o re-uso de funcionalidades e futuras manutenções.

1.3 Organização

Esta dissertação foi dividida em sete capítulos. Os três primeiros descrevem tecnologias, projetos e padrões analisados para fundamentar este trabalho. E os demais apresentam a proposta, exemplos de uso e conclusões, conforme descrito a seguir.

- Capítulo 1: Introdução - Apresenta uma visão geral da dissertação, incluindo a motivação para a pesquisa e a proposta de trabalho.

- Capítulo 2: Técnicas de Persistência de Objetos - Este capítulo é bastante extenso, por conter informações relevantes para esta pesquisa. Muitas abordagens, padrões e experiências de projetos relacionados a persistência de objetos foram desenvolvidos ao longo de vários anos. Estes conhecimentos precisavam ser absorvidos para fundamentar o mecanismo proposto. Alguns *design patterns* podem ser aplicados a soluções de persistência. A grande vantagem de adotá-los é obter modelos prontos, que apresentam soluções padrões a respeito de determinado assunto. Soluções de persistência precisam estar preparadas para interagir com a diversidade de meios de armazenamento secundário, então foram estudados alguns sistemas gerenciadores de bancos de dados, tipicamente usados no desenvolvimento de aplicações orientadas a objetos.

- Capítulo 3: Técnicas de Separação de Interesses - Reflexão Computacional e Programação Orientada a Aspectos (AOP) são técnicas usadas para separar interesses, ou seja separar atividades do sistema de atividades funcionais. São descritos conceitos, benefícios e projetos que utilizam reflexão computacional e AOP, observando principalmente sua capacidade para separar o interesse de persistência do interesse da aplicação de negócio, assim como a viabilidade de fazer isso de forma dinâmica.

- Capítulo 4: Um Ambiente Reflexivo - Virtuosi. Descreve o ambiente de execução para o qual foi projetado o mecanismo proposto neste trabalho. A leitura deste capítulo é fundamental para o entendimento da proposta.

- Capítulo 5: Mecanismo Proposto - Apresenta o mecanismo proposto para persistência de objetos, justificando a escolha de suas características, de acordo com o estudo realizado.

- Capítulo 6: Exemplos de uso - Reúne códigos e árvores de programas que exemplificam o uso do mecanismo proposto. Isto torna possível a validação do mecanismo e da dissertação.

- **Capítulo 7: Conclusões e Trabalhos Futuros - Apresenta conclusões resultantes deste trabalho, e possibilidades de prosseguir com o mesmo.**

Capítulo 2

Técnicas de Persistência de Objetos

Este capítulo é dedicado a técnicas de persistência de objetos. Ele inicia apresentando abordagens para persistência Java e uma forma trivial e bastante conhecida de implementá-la, denominada *serialização*. Em seguida descreve JDBC (*Java Database Connectivity*), o serviço de persistência padronizado pela OMG, persistência de objetos em arquiteturas de três camadas e aspectos relevantes de alguns projetos bem conhecidos na literatura: *PJama, Drastic, PerDiS* e *Arjuna*. Finalmente, apresenta dois design patterns para persistência (*Serializer* e *Memento*) e descreve sistemas de gerenciamento de dados, como meio de armazenamento secundário.

2.1 Abordagens para Persistência Java

Moss e Hosking apresentam abordagens para acrescentar persistência ao Java [MH96], explorando mais as técnicas que oferecem maiores graus de transparência e ortogonalidade. Transparência é considerada a propriedade que permite a manipulação de objetos transientes e persistentes de modo semelhante. Dificilmente a transparência é completa, além disso ela pode não ser desejada, pois muitas vezes o programador necessita um certo grau de controle. Por ortogonalidade, entende-se que persistência é uma propriedade independente do tipo. Ortogonalidade completa também não pode ser atingida e pode não ser desejada. Por exemplo, não faz sentido transferir para a memória estável algumas estruturas de dados tipicamente transientes.

As seguintes opções de modelos de persistência ortogonal são apresentadas em [MH96]:

(a) A primeira opção considera que a persistência se dá por alcançabilidade ao invés de outros meios de designar quais objetos vão persistir. Eles

observam que pode ser desejável permitir aos programadores especificarem alguns objetos que não devem persistir, alguns tipos cujas instâncias nunca devem persistir, alguma fronteira que não deve ser seguida no fechamento de um rastreamento de persistência por alcançabilidade.

(b) Esta opção de modelo aborda as vantagens e desvantagens de persistir o código (do objeto). É observado que a inclusão do código tem um grau de apelo intuitivo por ser mais completo e elegante. Entretanto, conduz a duplicação, pois o código freqüentemente continuará existindo em arquivos do sistema. Isto pode provocar o crescimento do armazenamento além do desejado. Não incluir código é mais simples, mas leva ao problema de talvez necessitar-se encontrar e carregar dinamicamente o código quando objetos são puxados do armazenamento, e o programa que está rodando não possui as classes necessárias já carregadas. Outra dificuldade é manter o controle, a trilha de nomes dos arquivos de código, e carregar código a partir de um sistema de arquivos, realizando ainda a checagem de que o código realmente corresponde aos objetos salvos.

(c) Nesta opção, é discutido se o estado de um programa em execução deve ou não ser persistente. Persistir o estado de programas e threads também pode ser interessante, mas não existe uma definição padrão de seu formato. Além disso, requer que o implementador de persistência tenha um conhecimento profundo da máquina virtual existente para suportar persistência do estado de execução. Alguns aspectos do estado de execução, tais como canais de arquivos abertos, podem não ser sensíveis ao salvamento.

(d) Esta opção discute o que é um modelo de transação. Define o que se torna persistente em que momento, quem pode ver isto e quando. Os autores apontam para a oferta de funcionalidade rica e flexível, na premissa de que para o largo uso de persistência tais capacidades serão necessárias, enquanto que outros grupos optam por projetar um sistema suportando uma extensão com novos modelos de controle de concorrência e recuperação.

A seguir são apresentadas as opções de implementação de persistência transparente definidas por [MH96]:

(a) Esta opção investiga a necessidade de modificar o compilador Java. Se a linguagem fonte é alterada, então é preciso alterar o compilador, ou fornecer um pré-processador da linguagem alterada para o Java padrão. Alterações de linguagem podem ser evitadas fazendo-se extensões em outras partes do sistema, ou tornando as facilidades requisitadas disponíveis sem extensões de linguagem, por exemplo, através de novas interfaces de biblioteca. Mesmo que alterações de linguagens sejam evitadas pode ser razoável usar um

pré-processador Java-para-Java para acrescentar persistência; isto não é tão transparente ou agradável a nível de depuração de código, mas pode reduzir o esforço em outros aspectos e/ou aumentar a portabilidade.

(b) Esta opção analisa alterações no interpretador Java e no sistema de run-time. É evidenciado que qualquer abordagem resulta no aumento da implementação, mas pode ser possível fazê-lo sem alterar o interpretador, carregando-se bibliotecas Java adicionais e providenciando que elas sejam chamadas nos locais apropriados. Entre outras limitações, essa abordagem dificulta o suporte a persistência de código e de threads.

As abordagens apresentadas por Moss e Hosking são relevantes para persistência de objetos. São fundamentadas por diversos outros trabalhos dos mesmos autores e exploram também as contribuições resultantes do projeto PJama, que será visto nas próximas subseções.

2.2 Serialização de Objetos

A serialização de objetos é uma forma simples de implementar persistência, que foi adicionada a Java desde a versão 1.1. A serialização permite salvar um objeto em uma seqüência de bytes, que mais tarde pode ser recuperada, restaurando-se novamente o objeto e seu estado [Eke00].

A serialização de objetos pode ser usada para diversos fins. Por exemplo, poderia ser usada para transferir um objeto de um computador Windows para um computador Unix conectados através de uma rede. Serialização foi adicionada a Java para suportar outras características importantes da linguagem: *RMI - Remote Method Invocation* e *Java Beans*.

Para serializar um objeto é preciso inicialmente indicá-lo como serializável, implementando-se a interface *Serializable*. Depois, criar um objeto de saída *OutputStream* e encapá-lo dentro de um objeto *OutputStreamObject*. Em seguida, usar o método *writeObject()* para serializar o objeto. Para realizar o processo contrário encapa-se o objeto *InputStream* dentro de *InputStreamObject* e chama-se o método *readObject()*.

Os exemplos a seguir foram extraídos do Tutorial Java [CWH00]. A figura 2.1 mostra um exemplo de escrita no qual obtém-se a hora atual, em milisegundos, e constrói-se um objeto Date que é serializado em um arquivo chamado *theTime*.

A figura 2.2, apresenta o processo de leitura e reconstrução de objetos, de acordo com o exemplo anterior.

Quando um objeto é serializado todos os objetos referenciados por ele tam-

```
FileOutputStream out = new FileOutputStream("theTime");
ObjectOutputStream s = new ObjectOutputStream(out);
s.writeObject("Today");
s.writeObject(new Date());
s.flush();
```

Figura 2.1: Exemplo de Serialização em Java

```
FileInputStream in = new FileInputStream("theTime");
ObjectInputStream s = new ObjectInputStream(in);
String today = (String)s.readObject();
Date date = (Date)s.readObject();
```

Figura 2.2: Exemplo de Deserialização em Java

bém são salvos. Porém, existem formas de impedir que um atributo ou objeto não seja serializado automaticamente. Uma maneira é indicar que um atributo é transiente, através da palavra-chave *transient*; outra, é utilizar a interface *Externalizable*, ao invés de *Serializable*. Esta interface acrescenta os métodos *readExternal* e *writeExternal*, que são chamados automaticamente durante a serialização e deserialização de objetos, permitindo executar operações especiais.

Serialização de objetos é uma maneira simples de salvar e recuperar o estado de objetos, porém possui algumas limitações.

Um objeto serializado (seqüência de bytes) pode ser transferido para outra máquina, armazenado em arquivo ou outro meio. Mais tarde, durante o processo de leitura, novos objetos são criados com tipos equivalentes aos originais e contendo o mesmo estado anterior, porém com identificação diferente. Então, sucessivas serializações resultam na impossibilidade de compartilhar estruturas comuns[ADJ+96].

Serializações de grandes seqüências de objetos consomem um tempo considerável, devido ao processo de tradução. Conseqüentemente, o programador precisa particionar as estruturas de dados de acordo com suas necessidades de negócio de modo que durante uma leitura, por exemplo, em que deseje ler apenas alguns objetos, não tenha que sofrer a sobrecarga de uma leitura completa que recupere objetos desnecessários.

Atkinson et. al [ADJ+96] não considera serialização um bom mecanismo para persistência por duas razões: não escala bem, porque não suporta al-

goritmos incrementais; viola o princípio de persistência independente, com a mudança de identidade e perda de sub-estrutura de compartilhamento, requerendo que os programadores tenham que superar mudanças semânticas após a serialização.

2.3 JDBC

JDBC - *Java Database Connectivity* provê uma API padrão para executar comandos SQL em Java. Permite enviar comandos SQL para qualquer banco de dados relacional [Mic97]. Ou seja, não há necessidade de escrever um programa específico para acessar banco de dados Sybase, outro para acessar banco de dados Oracle e outro para acessar banco de dados Informix. Além disso, por ser escrito na linguagem de programação Java, não é preciso escrever diferentes aplicações para rodar em plataformas distintas.

Um exemplo de aplicação usando Java e JDBC é uma página web contendo um *applet* que usa informações obtidas de um banco de dados remoto. Outro exemplo, é uma empresa que usa JDBC para conectar todos os seus empregados a um ou mais bancos de dados internos, via intranet. Isto é possível mesmo que os empregados utilizem máquinas de plataformas distintas como Windows, Unix e Macintosh.

A combinação de Java e JDBC é um sucesso por tornar a disseminação da informação fácil e econômica. As aplicações de negócio podem continuar a usar seus bancos de dados instalados e acessar facilmente as informações armazenadas em diferentes sistemas de gerenciamento de banco de dados.

De forma genérica, JDBC torna possível estabelecer uma conexão com um banco de dados, enviar comandos SQL e processar os resultados. Isto é exemplificado no fragmento de código da figura 2.3.

2.3.1 JDBC e ODBC

ODBC - *Open DataBase Connectivity*, da Microsoft é provavelmente a interface de programação mais largamente usada para acessar banco de dados relacional. Ela oferece a habilidade de conexão com quase todos os bancos de dados em quase todas as plataformas. Então, uma questão levantada é: Por que não usar ODBC através de Java? A resposta é que pode-se usar ODBC de Java, mas a melhor maneira é com a ajuda de JDBC, na forma de uma ponte JDBC-ODBC, pelas seguintes razões:

```
Connection con = DriverManager.getConnection("jdbc:odbc:wonbat",
                                             "login", "password");
Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery ("SELECT a, b, c FROM Table1");
while (rs.next())
{
    int x = getInt("a");
    String s = getString ("b");
    float f = getFloat ("c");
}
```

Figura 2.3: Exemplo JDBC

- **ODBC não é apropriada para uso direto de Java, porque usa uma interface C. Chamadas de Java para código C nativo possuem deficiências na segurança, implementação, robustez e portabilidade automática de aplicações.**
- **Uma tradução literal da API ODBC em C para a API Java não seria desejável. Por exemplo, Java não tem ponteiros, e ODBC faz uso deles.**
- **ODBC é difícil de aprender. Mistura características simples e avançadas e têm opções complexas mesmo para pesquisas simples.**
- **Quando ODBC é usada, o driver deve ser manualmente instalado em toda máquina cliente. Entretanto, quando o driver JDBC é escrito completamente em Java, o código JDBC é automaticamente instalável, portável e seguro em todas as plataformas, de rede de computadores a mainframes.**

2.3.2 Modelos de Duas e Três Camadas

A API JDBC possibilita modelos de duas e três camadas para acesso a banco de dados. No modelo de duas camadas, tipicamente um *applet* Java ou aplicação conversa diretamente com o banco de dados, como mostra a figura 2.4. Isto requer um driver JDBC capaz de se comunicar com o sistema de gerenciamento de dados específico. Os comandos SQL do usuários são enviados ao banco de dados, e os resultados retornados, independente do local onde o usuário está conectado.

No modelo de três camadas, como exemplificado na figura 2.5, as requisições do cliente são enviadas para a camada intermediária de serviços, que

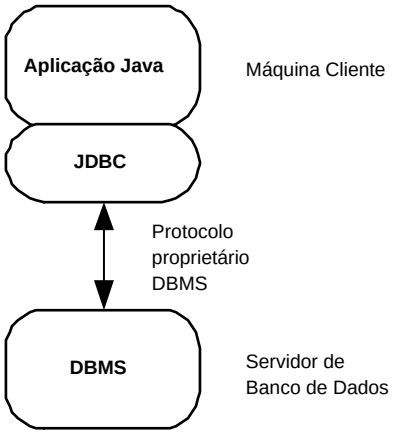

Figura 2.4: JDBC em arquitetura de 2 camadas

encaminha os comandos SQL para o banco de dados. Após o processamento, o resultado percorre o caminho inverso.

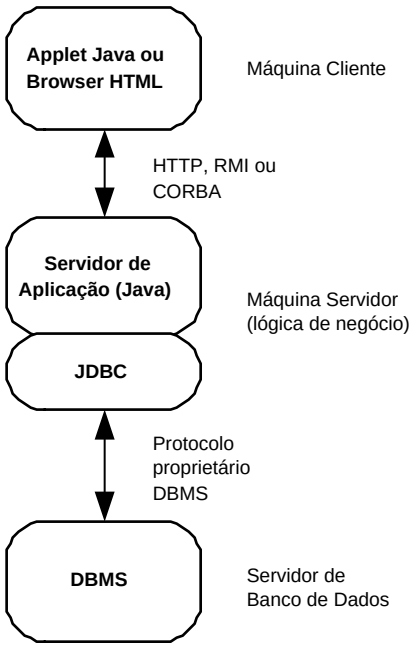

Figura 2.5: JDBC em arquitetura de 3 camadas

A grande vantagem do modelo de três camadas é manter o controle sobre os acessos e os tipos de alterações que são feitos sobre os dados da corporação.

2.4 Padrão da OMG para Persistência

Ser compatível com CORBA [1], trabalhar com o serviço de transação, ser portável, razoavelmente fácil de implementar e ser consistente com outros padrões, como SQL-3 e ODMG são as principais metas do *Persitent State Service*, padrão da OMG [2] para persistência [Gro99].

2.4.1 Conceitos

O *Persitent State Service* introduz alguns conceitos que são descritos a seguir:

Datastore é uma entidade que gerencia dados, tais como, banco de dados, um conjunto de arquivos ou um esquema em banco de dados relacional. Um *datastore* é composto de vários *storage homes*. Cada *storage home* tem um tipo e armazena dados de objetos daquele mesmo tipo. Quando armazenados estes objetos são denominados *storage objects* e possuem identificação única dentro de seu *storage home* (*short-pid*) e identificação global (*pid*), cujo escopo é seu catálogo.

Storage object incarnation é o nome dado ao objeto em memória que está associado ao objeto armazenado no *datastore*. Ele provê acesso direto a seu estado armazenado, refletindo qualquer modificação em memória para o *datastore*.

Storage home instance é a forma que a linguagem de programação provê acesso a *storage homes*

Sessão é a conexão lógica entre um processo e o *datastore* que contém o storage home com o estado do objeto de interesse.

O gerenciamento de sessão pode ser explícito ou implícito. No modo explícito, o desenvolvedor cria e gerencia a sessão. No modo implícito, o desenvolvedor cria um ou mais *pools* de sessão que realizam o gerenciamento. Os tipos de catálogo *session* e *session pools* são definidos pela especificação para este fim.

Todos estes conceitos estão representados na figura 2.6 [Gro99]:

2.4.2 Descrição do Serviço

O desenvolvimento de aplicações com *Persistent State Service* requer a especificação do tipo de objetos para armazenamento e *storage homes*. Isto pode

[1] Common Object Request Broker Architecture
[2] Object Management Group

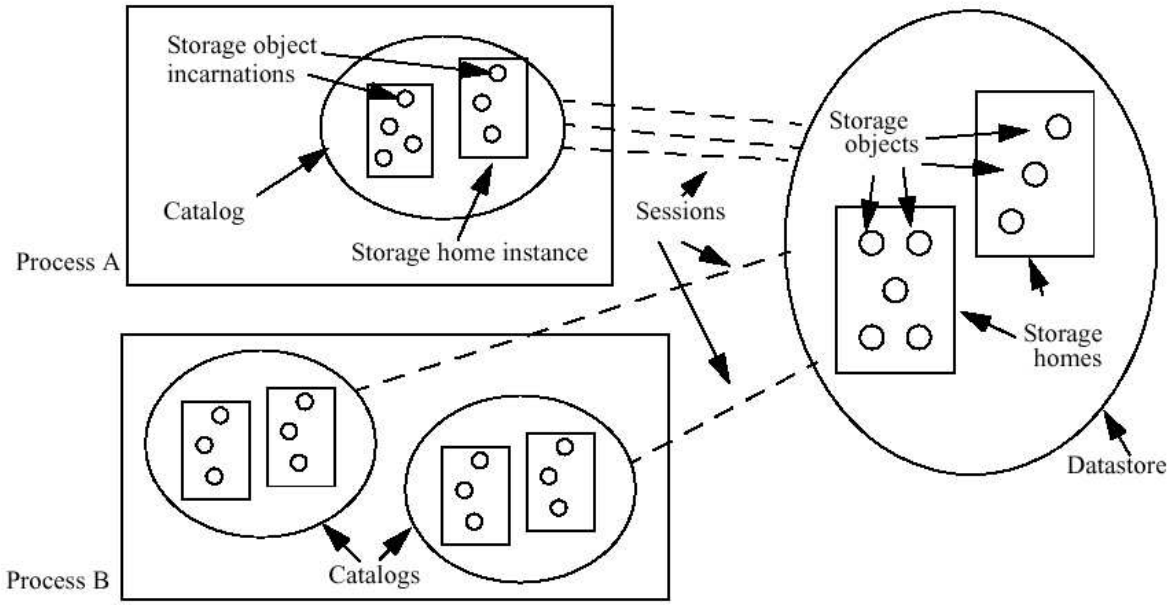

Figura 2.6: PSS - Conceitos fundamentais

ser definido usando a Linguagem de Definição de Estado Persistente (*Persistent State Definition Language - PSDL*), ou diretamente em uma linguagem de programação. A segunda maneira é conhecida como Persistência Transparente (*Transparent Persistent*). PSDL é uma extensão da IDL [3] e adiciona os seguintes novos construtores: *storagetype, storagehome, catalog, abstract storagetype* e *abstract storagehome*.

O código da figura 2.7 exemplifica definições feitas em PSDL.

O uso de PSDL requer uma ferramenta para interpretar o arquivo e gerar código correspondente na linguagem de programação. Essa ferramenta deve ser provida pelo fornecedor do *Persistent State Service*.

As mesmas definições do exemplo anterior, são demonstradas usando o modo transparente de persistência, no código da figura 2.8, escrito em Java.

O benefício mais visível da persistência transparente é que os estados dos membros podem ser diretamente representados em campos, ao invés de requerer um acessor e métodos modificadores que fazem a chamada a implementação PSS.

Uma implementação do PSS que pretende suportar persistência transparente precisa garantir que uma encarnação de um objeto é carregada antes do programa tentar acessar o estado de um de seus membros. Ele necessita também estar apto a determinar quais objetos foram modificados e precisam

[3] Interface Definition Language

```
// In file People.psdl
abstract storagetypePerson
{
    readonly state long social_security_number;
    state string full_name;
    state string phone_number;
};
abstract storagehome PersonHome of Person
{
     Person create (in long ssn, in string full_name, in string phone)
};
catalog People
{
   provides PersonHome person_home;
};
```

Figura 2.7: Definições em PSDL

ser efetivados. A especificação [Gro99] sugere algumas técnicas para prover persistência transparente para Java:

- um pré-processador Java insere código para recuperar objetos do banco de dados antes de toda leitura de campos de classes suscetíveis a persistência, e cria um código para marcar objetos sujos antes de toda escrita a estes campos;
- um compilador especial para Java realiza estes mesmos tipos de modificações;
- um pós-processador realiza modificações similares, mas para o *bytecode* que é gerado pelo compilador Java, não para código fonte;
- uma máquina virtual especial para Java usa ganchos não padronizados para recuperar e "sujar"(marcar) objetos quando eles são lidos ou modificados.

A implementação de um PSS poderá oferecer diversos modos de definir os tipos de armazenamento, inclusive ferramentas gráficas que permitem o mapeamento entre o estado de membros de objetos para colunas relacionais e *storagehomes* abstratos para tabelas relacionais.

A criação de sessões ou *pool* de sessões deve ser provida através de um conector local. O código da figura 2.9, apresentado na especificação [Gro99],

```
// Java
public interface Jperson
{
    public long socialSecurityNumber();
    public String fullName();
    public void fullName(String newName);
    public String phoneNumber();
    public void phoneNumber(String newNumber);
}
```

Figura 2.8: Definições em Java

mostra um programa Java que recupera o conector default do PSS, associado com o ORB *myOrb*, cria uma sessão, encontra um *storage home* e insere uma nova pessoa nele.

2.4.3 Transações

Objetos armazenados podem ser acessados no contexto de transações gerenciadas pelo OMG Transaction Service.

No gerenciamento explícito de sessões, uma encarnação de um objeto armazenado e uma transação são ligados através de uma sessão transacional. Na maioria das vezes, o gerenciamento de sessões e suas associações com transações é algo que o desenvolvedor não quer se preocupar. Então, alguns fornecedores oferecem mapeamento transacional de alto desempenho e *caching* baseado em gerenciamento de sessão altamente otimizados e complexos.

No gerenciamento implícito de sessões, a implementação do PSS faz todo o trabalho. Porém, o desenvolvedor não tem controle sobre associações entre transação e sessão.

O *Persistent State Service*, padrão da OMG para persistência, foi elaborado por um grupo composto por representantes de várias empresas, de reconhecida experiência no desenvolvimento de produtos de software. O fato de ser um padrão mundialmente conhecido e livremente distribuído contribui para que seja seguido pelas empresas que desenvolvem componentes de software. Porém, mesmo após alguns anos de sua conclusão, poucas empresas desenvolveram implementações do serviço e nem todas seguiram integralmente as diretrizes descritas na especificação.

```
import org.omg.*;
CORBA.ORB myOrb = CORBA.ORB.init();
CosPersistentState.ConnectorRegistry connectorRegistry
= CosPersistentState.ConnectorRegistryHelper.narrow(
       myOrb.resolve_initial_references("PSS"));

CosPersistentState.Connector connector
   = connectorRegistry.find_connector("");

//create session
CosPersistentState.Session mySession
   = connector.create_basic_session
             (org.omg.CosPersistentState.READ_WRITE, "", parameters);

// find person home
// personHome is a storage home instance)
PersonHome personHome = (PersonHome)
   mySession.find_storage_home ("PSDL:PersonHomeImpl:1.0") ;

//create person Joe Bloggs
Person joe = personHome.create (12345678, "Joe Bloggs", "(617)949-9000");
```

Figura 2.9: Exemplo Java de conexão com sessão

2.5 Persistência de Objeto Java em Arquitetura de 3 Camadas

Bretl et.al. [BOS+] aborda um padrão de arquitetura em três camadas, que consiste da apresentação e aplicação lógica no cliente, lógica da aplicação e do negócio e uma camada intermediária servidora da aplicação. O código da camada intermediária conduz pesquisas de dados, atualizações, transações e implementa a lógica de negócio comum.

A manipulação de dados executada pela aplicação é tipicamente feita em representações de objetos da terceira camada. Os dados são trazidos através de pesquisas (*queries*), ou através de APIs de manipulação de dados, ou código SQL que opera no servidor de banco de dados. Esta arquitetura está representada na figura 2.10.

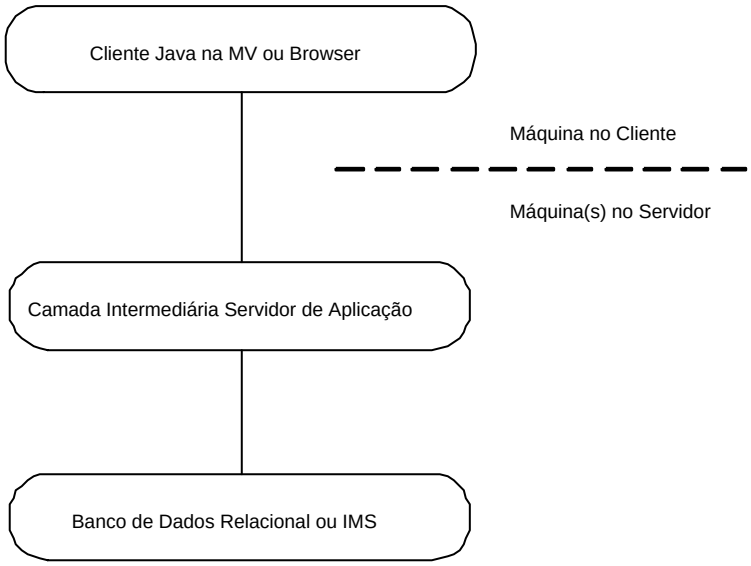

Figura 2.10: Arquitetura em 3 camadas

O servidor de aplicação na camada intermediária representa uma evolução da camada 2 em aplicações cliente-servidor. Em sistemas de duas camadas cliente-servidor a lógica de negócios também está no cliente, ou em procedimentos armazenados no servidor de base de dados relacional.

Um sistema de duas camadas geralmente requer transporte de dados em quantidade significativa para o cliente, por causa da carência de um modelo de programação completo no servidor de banco de dados. Isto conduz a problemas com o consumo de muita memória no cliente. Também pode ocorrer excessivo consumo de tempo de CPU executando software de mapeamento objeto-relacional. O servidor de aplicação na camada intermediária coordena, consolida e apresenta a visão de dados de objetos para as aplicações.

Uma aplicação escalável necessita manipular grande número de clientes e acessar grandes volumes de dados da terceira camada. Isto implica muitas máquinas virtuais Java, muitas threads por máquina virtual, e grande espaço de memória para objetos.

2.5.1 GemStone/J

GemStone/J provê um ambiente para execução da camada intermediária de aplicações Java e CORBA, de 3-camadas. Oferece carga equilibrada, corretor de acesso a processos de recursos Java, objetos persistentes Java compartilhados para modelos de dados avançados, coordenação e ocultação de dados, acesso gerenciado a 3ª camada de base de dados e serviços para desenvolvimento e gerenciamento de execução de sistemas [BOS+]. Um conjunto de

interfaces para cliente prove acesso ao servidor de aplicação de uma variedade de plataformas de cliente, oferecendo serviços do CORBA para sessões interativas e gerenciamento de transações escaláveis.

GemStone/J implementa persistência ortogonal por alcançabilidade. Qualquer objeto Java pode ser persistente, sem trabalho prévio em tempo de desenvolvimento para identificar classes de objetos que estão para ser persistentes, e sem requerer pós-processamento de classes Java cujas instâncias possam se tornar persistentes. Um objeto pode ser persistido se ele é um elemento de um objeto raiz conhecido no repositório, ou se ele é referenciado por outro objeto o qual é auto referenciado por fechamento transitivo de um objeto raiz.

2.5.2 *Enterprise JavaBeans*

J2EE - *Java2 Enterprise Edition*, é uma plataforma para desenvolvimento de aplicações corporativas. Por ser uma especificação, é independente de fornecedor. J2EE propõe uma arquitetura em camadas distribuídas, sendo que a camada de negócios fica no servidor e é implementada tipicamente por componentes EJB - *Enterprise JavaBeans*.

A arquitetura EJB é um modelo de componente para plataforma Java, cuja proposta é facilitar o desenvolvimento de aplicações distribuídas que requerem confiabilidade, escalabilidade, segurança, rapidez na construção e que suportem ambientes heterogêneos [Rot98]. EJB permite ao desenvolvedor focar somente na lógica de negócio, sem se preocupar com a implementação de comportamento transacional, segurança, *pooling* de conexões, ciclo de vida do objeto, acesso remoto e persistência porque a arquitetura delega esta responsabilidade ao fornecedor do servidor.

Benefícios da Tecnologia

Os principais benefícios que a tecnologia oferece são:

- *Produtividade*: Os desenvolvedores serão mais produtivos pelo uso da plataforma Java e foco na lógica de negócio;
- *Suporte da industria*: A tecnologia tem sido adotada em produtos de diversas empresas;
- *Independência arquitetural*: O desenvolvedor não precisa conhecer o *middleware*, apenas a plataforma Java. Isto permite aos fornecedores melhorar e modificar a camada do *middleware*, sem perturbar o usuário de aplicações EJB;

- *Uso de Java no servidor*: a facilidade de escrever uma vez e executar em qualquer lugar, agora disponível no lado servidor.

Modelos de Aplicação

Há dois modelos fundamentais para construções de aplicações. No primeiro, o cliente inicia uma sessão com um objeto que age como uma aplicação executando uma unidade de trabalho em favor do cliente, possivelmente incluindo múltiplas transações de banco de dados. No segundo modelo, o cliente acessa um objeto que representa uma entidade em um banco de dados.

Session Beans cobre o primeiro modelo. Um *session bean* é um objeto que representa uma conversação transiente com um cliente, e ele executa leitura e escrita ao banco de dados para o cliente, que pode estar no contexto de uma transação. Os campos do *session bean* contém o estado da conversação e são transientes, portanto se ocorre falha no servidor ou no cliente, o *session bean* é perdido.

Entity Beans cobre o segundo modelo. Um *entity bean* representa dados em um banco de dados paralelos aos métodos que agem nos dados. Em um contexto de banco de dados relacional, para uma tabela de informações de empregados há um *bean* para cada linha na tabela. *Entity beans* são transacionais e persistentes.

Arquitetura EJB

A figura 2.11 ilustra a arquitetura da tecnologia EJB. O servidor possibilita o uso de múltiplos protocolos tais como RMI, IIOP(CORBA) e DCOM. Isto implica que um cliente de um servidor EJB não precisa ser escrito na linguagem Java. O servidor EJB oferece uma coleção de serviços como gerenciamento de transações distribuídas, gerenciamento de objetos distribuídos, invocações distribuídas nestes objetos e serviços de sistemas de baixo nível. Um servidor EJB pode prover uma implementação de um *container*. Um *container* EJB é uma residência (home) para componentes EJB. Um *container* é onde um *bean* vive, assim como um registro vive em um banco de dados. Ele provê um ambiente seguro, escalável e transacional, no qual *beans* podem operar. Ele é o *container* que manipula o ciclo de vida do objeto, incluindo a criação e destruição de um objeto. O *container*, também manipula o gerenciamento de estado dos *beans*.

Um *container* é transparente para o cliente, não existindo API para ele. Quando um *bean* é instalado em um *container*, duas implementações são provi-

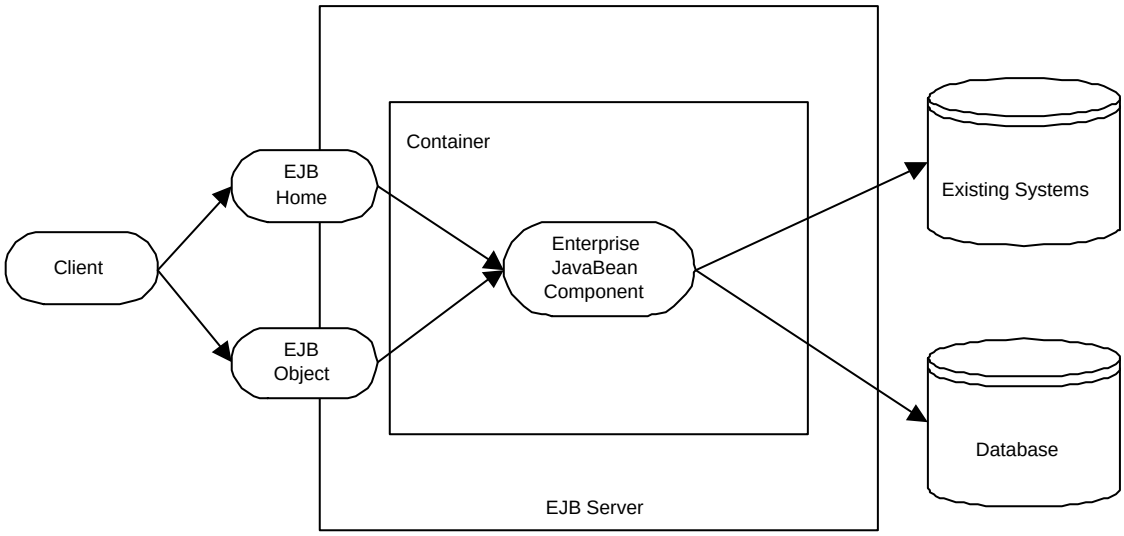

Figura 2.11: Arquitetura EJB

das: uma implementação da *interface EJBHome* e uma *interface* remota.

Para construir um *bean*, deve-se primeiro implementar métodos de negócio. Por exemplo, se está sendo escrito um *bean conta*, pode-se implementar um método débito como parte de sua *interface*. Deve-se também implementar um dos tipos de *interfaces* EJB: *SessionBean* ou *EntityBean*. Essas *interfaces* incluem métodos relacionados ao conjunto de gerenciamento de trabalho, por exemplo, e não são expostas ao cliente.

Para este fim, quando um *bean* é instalado em um servidor, a *interface* remota geralmente chamada de *skeleton* no CORBA é automaticamente gerada. A implementação da *interface* remota é chamada *EJBObject* e é um objeto que expõe somente a *interface* remota especificada pelo programador. O *EJBObject* age como um proxy, interceptando invocações remotas ao objeto e invocando os métodos apropriados na instância do *enterprise bean*.

Um *container* EJB implementa uma *interface EJBHome* para os *beans* instalados no *container*. Ele permite a criação e remoção de um *bean* e pesquisa informações ou metadados sobre um *bean*. O *container* torna as *interfaces* EJBHome disponíveis para o cliente através de JNDI [4]. Para *entity beans*, a *interface EJBHome* também contém um ou mais métodos "descobridores"que permitem a um cliente encontrar um *bean* por uma chave primária.

Uma característica chave da tecnologia EJB é seu suporte para transações distribuídas. Ela permite escrever aplicações que acessam múltiplos bancos de dados distribuídos através de diversos servidores EJB. É possível especificar o comportamento transacional de forma declarativa, em tempo de implantação.

[4]Java Naming Directory Interface

A tecnologia Enterprise JavaBeans apresenta um novo modo de desenvolver, implantar e gerenciar aplicações de negócios distribuídas. Ela torna fácil ao desenvolvedor escrever aplicações de negócios como componentes reusáveis no servidor e elimina a preocupação com programação de nível de sistema.

Persistência

Recentemente, a versão EJB 2.0 introduziu um componente para gerenciamento de persistência e relacionamento - *Container-Managed Persistence (CMP)* e *Container-Managed Relationship (CMR)* . Estas facilidades são aplicáveis a *entity beans*, que normalmente são persistentes.

O novo componente promete facilitar a implementação de persistência, que antes requeria a criação de um sistema de baixo nível baseado em JDBC, ou seja, era preciso codificar o acesso ao banco de dados *(Bean-Managed Persistence)*.

2.6 Projeto PJama

O projeto PJama [5] foi realizado em conjunto pela Universidade de Glasgow, na Escócia, e Sun Microsystem, na Califórnia, sob liderança do professor Malcolm Atkinson e Dr. Mick Jordan. Conhecido formalmente por PJava, o projeto teve diversos trabalhos publicados propondo persistência ortogonal para Java.

Persistência ortogonal [ADJ+96] é proposta como uma solução particular para prover facilidades para bancos de dados. Persistência permite aos dados ter duração de vida que varia de transiente para indefinida, e é ortogonal se a disponibilidade de durações de vida são as mesmas para todos os tipos de dados.

A linguagem Java foi escolhida por possuir características, tais como: tipagem forte, herança simples, um modelo orientado a objetos, gerenciamento automático de memória, manipulação de ponteiros não explícita e validações para melhorar segurança, precisão e produtividade, que são de interesse particular para gerenciamento de dados persistentes. O objetivo era acrescentar à Java facilidades para prover persistência de uma maneira que melhorasse a produtividade do programador de aplicações.

[5] formalmente conhecido como PJava

Persistência Ortogonal Java (PJava) foi projetada com a meta de otimizar o trabalho desses programadores de aplicação tanto na construção, quanto na manutenção de componentes. Os três princípios que conduzem à persistência ortogonal e facilitam o re-uso de classes de forma significativa são: persistência ortogonal, transitiva e independente [ADJ+96], [JA98].

Persistência ortogonal é a provisão de persistência para todos os dados independente de seu tipo, abrangendo os objetos de qualquer que seja a classe. Todos os códigos organizados em métodos de classe e todos os meta-dados descrevendo objetos são capazes de persistir.

Persistência transitiva requer que a persistência dos objetos seja determinada por seus alcances. Durante a execução normal de um programa Java objetos continuam a existir se eles estão ao alcance de alguma variável. Isto é implementado usando *garbage colection*. Ciclos de vida são terminados quando objetos não são alcançáveis por longo tempo. Isto geralmente ocorre porque a última referência a um objeto é sobrescrita.

Em PJava, deve-se identificar um objeto como *raiz persistente* para indicar que ele precisa sobreviver a execução de um programa. Todas as raízes persistentes são guardadas e todo objeto atingível via uma seqüência de operações daqueles objetos raízes são armazenados. Os espaços são recuperados mais tarde por um coletor de lixo em disco.

O terceiro princípio, persistência independente, requer que o código tenha exatamente a mesma semântica se ele está operando com dados persistentes ou transientes. Este princípio é particularmente importante para capacitar o re-uso de software.

Os três princípios descritos acima não são recentes, mas sua aplicação tornou-se possível comercialmente somente com a segurança de tipo de Java.

PJava foi implementado sem qualquer modificação na linguagem Java, nas classes do núcleo Java ou no compilador. Os mecanismos para persistência foram implementados em uma API adicional.

Atkinson et all [ADJ+96] apresenta exemplos que mostram a simplicidade de PJava. No primeiro exemplo, Figura 2.12, pode-se observar que o sistema PJava inicia implicitamente uma transação quando o método *main* é ativado. No momento da criação, as estruturas de dados *sp1* e *sp2* são transientes, sendo mais tarde *sp1* definido como uma raiz persistente. Então, quando o código for executado e atingir o fim de *main*, o *commit* será executado, tornando persistente o estado de *sp1* e de todos os objetos alcançáveis por ele. O estado de *sp2* será descartado neste ponto, porque ele não é uma raiz persistente, e nem é alcançável de algum objeto persistente. O tratamento a

exceções pode ser feito capturando-se *PJSException*. Se uma exceção que não é capturada ocorre dentro de *main*, então a transação associada com *main* é automaticamente abortada e o estado do meio de armazenamento restaurado para o estado original do início da execução de *main*.

```
public class SaveSpag
{
   public static void main (String[] args)
   {  // inicia a transação
      Spaghetti sp1 = new Spaghetti(27); // cria um novo objeto
      Spaghetti sp2 = new Spaghetti(5);  // e outro
      sp1.add ( "Pesto" );                // modifica sp1
      sp1.add ( "Pepper" );              // modifica sp1 novamente
      sp2.add ( "Quattro Formaggio" );   // modifica sp2
      try  // captura exceções de armazenamento
      {
         // obtém o objeto que representa o meio de armazenamento
         PJavaStore pjs = PjavaStore.getStore();
         pjs.newRoot ("Spag1", sp1); // cria uma raiz persistente
      }
      catch (PJSException e ) { ... } // manipula exceções
   } // fim de main
} // fim de SavePag
```

Figura 2.12: Criando uma estrutura de dados persistente

O exemplo da figura 2.13, mostra como recuperar o estado de um objeto previamente gravado no meio de armazenamento. Pelo uso do método getPRoot(), disponível na API de PJava, é possível localizar um objeto gravado como raiz através do nome que o identifica e ler o seu estado. É preciso usar um *cast* para especificar o tipo do objeto. Então, é feita uma verificação para garantir que a classe referida pelo objeto no meio de armazenamento é a mesma classe que *Spaghetti* se refere no programa.

O código da figura 2.14, exemplifica o processo de atualização de dados que já se encontram no meio de armazenamento. Algumas alterações são feitas através dos métodos *add* e *stir*. Outras são por meio do método *userArrangement*, que permite a um usuário fazer mudanças arbitrárias que modificam as estruturas de dados alcançáveis de *sp* e possivelmente adicionam, removem ou substituem objetos alcançáveis dele. Ao atingir o fim de *main*, a transação é

```
public class SpagShow
{
   public static void main (String[] args)
   {   // inicia a transação

     try // captura exceções de armazenamento
     {
         // obtém o objeto que representa o meio de armazenamento
         PJavaStore pjs = PjavaStore.getStore();
         // obtém uma raiz persistente
         Spaghetti sp = (Spaghetti)pjs.getPRoot ( "Spag1" );
         sp.display();
     }
     catch (PJSException e ) { ... } // manipula exceções
   } // fim de main
}  // fim de SpagShow
```

Figura 2.13: Usando estruturas de dados preservadas

efetivada salvando todos os objetos que tenham sido modificados e gravando todos os novos objetos que são alcançáveis destes objetos atualizados para o meio de armazenamento.

Diversos conjuntos de medidas e testes de carga mostraram desempenho razoável de PJava, e vários casos foram identificados nos quais o uso de persistência teve ganho significativo em relação a soluções não persistentes.

```
public class EditSpag
{
   public static void main (String[] args)
   {  // inicia a transação
      try  //  captura exceções de armazenamento
      {
         // obtém o objeto que representa o meio de armazenamento
         PJavaStore pjs = PJavaStore.getStore();
         // obtém uma raiz persistente
         Spaghetti sp = (Spaghetti) pjs.getPRoot ( "Spag1" );
         // modifica sp usando métodos
         sp.add ( "Garlic" );  sp.stir ( 5 );
         sp.userArrangement();  // e entrada por um usuário
     } catch ( PJSException e ) { ... } // manipula exceções
   }  // fim de main
}   // fim de EditSpag
```

Figura 2.14: Atualizando uma estrutura de dados persistente

2.7 Drastic

O objetivo deste projeto é desenvolver e demonstrar uma arquitetura de sistemas de software heterogêneos, escaláveis e distribuídos. A arquitetura é baseada em uma abstração de localidade denominada zona (*zone*), e é projetada para suportar grandes sistemas de informações integrados e evolutivos [ED97].

Drastic pretende suprir a carência das plataformas existentes, provendo gerenciamento de mudanças e evolução de aplicações críticas. Além disso, prover facilidades para gerenciamento de sistemas distribuídos como coleta de lixo, recuperação, reorganização e outros.

Um sistema pode ser decomposto em sub-domínios chamados zonas. Esta decomposição é feita com o intuito de tornar o gerenciamento mais fácil, até mesmo porque as próprias organizações tendem a ser divididas em pequenos grupos como departamentos, por exemplo. Assim, mudanças em uma zona podem ser feitas de forma autônoma, sem interferir em outras zonas onde a mudança em questão pode não fazer sentido.

Contratos compartilhados entre duas zonas são usados para forçar a

evolução de tipos, descrevendo se essas zonas podem trocar objetos de certos tipos e se objetos de certos tipos podem migrar entre essas duas zonas.

Inicialmente, Drastic implementava persistência utilizando serialização de objetos. Mais tarde, passou a suportar persistência ortogonal por alcançabilidade [ES98] proposta pelo projeto PJama, descrito na seção anterior. Assim, qualquer objeto designado como raiz persistente ou objetos alcançáveis a partir desta raiz são persistidos periodicamente de forma automática.

2.8 PerDiS - Persistent Distributed Store

O Projeto PerDiS tem como objetivo prover suporte para aplicações distribuídas, de engenharia colaborativa, em grande escala.

A engenharia colaborativa aumenta casos de sistemas relacionados ao compartilhamento de grandes volumes e de fina granularidade, com objetos complexos, através de redes distantes e fronteiras administrativas.

PerDiS provê persistência, transação, controle de concorrência, segurança e distribuição. Simplifica a programação de aplicações distribuídas porque usa a mesma abstração de memória, como em aplicações centralizadas [PF98].

2.8.1 Modelo Conceitual de PerDiS

Uma aplicação mapeia um processo distribuído, compartilhado, de memória persistente. A memória é acessada transacionalmente e é dividida em *clusters*, contendo objetos. Raízes nomeadas provêem os pontos de entrada e os objetos são conectados por ponteiros. Os objetos alcançáveis são armazenados em disco; os não alcançáveis são eliminados pelo coletor de lixo. PerDiS provê acesso direto em memória para objetos persistentes e distribuídos. E oferece controle total sobre concorrência e distribuição, que é útil a programadores experientes.

O modelo de espaço de endereçamento compartilhado é natural e fácil de usar porque provê a mesma abstração de memória como no caso centralizado. Ele facilita o compartilhamento de dados entre programas somente por nomeação, designando e removendo a referência a ponteiros. PerDiS provê a ilusão de uma memória compartilhada através da rede e do tempo, fazendo *caching* dos dados em memória de forma transparente, lendo e escrevendo em disco [PF98].

Clusters agrupam objetos que pertencem a uma mesma localidade, controle de concorrência, proteção e coleta de lixo. Os objetos devem ser alocados pela

aplicação dentro de *clusters* da memória compartilhada, de modo natural e que proporcione a melhor localização.

A figura 2.15 representa uma abstração de PerDiS.

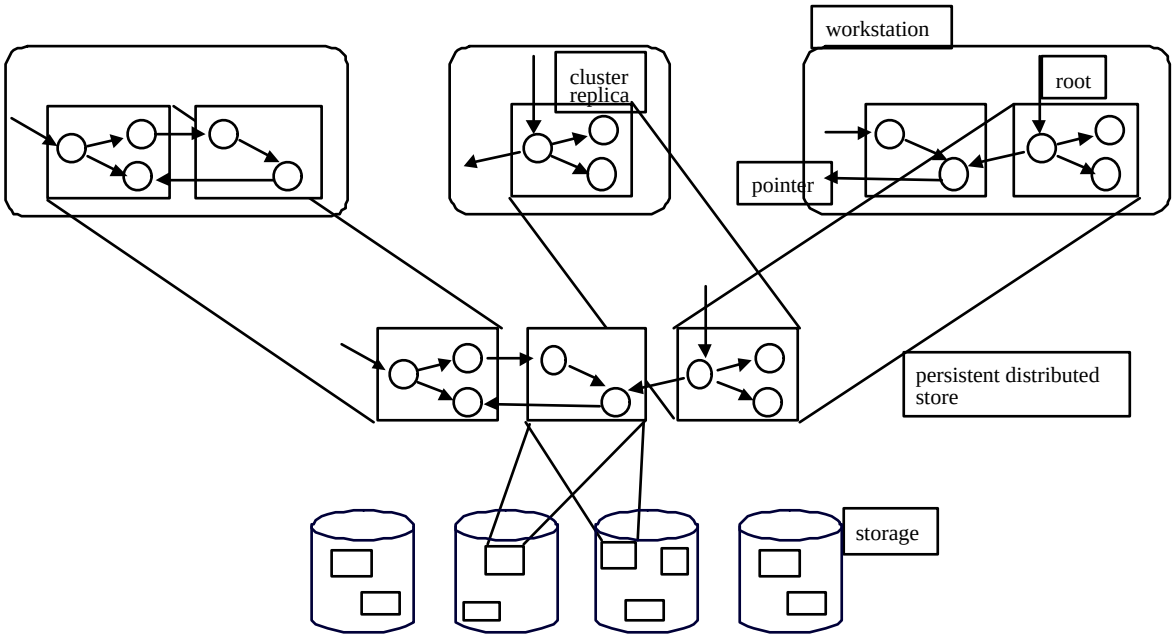

Figura 2.15: Abstrações de PerDiS

2.8.2 Estudo de Caso

PerDiS foi experimentado em aplicações CAD, mono-usuário, distribuídas fisicamente. O objetivo foi demonstrar projetos(CAD) cooperativos de construção, dentro de uma empresa virtual. Os autores definem empresa virtual como um consórcio de pequenas empresas, ou de pequenos departamentos de grandes empresas, trabalhando juntos em um projeto de construção sendo que os membros estão freqüentemente dispersos geograficamente. Um projeto para construção contém muitos objetos de fina granularidade tipicamente rodando em *megabytes*, mesmo para pequenas construções. Objetos são interconectados por ponteiros.

2.8.3 Persistência

A memória de PerDiS é persistente. Sua persistência é por alcançabilidade, ou seja, indica-se um objeto raiz que é persistente, então o processo navega de objeto a objeto, seguindo todos os ponteiros. Assim, qualquer objeto atingível através de uma raiz persistente tem seu estado persistido automaticamente.

Não importa se esta navegação envolve objetos de *clusters* distintos, todos eles são persistidos.

PerDiS usa somente serviços de arquivo local para armazenar seus dados.

2.8.4 Transação, Controle de Concorrência e *Cache*

Em PerDiS uma aplicação roda como uma seqüência de uma ou mais transações. A leitura e escrita na memória ocorre sem interferência de processos concorrentes. Transações garantem que se uma única transação atualiza múltiplos *clusters*, ou ela termina (com *commit*) e todas as atualizações são aplicadas, ou ela é abortada e nenhuma atualização é feita.

PerDiS suporta transações com a semântica transacional comum ACID [6]. Um *daemon* coloca os dados em *cache* e bloqueia o acesso por transações executando neste *site*. A implementação atual usa um *cache* seqüencial , cuja granularidade é uma página. Transações rodam no topo desse *cache* coerente. Um processo da aplicação executando uma transação obtém uma cópia privada das páginas que ele acessa. Uma aplicação pode atualizar sua cópia se sua intenção de escrita tiver sido concedida [PF98].

O controle de concorrência pode ser pessimista ou otimista. No modo pessimista o bloqueio dos dados envolvidos é feito no início da transação garantindo que, ao final da mesma, ela possa efetivar as atualizações sem problemas de concorrência. Seu desempenho é inferior ao modo otimista, que faz o bloqueio somente no momento da atualização, porém não garante a efetivação dos dados porque outro processo concorrente pode ter bloqueado o acesso aos mesmos.

2.8.5 Segurança

Cada *cluster* é armazenado em seu próprio arquivo, com acesso restrito ao *daemon*, que é protegido pelo sistema operacional.

Para proteger dados e comunicações há uma lista de controle de acesso (ACL [7]) para cada *cluster*. Um processo de uma aplicação só ganha acesso a um *cluster* se o usuário apresentar credenciais, de acordo com a lista de controle de acesso. A comunicação entre nós confiáveis é feita por um protocolo simples. Por outro lado, a comunicação entre nós não confiáveis é verificada duplamente: os dois nós verificam se o outro tem permissão para acessá-lo.

[6] Atomicidade, Consistência, Isolamento e Durabilidade
[7] Access Control List

2.9 Arjuna

Arjuna provê um conjunto de ferramentas para a construção de aplicações distribuídas, tolerantes a falhas. Suporta um modelo orientado a objetos com transações atômicas e controla operações sobre objetos persistentes [SGG91].

Objetos persistentes quando em uso são considerados ativos, caso contrário, são denominados passivos. Um objeto passivo tem seu estado armazenado em um repositório de objetos estável denominado *object store*. Quando uma invocação é feita para um objeto passivo é preciso primeiro localizá-lo, através do serviço de *binding*, que utiliza seu identificador único (UID). Uma vez localizado, a invocação pode ser feita pelo serviço de RPC [8] [LS98].

O serviço de *binding* provê um mapeamento de UIDs de objetos para sua localização (ex: nome do *host* do objeto servidor, nome do *host* do *object store*).

Para cada objeto persistente existe um nó (dito a), que quando em funcionamento, é capaz de executar um servidor para aquele objeto. O estado do objeto pode ser acessado a partir de outro nó (b), que contém seu *object store*.

Assim, o serviço de *binding* mapeia um dado UID para localizar a informação relacionada, nomeando o par <a,b>. A figura 2.16 ilustra os passos envolvidos para o caso de um objeto que é passivo: o cliente C1 apresenta o UID de um objeto para o serviço de *binding*; envia a invocação (para executar a operação op) para o nó a. Este nó, aloca um servidor S e pega o estado do objeto no *object store*. Ao término de uma aplicação, os novos estados dos objetos são migrados para seus *object stores* [LS98].

Arjuna utiliza replicação de objetos para prover tolerância a falhas. Um modelo geral para gerenciamento de objetos persistentes replicados é apresentado em [MLS93] e [LS98].

Falhas como a queda de um servidor de objeto, ou da partição de rede, podem tornar um objeto persistente indisponível temporariamente. Para melhorar a disponibilidade de um objeto ele pode ser replicado em nós distintos. No caso de consistência forte todas as réplicas disponíveis devem ser consistentes, ou seja, obrigatoriamente idênticas.

O serviço de *binding* precisa manter informações sobre réplicas tais como a lista de nós que podem executar um servidor para o objeto, e uma lista de nós que contém os *object stores* para o objeto.

[8] Remote Procedure Call

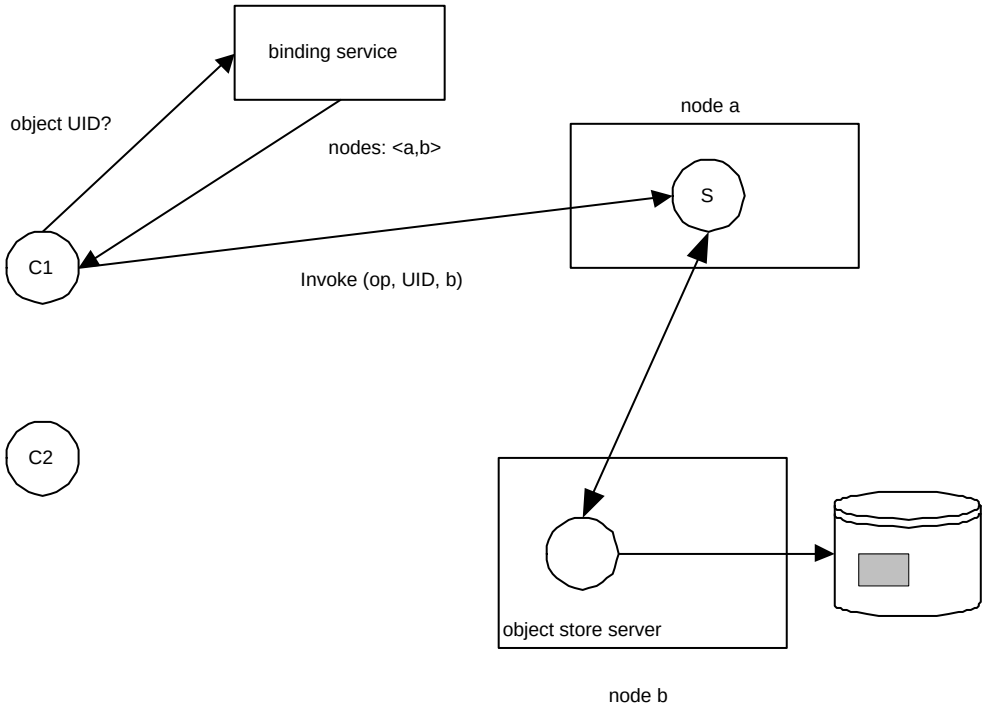

Figura 2.16: *Binding* de objeto em Arjuna

2.10 *Design Patterns*

Design Patterns são uma técnica recente de engenharia de software para resolver problemas [Cop02]. Surgiu da comunidade de orientação a objetos, cuja meta é construir um corpo de literatura para suportar projeto e desenvolvimento em geral. A técnica se tornou popular com o livro Design Patterns [GHRV95], mas tem sido usado para domínios tão diversos como organização de desenvolvimento e processo, exposição, ensino e arquitetura de software.

Um *design pattern* tem o propósito de resolver um problema capturando soluções, não somente princípios abstratos ou estratégias. Design Patterns não descrevem somente módulos, mas estruturas e mecanismos.

Os quatro elementos essenciais de um *design pattern* são [GHRV95]: o nome, o problema, a solução e as conseqüências. O *nome* do *design pattern* descreve um problema de projeto, sua solução e conseqüências, em poucas palavras. Esta nomeclatura possibilita a padronização de vocabulário comum, facilitando a comunicação entre os profissionais. O *problema* descreve um problema de projeto, seu contexto e aplicação. A *solução* apresenta os elementos que compõem o projeto, seus relacionamentos e contexto. E finalmente, as *conseqüências* são os resultados da aplicação do *design pattern*. Tanto os aspectos positivos quanto os negativos são discutidos neste item.

2.10.1 *Design Pattern Serializer*

O propósito do *design pattern Serializer* [MRB98] é padronizar a leitura e escrita de objetos independente do meio de armazenamento. Pode-se persistir objetos através de uma interface única que serve tanto para arquivo simples, quanto para banco de dados relacional, áreas para transporte de dados na rede ou outro meio de armazenamento. Este *design pattern* também é conhecido como *Atomizer*, *Streamer* e *Reader/Writer*.

Os autores apresentam um exemplo simples para ilustrar o uso do *design pattern*. Trata-se da modelagem da classe *Customer*, que possui os atributos nome e uma lista de contas. Devido a necessidade de troca de informações entre as agências do banco, os objetos *Customer* e *Account* devem ser persistentes. O banco precisa fazer uso de mais de um meio de armazenamento. Para atividades operacionais utiliza banco de dados relacional, porém para visitas em casa de clientes é mais adequado manter os dados em arquivos simples, no próprio notebook do funcionário.

Assim como no exemplo, a maioria das aplicações reais necessita utilizar mais de um meio de armazenamento e conseqüentemente, usar mais de um

formato para representar os objetos externamente. O conhecimento dessa representação externa dos objetos não deve ficar nas classes de negócio, pois torna o código difícil de entender e de manter. Para isolar o código de negócio e torná-lo independente do código de persistência, o *design pattern* propõe o uso de um protocolo genérico para leitura e outro para escrita, denominados *Reader* e *Writer*, respectivamente. As implementações particulares para cada meio de armazenamento devem ser inseridas apenas nas classes especializadas que herdam de *Reader* e *Writer*. Um exemplo dessa classe hierárquica é mostrado na figura 2.17.

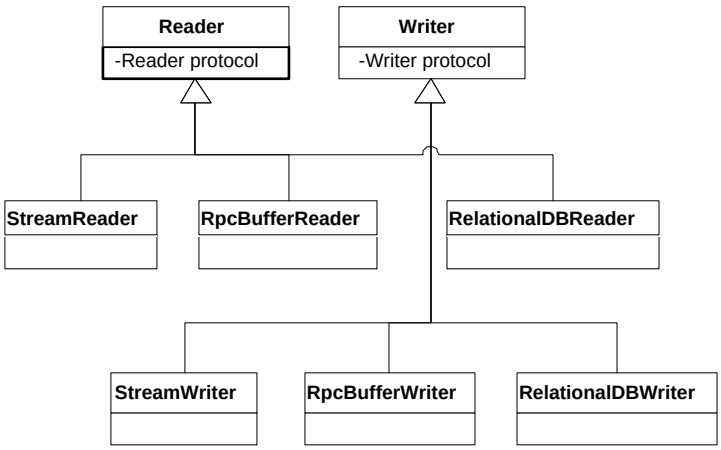

Figura 2.17: Exemplo de uma classe hierárquica *Reader/Writer*

O código de persistência também não deve conhecer as classes da aplicação, mas tem que ter acesso ao estado dos objetos. Assim, toda classe da aplicação que desejar ser persistente deve prover uma *interface* chamada *Serializable*, como no exemplo mostrado na figura 2.18. Através dos métodos *readFrom* e *writeTo* de *Serializable*, são implementadas interfaces que aceitam objetos do tipo *Reader* e *Writer*. O *design pattern Serializer* permite leitura e escrita de valores primitivos e de referências para objetos.

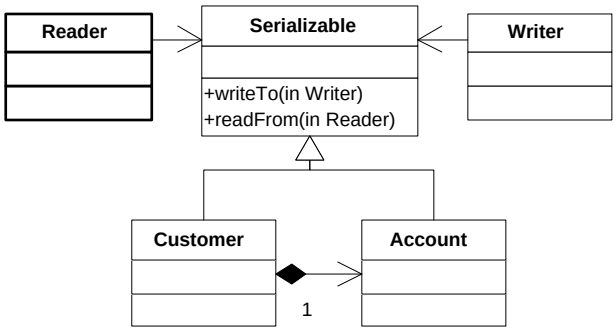

Figura 2.18: A interface *Serializable*

A figura 2.19 apresenta a estrutura completa do *design pattern Serializer*, onde pode-se observar os objetos de negócio, *ConcreteElementA* e *ConcreteElementB*, que indicam sua condição de objetos persistentes, por implementar a interface *Serializable*. O *Reader* e *Writer* oferecem métodos para leitura e escrita de tipos primitivos e referências para objetos. A implementação de protocolos específicos para cada meio de armazenamento *(Backends)* está representada pelos leitores e escritores denominados concretos (*ConcreteReaders* e *ConcreteWriters*).

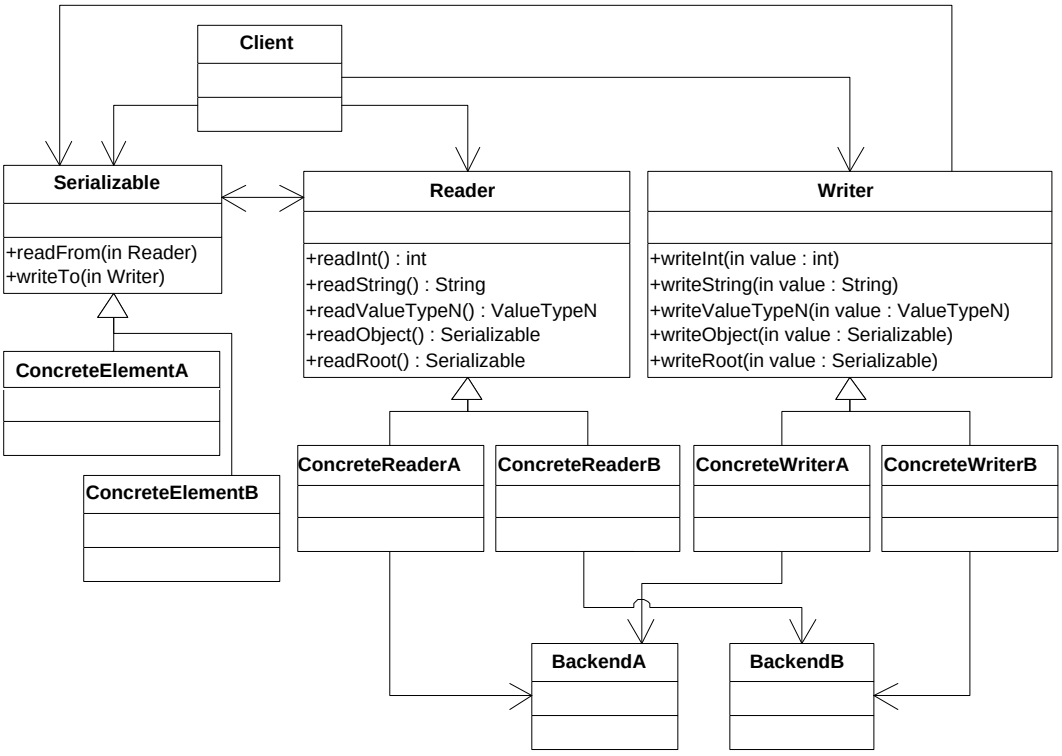

Figura 2.19: Estrutura do *design pattern Serializer*

Durante o processo de escrita cada objeto escreve seus atributos por chamar o método *write* do *Writer*. As referências para objetos são manipuladas de acordo com a política de profundidade estabelecida. Na leitura de um objeto o *Reader* cria uma nova instância da classe apropriada. Este novo objeto chama o método *read* apropriado para cada atributo, obtendo assim o estado completo do objeto vindo do meio de armazenamento.

Considerações Sobre Persistência

Um *Reader* e *Writer* pode seguir ou não as referências de objetos durante processos de leitura e escrita. Em meios de armazenamento como banco de dados, o mais adequado parece ser não seguir as referências. Esta forma

de implementação é sem dúvida mais complexa do que seguir todas as referências e persistir todo um conjunto de objetos relacionados de uma só vez, porém mostra-se mais eficiente e proporciona maior controle. Essa é uma decisão importante sobre persistência, então é abordada separadamente na seção *Políticas de Profundidade para Persistência.*

Outro fator a considerar diz respeito a referências, que em geral objetos têm para outros objetos. No meio de armazenamento essas referências podem ser representadas por identificadores, não ambíguos. Assim, cada objeto deve ser persistido com seu identificador para mais tarde ser localizado corretamente. Uma forma conhecida de implementar identificação para objeto é usar um contador global. Neste caso, a cada objeto criado é atribuído um valor do contador que é incrementado em seguida gerando o próximo identificador. Outra possibilidade é utilizar mecanismos de geração de ids oferecidos por meios de armazenamentos como sistemas de bancos de dados.

Também é fundamental considerar a manipulação de referências pendentes. Quando se lê parcialmente uma estrutura de um objeto nem todas as referências são resolvidas. As formas mais conhecidas de tratar essas referências pendentes são através do uso de *proxy* ou pela alteração no mecanismo de interpretação de referências no ambiente de execução.

Um *proxy* é um substituto para o objeto real. Pode ser um objeto do tipo correto, mas sem a inicialização de seus atributos, ou pode ser um objeto especial do tipo *proxy*. As chamadas de operações para o objeto devem ser interceptadas e remetidas ao responsável pela leitura do objeto real, antes de executar a operação.

No caso de alteração no mecanismo de interpretação de referências, pode-se substituir ou modificar o sistema de execução ou o compilador para interpretar as referências de um modo avançado, que verifique se a área apontada corresponde a um objeto válido ou não. Se for inválido o valor deve ser tratado para prover um identiticador de banco de dados ou alguma associação para o objeto real. Esta solução só deve ser adotada quando o uso de *proxies* for inadequado por questões de desempenho.

Durante o processo de escrita é importante analisar quais informações podem ser adicionadas junto ao estado dos objetos que possam ser úteis e até mesmo otimizar o processo de leitura posterior. Por exemplo, se for escrito apenas o valor dos atributos no meio de armazenamento a posterior leitura pode ficar comprometida, tendo que ser feita sempre de forma seqüencial. O ideal seria acrescentar o nome do atributo para tornar viável a localização rápida e precisa da informação que se deseja. Além disso, alguns meios de

armazenamento, como banco de dados relacional, requerem nomes de colunas para salvar e recuperar informações das tabelas, o que poderia ser resolvido com uma associação entre o nome do atributo e o nome da coluna na tabela. Outra informação que poderia ser adicionada para facilitar a implementação seria um *tag* indicando se o atributo é persistente ou transiente. Ainda, pensando em prover ao sistema um suporte mínimo para evolução, poderia-se escrever um número de versão associado a cada objeto.

Um gerenciador de objetos pode desempenhar o importante papel de controlar os objetos que já foram lidos ou escritos, evitando a ocorrência de *loops* caso hajam referências circulares entre objetos. Isto poderia ser implementado com o auxílio de tabelas que contivessem o mapeamento entre as referências de objetos já lidos/escritos e seus respectivos identificadores.

No caso de ambientes que oferecem protocolo de metaobjetos e que permitem acesso aos atributos dos objetos, é possível implementar operações de leitura e escrita diretamente em *Serializable*. A desvantagem é que em geral não é possível marcar os atributos persistentes e transientes a não ser que se esteja usando Java, que provê esses *flags*.

O uso de uma área de dados como meio de armazenamento pode tornar genérico o processo de persistência, pois separa o *Serializer* dos serviços concretos para o meio de armazenamento definitivo.

Algumas operações podem ser úteis quando adicionadas ao processo de inicialização de um objeto. Por exemplo, o tratamento a atributos transientes pode ser acrescentado logo após a leitura de um objeto.

Caso ocorra uma falha irrecuperável durante a leitura ou escrita, o processo tem que ser interrompido e todas as mudanças têm que ser desfeitas de modo que o sistema volte ao estado anterior de forma consistente. A melhor forma de resolver esta questão é tratar a leitura ou escrita de um objeto como uma transação, ou seja, o novo estado do objeto só é efetivado ao final se todas as alterações forem realizadas com sucesso, ou então, em caso de falha, todas as modificações devem ser desfeitas. O ideal seria dispor de um gerenciador de transações embutido no ambiente e apto a executar *rollback* na ocorrência de exceções.

Políticas de Profundidade para Persistência

Na política *Superficial* um objeto é persistido somente no primeiro nível, sem que nenhuma referência seja seguida. Esta política deve ser adotada quando for alto o custo de persistir mais que um objeto.

A política de *Nível de Profundidade Fixo* inicia a persistência de um objeto raiz, e segue toda referência até atingir a profundidade pré-definida. Esta solução também pode ser aplicada quando a persistência em muitos níveis for de alto custo e não houver informações sobre a estrutura de objetos que torne possível a especificação de uma estratégia melhor.

A *Parcial* ocorre a partir de um grafo que define os objetos que a serem atingidos pela persistência e quais referências devem ser deixadas pendentes. Esta é a melhor solução, porque deixa o programador mapear os requisitos específicos em relação ao comportamento da persistência.

A *Adaptativa* é uma especialização da *Parcial*, entretanto a especificação do que deve ser persistido não deriva dos requisitos de negócio, mas da especificação dinâmica determinada pelo uso atual do cliente de uma estrutura de objeto. Esta especificação dinâmica, inicia-se a partir da persistência *Superficial* e segue coletando informações sobre a freqüência com que os objetos requisitam dados de outros objetos referenciados, até obter a especificação mais adequada. Durante a escrita da estrutura de um objeto, faz sentido escrever apenas os objetos que foram modificados. Principalmente quando o meio de armazenamento for um banco de dados. Isto pode ser provido através de *flags* ou outras técnicas para indicar que o estado de um objeto foi modificado.

2.10.2 *Design Pattern Memento*

O *design pattern Memento* [GHRV95] mostra a importância de gravar o estado interno de um objeto, sem contudo expor o seu estado e violar o seu encapsulamento.

Um editor gráfico é usado para exemplificar a necessidade de manter o estado de objetos. Quando deseja-se desfazer a movimentação de figuras, inclusive de conexões entre elas, é preciso recuperar seu posicionamento e características anteriores que poderiam estar salvas em algum meio de armazenamento.

O *design pattern* é composto por um objeto denominado *memento*, que armazena o estado interno de um outro objeto: o *originator* do *memento*. Assim, o mecanismo de persistência deve solicitar a criação de um *memento* para cada objeto a ser persistido. Este objeto, o *originator*, é que cria um *memento* com seu estado atual e somente ele pode ler ou alterar o estado do *memento*. Desta forma, o encapsulamento do objeto é preservado e seu estado não é exposto aos demais objetos da aplicação nem ao mecanismo de

persistência.

A estrutura do *design pattern* está representada na figura 2.20 e a figura 2.21 mostra um exemplo de uso, através de um diagrama de seqüência.

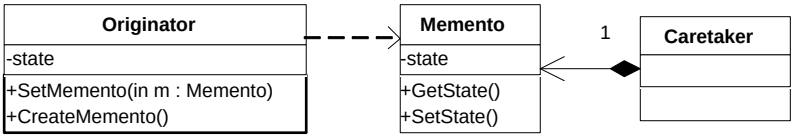

Figura 2.20: Estrutura do *design pattern Memento*

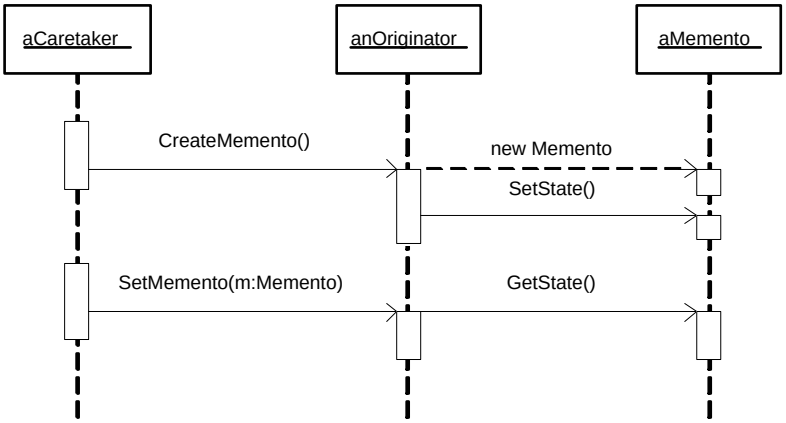

Figura 2.21: Diagrama de seqüência - *design pattern Memento*

O *caretaker* é o elemento responsável por solicitar ao objeto da aplicação (*originator*) que inicie a criação do *memento*, e salve seu estado atual. Quando ocorrer outra alteração de estado no *originator*, ou necessidade de recuperar seu estado anterior, o *caretaker* deve solicitar ao *originator* que ele interaja com o *memento* e efetive a operação. No exemplo do editor gráfico, o *caretaker* é o mecanismo de *undo*, ou de forma geral, o próprio mecanismo de persistência.

2.11 Banco de Dados

Sistemas de gerenciamento de dados são contemplados por serem os meios de armazenamento secundário mais utilizados por aplicações persistentes. Embora arquivos simples (*flat files*) possam ser considerados válidos e muitas vezes eficientes para manter dados, na maioria dos casos um sistema gerenciador de banco de dados é mais adequado principalmente para aplicações que requerem segurança, alta disponibilidade e confiabilidade nas informações. Algumas dificuldades são identificadas no uso de sistemas de gerenciamento de

dados convencionais para o desenvolvimento de aplicações orientadas a objetos. Sob este aspecto são abordados os sistemas de gerenciamento de dados relacional, orientado a objetos e objeto-relacional.

2.11.1 Necessidades das Aplicações Orientadas a Objetos

As novas aplicações estão exigindo recursos mais complexos do que os oferecidos por sistemas de gerenciamento de dados (SGBD) convencionais [KRDB91].

Os sistemas de gerenciamento de dados convencionais fornecem somente um pequeno conjunto de tipos de dados, como *integer, real, string* e estrutura de dados como *registros* e *relações*. Estes tipos de dados e estruturas têm sido inadequados para representar estruturas complexas como mapas, documentos e programas presentes nas novas aplicações. Outro problema é que eles provêem somente um repertório fixo de operações para manipulação de registros ou relações e um conjunto fixo de métodos de acesso e estratégias de processamento para implementação eficiente dessas operações.

Por outro lado, as linguagens de programação convencionais modernas são inadequadas para aplicações que requerem acesso compartilhado aos dados, pois provêem somente arquivos para armazenar e recuperar objetos persistentes. Antes de um objeto ser manipulado, ele deve ser recuperado do arquivo e traduzido para um formato interno, depois ele deve ser traduzido novamente ao formato externo que se ajuste ao armazenamento secundário para então ser armazenado no arquivo.

A solução convencional para suportar o desenvolvimento de aplicações, tem sido embutir uma linguagem de pesquisa em banco de dados na linguagem de programação, como SQL embutido em PL/1 ou C. A tradução pode ser ineficiente, pois cada operação complexa pode requerer diversas operações de banco de dados, as quais não poderão ser otimizadas pelo sistema gerenciador de banco de dados.

As chamadas são feitas do programa para o SGBD, para armazenar dados em registros no banco de dados. Esta solução não elimina o problema da tradução. O programador tem que aprender duas linguagens diferentes. A meta do sistema gerenciador de banco de dados orientado a objetos é simplificar o desenvolvimento de aplicações, reduzindo, e até eliminando o problema da tradução.

Dayal [KRDB91] apresenta duas abordagens para o desenvolvimento de sistemas de banco de dados orientado a objetos. As duas abordagens diferem

em seus pontos de início: uma abordagem procura estender conceitos de sistemas de bancos de dados existentes com dados e abstrações procedurais; a outra abordagem enriquece as linguagens de programação existentes com persistência e compartilhamento.

2.11.2 Banco de Dados Relacional

O modelo relacional é o desenvolvimento isolado mais importante em toda a história de banco de dados. Ele é baseado em sólidos fundamentos matemáticos [Dat95].

Mapeamento Objeto-Relacional

Mapeamento objeto-relacional é o processo de transformação entre o modelo de objetos e o relacional e entre os sistemas que suportam essas abordagens. Um bom trabalho de mapeamento requer o conhecimento de similaridades e diferenças entre os dois modelos [Fus97].

Scott Ambler, apresenta as seguintes estratégias para mapeamento objeto-relacional [Amb00]:

Atributos

Um atributo de uma classe pode ser mapeado a zero ou mais colunas, em um banco de dados relacional.

Heranças

São apresentadas três estratégias para implementar estruturas de herança para banco de dados relacional:

- Usar uma entidade de dados para uma hierarquia de classes. Assim, todos os atributos das classes envolvidas, são incluídos em uma única entidade. Além disso, são acrescentados o identificador do objeto (chave primária) e um indicador de tipo para o objeto, que identifica a classe que originou aquela instância. Esta abordagem é considerada simples e facilita a pesquisa dos dados, porque reúne todas as informações em uma única tabela. Porém, devido ao acoplamento existente entre as classes envolvidas na hierarquia, um erro ocorrido em apenas uma delas afeta as demais. Esta solução tende a ocupar mais espaço de armazenamento [Amb00].

- Usar uma entidade de dados por classe concreta. Nesta abordagem, cada entidade de dados inclui os atributos da classe que ela representa e também os atributos herdados. Isto facilita a pesquisa, porque todos os dados necessários estão em uma única entidade que corresponde a uma única tabela. Porém, várias desvantagens são citadas: alterações em uma classe implicam em alterações na tabela correspondente, e em todas as tabelas de suas subclasses; se um objeto modifica seu papel (passa a ser de outro tipo), é necessário copiar seus dados na tabela apropriada e designar um novo OID para ele; é difícil suportar múltiplos papéis e ainda manter a integridade dos dados.

- Usar uma entidade de dados por classe. Neste caso, para cada classe é criada uma tabela que irá conter os mesmos atributos da classe, mais o identificador do objeto. A grande vantagem é o suporte aos conceitos de orientação a objetos, como polimorfismo. A alteração em uma classe requer modificação em apenas uma tabela, sem afetar as demais. As desvantagens são: grande quantidade de tabelas no banco de dados; a leitura e escrita dos dados, assim como as pesquisas, podem ser lentas e complexas porque requerem acesso a múltiplas tabelas [Amb00].

Relacionamentos

O relacionamento entre objetos também deve ser mapeado para o banco de dados. Isto é implementado através de chaves estrangeiras.

No relacionamento um-para-um, basta apenas uma chave estrangeira, que relaciona uma entidade a outra. Porém, no caso de relacionamento bi-direcional devem haver duas chaves estrangeiras, uma em cada entidade.

Na associação um-para-muitos, a chave estrangeira deve ser inserida na entidade do lado muitos.

Em associações muitos-para-muitos, é preciso criar uma tabela associativa, cuja finalidade é manter a associação entre duas ou mais tabelas em um banco de dados relacional. Os atributos contidos na tabela associativa devem ser a combinação das chaves das tabelas, envolvidas no relacionamento.

2.11.3 Camada de Persistência para Banco de Dados Relacional

Ambler descreve três tipos de camadas de persistência para banco de dados relacional [Amb97]. O primeiro contém código SQL [9] embutido nas classes de negócio. Esta abordagem é a mais comum e permite escrever código rapidamente, sendo viável para aplicações pequenas ou protótipos. Porém, *hard-coded* SQL em classes de negócio resultam em código difícil de manter e estender.

O segundo tipo mantém o código SQL separado em classes de dados ou em *stored procedures*. Esta abordagem é um pouco melhor que a anterior, porque encapsula o código que manipula interações *hard-coded* em um único lugar, a classe de dados. Da mesma forma, só é adequada a pequenos sistemas ou protótipos.

A terceira abordagem proposta por Ambler é uma camada de persistência robusta, que mapeia objetos para mecanismos de persistência utilizando banco de dados relacional. Com esta camada, os programadores não precisam conhecer sobre o esquema de banco de dados relacional. Segundo o autor, ela é adequada ao desenvolvimento de aplicações críticas de grande escala. Entretanto, o impacto no desempenho das aplicações é considerado uma desvantagem.

A seguir são relacionados os requisitos para uma camada de persistência ser considerada robusta [Amb97]:

- Oferecer tipos diferentes de armazenamento, tais como *flat files*, banco de dados relacional, objeto-relacional, hierárquico, de rede e base de objetos;
- Encapsular completamente o mecanismo de persistência, oferecendo os serviços através de envio de mensagens ao objeto, sem escrever qualquer código adicional;
- Direcionar ações a multi-objetos, permitindo a recuperação de diversos objetos de uma só vez, ou remoção de objetos que atendam a determinado critério;
- Suportar transações;
- Suportar extensões, permitindo tanto a adição de novas classes de negócio, quanto alterações no mecanismo de persistência;
- Utilizar identificadores únicos de objetos;

[9] Structured Query Language

- Prover cursores para recuperar objetos de forma controlada;
- Suportar *proxies* para obter informações de objetos, sem a sobrecarga que o próprio objeto poderia representar;
- Disponibilizar informações em forma de registros, e não como coleção de objetos. Isto pode ser útil para ferramentas de relatórios, por exemplo;
- Suportar várias versões de bancos de dados e/ou fornecedores;
- Suportar múltiplas conexões, permitindo que a aplicação acesse outros mecanismos de persistência;
- Prover suporte a *drives* nativos e não-nativos. No caso de banco de dados relacional, utilizar ODBC, JDBC e drives nativos suportados por fornecedores de banco de dados;
- Suportar pesquisas SQL que permitam obter resultados com maior desempenho, embora o uso de SQL no código orientado a objeto viole o encapsulamento;

2.11.4 Banco de Dados Orientado a Objetos

O manifesto escrito por Atkinson et all [ABD+89] relaciona os requisitos obrigatórios para bancos de dados orientados a objetos. Considera-se fundamental que ele seja um sistema gerenciador de banco de dados, fornecendo persistência, gerenciamento de armazenamento secundário, concorrência, recuperação e instrumento de pesquisa (*query facility*). Além disso, deve suportar características de sistemas orientados a objetos, tais como objetos complexos, identidade dos objetos, encapsulamento, tipos ou classes, herança, sobrecarga combinada com ligação tardia (late binding), capacidade de extensão e integridade computacional.

Tuplas, conjuntos, listas e *arrays*, são considerados objetos complexos. O sistema deveria prover, no mínimo, conjunto, lista e tupla. Conjuntos são importantes porque representam coleções no mundo real. Tuplas podem representar propriedades de uma entidade. Listas e arrays capturam a ordem na qual ocorrem no mundo real, e aparecem em muitas aplicações científicas onde as pessoas necessitam matrizes ou seqüência de dados.

A identidade de objetos existe há muito tempo em linguagens de programação, porém em banco de dados o conceito é mais recente. Em um modelo com identidade de objeto, um objeto tem uma vida que é independente de

seus valores. Assim, há duas noções de objetos equivalentes: dois objetos podem ser idênticos (eles são o mesmo objeto) ou eles podem ser iguais (eles têm o mesmo valor). Isto tem duas implicações: uma é o compartilhamento do objeto e a outra é a atualização do objeto [ABD+89].

A idéia de encapsulamento, em linguagens de programação veio de tipos de dados abstratos. Nesta visão, um objeto tem uma parte de interface que corresponde a especificação das operações que podem ser executadas no objeto. E outra parte, de implementação, composta de dados e procedimentos. Os dados são o estado do objeto e os procedimentos descrevem a implementação de cada operação.

O encapsulamento considerado adequado é obtido somente quando as operações são visíveis e os dados e a implementação de operações estão escondidos nos objetos.

O sistema deveria oferecer algum mecanismo para estruturar os dados, seja ele classes ou tipos. Assim, a noção clássica de esquema de base de dados é substituída por um conjunto de classes ou de tipos.

Heranças têm duas vantagens: são uma ferramenta poderosa de modelagem porque fornecem uma descrição precisa e concisa do mundo, e ajudam a fatorar especificações compartilhadas e implementações em aplicações.

A seguir são descritos, de forma resumida, dois bancos de dados orientados a objetos: Jasmine e O2.

Jasmine

Jasmine é um banco de dados orientado a objetos que age como um repositório para definir, administrar e acessar de forma concorrente, estruturas de dados complexas e grandes volumes de dados.

O acesso a banco de dados é provido por *Object Database Query Language* (ODQL), que suporta todas as capacidades comuns a linguagens orientadas a objetos modernas. Pode ser usada sozinha, ou em combinação com uma linguagem hospedeira como C ou C++. ODQL fornece armazenamento estruturado para multimídia, através de classes com definições dos tipos mais comuns como vídeo, áudio e fotografias [Com01].

Jasmine provê um conjunto completo de serviços para gerenciamento de dados como bloqueio e controle de concorrência, gerenciamento de transações, recuperação e controle de acesso.

O ambiente de desenvolvimento é fácil de usar e possibilita criar aplicações de forma visual, sem requerer experiência avançada em programação.

A arquitetura utilizada é cliente/servidor, que permite a um servidor suportar múltiplos bancos de dados. As aplicações podem executar em estações de cliente ou como aplicações *stand-alone*, ou ainda, como *plug-ins* para *web browsers*. As aplicações se comunicam com o servidor de banco de dados, o qual executa a lógica de negócio e provê o armazenamento para objetos multimídia e outros dados da aplicação [Com01].

Jasmine permite a integração com outros bancos de dados, inclusive com bancos de dados relacionais, como Oracle, Sybase e Informix.

O2

O2 é um SGBD orientado a objetos, fundamentado em pesquisas do Consórcio Altair. Ele provê um mecanismo para banco de dados orientado a objetos e interfaces para linguagens de programação, como C e C++. Também oferece uma linguagem de pesquisa, um gerador de interfaces para usuário e um ambiente de programação gráfico contendo *browser* para banco de dados e *debugger* [Man94].

Seu modelo de objetos combina tecnologia relacional e de objetos, incluindo objetos no entendimento convencional de orientação a objetos e valores estruturados. Os valores podem conter objetos e os objetos podem conter valores, e estas estruturas podem ser aprofundadas e aninhadas conforme a necessidade. Porém, valores são diferentes de objetos: estruturas de valores, enquanto similar a de objetos, não são encapsuladas; valores não têm identidade; valores são manipulados por operadores e não por métodos.

2.11.5 Banco de Dados Objeto-Relacional

Sistema Gerenciador de Banco de Dados Objeto-Relacional tenta combinar as capacidades de um SGBD orientado a objetos e um SGBD relacional. Essa combinação de características, faz com que não haja um conceito que defina exatamente o que é um SGBD Objeto-Relacional [Man94].

SGBD O-R geralmente fornece suporte completo a pesquisa relacional e adiciona objetos ou tipos de dados abstratos como conceitos de estruturas de dados adicionais, que podem ser usados para generalizar tabelas relacionais e tipos usados dentro dele.

2.12 Conclusão

As abordagens para persistência de objetos descritas neste capítulo são importantes para fundamentar nossa proposta, pois resultam de muitos anos de pesquisas e experimentos no assunto.

O estudo mostrou que persistência representa alto custo no desenvolvimento de aplicações orientadas a objetos e que atualmente o desenvolvedor ainda gasta muito tempo com essas implementações, quando deveria preocupar-se apenas com definições e codificações a respeito do negócio da aplicação.

Observou-se que todas as abordagens propõem soluções para persistência de objetos de maneira a reduzir interferências na implementação do negócio. Contudo, elas não permitem o desenvolvimento de aplicações modulares porque requerem chamadas a bibliotecas ou APIs que ficam espalhadas por todo o código de negócio. Assim, o código da aplicação torna-se uma mistura de código de negócio e persistência, inviabilizando a localização de funcionalidades que possam ser reutilizadas em outras aplicações. Além disso, a ilegibilidade do código fonte e falta de flexibilidade prejudicam principalmente a fase de manutenção da aplicação.

Capítulo 3

Técnicas de Separação de Interesses

Reflexão Computacional e Programação Orientada a Aspectos (AOP) são técnicas usadas para separar interesses, ou seja separar atividades do sistema de atividades funcionais. Neste capítulo são descritos conceitos, benefícios e projetos que utilizam reflexão computacional e AOP, observando principalmente sua capacidade para separar o interesse de persistência do interesse da aplicação de negócio, assim como a viabilidade de fazer isso de forma dinâmica.

O capítulo inicia revisando conceitos de reflexão computacional e o funcionamento básico de alguns ambientes e linguagens de destaque, tais como Guaraná, Iguana, Open C++, OpenJava, MetaXa e Javassist. Em seguida, descreve AOP e uma extensão para a linguagem de programação Java, conhecida como AspectJ, que implementa os conceitos de programação orientada a aspectos.

3.1 Reflexão Computacional

Em [Mae87] reflexão computacional é definida como a atividade executada por um sistema computacional que faz computação sobre si mesmo. Para melhorar esta definição outros conceitos são citados: sistema computacional é tido como um sistema baseado em computador, com o propósito de resolver e suportar ações sobre seu domínio. Ele incorpora estruturas internas de representação do domínio, tais como entidades e relações e um programa prescrevendo como esses dados podem ser manipulados; computação é a execução deste programa. Um sistema reflexivo incorpora as estruturas de representação dele mesmo. Isto é chamado de auto-representação.

Maes [Mae87] defende a importância da reflexão, contrariando a visão da maioria, que a via como um assunto fascinante e misterioso, mas sem im-

portância técnica. São citados alguns exemplos de aplicação: manter estatísticas de uso, manter informações para propósitos de depuração, auto-otimização, auto-modificação e auto-ativação.

As linguagens de programação daquela época não proviam suporte adequado a reflexão. Para ser considerada reflexiva, uma linguagem deve reconhecer reflexão como um conceito de programação fundamental, e prover ferramentas para manipulação de computação reflexiva. Assim, o interpretador da linguagem tem que permitir o acesso aos dados de representação do próprio sistema. Além disso, tem que garantir que a conexão casual entre esses dados e os aspectos do sistema que ele representa esteja completa.

Em uma arquitetura reflexiva um sistema computacional é visto como a combinação de uma parte objeto e uma parte reflexiva. A tarefa do objeto é resolver problemas sobre o domínio externo, enquanto que a do nível reflexivo é solucionar problemas sobre a computação do objeto.

As arquiteturas reflexivas existentes são: baseada em processo, baseada em lógica e baseada em regra. A maioria das linguagens nelas utilizadas operam por meio de um interpretador metacircular, que é uma torre infinita de interpretadores circulares. Um interpretador metacircular apresenta uma maneira fácil e completa para o requisito de conexão causal.

A complexidade da reflexão procedural é devido ao fato da auto-representação servir para duas finalidades: atender ao propósito do sistema e ao mesmo tempo auto-representá-lo. Então, ao se projetar o sistema, deve-se cuidar para que os dados para computação reflexiva sejam representativos para pensar sobre o ele, além de atender aos aspectos do negócio de forma eficiente. Esta dificuldade resultou no desenvolvimento da reflexão declarativa, na qual a auto-representação não poderia estar implementada no sistema. Neste tipo de arquitetura é mais difícil de realizar a conexão causal, sendo necessário o emprego de interpretadores mais inteligentes.

O conceito de reflexão computacional se ajusta mais naturalmente em linguagens orientadas a objetos, pois sua abstração permite que objetos estejam livres para realizar seu papel em todo o sistema, de qualquer modo que eles queiram. Assim um objeto pode executar computação sobre seu domínio, mas também sobre como ele pode realizar esta computação.

3.1.1 API de Reflexão Java

A API de Reflexão Java [CWH00] permite obter metainformações de um objeto Java em tempo de execução. É possível acessar informações sobre a definição da classe, incluindo os atributos e métodos da mesma, sem quebrar a fronteira de segurança de Java. Os elementos do sistema de execução Java são todos objetos que também podem acessar suas metainformações, permitindo assim modelos de acesso consistente.

A API mantém um metaobjeto para cada um dos principais elementos da linguagem, tais como uma classe, método e atributo (campo). Tudo isto centraliza-se em uma classe chamada *Class*. Há uma instância da classe *Class* para cada classe carregada no ambiente de execução Java. O código do usuário pode obter o metaobjeto invocando o método *getClass*, dado um objeto. Uma vez que o metaobjeto é obtido, um número de métodos são disponíveis para acessar o estado da informação em objetos.

O código do usuário pode acessar o nome e o valor de um atributo através do objeto *Field*. Similarmente o nome de um método pode ser obtido e ele pode ser invocado através do objeto *Method*. A classe *Class* suporta os métodos *getMethods*, *getMethod*, *getDeclaredMethods*, *getDeclaredFields*, *getFields* e *getField*. A API também suporta uma classe estática chamada *Array*, para os *arrays*. Não existe instância de *Class* para um *array*, mas a classe *Array* cuida da tarefa de manter as metainformações para um *array*.

A API de reflexão Java oferece entre outros serviços: informações sobre modificadores de classe, atributos, métodos, construtores, e super-classes; localização de constantes e declarações pertencentes a uma *interface*; criação de instância de uma classe cujo nome seja desconhecido até que o programa esteja em execução; consulta e alteração do valor de um atributo mesmo que o nome do atributo seja desconhecido até que o programa esteja em execução; invocação de método de um objeto, mesmo que o método seja desconhecido até que o programa esteja em execução; criação e alteração de *array* cujo tamanho e tipo de componente sejam desconhecidos até a execução do programa.

Persistência com a API de Reflexão Java

Lee e Shin [HS98] reconhecem em seu trabalho as vantagens de utilizar um mecanismo para persistência de objetos ao invés de outros sistemas tradicionais, como banco de dados relacional. Uma das vantagens citadas é a de não ter que utilizar diversas linguagens para implementar o acesso aos dados, como *DDLs - Data Definition Languages* e *DMLs - Data Manipulation Languages*.

Com o serviço de persistência, basta invocar métodos escritos na própria linguagem de programação. Entretanto, mesmo utilizando persistência de objetos, o código cliente deve prover explicitamente metainformações dos objetos que tornem possível salvar e carregar cada campo do objeto quando necessário. Este tipo de metainformação geralmente envolve tempo demais e as definições das classes sofrem mudanças frequentes, agravando ainda mais o problema. Com o uso da API de reflexão suportada por Java grande parte deste problema pode ser simplificado, o que levou os autores a descrever um projeto e implementação para persistência de objetos usando a API de reflexão.

O projeto teve como meta a persistência de objetos destinada a aplicações que requerem pequena a média quantidade de dados, em torno de 50.000 objetos de tamanho típico, através de uma *interface* de programação fácil de usar. Buscaram também prover persistência da forma mais transparente possível. Foram adotados arquivos de acesso seqüencial como estrutura de armazenamento secundário. Todos os objetos persistentes são salvos e carregados em modo batch. O sistema consiste da classe *PersistentRoot*, um *Gerente de Índice* e um *Gerente de Armazenamento de Objetos*. A classe *PersistentRoot* é vista como a *interface* do usuário para persistência de objetos. Todas as classes persistentes devem herdar da classe *PersistentRoot*. O Gerente de Índice é responsável por manter uma estrutura interna para objetos e também tratar a função de busca para programadores. O *Gerente de Armazenamento de Objetos* é invisível para o código cliente e faz o papel de conectar objetos persistentes a seu sistema de armazenamento secundário. Ele provê uma simples API que lê um bloco de dados de origem e escreve para um arquivo de armazenamento persistente.

O ciclo de vida de um objeto no ambiente *POS*, inicia-se quando um objeto é criado de uma classe persistente que herda a classe *PersistentRoot*, parecendo um objeto comum para o código cliente, no modo de ser acessado. Entretanto, o sistema executa algum trabalho extra internamente. Um objeto criado é registrado como persistente no sistema *POS* pelo construtor da classe. Um identificador único para objeto (OID) é também designado neste ponto. Enquanto um objeto persistente está sendo acessado, o código do cliente invoca um método *saveAll* para salvar todos os objetos persistentes para o *POS*. Isto é feito em chamadas ao método *save* para cada objeto registrado no POS. O método *save* tem que conhecer que parte do objeto tem que ser salva, então alguma metainformação é necessária. Isto é obtido através da API de reflexão, que permite acessar a metainformação e o valor de um campo. Assim, é possível salvar o estado corrente do objeto acessando os

campos declarados para ser *public* ou *protected*. A implementação atual da API de reflexão Java, não permite acessar campos declarados como *private*, além do nome do campo. Então, o projeto desenvolveu um mecanismo *ad hoc* para compensar esta situação. Quando um campo de um objeto tem uma referência para um objeto como parte de seu valor, a referência é convertida para um identificador lógico(OID) antes de ser salvo.

Quando uma aplicação é iniciada novamente, ou uma aplicação em execução necessita carregar objetos salvos, o código do cliente pode invocar o método *loadAll* . Esse método lê cada objeto salvo verificando a classe que originou o objeto e então cria uma nova cópia do objeto na memória. Neste ponto, todas as metainformações necessárias para criar o objeto são obtidas da API de reflexão Java. Depois de criado o objeto tem seu estado definido através de valores dos campos obtidos por invocar o método *load* do objeto. Enquanto os objetos são carregados um identificador lógico(OID) salvo como parte do valor do campo é convertido novamente para o endereço físico.

A classe *PersistentRoot* age como a *interface* do programador para objetos persistentes, sendo que as principais operações são oferecidas através dos métodos *save*, *load* e do construtor. O construtor registra objetos persistentes para gerenciar índice. Deste modo, todos os objetos persistentes são implicitamente registrados durante seu processo de instanciação. Para cada objeto persistente é designado um identificador único de objeto(OID). O método *save* salva o estado corrente de um objeto para o meio de armazenamento secundário. O estado do objeto em questão é obtido por meio da API de reflexão Java. Os valores dos atributos, sejam de tipos de dado atômico ou referências para objetos, são salvos e identificados com o uso de OID, o endereço lógico do objeto persistente. Assim todos os objetos alcançáveis de um objeto persistente são salvos.

O método *load* carrega um objeto salvo de um meio de armazenamento secundário para a memória. No momento em que um objeto persistente é carregado, a definição da classe para o objeto salvo já deveria estar no sistema de execução Java. Então, um objeto é criado para o objeto salvo e os atributos são instanciados com o estado que tinham salvo anteriormente. As referências para objetos que foram convertidas de endereços físicos para endereços lógicos (OID's), são revertidas novamente para o novo endereço físico na memória quando os objetos são carregados, assim os OID's são convertidos em referências para objetos físicos novamente.

O *Gerenciador de Indexação* é projetado com um *B+ Tree*. Cada objeto persistente é inserido em árvores e seu OID é usado como a chave para indexar

o objeto. Uma das funcionalidades do *Gerenciador de Indexação* é prover o método *search*, que procura um objeto pelo nome do atributo e seu valor, que é passado como parâmetro. Esta operação também é feita usando a API de Reflexão Java.

Algumas limitações foram encontradas devido ao mecanismo de proteção adotado por Java. A API de reflexão, do *JDK 1.1* não pode acessar os atributos e métodos declarados para ser *private*. Então, no projeto em estudo [HS98] foi utilizada uma solução *ad hoc*, que incluiu dois métodos para a classe *PersistentRoot*: *userSave* que move todos os valores de atributos *private* para um array público e *userLoad*, que restaura o estado de todos os atributos *private*. O método *userSave* é invocado antes de chamar o método *save* em um objeto persistente e o método *userLoad* é invocado diretamente após o método *load* carregar um objeto persistente do meio de armazenamento. A sugestão dos autores a este respeito é que a semântica da API de reflexão Java seja estendida para que campos private também possam ser acessados, sem que haja comprometimento do modelo de segurança de Java. Uma solução apresentada seria permitir uma definição de classe para especificar o que pode ser acessado pela API de reflexão.

3.1.2 Guaraná

Guaraná descreve uma arquitetura reflexiva que busca independência de linguagem e definição de um protocolo de nível meta, projetado para prover de forma segura, flexibilidade e capacidade de re-configuração do comportamento de objetos de nível meta. Sua implementação é feita através da modificação de uma máquina virtual Java, com código aberto [OGB98].

A reflexão computacional é obtida por processar em dois níveis bem definidos: nível funcional, também conhecido como nível de aplicação e nível de gerenciamento ou meta. Aspectos do nível de aplicação são representados como objetos no nível meta em um processo chamado reificação (*materialização*). Guaraná oferece também uma biblioteca de componentes de nível meta para construção de aplicações distribuídas.

Reificação

Para o nível meta ser capaz de refletir diversos objetos, especialmente se eles são instâncias de diferentes classes, ele deve ter informações sobre a estrutura interna dos objetos. Este objeto de nível meta deve ser capaz de encontrar os métodos implementados pelo objeto, assim como os atributos definidos por este objeto. Tal como a representação do nível de aplicação que está disponível para o nível meta, é chamada metainformação estrutural. A representação em forma de objetos de conceitos de linguagem abstrata, tais como classes e métodos, é chamada reificação.

Arquitetura e Protocolo de Nível Meta

A arquitetura básica de Guaraná (seu kernel) realiza os mecanismos básicos de: intercepção de operação e reificação; ligação dinâmica e invocação para objetos do nível meta; manutenção de metainformação estrutural.

O protocolo de nível meta do Guaraná é responsável pela comunicação e modelo de acoplamento, induzindo desenvolvedores de software a criar configurações de metaobjetos bem estruturados e adaptáveis.

Guaraná define um metaobjeto como um objeto que faz parte do nível meta, responsável por implementar parte das propriedades reflexivas de uma aplicação. Cada objeto pode ser diretamente associado com zero ou um metaobjeto, chamado metaobjeto primário daquele objeto. Sua regra é observar todas as operações endereçadas a seu objeto associado, bem como seus resultados. Esta observação é feita pela intercepção e mecanismos de reificação implementados no kernel.

Uma classe também pode ser associada a um metaobjeto primário, que irá observar todas as operações relacionadas à classe e a suas instâncias. Assim, os metaobjetos de classes e suas instâncias são independentes um do outro.

Em Guaraná objetos de nível base não podem fazer referência a seus metaobjetos. A ligação entre objetos e metaobjetos é suportada pela interceptação e reificação de operações por um mecanismo de binding dinâmico.

Composição de Metaobjetos

Guaraná permite associar múltiplos metaobjetos a um objeto da aplicação. Conseqüentemente, é preciso organizar o fluxo de operações delegadas aos metaobjetos. Isto pode ser feito através de *composers*, que permitem o agrupamento de metaobjetos que são comumente usados juntos. Estes grupos podem ser compostos mais adiante formando hierarquias de metaobjetos, recursivas e potencialmente infinitas.

O *composer* implementado por Guaraná é seqüencial. Ele organiza metaobjetos em seqüência como em uma pilha. As operações são sustentadas para metaobjetos descendendo na pilha, apesar de que os resultados são apresentados na ordem reversa aos metaobjetos que têm requisitado para inspecioná-los ou modificá-los.

3.1.3 Iguana

O projeto Iguana [GBCV96] é desenvolvido pelo Grupo de Sistemas Distribuídos, do Departamento de Ciência da Computação, no Trinity College Dublin. A meta do projeto é prover suporte para a construção de sistemas que podem ser dinamicamente customizados para receber mudanças em requisitos não funcionais. Essas mudanças são tratadas em tempo de execução. Este tipo de customização é particularmente importante no contexto de middleware, tais como gerenciamento de sistemas de bancos de dados, monitores de transação on-line, e ORBs distribuídos. Esses sistemas são caracterizados pela necessidade de suportar aplicações de usuários finais, cada uma com requisitos diferentes, simultaneamente e beneficiando usuários diferentes. Com customização dinâmica, o comportamento do sistema é adaptado para as necessidades atuais das aplicações e usuários em tempo de execução. Iguana busca este tipo de customização no uso de reflexão computacional e protocolos de metaobjetos no contexto de programação orientada a objetos. Iguana desenvolve um modelo de programação reflexiva, implementando para C++ e Java (Iguana/C++ e Iguana/J). Além disso, conduz estudos de casos rela-

cionados ao uso de reflexão para suportar customização dinâmica de sistemas de objetos distribuídos e gerenciamento de aplicações em telecomunicações.

Modelo de Programação Reflexivo

Iguana é uma linguagem dependente do modelo de programação reflexivo, projetado com suporte para customização dinâmica de sistemas de software como uma de suas principais metas [GBCV96]. A linguagem de programação Iguana estende C++ com as características de Protocolos de Metaobjetos (MOPs). Seu desenvolvimento foi motivado pela pesquisa em um sistema operacional adaptável. O uso de reflexão foi o mecanismo escolhido para implementar sistemas de componentes adaptáveis dinamicamente. A opção por C++, deve-se ao fato da linguagem suportar as complexidades de implementação de sistemas operacionais. Assim só precisariam adicionar as características reflexivas.

O MOP especifica a implementação de um modelo de objetos. Em Iguana o nível meta do programa é considerado uma implementação do modelo de objetos suportado por aquela linguagem. Assim, um MOP especifica quais objetos do nível meta são necessários para implementar um modelo de objetos. O próprio MOP pode ser instanciado, através da instanciação de cada um de seus objetos de nível meta. Isto significa que as interfaces exportadas por cada um dos metaobjetos tornam-se a interface daquele MOP.

Características chave do modelo de programação:

- Múltiplos MOPs de fina-granularidade: objetos dentro de um programa simples podem usar modelos de objetos diferentes.
- Classes e objetos de nível meta.
- Categorias de reificação: programas podem escolher quais características do modelo de objetos podem sofrer reificação;
- Declaração de MOP: Iguana define a sintaxe para definição de MOPs consistindo de classes de nível meta e reificação de declarações de categorias;
- Seleção de MOPs: Iguana define mecanismos para associar objetos da aplicação com um ou mais MOPs;
- Mecanismos para invocação de objetos do nível meta.
- Uma biblioteca de classes contendo implementações MOP.

Linguagens reflexivas em geral são interpretadas. Isto se deve ao fato de serem adequadas ao processo de reflexão, que em tempo de execução deve ser capaz de adaptar o software. Estender um interpretador para ser reflexivo requer somente a adição de suporte para expor informações do nível meta ao programa de nível base, permitindo ao programa influenciar a decisão do processo do interpretador e efetivar seu próprio comportamento através de modificações de informações do nível meta. Para acrescentar reflexão a uma linguagem compilada as informações do nível meta devem ser mantidas além do processo de compilação, além disso, o código gerado deve conter os links apropriados para as informações do nível meta que controlam seu comportamento. A geração de informações de nível meta para cada objeto do modelo não é uma boa solução, pois consome tempo de compilação e espaço de armazenamento, além de queda de desempenho. Para resolver este problema, Iguana suporta categorias de reificação e múltipla fina-granularidade de MOPs. Categorias de reificação permitem ao programador, selecionar os elementos do modelo de objetos que necessitam ser reificados. Múltipla fina-granularidade de MOPs possibilita a uma aplicação ter objetos que usam modelos de objetos distintos.

3.1.4 Open C++

Open C++ é uma variação da linguagem C++ que inclui um protocolo de metaobjetos (MOP - Metaobject protocol). Em Open C++ a implementação de uma chamada de método é feita pelo MOP [CM93]. O interesse particular pela chamada de método é devido as facilidades que ela pode suportar para implementar computação distribuída. Ou seja, programadores não especializados em técnicas de reflexão podem implementar uma chamada de método dentro de um programa para obter várias funcionalidades para comunicação remota.

A implementação de chamadas de métodos é feita através de metaobjetos. Um objeto do nível base tem seu correspondente metaobjeto no nível meta, que controla a execução de seus métodos. Devido a sobrecarga provocada pelo desvio na execução de métodos, Open C++ permite que as aplicações utilizem tanto objetos comuns, que não sofrem qualquer reflexão, quanto objetos reflexivos, que possuem um metaobjeto associado.

Protocolo de Metaobjeto

Um metaobjeto deve herdar da classe base *MetaObj*, que define dois métodos: *Meta_MethodCall (Id method, Id category, ArgPac arg, ArgPac reply)* e

Meta_HandleMethodCall (Id method, Id category, ArgPac args, ArgPac reply). O primeiro implementa uma chamada de método no nível base. Ele é invocado se um método refletido é chamado. O segundo é usado para executar um método refletido.

A figura 3.1 é apresentada em [CM93] para exemplificar os desvios controlados pelo MOP de Open C++ durante a chamada de método de um objeto reflexivo.

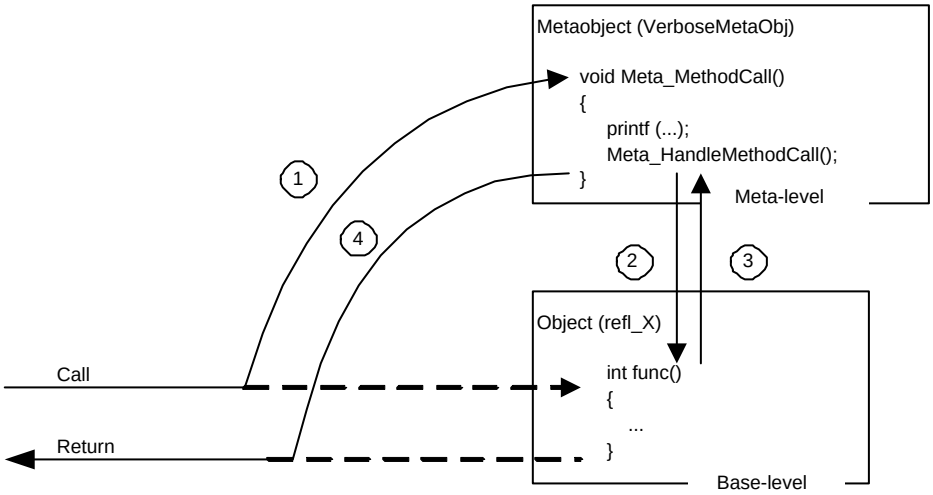

Figura 3.1: Protocolo de metaobjeto de Open C++

Processamento de Macros

Muitas experiências foram realizadas sobre Open C++ tornando seu propósito inicial mais abrangente. Recentemente Open C++ tem sido definido como um sistema de macros para C++ [Chi98b].

O uso de macros habilita a implementação de várias facilidades de linguagens, tais como controle de sentenças e abstrações de dados. Além disso, são úteis para executar expansões *inline* de uma função simples para melhorar o desempenho em tempo de execução.

Open C++ não utiliza a implementação tradicional de macros, por não ser adequada à programação orientada a objetos. Então, provê *metaobjetos* para processamento de macros.

Um programa é representado por uma coleção de metaobjetos que correspondem a uma estrutura lógica daquele programa. Nesta representação lógica, todas as ocorrências de construtores de linguagens associados a cada outro, mas espalhados pelo programa, são coletados para formar uma simples agregação. Por exemplo, uma classe metaobjeto inclui ligações para todas as

ocorrências de construtores de linguagem [1]envolvidos com a classe representada por aquele metaobjeto. Macros podem facilmente acessar e manipular aqueles construtores através de ligações de metaobjetos [Chi98b].

A representação de uma estrutura lógica provida pelo compilador de Open C++ primeiro compila um programa fonte em uma árvore, em seguida converte a árvore em uma coleção de metaobjetos. Então as macros escritas por programadores, que são chamadas de programas de nível meta, acessam e modificam esses metaobjetos. Os modificadores aplicados aos metaobjetos são finalmente refletidos no programa fonte. O compilador de Open C++ reconverte os metaobjetos para um programa fonte de acordo com as alterações e transfere este fonte para um compilador comum de C++.

3.1.5 OpenJava

OpenJava é um sistema de macros para Java que pode acessar estruturas de dados representando uma estrutura lógica de programas.

O uso de macros não é feito da forma tradicional através de árvores de sintaxe abstratas (abstract syntax trees) para representar códigos fontes, mas através de metaobjetos, que normalmente são usados em reflexão computacional. Com essa combinação, pode-se evitar as penalidades típicas de reflexão computacional, como baixo desempenho, aplicando-se tanto quanto possível as alterações em tempo de compilação, com o uso de macros [MTI00].

Em OpenJava a estrutura lógica que define cada classe do código fonte é representada por um objeto chamado metaobjeto da classe. A classe que representa o metaobjeto é chamada *metaclass*. O metaobjeto da classe também é responsável pela expansão relacionada a classe que ele representa.

OpenJava herdou muitas características de Open C++, porém a principal estrutura de dados usada em OpenC++ ainda é árvore de sintaxe abstrata.

API de OpenJava

A classe raiz para metaobjetos é *OJClass*. Os métodos membros de *OJClass* para obter informações sobre a classe são mostrados na figura 3.2 e 3.3, apresentadas em [MTI00]. Na sequência, são mostradas outras tabelas contendo os métodos restantes oferecidos pela API de OpenJava (Figuras 3.4 - 3.6).

[1]Construtores de linguagens são classes base, funções membro, membros de dados, expressões para chamar uma função membro em uma instância daquela classe, etc.

```
boolean isInterface()
    Testa se this representa um tipo de interface.
boolean array()
    Testa se this representa um tipo de array.
boolean isPrimitive()
    Testa se this representa um tipo primitivo.
OJClass getComponentType()
    Retorna uma classe metaobjeto para o tipo de componente array.
```

Figura 3.2: Métodos membros em OJClass para tipos que não são classes

```
String getPackageName()
    Retorna o nome do pacote ao qual esta classe pertence.
String getSimpleName()
    Retorna o nome desta classe de modo não qualificado.
OJModifier getModifiers()
    Retorna os modificadores para esta classe.
OJClass getSuperclass()
    Retorna a superclasse declarada explícita ou implicitamente.
OJClass[] getDeclaredInterfaces()
    Retorna todas as super-interfaces declaradas.
StatementList getInitializer()
    Retorna todas as declarações estáticas  de inicialização.
OJField[] getDeclaredFields()
    Retorna todos os campos declarados.
OJMethod[] getDeclaredMethods()
    Retorna todos os métodos declarados.
OJConstructor[] getDeclaredConstructors()
    Retorna todos os construtores declarados explícita ou implicitamente.
OJClass[] getDeclaredClasses()
    Retorna todas as classes membro (classes internas).
OJClass getDeclaringClass()
    Retorna a classe que declara esta classe (classe externa).
```

Figura 3.3: Métodos membro em OJClass para introspeção (1)

```
String setSimplename(String name)
    Atribui o nome desta classe sem qualificação.
OJModifier setModifiers(OJModifier modifs)
    Atribui modificadores da classe.
OJClass setSuperclass (OJClass class)
    Atribui a superclasse.
OJClass[] setInterface (OJClass[] faces)
    Atribui as super-interfaces para serem declaradas.
OJField removeField (OJField field)
    Remove o dado campo da declaração desta classe.
OJMethod removeMethod (OJMethod method)
    Remove o dado método da declaração desta classe.
OJConstructor removeConstructor (OJConstructor constr)
    Remove o dado construtor da declaração desta classe.
OJField addField (OJField field)
    Adiciona o dado campo à declaração desta classe.
OJMethod addMethod (OJMethod method)
    Adiciona o dado método à declaração desta classe.
OJConstructor addConstructor  (OJConstructor constr)
    Adiciona o dado construtor à declaração desta classe.
```

Figura 3.4: Métodos membro em OJClass para modificar a classe

```
String getName()
    Retorna o nome deste método.
OJModifier getModifiers()
    Retorna os modificadores para este método.
OJClass getReturnType()
    Retorna o tipo de retorno.
OJClass[] getParameterTypes()
    Retorna os tipos de parâmetros  na ordem da declaração.
OJClass[] getExceptionTypes()
    Retorna os tipos de exceções declaradas para serem lançadas.
String [] getParameterVariables()
    Retorna os nomes das variáveis de parâmetros na ordem da declaração.
StatementList getBody()
    Retorna as declarações do corpo do método.
String setName(String name)
    Atribui o nome deste método.
OJModifier setModifiers (OJModifiers modifs)
    Atribui os modificadores do método.
OJClass setReturnType()
    Atribui o tipo de retorno.
OJClass[] setParameterTypes()
    Atribui os tipos dos parâmetros na ordem de declaração.
OJClass [] setExceptionTypes()
    Atribui os tipos de exceções declaradas para serem lançadas.
String[] setParameterVariables()
     Atribui os nomes das variáveis de parâmetros na ordem da declaração.
StatementList setBody()
     Atribui as declarações do corpo do método.
```

Figura 3.5: Métodos básicos em OJMethod

```
OJClass[] getInterfaces()
     Retorna todas as interfaces implementadas por esta classe ou todas
     as super-interfaces desta interface.
boolean is AssignableFrom (OJClass class)
     Determina se esta classe/interface é a mesma que, ou é uma superclasse ou
     superinterface da dada classe/interface.

OJMethod[] getMethods(OJClass situation)
     Retorna todas as classes disponíveis de uma dada situação, incluindo as
     declaradas e aquelas herdadas de superclasse/superinterface.

OJMethod getMethod(String name, OJClass[] types, OJClass situation)
     Retorna o método especificado, disponível de uma dada situação.

OJMethod getInvokedMethod(String name, OJClass[] types, OJClass situation)
     Retorna o método, de um dado nome, invocado por certos tipos de
     argumentos e disponível em dada situação.
```

Figura 3.6: Métodos membro em OJClass para introspeção (2)

3.1.6 MetaXA

O modelo computacional proposto em MetaXA [2] [JK96] consiste de sistema operacional, programa de aplicação (sistema base) e sistema meta. Conforme mostra a figura 3.7, a computação no nível base levanta eventos que são entregues ao sistema meta, que os avalia e reage de maneira específica. Todos os eventos são manipulados de forma síncrona, então a computação do nível base é suspensa enquanto o evento é processado pelo metaobjeto. Isto dá ao nível meta total controle sobre a atividade no nível base.

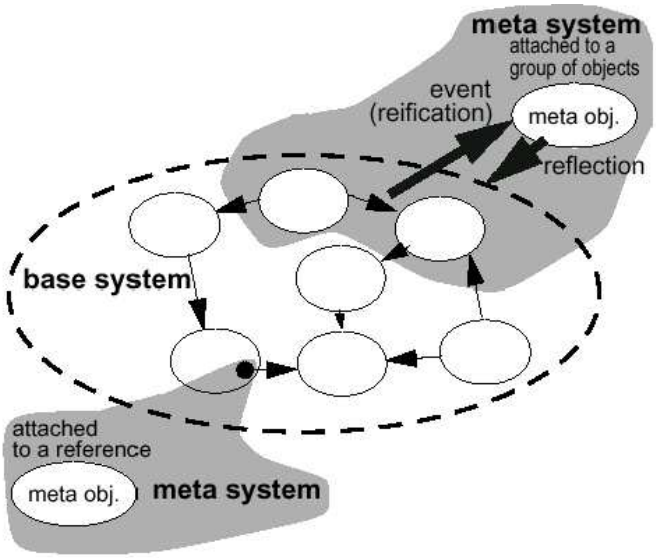

Figura 3.7: Modelo computacional de comportamento reflexivo

Um objeto do nível base também pode invocar um método de um metaobjeto diretamente. Isto é chamado *metainteração explícita* , e é usada para controlar o nível meta a partir do nível base. Nem todo objeto deve ter um metaobjeto vinculado a ele. Metaobjetos podem ser vinculados dinamicamente a objetos do nível base em tempo de execução. Desta forma, a arquitetura proposta evita sobrecarga, porque as funcionalidades do nível meta podem estar vinculadas somente onde realmente houver necessidade. Um metaobjeto pode ser vinculado a uma referência, um objeto ou uma classe. Se for vinculado a uma classe, todas as instâncias dessa classe serão reflexivas.

Para cumprir sua tarefa o metaobjeto tem acesso a um conjunto de métodos, com os quais pode manipular o estado interno da máquina virtual. Somente a classe *MetaObject*, suas subclasses e membros do pacote *meta* podem invocar esses métodos. O quadro da figura 3.8 relaciona os principais métodos.

[2] MetaXa é conhecido formalmente por MetaJava.

3.1.7 Javassist

Javassist é uma ferramenta de programação para ajudar programadores Java. Permite escrever meta-programas para automatizar alguns tipos de definição de classes em Java, que podem ser executados em tempo de compilação ou em tempo de execução[Chi98a].

Javassist provê uma biblioteca de classes para habilitar reflexão estrutural em Java e não requer alteração no sistema de execução ou compilador. Reflexão estrutural permite a um programa alterar as definições de estruturas de dados, tais como classes e métodos. A figura 3.9 apresenta os métodos disponíveis na API de Javassist para reflexão estrutural, que possibilitam alterar classes (definidos na classe denominada *CtClass*)[Chi00].

A idéia básica desta arquitetura é que reflexão estrutural é executada através da transformação de *bytecodes* em tempo de compilação ou em tempo de carga, embora o usuário de Javassist não necessite conhecer profundamente *bytecodes* Java.

Para prover reflexão estrutural sem modificar o sistema de execução ou compilador, Javassist disponibiliza a reflexão somente antes do programa ser carregado no sistema de execução, que é em tempo de carga (*load time*). Os *bytecodes* resultantes de uma compilação em Java são armazenados em arquivos de classes e transformados por Javassist para executar reflexão estrutural. Após a transformação, os arquivos de classes modificados são carregados na JVM e nenhuma alteração é permitida a partir deste momento.

Javassist também provê API para reificação, reflexão e introspecção semelhante a API de OpenC++ e OpenJava, porém seu diferencial em relação a outras arquiteturas é prover reflexão estrutural ao invés de comportamental sem requerer alteração no código fonte. Javassist é uma extensão para a API de reflexão do Java.

As três metas de projeto de Javassist são:

- prover abstração de código fonte para o programador sem requerer conhecimento em *bytecodes*;
- executar reflexão estrutural da forma mais eficiente possível;
- ajudar programadores a executar reflexão estrutural de uma maneira segura em termos de tipos.

```
void attachObject (MetaObject meta, Object base)
   Liga um metaobjeto a um objeto base.

MetaObject findMeta (Object base)
   Encontra o metaobjeto responsável pelo objeto base.

Object continueExecutionObject (EventMethodCall event)
   Continua a execução do método do nível base. Isto chama o método não
   reflexivo. Nenhum evento é gerado, por outro lado a reflexão não
   deveria terminar.

Object doExecuteObject (EventMethodCall event)
   Executa um método. É o oposto do método anterior. Ele chama o método
   como se ele fosse chamado por um objeto comum do nível base.

Object createNewInstance (EventObjectCreation event)
   Cria uma nova instância de uma classe. O nome da classe é passada como
   String (como uma parte do parametro evento).

void createStubs (Object obj, String methods[])
   Cria método stubs para o objeto base, o qual substitui o método
   especificado. Se tal método é chamado, o stub delega o controle para o
   método manipulador de evento do metaobjeto vinculado.

Object cloneReference (Object ref)
   Cria uma nova referência que aponta para o mesmo objeto que o objeto passado.

Class cloneClass(Object ref)
   Copia a classe do objeto passado como parâmetro. A cópia torna-se a classe
   deste objeto. Este método é usado quando um metaobjeto é vinculado a um
   objeto para evitar interferência com outras instâncias da mesma classe.
```

Figura 3.8: Métodos selecionados da interface da máquina virtual de MetaJava

```
void bePublic()
    Torna a classe public

void beAbstract()
    Torna a classe abstract

void notFinal()
    Remove o modificador final da classe

setName(String name)
    Altera o nome da classe

void setSuperClass(CtClass c)
    Altera o nome da superclasse

void setInterfaces(CtClass[] i)
    Altera as interfaces

void addConstructor()
    Adiciona um novo construtor

void addDefaultConstructor()
    Adiciona um construtor default

void addAbstractMethod()
    Adiciona um método abstract

void addMethod()
    Adiciona um novo método

void addWrapper()
    Adiciona um novo método wrapper

void addField()
    Adiciona um novo campo
```

Figura 3.9: Métodos de Javassist para alteração de classes

3.2 Programação Orientada a Aspectos

O paradigma de programação orientado a objetos tem sido dominante nos últimos anos. Ele permite a construção de um sistema através da decomposição de um problema em objetos, que abstraem o comportamento e os dados em uma única entidade. A oferta de metodologias, ferramentas de análise, projeto e desenvolvimento contribuem para o sucesso do paradigma, possibilitando o desenvolvimento de sistemas complexos como interfaces gráficas para usuários, sistemas operacionais e aplicações distribuídas.

Apesar de todas as qualidades, a orientação a objetos é deficiente quando se quer representar certas áreas de interesses como segurança, tolerância a falhas e *log* do sistema, porque elas atuam em diversas classes espalhadas pelo sistema, o que torna complexa a programação e manutenção do código. Essa dificuldade de trabalhar com áreas de interesses, ou *concerns*, contribuiu para o nascimento da programação orientada a aspectos [EEB01][KLM+97].

Programação orientada a aspectos reconhece as qualidades de orientação a objetos e complementa seus benefícios, facilitando outro tipo de modularidade, que agrupa áreas de interesse [EEB01]. A proposta da programação orientada a aspectos é separar interesses durante o desenvolvimento e posteriormente compor o sistema, de forma coerente, através de um processo denominado *weave*, que consiste em entrelaçar todos os aspectos. A separação do código em interesses facilita a inclusão, alteração e exclusão de um aspecto em tempo de compilação.

AOP permite a modularização de *crosscutting concerns*, que é o comportamento que atravessa as divisões típicas de responsabilidade em dado modelo de programação. Como exemplo, pode-se citar: *log*, manipulação de erros sensível ao contexto e otimizações de desempenho.

3.2.1 AspectJ

AspectJ foi desenvolvida pela Xerox PARC, oferecendo vantagens típicas de programação orientada a aspectos para desenvolvedores Java.

O exemplo mostrado nas figuras 3.10 e 3.11 foi apresentado em [Les02], e torna claros os benefícios da programação orientada a aspectos. Trata-se de um *log* que simplifica a depuração e testes de componentes Java no lado servidor.

Sem o uso de aspectos, o programador teria que introduzir manualmente as linhas de código que fazem a chamada ao *log*, em todos os seus métodos, registrando inclusive, os parâmetros de cada método. Isto acarretaria dificuldades de implementação e manutenção. Essas linhas de código estão representadas no exemplo da figura 3.10.

```
public void doGet (JspImplicitObjects theObjects) throws
                                                  ServletException
{
  logger.entry ("doGet(...)");
  JspTestController controller = new JspTestController();
  controller.handlerRequest(theObjects);
  logger.exit("do.get")
}
```

Figura 3.10: Chamadas ao *log* inseridas manualmente em todo método

Com o uso de AOP bastaria definir um aspecto responsável pelo *log*, evitando a repetição deste controle, que não estaria mais espalhado por todo o código de negócio. Aspectos encapsulam comportamentos que afetam múltiplas classes, em módulos reusáveis.

A figura 3.11 exemplifica a definição de um aspecto para *log* onde pode-se observar a semelhança entre a declaração de um aspecto e a declaração de uma classe. Um aspecto contém *pointcuts* e *advices*. *Pointcut* é um construtor da linguagem que separa um conjunto de *join points* baseados em um critério definido. *Advice* é o código que executa antes, depois ou ao redor de um *join point*. Alguns comentários foram adicionados ao exemplo a fim de auxiliar o entendimento.

Join point é o termo utilizado em programação orientada a aspectos para nomear o local onde o código do aspecto interage com o resto do sistema. Chamadas a métodos, acesso a membros de classes e a execução de manipuladores de blocos de exceção são exemplos de *join points*. *Join points* podem

conter outros *join points*. Por exemplo, uma chamada de método pode resultar em outras chamadas de métodos antes que ela retorne.

Experiências com otimizações de performance [KLM+97] obtiveram resultados satisfatórios com o uso de AOP. Programas contendo 35.213 linhas foram reescritos com técnicas orientadas a aspectos e passaram a ter 1039 linhas, conservando a maioria dos benefícios de desempenho.

Kiczales et all [KLM+97] considera reflexão como uma poderosa ferramenta para programação orientada a aspectos. E descreve meta-linguagens como linguagens de aspectos de baixo nível, cujos *join points* são os ganchos que o sistema reflexivo provê.

3.2.2 Alterações Dinâmicas em AOP

Um domínio importante para uso de AOP é a adaptação de serviços em resposta às mudanças. O ideal é que as adaptações ocorram dinamicamente, em tempo de execução. Entretanto, a maioria das abordagens para AOP não permitem alterações dinâmicas, porque realizam o processo de entrelaçamento de aspectos em tempo de compilação.

PROSE [3] [PGG01] é uma plataforma para programação orientada a aspectos baseada em Java que suporta alteração dinâmica, em tempo de execução. Isto é provido através de uma JVMAI [4], que permite a inserção de aspectos no estado de uma JVM [5], entrelaçando a execução e não o código.

A implementação da JVMAI de PROSE é baseada em uma JVMDI (*Java Virtual Machine Debugger Interface*). Logo após a inserção de um aspecto, o núcleo de PROSE inspeciona a JVM e o novo aspecto para encontrar seus *join-points*. Para cada *join-point* encontrado é requisitada uma notificação específica da camada de *debugger* da JVM. A funcionalidade de resposta para o *join-point* também é registrada no núcleo de PROSE durante a inserção. Quando a JVM atinge um *join-point* registrado, a execução da *thread* atual é temporariamente suspensa e o *debugger* passa o controle de execução ao PROSE. Neste ponto, PROSE realiza chamadas lógicas para funcionalidades adicionais nos pontos que atravessam os aspectos dos *join-points* registrados. Durante este processo PROSE pode inspecionar a pilha de *threads* atuais e passar informações para os métodos *advice* dos pontos que atravessam os aspectos. Após a execução dos *advices*, o controle é retornado para a aplicação.

[3] *PROSE - PROgrammable extenSions of sErvices*
[4] *JVMAI -Java Virtual Machine Aspect Interface*
[5] *JVM - Java Virtual Machine*

```
public aspect AutoLog
{
 // seleciona a execução de todos os métodos públicos no pacote
 // org.apache.cactus
 pointcut publicMethods():execution(public *org.apache.cactus)(..));

 // seleciona todos os métodos da classe Logger
 poincut logObjectCalls(): execution (* Logger.* (..));

 // filtra todos os métodos selecionados no primeiro pointcut,
 // que não fazem parte da implementação do próprio log (para
 // evitar recursividade infinita)
 pointcut loggableCalls(): publicMethods() && !logObjectCalls();

 // solicita que o código seja executado antes de cada chamada de
 // método que se deseja registrar no log
 before(): loggableCalls()
 {
   Logger.entry (thisJoinPoint.getSignature().toString());
 }
 // solicita que o código seja executado após cada chamada de
 // método que se deseja registrar no log
 after() : loggableCalls()
 {
   Logger.exit(thisJoinPoint).getSignature().toString()
 }
}
```

Figura 3.11: Chamadas ao *log* aplicadas automaticamente para todo método

3.2.3 Persistência Ortogonal Usando Aspectos

Vandenborre et. all [KVH02] demonstram uma forma de implementar persistência ortogonal através de aspectos. Eles apresentam um exemplo de sistema de faturamento simples, cuja modelagem é expressa em UML, como mostra a figura 3.12.

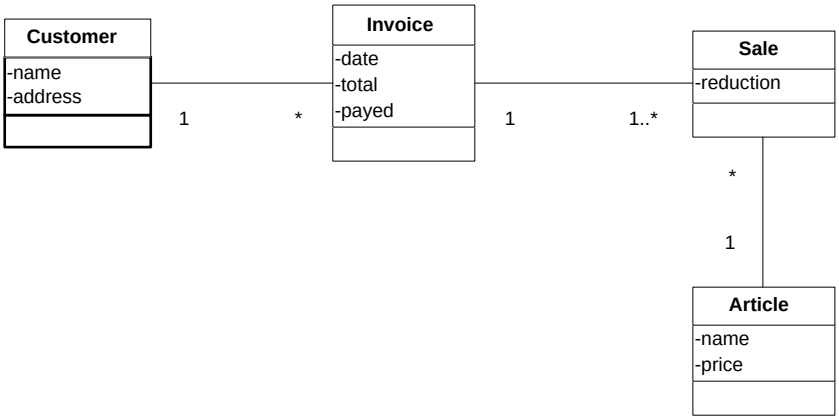

Figura 3.12: Sistema simples de faturamento

A implementação da persistência com o uso de aspectos é exemplificada nas figuras 3.13 e 3.14, escritas em Aspect/J [KVH02].

Aspect/J provê duas maneiras para capturar *crosscutting concerns*: por introdução ou recomendação (*advice*). E é justamente a combinação dessas duas formas que os autores propõem para obter ortogonalidade completa. A primeira forma - por introdução - permite a inclusão de novos atributos e métodos em classes existentes. A segunda - por recomendação - provê a habilidade de executar códigos extra, em certos pontos e momentos definidos (*pointcuts*).

A introdução é usada para inserir em classes existentes as características necessárias para obter uma classe persistente. Então, é preciso definir "o que" e "onde" será inserido. Eles sugerem a introdução de um atributo para identificação do objeto e de métodos como *read(), write(), update()* e *delete()* para interagir com o meio de armazenamento. Entretanto, esses métodos devem ter sua implementação vazia para evitar que toda classe tenha que implementar sua persistência, ou então criar métodos genéricos, que podem ser extremamente complexos. Para definir "onde" será inserido, utiliza-se a segunda técnica - *advice* - que define um conjunto de pontos e o momento adequado para executar o código extra que implementa a persistência.

O exemplo da figura 3.13 considera que as classes de negócio *Customer* e *Invoice* são persistentes. Ele mostra o aspecto que realiza a introdução de: um

atributo para identificar o objeto; um método privado para obter o identificador do objeto; métodos genéricos para escrever, atualizar e remover objetos persistentes cujo retorno é um booleano indicando o sucesso da operação; um método genérico para leitura, que pode retornar um ou muitos objetos, cujo tipo de retorno é um vector.

```
public aspect Persistent Introductor
{
   declare parents: Invoice implements Persistent;
   declare parents: Customer implements Persistent;

   private Long Persistent.oID = new Long(Math.round Math.random()* 1000000);

   private Long Persistent.getOID()
      {return oID;}
   public Boolean Persistent.write(Persistent p)
      {return new Boolean(false);}
   public Vector Persistent.read(Long i)
      {return new Vector();}
   public Boolean Persistent.update (Long i)
      {return new Boolean(false);}
   public Boolean Persistent.delete (Long i)
      {return new Boolean(false);}
}
```

Figura 3.13: Aspecto Introductor

Até esse ponto, foram introduzidos apenas métodos genéricos, que não executam código de persistência. Isto é resolvido com a criação de outro aspecto, que precisa: ter privilégio para acessar o método getOID() e os atributos privados do objeto; definir os pointcuts; realizar a conexão com o banco de dados. Isto é exemplificado pelo aspecto implementado para a classe Invoice, que pode ser observado na figura 3.14.

```
public privileged aspect PInvoice
{
   pointcut reader(Invoice p): target(p) && call (public Vector read(..));
   pointcut writer(Invoice p): target(p) && call (public Boolean write(..));
   pointcut updater(Invoice p): target(p) && call (public Boolean update(..));
   pointcut deleter(Invoice p): target(p) && call (public Boolean delete(..));

   private Connection con = null;
   private void setConnection()
       /* Conecta ao banco de dados */

   after (Invoice p ) returning (Boolean success) : reader(p);
   /* Obter uma conexão com o banco de dados, construir um PreparedStatement
      para ler a fatura da tabela invoice e associar ao cliente da tabela
      customer, construir um objeto invoice e colocar este objeto no vetor v
      retornado pelo método original sendo o sujeito deste advice  */

   after (Invoice p ) returning (Boolean success) : writer(p);
   /* Obter uma conexão com o banco de dados, construir um PreparedStatement
      para escrever o objeto invoice, na tabela apropriada e retornar para o
      método original sendo o sujeito deste advice */

   /* after advices semelhantes para updater e deleter */
}
```

Figura 3.14: Aspecto PInvoice

3.3 Conclusão

A API de reflexão Java oferece muitas facilidades interessantes para aplicar reflexão computacional à linguagem Java. Entretanto, algumas limitações devido ao mecanismo de proteção adotado por Java dificultam o uso da API para implementar persistência, porque ela permite obter metainformações somente de objetos com atributos públicos.

Guaraná é um exemplo da viabilidade do uso de máquina virtual como meio de prover reflexão computacional, mas a implementação atual baseia-se em Java bytecode, o que dificulta a obtenção de meta-informação em tempo de execução.

O sistema MetaXa tem diversas vantagens como o uso de uma linguagem popular, que oferece bibliotecas confiáveis e ambiente de desenvolvimento sofisticado. Permite a separação de código do nível base e nível meta, com relação *n para n* entre classes e instâncias de metaobjetos da classe. O projeto MetaXa foi descontinuado, porém ainda é usado como base para outros trabalhos, como a definição de um sistema operacional de arquitetura flexível, gerencia de réplicas de objetos, etc.

Open C++ separa claramente a computação distribuída de outra computação, que é mais substancial ao programador. Isto é obtido pelo uso de reflexão computacional, que mantém em um nível meta toda a computação responsável pela comunicação e sincronização [CM93]. Open C++, possui a desvantagem da sobrecarga associada ao protocolo de metaobjetos, que trabalha com desvios em chamadas de métodos. Essa sobrecarga é negligenciada quando Open C++ é usada para computação distribuída, pois mesmo não sendo pequena, ela não é significativa quando comparada ao tempo de latência de rede [CM93]. Em sua recente versão [Chi98b], Open C++ provê um eficiente sistema de macros adequado a programação orientada a objetos.

OpenJava é um sistema de macro com uma estrutura de dados representando uma estrutura lógica de um programa orientado a objetos. A efetividade de OpenJava foi mostrada através da implementação de macros para suportar *design patterns*. Entretanto, também foram encontradas limitações de OpenJava, como a necessidade de tratar diversas classes como uma entidade simples. Isto deve ser resolvido em trabalhos futuros, incorporando técnicas de programação orientada a aspectos [MTI00].

Enfim, todos os ambientes reflexivos contribuiram para a experimentação de conceitos e o aperfeiçoamento de técnicas de reflexão computacional, mostrando a viabilidade de seu uso para sistemas de comportamento e es-

trutura dinâmicos.

Programação orientada a aspectos complementa a programação orientada a objetos porque facilita outro tipo de modularidade, que junta toda a implementação espalhada de *crosscutting concerns* em simples unidades denominadas *aspectos*. A divisão do código em aspectos torna mais fácil tratar *crosscutting concerns*. Aspectos de um sistema podem ser modificados, inseridos ou removidos em tempo de compilação e mesmo reusados.

A proposta de AOP de separar interesses durante o desenvolvimento e posteriormente compor o sistema, de forma coerente está de acordo com nossos objetivos. Portanto resolvemos adotar o conceito de *weave*, usado neste paradigma, que consiste em entrelaçar todos os aspectos.

Capítulo 4

Um Ambiente de Execução Reflexivo - Virtuosi

Este capítulo apresenta o ambiente de execução sobre o qual foi projetado o mecanismo de persistência proposto por este trabalho.

Inicialmente são revisados os conceitos essenciais de Virtuosi, uma arquitetura proposta por Calsavara [Cal00]. Em seguida é apresentada uma visão geral de Larva[NC01], estudo realizado para definição de uma linguagem de programação reflexiva orientada a objetos, que baseia-se em muitos conceitos do ambiente.

Por fim, é descrita o metamodelo da Virtuosi, que atualmente está sendo aperfeiçoado e validado por uma equipe de trabalho coordenada pelo professor Dr. Calsavara e composta por alunos de mestrado. Embora bastante discutida, a versão atual do metamodelo não é considerada imutável, podendo sofrer alterações em função dos estudos e experimentos a serem realizados pelas dissertações em andamento.

4.1 Virtuosi

Virtuosi propõe uma arquitetura de um ambiente de execução distribuído para sistemas orientados a objetos. Este ambiente é composto de máquinas virtuais cooperantes, que executam em computadores conectados por uma rede sobre sistemas operacionais padrão. Suas principais características são o suporte direto aos princípios de orientação a objetos, a comunicação entre objetos distribuídos e o uso de reflexão computacional [Cal00]. O conceito de reflexão computacional serve como meio de concretizar diversos mecanismos do ambiente de execução e permite a redefinição do comportamento do

ambiente.

Nas próximas subseções são descritos alguns conceitos, notação gráfica e terminologias propostos pela Virtuosi. Em seguida são apresentadas as características principais de Virtuosi, que estão diretamente ligadas ao assunto desse trabalho, tais como distribuição, mecanismos do ambiente de execução e outros.

4.1.1 Conceitos e Terminologias

A seguir são relacionados alguns conceitos e terminologias relativos à organização dos objetos:

Composição: Objetos podem ser organizados de modo que, logicamente, uns contenham outros. Está técnica é utilizada para representar objetos complexos ou compostos.

Os tipos de objetos com relação à propriedade de composição, são os seguintes:

- Atômico: Um objeto que não contém outro objeto.
- Isolado: Um objeto atômico não contido em outro objeto.
- Contido: Um objeto contido em outro objeto.
- Contentor: Um objeto que contém outro objeto.
- Contentor global: Um objeto contentor não contido em outro.
- Contentor local: Um objeto contentor contido em outro.
- Contentor direto: O objeto contentor direto de um objeto contido x é o objeto que contém x diretamente.
- Contentor indireto: O objeto contentor indireto de um objeto contido x é um objeto que contém indiretamente, isto é contém direta ou indiretamente o objeto contentor direto de x.

4.1.2 Notação Gráfica

Na notação proposta um objeto é representado por um círculo dividido em duas partes, simbolizando seu estado e seu comportamento. Sua identidade, que deve ser única, é anotada através de um número inteiro.

As referências entre objetos são anotadas através de uma linha tracejada, terminada com uma seta indicando o objeto referenciado. A composição lógica de objetos, também pode ser representada através de referências, diferenciando-se pela seta cheia: um objeto contentor mantém referências para todos os objetos nele contidos. Outra forma é apresentar os objetos contidos diretamente no interior do objeto contentor. As duas formas equivalentes são mostradas no exemplo (figura 4.1) apresentado por Calsavara.

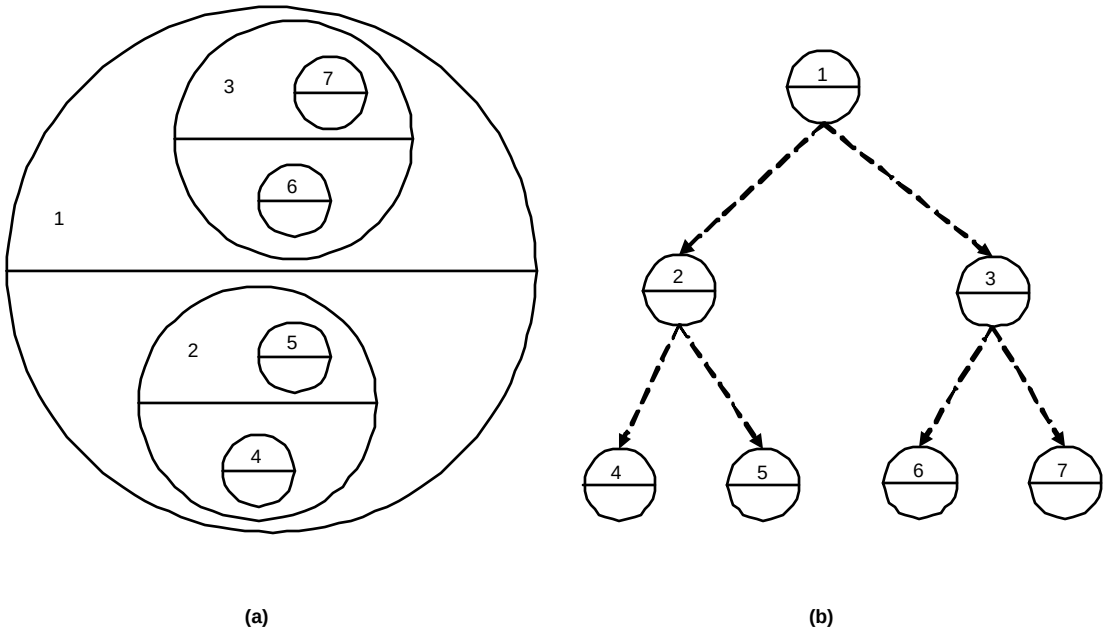

Figura 4.1: Representações alternativas para composição de objetos

Embora existam diversos outros elementos presentes na notação proposta, serão descritos somente aqueles utilizados nos exemplos desse trabalho. Sendo assim, resta ainda mencionar dois importantes elementos: a representação de um objeto de fronteira e de metacomponentes.

Um objeto de fronteira é semelhante a um objeto comum, mas possui arcos em sua camada inferior. Os objetos de fronteira são descritos na seção *Interação com dispositivos externos*.

Metacomponentes também são documentados em sua própria seção. Sua forma é idêntica a de um objeto comum, porém a atuação de um metacomponente sobre outro objeto é representada por uma referência com linha cheia. O exemplo da figura 4.3 contém a notação gráfica aplicada tanto a objetos de fronteira como a metacomponentes.

4.1.3 Distribuição na Virtuosi

Todo objeto existe dentro do espaço de endereçamento de uma máquina virtual em particular, a qual situa-se em um computador real. Um mesmo computador, por sua vez, pode abrigar diversas máquinas virtuais. Assim, os objetos podem estar distribuídos pelos diversos computadores de uma rede e ainda em um mesmo computador distribuídos em diversas máquinas virtuais. A alocação de objetos nas máquinas virtuais pode ocorrer de maneira totalmente arbitrária, com apenas uma restrição: um objeto contentor e todos os objetos nele contidos residem em uma mesma memória virtual. Essa restrição visa somente otimização, uma vez que há maior probabilidade de alta interação entre objetos que formam um objeto contentor.

A figura 4.2 exemplifica a distribuição de objetos na Virtuosi.

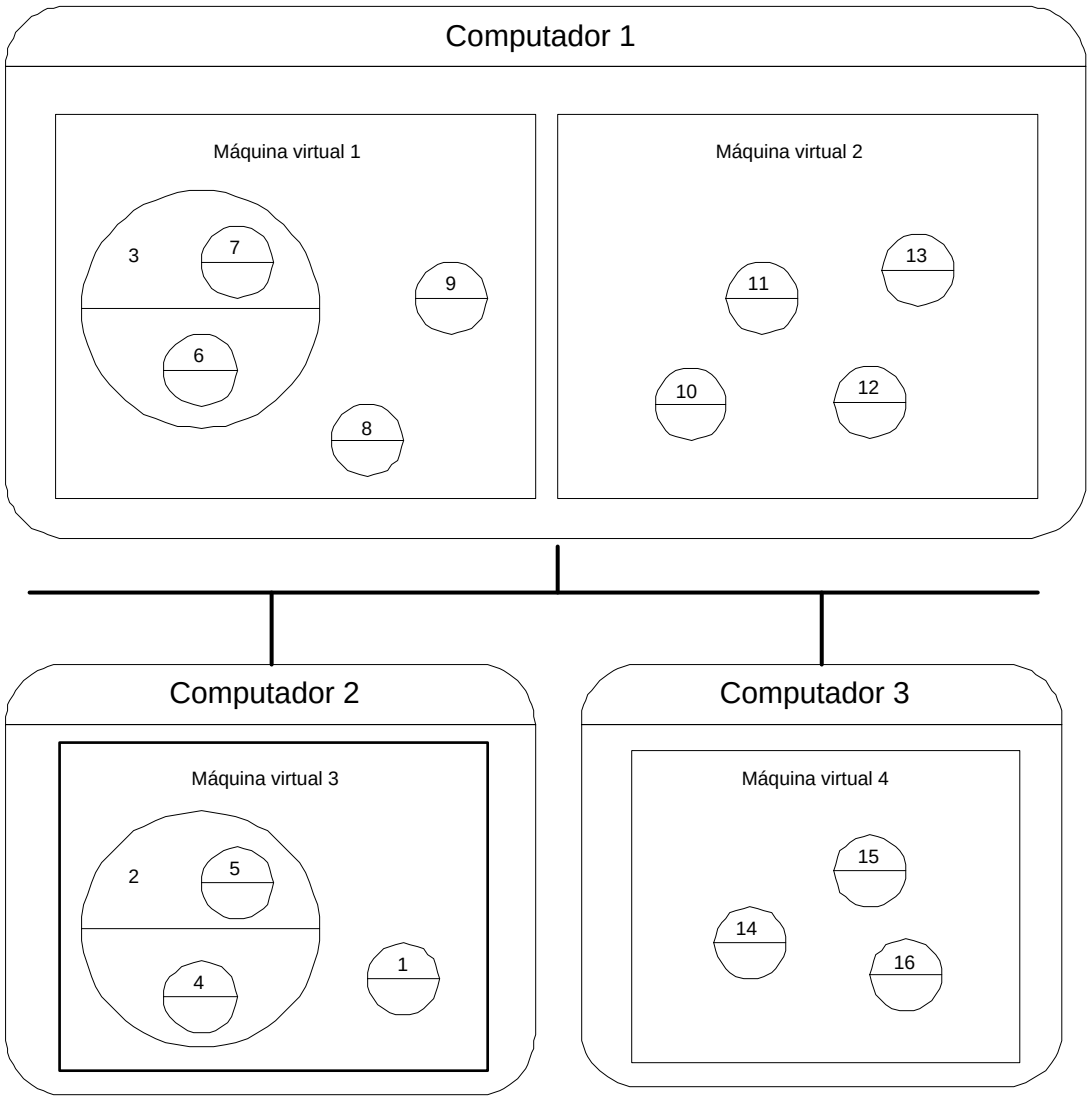

Figura 4.2: Objetos distribuídos na *Virtuosi*

4.1.4 Interação com Dispositivos Externos

Os objetos da Virtuosi interagem com dispositivos do ambiente de execução, tais como impressoras, discos e outros. Esses dispositivos não se restringem a elementos de hardware - um gerenciador de banco de dados, por exemplo, também pode ser considerado como um dispositivo externo [Cal00]. A interação entre objetos da Virtuosi e dispositivos externos ocorre através de objetos especialmente projetados para esse fim. Esses objetos situam-se logicamente na fronteira entre a Virtuosi e o sistema real de execução, sendo por essa razão denominados objetos de fronteira. Essa interação pode ocorrer nos dois sentidos, ou seja, é possível que os objetos de fronteira sejam afetados por eventos gerados no contexto do dispositivo externo e, como conseqüência, invoquem atividades dos objetos normais.

4.1.5 Metacomponentes

O conceito de reflexão computacional serve como meio para descrever, controlar e adaptar o comportamento do sistema computacional através da associação a *metacomponentes* dos objetos. Os metacomponentes podem ser definidos tanto para refletir sobre o comportamento dos aspectos funcionais de uma aplicação, quanto para refletir sobre os seus aspectos de sistema, como persistência, segurança, falhas e outros.

O fato de todo metacomponente também poder ser um objeto, torna a Virtuosi recursiva. Isso permite que os metacomponentes possam cooperar utilizando os próprios mecanismos de comunicação disponíveis para os objetos de aplicação. Também torna-se viável a definição de um metacomponente que coopera com outros metacomponentes, possivelmente associados a outros objetos e residentes em outras máquinas virtuais. Como exemplo, pode-se citar um metacomponente de gerenciamento de transação atômica que para realizar seu trabalho coopera com metacomponentes de persistência, controle de concorrência e recuperação.

Um exemplo de metacomponente atuando sobre objetos pode ser visto na figura 4.3. Trata-se de um metacomponente de persistência que acessa os dados dos objetos da aplicação e transfere-os ao objeto de fronteira, ou adaptador, que por sua vez, converte as informações para o formato entendido pelo DBMS.

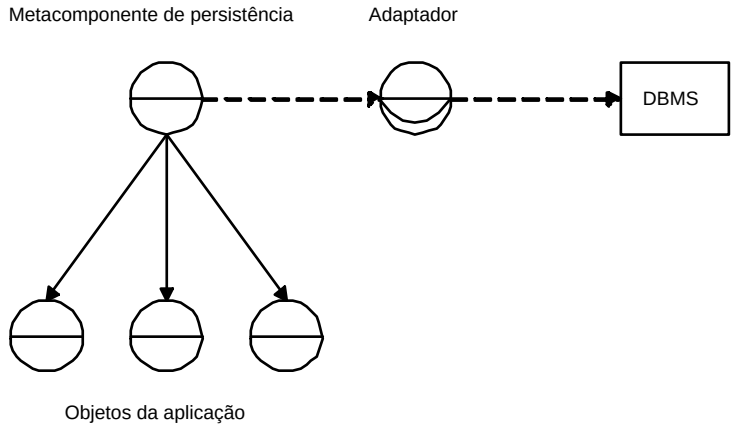

Figura 4.3: Interação do metacomponente de persistência com DBMS

4.2 Larva

Larva apresenta um estudo para definição de uma linguagem de programação reflexiva orientada a objetos e o seu correspondente ambiente de execução, tal que não se manifestem as restrições semânticas e de desempenho tipicamente encontradas em linguagens e ambientes similares [NC01].

Em Larva a reflexão computacional é realizada através da modificação dinâmica de árvores de programa. Assim, todo e qualquer elemento da linguagem pode sofrer reflexão computacional. Esta abordagem dispensa qualquer desvio extra no fluxo de execução de aplicações.

Foi estabelecido um modelo básico de objetos: subconjunto de Virtuosi, uma forma de representação de programas orientados a objetos através de árvores de programas e um mecanismo de execução baseado em técnicas tradicionais de gerenciamento de contextos, através de pilhas de execução.

As principais características seguidas por Larva são: fidelidade à orientação a objetos, tipagem forte, extensibilidade e reflexão computacional.

4.2.1 Modelo de Objetos

Os conceitos de orientação a objetos suportados por Larva assim como seus relacionamentos são expressos através de um metamodelo, mostrado na figura 4.4.

São definidos dois tipos de classe: atômica e composta, representadas pelas classes *AtomicClass* e *ComposedClass* respectivamente. Uma classe atômica encapsula um bloco de dados (*DataBlock*) e contém um conjunto de métodos que definem a semântica do bloco de dados encapsulado. Um bloco de dados é um objeto que fornece acesso e possibilita a manipulação de uma área de

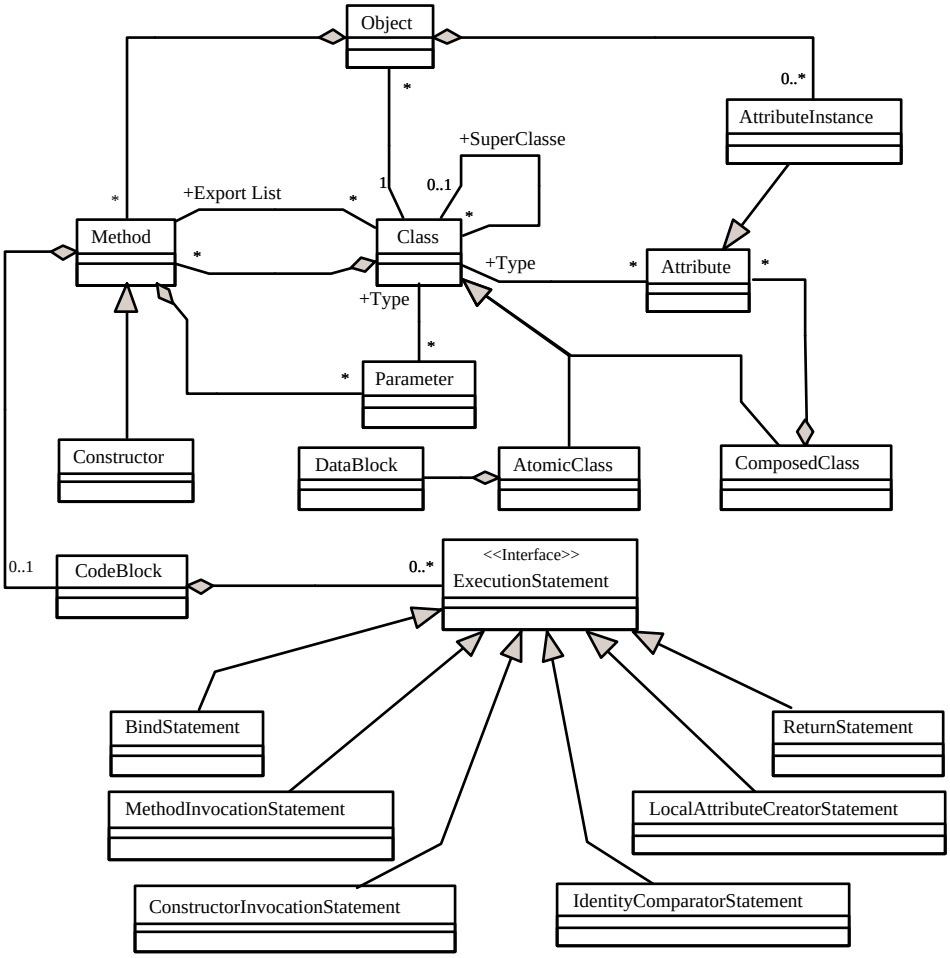

Figura 4.4: Metamodelo de Larva

memória. Assim, uma classe atômica pode ser utilizada para a implementação de um tipo primitivo, tal como um inteiro ou um real. Uma classe composta permite a definição de objetos que possuem atributos, os quais podem ser outros objetos (atômicos ou compostos).

A herança entre classes permite que a classe herdeira possa acessar o estado e os métodos definidos pela classe ancestral. Larva permite apenas a herança simples entre classes.

Os atributos são representados pelas classes *Attribute* e *AttributeInstance*. A primeira mantém o nome e o tipo do atributo, a segunda mantém o valor para cada instância do atributo.

Métodos contêm sua assinatura (nome, tipo de retorno e parâmetros de entrada) e uma lista de exportação, contendo as classes clientes que têm permissão para invocá-lo. A implementação de um método é definida por um bloco de execução (*CodeBlock*), que é executado no próprio contexto de exe-

cução em que é invocado, ou seja, é apenas uma lista de comandos (*ExecutionStatement*), que podem ser declaração de variáveis locais (*LocalAttributeCreatorStatement*), comparação de objetos (*IdentityComparatorStatement*), retorno em método (*ReturnStatement*), invocação de métodos (*MethodInvocationStatement*), invocação de construtores (*ConstructorInvocationStatement*) e atribuição de valores (*BindStatement*). A associação de um bloco de execução a um método é opcional; quando existe, o método é concreto, caso contrário, é abstrato.

4.2.2 Árvores de Programa e Reflexão

Árvore de programa é um grafo de objetos instanciados a partir das classes do metamodelo.

Uma árvore de programa possibilita obter reflexão computacional pela manipulação direta dos objetos que a constituem. Assim, é possível fazer introspecção sobre a estrutura de um objeto verificando-se os objetos que descrevem a sua classe [NC01].

O comportamento das instâncias de uma classe pode ser alterado pela edição dos blocos de execução dos seus métodos.

4.2.3 Execução de Programas

As aplicações feitas em Larva iniciam sua execução através da criação de algum objeto. O interpretador executa blocos de execução conforme os métodos e os construtores sejam invocados. Cada execução de um bloco requer a criação de um contexto de execução, ou seja, uma região que armazena uma referência para o objeto sobre o qual se aplica o bloco, os parâmetros de execução e os atributos locais. O gerenciamento de contextos é realizado através de uma pilha de execução, que a cada novo bloco iniciado cria e insere na pilha o contexto correspondente. No encerramento do bloco o contexto é retirado da pilha.

4.3 Metamodelo da Virtuosi

Dando continuidade ao trabalho iniciado em Virtuosi e Larva [Cal00] [NC01] alguns alunos de mestrado estão trabalhando na definição de um metamodelo mais abrangente, capaz de representar de forma genérica, elementos estruturais e dinâmicos de sistemas orientados a objetos. Através das metainformações presentes nesses elementos o sistema de execução poderá agir de forma reflexiva sobre as aplicações, e estará apto a acoplar serviços básicos de sistemas, tais como, persistência, eventos, segurança e controle de transações, que poderão ser projetados para este ambiente.

O metamodelo define o formato e a semântica do código de execução. Sua representação em UML pode ser vista na figura 4.5.

4.3.1 Árvores de Programa

Árvores de Sintaxe Abstratas (Abstract Syntax Trees) são utilizadas para representar códigos de execução de acordo com as definições do metamodelo. Para simplificar, essas árvores são denominadas *árvores de programas*.

Árvore de programa é uma representação intermediária de software, ou seja, é o resultado de uma análise léxica e sintática realizada sobre um conjunto de código fonte. A partir dessa árvore é possível gerar código objeto ou executá-la diretamente através de um interpretador (máquina virtual).

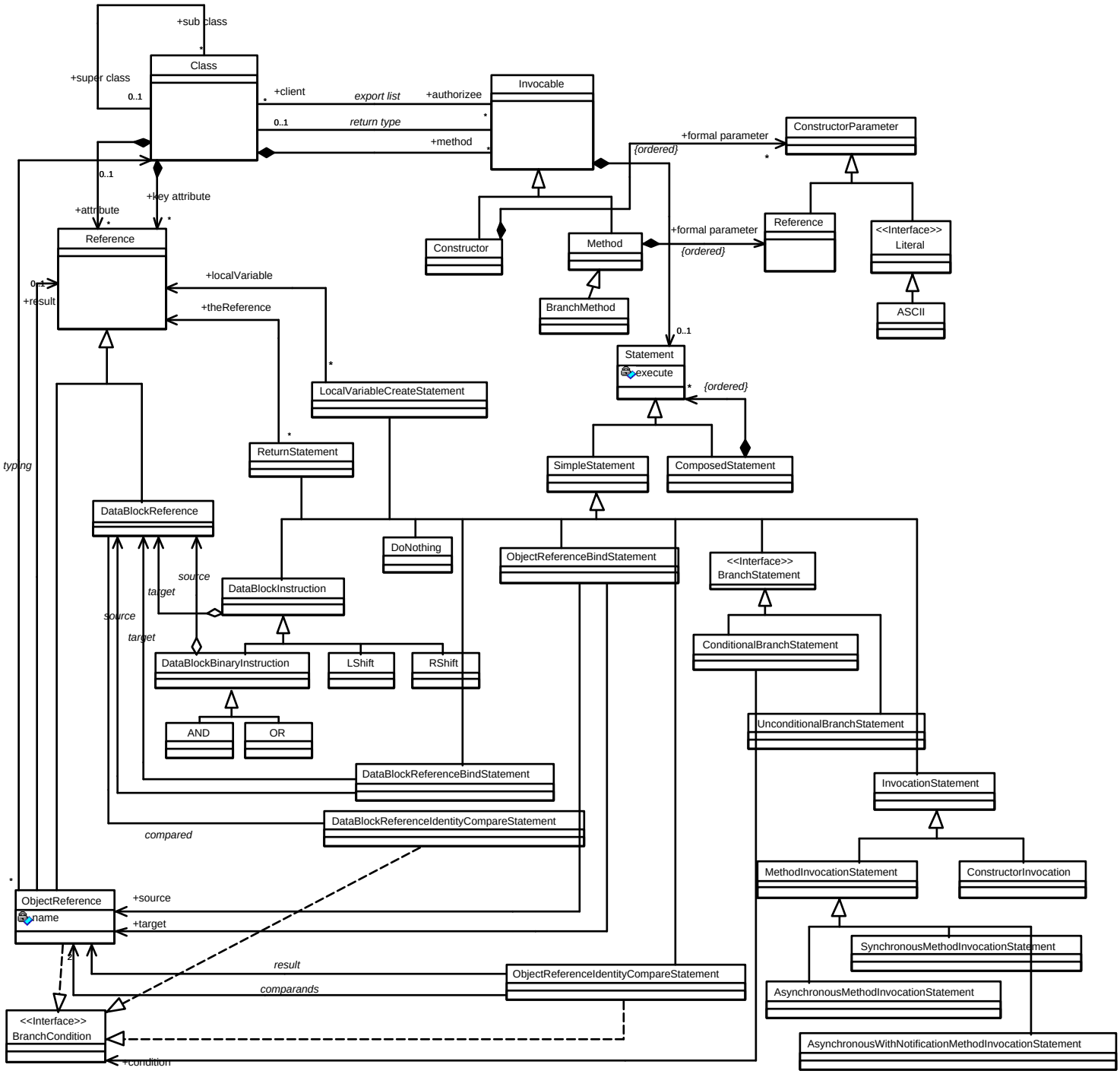

Figura 4.5: Metamodelo da Virtuosi para árvores de programa

4.4 Conclusão

O ambiente Virtuosi, através da integração dos conceitos de orientação a objetos, sistemas distribuídos e reflexão computacional objetiva diversos benefícios, tais como a simplificação na compilação de programas escritos em linguagens orientadas a objetos, a otimização na geração de código executável, a simplificação na escrita de programas com relação à comunicação entre objetos distribuídos, a portabilidade do código gerado pelos compiladores em função de executar em uma máquina virtual, a possibilidade de se construir um mesmo software utilizando-se diferentes linguagens, a possibilidade de se executar um sistema sobre plataformas heterogêneas de hardware e sistema operacional, a simplificação na construção de software pela possibilidade de separação entre os aspectos funcionais de uma aplicação e os aspectos de sistema envolvidos, a possibilidade de se definir metacomponentes que especializem o comportamento padrão do sistema computacional e consequente flexibilidade na configuração desse sistema [Cal00].

O estudo realizado em Larva, demonstrou a possibilidade de concepção de uma linguagem de programação reflexiva orientada a objetos sem a imposição de restrições semânticas e sem prejuízo de desempenho na execução causado por desvios extras no fluxo de execução. O metamodelo proposto eliminou a necessidade dos tipos primitivos típicos de linguagens de programação possibilitando reflexão computacional ortogonal. E o emprego de árvores de programa possibilitou a edição do comportamento das aplicações antes ou durante a sua execução [NC01].

Capítulo 5

Mecanismo Proposto

Este capítulo descreve o mecanismo proposto para persistência de objetos com uso de reflexão computacional em um ambiente de execução composto por máquinas virtuais.

O mecanismo é baseado nos conceitos e experiências descritas nos capítulos anteriores e conserva as características do ambiente reflexivo (descrito no capítulo 7) para o qual foi projetado.

A seguir são apresentadas as diretrizes da proposta, a descrição do uso de reflexão computacional, as políticas e outras definições inerentes ao mecanismo de persistência.

5.1 Diretrizes

O mecanismo para persistência de objetos propõe o uso de reflexão computacional de um modo diferenciado, que tenta evitar a queda de desempenho típica de seu uso. Isto é discutido na próxima seção.

O fato de ser aplicado a uma arquitetura de ambiente de execução distribuído [Cal00] adiciona algumas vantagens a esta abordagem, tais como: estar apto a executar em um ambiente distribuído e orientado a objetos; ser portável, pois o ambiente é composto por máquinas virtuais cooperantes; utilizar mecanismos para redefinição do comportamento do ambiente, pelo uso de reflexão computacional.

Outras diretrizes adotadas para a persistência são transparência e ortogonalidade. O princípio de persistência transparente requer que o código tenha a mesma semântica, independentemente se ele está operando com dados persistentes ou transientes [ADJ+96]. Isto habilita o software para re-uso. Persistência ortogonal é a provisão de persistência para todos os dados, independente

do tipo, abrangendo objetos de qualquer que seja a classe [ADJ+96].

Assim como em diversas abordagens de persistência [MRB98, Gro99, LS98, HS98] utilizaremos identificadores únicos para objetos. Um identificador, aqui denominado *oid* (*object identifier*), facilita a localização de um objeto e será usado para associar um objeto em memória a seu estado persistido em um meio de armazenamento secundário.

Como discutido no capítulo 4 existem diversos meios de armazenamento secundário que podem ser usados para persistir objetos. Para que o serviço de persistência seja independente do meio de armazenamento, utilizaremos objetos de fronteira, ou adaptadores, conforme descrito por Virtuosi [Cal00].

5.2 Reflexão sem Desvio na Execução

Reflexão computacional tem sido experimentada em vários trabalhos [HS98, OGB98, GBCV96, JK96, CM93, MTI00] e muitos benefícios têm sido demonstrados, tais como separação de atividades de sistemas das atividades funcionais, manutenção de estatísticas de uso, manutenção de informações para propósitos de depuração, auto-otimização e auto-modificação.

Entretanto, uma desvantagem apontada por várias experiências é que reflexão computacional tende a degradar o desempenho dos sistemas. Considerando que a queda de desempenho é resultado de desvios ocorridos entre os níveis meta e base, nossa idéia é justamente eliminar esses desvios em tempo de execução.

Para ilustrar o problema a figura 5.1 (a) demonstra o que ocorre ao invocar um método de um objeto de negócio que está associado a um metacomponente de um ambiente reflexivo tradicional. Quando a invocação de método é solicitada, ao invés de executá-lo é feito um desvio para seu metaobjeto, que assume o controle e realiza algumas ações. Só depois o controle é devolvido para o nível da aplicação para que o objeto de negócio execute o método invocado.

A figura 5.1 (b) representa o objeto equivalente, que em nossa abordagem é composto de código de negócio e código correspondente a ações de seu metacomponente. Assim, quando uma invocação é feita o método de negócio é executado diretamente, sem necessidade de desvios, porque as ações reflexivas já estão embutidas em seu próprio código.

Uma forma de evitar o desvio de ambientes tradicionais seria trazer o código do metaobjeto e embutí-lo diretamente no próprio código da aplicação. Porém, se isso fosse feito no código fonte, ocasionaria ilegibilidade e mistura do código de negócio ao código das implementações básicas do sistema (reflexivas). Além

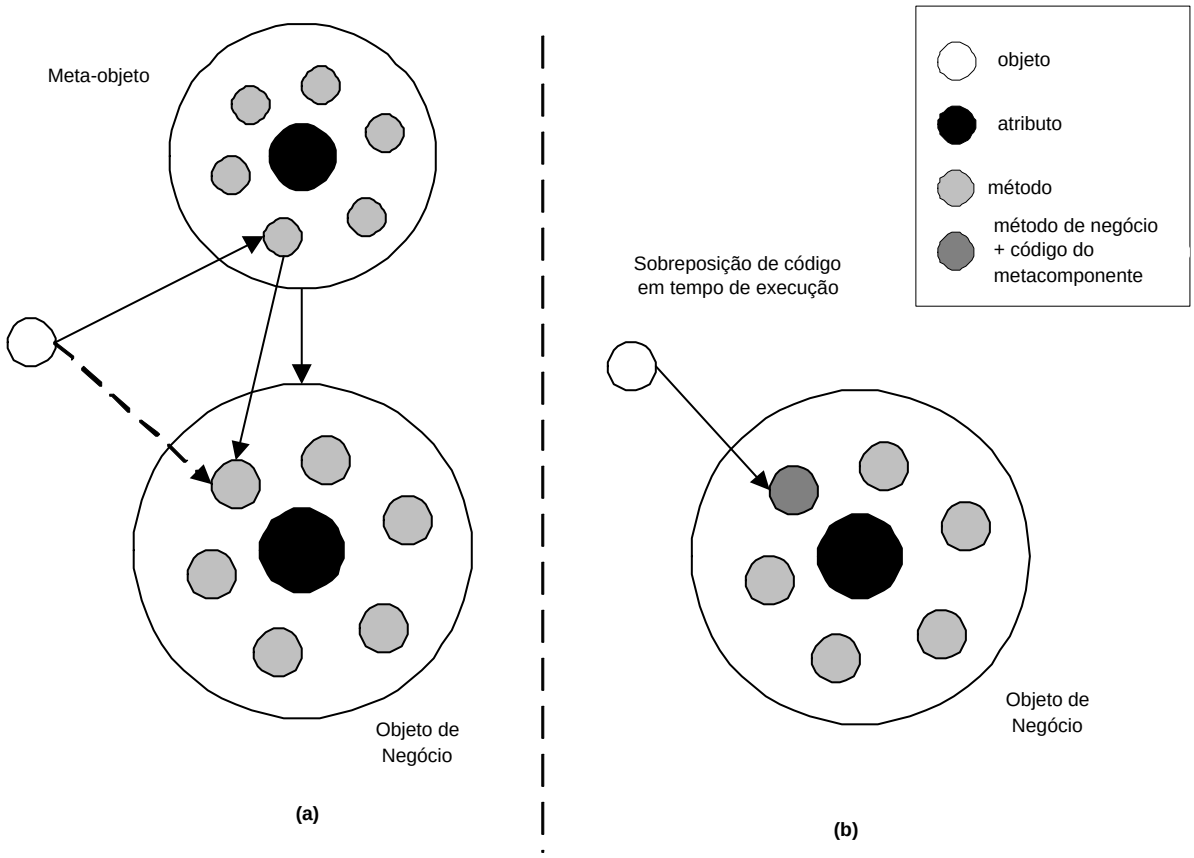

Figura 5.1: Reflexão: abordagem tradicional x proposta

disso, o código do metacomponente ficaria visível ao programador da aplicação.

Frente a essas dificuldades, decidiu-se incluir as funcionalidades inerentes ao nível meta não diretamente no código fonte da aplicação, mas em um código resultante de uma compilação ou interpretação. Assim, o código fonte da aplicação permanece independente do código do nível meta e as vantagens do uso de reflexão computacional continuam existindo em tempo de execução.

No caso de Java, por exemplo, este código resultante poderia ser o *bytecode*. Em nosso trabalho, entretanto, esse resultado é apresentado como uma árvore de programa (AST [1]). A árvore resultante é a integração entre a árvore que representa o negócio e a árvore que representa o metacomponente de persistência. Assim, conservamos a solução independente da plataforma de software.

A representação de código objeto em árvore foi explorada no ambiente descrito no capítulo anterior. O entrelaçamento entre árvores baseia-se no conceito de *weave* entre aspectos, proposto por programação orientada a aspectos (capítulo 3).

[1] Abstract Syntax Tree

A figura 5.2 resume nossa proposta de *weave* entre negócio e o aspecto de persistência. O grafo (a) representa o código objeto da aplicação de negócio, expresso em forma de árvore de programa. O grafo (b) mostra o código objeto de persistência, também em forma de árvore de programa. Em (c) são representados os pontos de entrelaçamento entre as árvores de negócio e persistência. E finalmente, (d) representa a árvore resultante ao final do processo de *weave* (entrelaçamento).

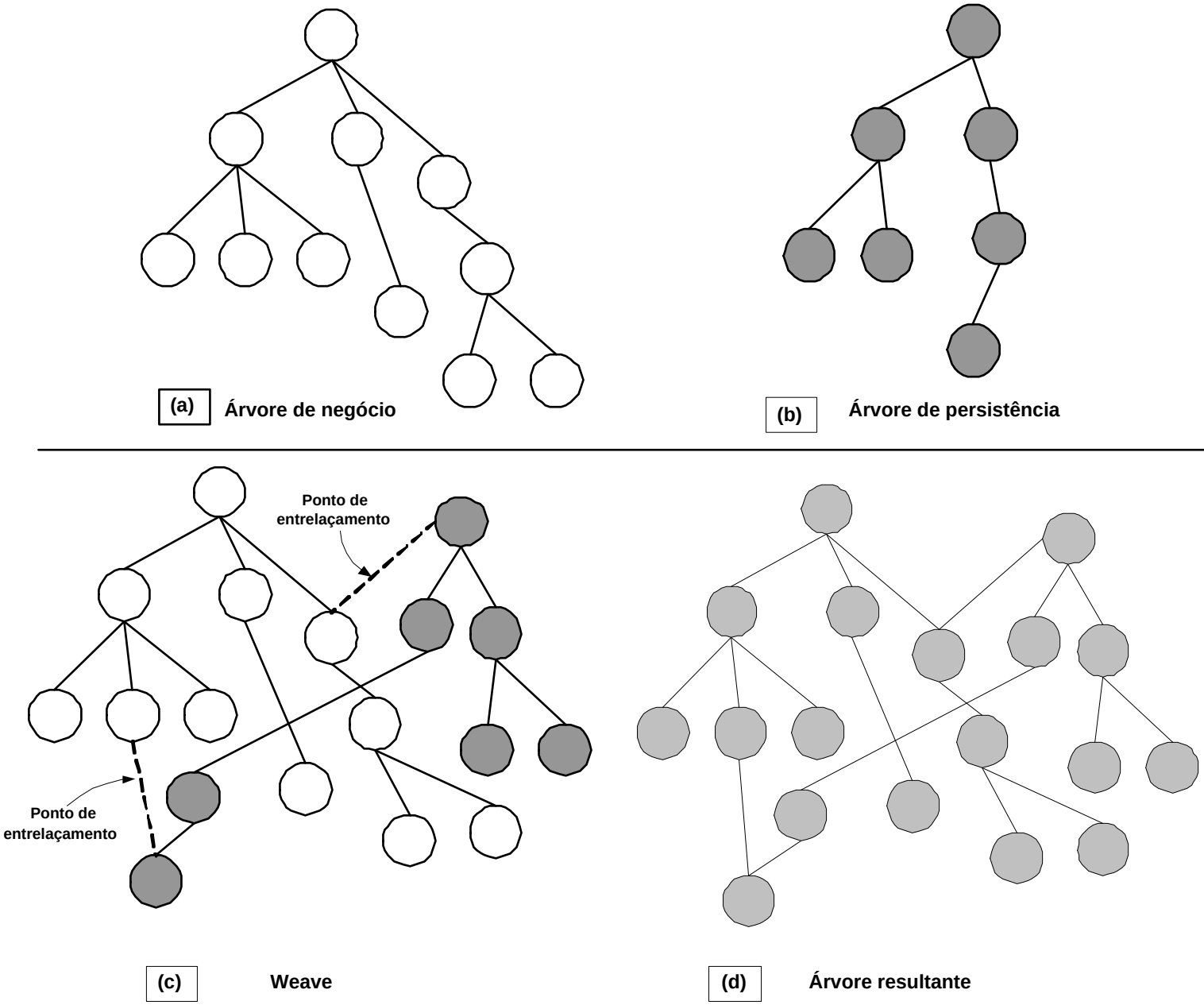

Figura 5.2: *Weave* entre negócio e persistência

O mecanismo de persistência evita o desvio típico de ambientes reflexivos para seguir o padrão definido na Virtuosi, onde evitar o desvio deverá im-

plicar em ganho de desempenho para outras funções do sistema. No caso de persistência, o tempo de desvio na execução é praticamente desprezível para o contexto do trabalho pelas seguintes razões:

- em um ambiente distribuído o tempo gasto em trocas de mensagens é relativamente muito maior.
- o tempo de acesso à memória estável (tipicamente disco) - operação intrínseca a qualquer mecanismo de persistência - é relativamente muito maior também.

Nenhuma verificação foi realizada para medir o desempenho da abordagem de uso de reflexão computacional sem desvio na execução. Isto deverá ser feito em trabalhos futuros, através do desenvolvimento de protótipos que possibilitem a comparação de desempenho entre implementações tradicionais e a forma proposta.

5.3 Políticas de Persistência

Segundo nossas observações, políticas de persistência podem oferecer maior flexibilidade ao usuário. Sugerimos algumas políticas com as quais o desenvolvedor pode definir qual a forma de persistência mais adequada às características de sua aplicação.

A definição da política deve acontecer em tempo de projeto das classes, pelo analista ou projetista do sistema. Deste modo, evita-se que essas decisões sejam tomadas pelo programador em fase de implementação, devendo o mesmo concentrar-se apenas em preocupações pertinentes ao negócio a ser implementado.

5.3.1 Persistência a Cada Alteração de Estado

Consiste em atualizar o estado do objeto no meio de armazenamento secundário toda vez que ele sofrer alteração, ou seja, a qualquer mudança no valor de um de seus atributos.

Tendo em mente nossa abordagem sobre reflexão computacional, considere o exemplo da figura 5.3. O lado esquerdo (a) representa a implementação do metacomponente de persistência, contendo os métodos (*readObject e writeObject*) específicos para esta política. O metacomponente está associado a uma classe de negócio (*Carro*), que possui um método (*modificaCor*) responsável

pela alteração do atributo cor. O lado direito (b) mostra a classe resultante que em tempo de execução contém os métodos de negócio devidamente adaptados para realizar, além de suas funções normais, as ações de persistência. Em outras palavras, durante a execução uma alteração no estado do objeto (mudança na cor do carro) provoca a atualização de seu estado no meio de armazenamento secundário. Isto é feito pelo próprio método *modificaCor*, que em tempo de execução tem o código de negócio e sentenças adicionais correspondentes a implementação do mecanismo de persistência.

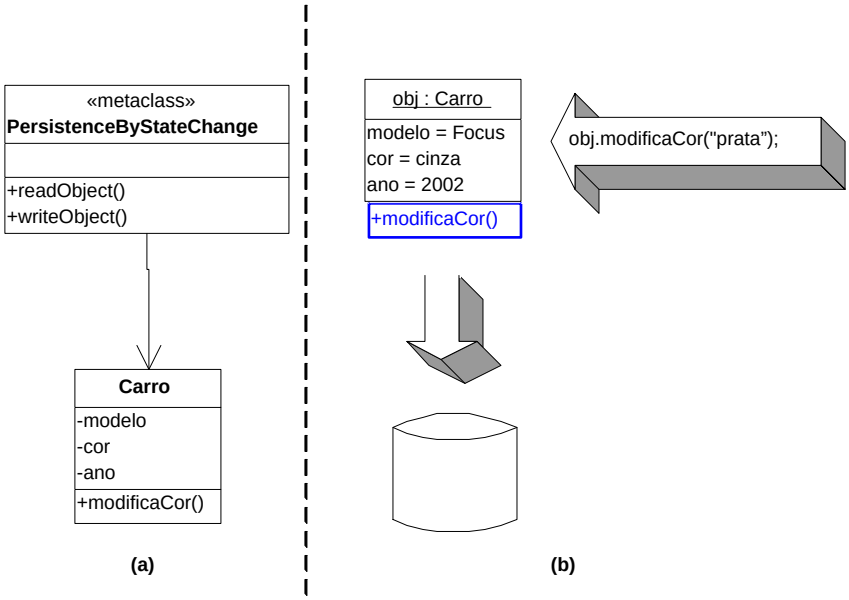

Figura 5.3: Exemplo de persistência a cada alteração de estado

Esta política tem a vantagem de ser simples de implementar. Entretanto, não é apropriada para sistemas que executam atualizações freqüêntes de seus dados. Poderia ocorrer queda de desempenho devido ao elevado número de acessos ao meio de armazenamento.

5.3.2 Persistência por Chamadas de Operação

Consiste em persistir o estado do objeto quando uma operação é solicitada por determinado número de vezes. Por exemplo, a cada três invocações para o método *saque()* o estado do objeto *Conta* deve ser persistido. Isto é ilustrado na figura 5.4. Em (a) pode-se observar o metacomponente de persistência, refletindo sobre a classe Conta. Em (b) um objeto da classe conta sofre invocações sobre um método que atualiza seu estado (método saque). Somente após atingir a quantidade de invocações determinada pelo usuário é que as atualizações são realizadas no meio de armazenamento permanente.

Esta política também é fácil de implementar. Para selecioná-la o usuário deve ter conhecimento sobre a freqüência com que essas operações são chamadas para evitar inconsistência prolongada entre os estados de objetos em execução e seus estados persistidos.

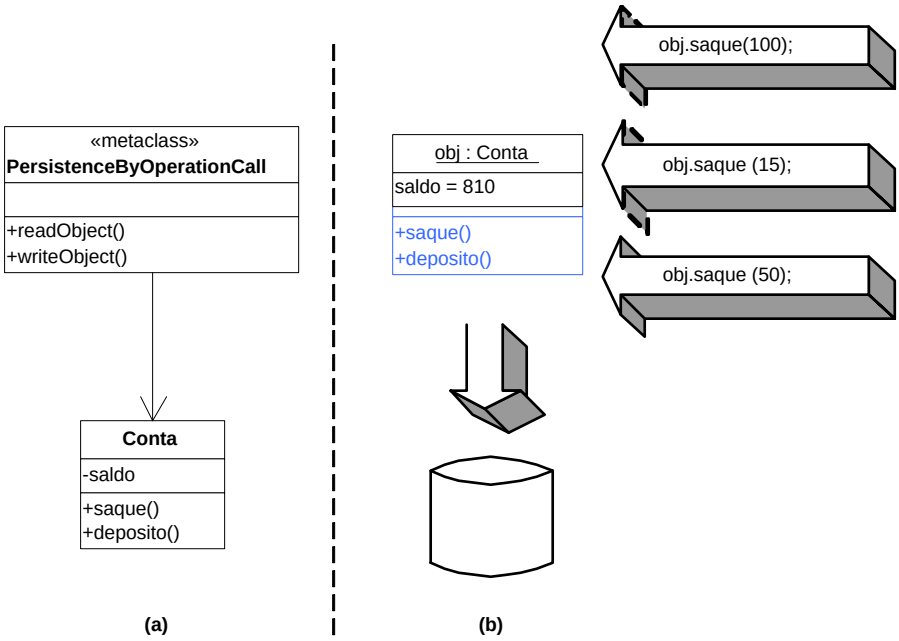

Figura 5.4: Exemplo de persistência por chamadas de operação

5.3.3 Persistência Por Período

Nesta política o estado dos objetos é persistido a cada intervalo de tempo. O usuário poderia por exemplo definir que os dados devem ser atualizados no meio de armazenamento secundário a cada 10 minutos. Quanto maior o período maior o desempenho, porém mais suscetível a falhas entre os intervalos. Portanto, a política é melhor aplicada quando o usuário conhece o intervalo de tempo mais adequado para a atualização de seus dados. Um exemplo típico de uso desta política são editores de texto, que salvam as alterações a cada período, evitando que o usuário se preocupe com esta tarefa enquanto concentra-se na elaboração de seu texto.

5.3.4 Persistência por Ação Atômica

Ações atômicas (transações atômicas) garantem que determinado conjunto de atualizações sejam realizadas integralmente ou, caso ocorra alguma falha

nenhuma atualização seja feita e o estado dos objetos envolvidos volte a ser exatamente como era antes da transação iniciar.

Transações atômicas podem ser bastante complexas de implementar, principalmente considerando ambientes distribuídos. Por esta razão muitas soluções tratam separadamente persistência e transação atômica. O padrão da OMG, por exemplo, propõe serviços (CORBA services) distintos para persistência e transação atômica, embora eles se relacionem. Da mesma forma, este trabalho propõe que o metacomponente de persistência interaja com outro metacomponente especializado em transações atômicas. As decisões sobre quando um objeto deve ser persistido deve ser responsabilidade do metacomponente de transações atômicas e o metacomponente de persistência deve apenas responder às suas solicitações, executando o que foi pedido e informando se a operação ocorreu com sucesso. Esta política pode ser útil para aplicações críticas, que requerem integridade dos dados.

5.3.5 Persistência em Horário Pré-Determinado

O estado do objeto é atualizado em um horário determinado pelo usuário. Por exemplo: às 18h30. Esta poderia ser uma solução para garantir a atualização dos dados de sistemas, antes de iniciar o backup diário.

5.3.6 Considerações

Algumas políticas poderiam fazer uso de um log, ou cache, cuja função seria manter os dados em uma área temporária, reduzindo o acesso aos meios de armazenamento secundários mais lentos. Além disso, poderiam garantir a integridade dos dados em caso de falha, desde que fosse acrescentado um algoritmo de recuperação de informações baseado neste log. Naturalmente isto aumentaria a complexidade de implementação dos componentes de persistência e levaria a estudos avançados sobre técnicas de tolerância a falhas [MLS93] e[MLS93], o que não é o objetivo deste trabalho.

5.4 Indicação de Classes Persistentes

Embora nosso objetivo seja oferecer persistência ortogonal (independente do tipo ou classe), em um modelo de negócios, pode não ser interessante persistir todas as classes. Sendo assim, o mecanismo de persistência deve prover um modo de indicar quais classes/atributos serão persistentes. Uma ferramenta gráfica parece ser a forma mais intuitiva para que o usuário possa expressar suas necessidades. O usuário deve ser preferencialmente o analista e/ou projetista do sistema e não o desenvolvedor.

5.5 Profundidade da Persistência

Algumas abordagens estudadas implementam persistência por alcançabilidade (ex: Serialização em Java, PJama, Drastic). Na persistência por alcançabilidade quando um objeto é persistido todos os objetos atingíveis através de referências também são persisidos.

Outras formas de profundidade mais elaboradas, em especial as apresentadas em [MRB98], parecem ser mais eficientes, porque permitem a definição de níveis de profundidade ou até mesmo a utilização de grafos para indicar os objetos persistentes. Sem dúvida essas técnicas proporcionam maior controle sobre o comportamento da persistência, porém são mais complexas de implementar exigindo ferramentas sofisticadas para interpretar as definições do usuário.

Como a indicação de classes persistentes também requer ferramenta gráfica semelhante, uma saída interessante é desenvolver apenas uma ferramenta para as duas finalidades. Poderia-se construir uma primeira versão simplificada, e melhorá-la de acordo com os resultados e necessidades observados em seu uso.

5.6 Mecanismo de Persistência

O mecanismo de persistência expressa as características descritas nas seções anteriores e segue alguns fundamentos usados pelo *design pattern Serializer*, visto no capítulo 2, seção 2.10.1.

5.6.1 Diagrama de Classes

O diagrama de classes (em UML [2]) da figura 5.5 representa de forma conceitual o mecanismo proposto para persistência de objetos.

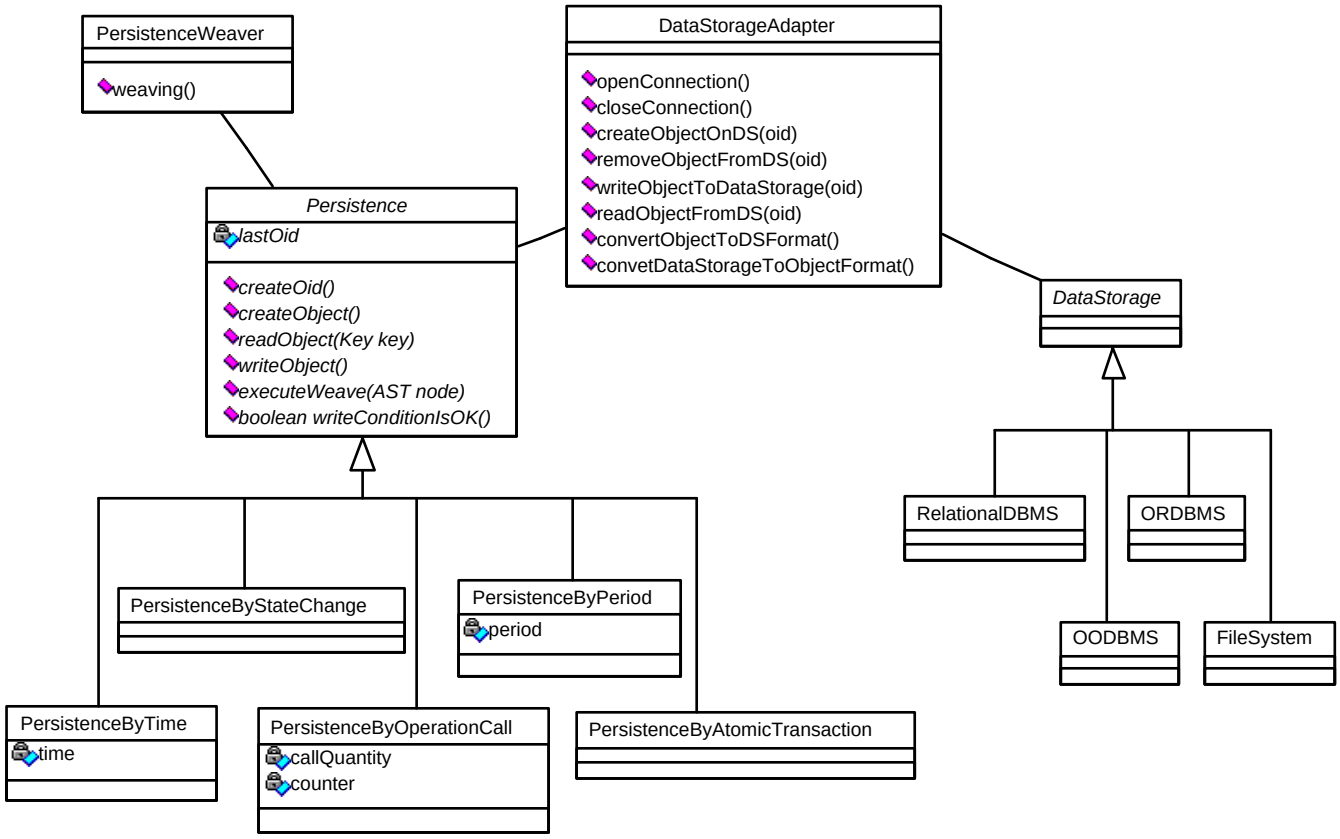

Figura 5.5: Mecanismo de persistência

Para melhor entendimento, as seções subsequentes descrevem os elementos apresentados no diagrama.

[2] Unified Modeling Language

5.6.2 Classe PersistenceWeaver

A classe *PersistenceWeaver* é responsável por gerar um código "executável", a partir do código de negócio e código de persistência.

Método weaving

Faz o entrelaçamento entre as árvores de negócio e de persistência.

5.6.3 Classe Persistence

A classe *Persistence* reúne todos os métodos comuns e define um método específico (writeConditionIsOK) que deve ser implementado por suas subclasses de acordo com cada política de persistência.

Método *createOid*

É o método responsável por gerar um identificador único para cada objeto persistente.

Método *createObject*

É o método responsável por criar o objeto no meio de armazenamento secundário. Ou seja, armazenar o estado do objeto assim que ele é instanciado.

Para desempenhar suas funções este método deve primeiramente criar um identificador único para o novo objeto e depois, solicitar a criação do objeto no meio de armazenamento específico invocando o método *createObjectOnDS* da classe *DataStorageAdapter*.

Método *readObject*

Este método faz a leitura do estado de um objeto no meio de armazenamento secundário, através de seu oid. Isto é feito pela invocação ao método *readObjectFromDataStorage* da classe *DataStorageAdapter*

Método *writeObject*

Este método faz a atualização dos dados do objeto no meio de armazenamento através de seu oid. Isto é feito pela invocação ao método *writeObjectToDataStorage* da classe *DataStorageAdapter*

Método *executeWeave*

Este método faz o *weaver* entre o código de negócio e o código de persistência, produzindo um executável. É invocado pelo método weaving e contém ações comuns a todas as políticas de persistência.

Método *writeConditionIsOK*

Executa as ações específicas de cada política de persistência.

5.6.4 Classe PersistenceByStateChange

Esta classe implementa a política de persistência a cada alteração de estado.

5.6.5 Classe PersistenceByPeriod

Esta classe implementa a política de persistência por período.

5.6.6 Classe PersistenceByTime

Esta classe implementa a política de persistência em horário pré-determinado.

5.6.7 Classe PersistenceByOperationCall

Esta classe implementa a política de persistência por chamadas de operação.

5.6.8 Classe PersistenceByAtomicTransaction

Esta classe implementa a política de persistência por ação atômica.

5.6.9 Classe DataStorageAdapter

DataStorageAdapter é uma classe que define um objeto de fronteira, cujo objetivo é encapsular a comunicação entre o componente de persistência e um meio de armazenamento específico.

Método *createObjectOnDS*

Este método cria o objeto no meio de armazenamento específico. Por exemplo, se for um banco de dados relacional, ele insere um novo registro contendo o estado do objeto e seu identificador.

A primeira ação deste método é converter os dados do objeto para o formato esperado pelo meio de armazenamento específico. Isto é feito por invocar o método *convertObjectToDataStorage*. Em seguida, pode criar o objeto no meio de armazenamento específico, salvando os dados e o identificador único do objeto.

Método *removeObjectOnDS*

Este método remove o objeto do meio de armazenamento específico. Por exemplo, se for um banco de dados relacional, ele apaga o registro correspondente.

Método *readObjectFromDataStorage*

O método faz a leitura do estado do objeto no meio de armazenamento específico, através de seu oid. No caso de banco relacional, por exemplo ele localiza o registro onde foi salvo o objeto e lê os dados (cláusula *select*).

Considerando que o resultado da leitura é retornado no formato específico do meio de armazenamento secundário, é preciso converter para o formato de objeto. Isto é feito invocando-se o método *convertDataStorageToObjectFormat*.

Método *writeObjectToDataStorage*

O método faz a atualização do estado do objeto no meio de armazenamento específico, através de seu oid. No caso de banco relacional, por exemplo, ele localiza o registro através de seu oid e atualiza os dados (cláusula *update*). Entretanto, antes de atualizar os dados, é preciso convertê-los para o formato esperado pelo meio de armazenamento secundário. Isto é feito por invocar o método *convertObjectToDataStorage*.

método *convertDataStorageToObjectFormat*

Converte dados que estão no formato do meio de armazenamento secundário, para o formato de objeto.

método *convertObjectToDataStorage*

Converte dados de um objeto para o formato esperado pelo meio de armazenamento secundário.

5.6.10 Classe DataStorage

Esta classe representa os meios de armazenamento que podem ser utilizados pelo mecanismo de persistência, através da implementação de adaptadores.

5.6.11 Classe RelationalDBMS

Representa sistema gerenciador de banco de dados relacional.

5.6.12 Classe OODBMS

Representa sistema gerenciador de banco de dados orientado a objetos.

5.6.13 Classe OODBMS

Representa sistema gerenciador de banco de dados objeto-relacional.

5.6.14 Classe FileSystem

Representa arquivos comuns de sistemas, para armazenamento de dados.

5.7 Passos para Utilização

Para utilizar o componente de persistência os seguintes passos devem ser seguidos: configuração de preferências e *weave* entre a árvore de negócio e a árvore de persistência.

5.7.1 Configuração de Preferências

Uma característica desejável em nossa abordagem é que persistência seja o mais transparente possível. Entretanto, é preciso que haja um mínimo de interação do usuário, para indicar suas preferências no que diz respeito a persistência de seu modelo de negócio. Então, é preciso escolher a política desejada e indicar as classes persistentes e a profundidade da persistência.

5.7.2 *Weave* da Árvore de Negócio e Árvore de Persistência

Conforme dito na seção 8.2, a reflexão do componente de persistência sobre os objetos de negócio ocorre através da combinação entre a árvore que representa o negócio e a árvore que representa o código de persistência. Neste

ponto, o usuário deve indicar que o *weave* pode ser executado, produzindo classes de negócio que se auto-persistem, conforme as definições de preferência. Após concluído o *weave* o sistema está apto a ser executado.

5.7.3 Alteração no Componente de Persistência

Embora não seja muito comum pode ocorrer alguma alteração ou upgrade do componente de persistência. Neste caso, não haveria necessidade de alterar as configurações de preferências, a não ser que uma nova política de persistência tenha sido criada. Mas, na maioria dos casos basta realizar novamente o *weave*.

Capítulo 6

Exemplos de Uso

O objetivo deste capítulo é apresentar exemplos que demonstrem o uso do metacomponente de persistência.

Como descrito na seção 5.2 nossa abordagem utiliza sobreposição do código de persistência nos objetos de negócio, em tempo de execução, que são representados por árvores de programas. Então, os exemplos deste capítulo serão apresentados em forma de código fonte Java e no formato de árvores de programa.

6.1 Árvores de Programa

As árvores de programas representam códigos que podem ser interpretados/executados pela máquina virtual. Entretanto, exibir todos os exemplos neste formato, que é bastante extenso, dificultaria o entendimento. Então, optou-se por apresentar códigos de negócio e do componente de persistência em Java, e o código resultante do *weave*, em Java e em formato de árvore. É bom lembrar que todas as vezes que mencionarmos código resultante do *weave* estamos nos referindo ao código "executável", mesmo que ele esteja exemplificado em formato de código fonte.

6.2 Entrelaçamento

A figura 6.1 é um exemplo de código que realiza o processo de entrelaçamento entre negócio e persistência. A árvore de negócio é percorrida a procura de classes persistentes. Para cada classe localizada é executado o método *executeWeave* (figura 6.2), contendo ações comuns a serem realizadas para todas as políticas de persistência. Ele identifica os pontos do código de negócio onde

devem ocorrer atividades inerentes ao mecanismo de persistência, tais como: criação de um identificador único para cada objeto persistente (nas linhas 7 a 11); criação de um objeto no meio de armazenamento (nas linhas 13 a 15); atualização (nas linhas 21 a 23) e recuperação de seu estado (nas linhas 17 a 19). Uma vez localizados estes pontos, é preciso entrelaçar o código de persistência que executa tais atividades, de acordo com a política escolhida.

```
1 public class  PersistenceWeaver
2 {
3    void weaving()
4    {
5       noAtual = arvoreNegocio.getRaiz();
6
7       // enquanto não atingir fim da árvore de negócio
8       while (noAtual != null)
9       {
10          if (noAtual.isPersistentClass()) // Para cada classe persistente
11          {
12             pers.executeWeave(noAtual);
14          }
15          noAtual = getNextNode();
16       }
17    }
18 }
```

Figura 6.1: Exemplo de *Weaver* para Persistência

As ações particulares a cada política de persistência são embutidas na implementação do método writeConditionIsOK(). Um exemplo pode ser visto na figura 6.3, onde as políticas de persistência a cada alteração de estado (linhas 1 a 7) e por chamada de operação (linhas 9 a 16) apresentam condições distintas para executar a atualização no meio de armazenamento. A primeira executa a atualização dos dados imediatamente após a alteração de estado do objeto. Então, o código de persistência para atualização é executado após cada sentença de alteração de estado encontrada no código de negócio. A segunda política realiza a atualização somente quando a condição de quantidades de chamadas ao método é satisfeita.

6.3 Código de Negócio - Pontos de Inserção

Como mencionado anteriormente é preciso localizar no código de negócio os pontos onde devem ser inseridos os códigos de persistência. Nas próximas subseções são detalhados os critérios usados para criação, atualização e leitura dos objetos no meio de armazenamento.

Considere o código de negócio da figura 6.4 e sua árvore correspondente na figura 6.5.

A geração de um identificador único para cada novo objeto persistente é feita através da criação de um novo atributo (oid) para cada classe de negócio persistente. Na árvore de programa, é inserido um nó para o atributo e a ele é associado o retorno do método que gera um novo valor para oid (método createOid). Isto pode ser observado na figura 6.6, que apresenta o atributo *oid* em destaque.

A figuras 6.7 e 6.8 mostram o código e árvore de programa que exemplificam a criação, atualização e recuperação do estado de objetos. Eles representam a parte comum a todas as políticas de persistência. A diferença entre uma política e outra é implementada pelo método *executeWeave*, como discutido anteriormente.

6.3.1 Criação de um Objeto no Meio de Armazenamento

Toda vez que um novo objeto é criado deve-se preparar o meio de armazenamento secundário e salvar seu estado atual. O ponto mais adequado para inserir o código de criação é no método construtor do objeto.

Considere novamente o código de negócio da figura 6.4 e o código de persistência da figura 6.7, responsável pela criação e escrita de um objeto em um meio de armazenamento secundário (linhas 3 a 14). O resultado do *weave* entre os dois códigos é representado na figura 6.9 que mostra como ficou a classe de negócio após sofrer reflexão e ser adaptada para se auto-persistir. Novamente, considere que este código resultante não é um programa fonte, só foi representado desta forma para fins de entendimento.

A figura 6.10 mostra o mesmo exemplo da figura 6.9, porém apresenta detalhes da criação do objeto (linhas 18 a 37). Neste detalhamento pode-se observar a introspeção realizada para obter metainformações do objeto, como o nome da classe e seus atributos com o objetivo de montar o comando que acessa o banco de dados relacional. Os comandos que obtém as metainformações do objeto representam a busca de informações contidas na árvore de programa em tempo de execução.

A árvore corresponde ao resultante do *weave* é mostrado na figura 6.11. Os elementos com preenchimento de fundo, representam o código de persistência e os demais correspondem ao código de negócio. A árvore foi representada de forma resumida, para facilitar o entendimento. O objetivo do exemplo não é traduzir todo o código em formato de árvore, mas demonstrar como ocorre o *weave* entre negócio e persistência.

6.3.2 Atualização do Estado do Objeto

As alterações de estado de um objeto precisam ser refletidas no meio de armazenamento secundário. Existem duas formas de alterar o estado de um objeto. Uma delas é modificando o valor de um ou mais atributos. A outra é fazendo com que um atributo, que é uma referência, passe a referenciar outro objeto. No metamodelo, a alteração de valor é feita pela modificação de um *DataBlock* e a alteração de uma referência através de *BindStatement*.

Na alteração de referência os locais do código que atualizam o estado de objetos podem ser identificados procurando-se por *BindStatements* cujo *target* seja uma referência para um atributo. Ou seja, se um atributo deixa de referenciar um objeto para referenciar outro, está caracterizada uma mudança de estado, que deve ser persistida. Então, este é um ponto onde precisa-se acrescentar o codigo de persistência responsável pela atualização do objeto no meio de armazenamento.

Considere novamente a classe Conta (figura 6.4) como código de negócio, sendo que o ponto onde ocorre alteração de estado está representado na linha 13 (*bindStatement: this.saldo = novoSaldo;*) no método *setSaldo*.

O código do metacomponente de persistência responsável pela atualização de um objeto no meio de armazenamento é mostrado na figura 6.7 entre as linhas 15 e 29. E o código resultante do *weave* entre negócio e persistência na figura 6.12, tendo sua árvore representada em 6.13.

6.3.3 Recuperação do Estado do Objeto

Todo objeto persistente pode ter seu estado recuperado. Para esta finalidade, o mecanismo deverá prover um construtor especial para recuperação, cujo objetivo é localizar o objeto persistido anteriormente, através de chaves de pesquisas semelhantes ao conceito de chaves primárias, usado em banco de dados relacional. O metamodelo indica quais atributos compõe a chave para localizar um objeto. Então o usuário deve implementar um tipo especial de construtor denominado construtor de recuperação, como em [Cal96]. Este

construtor cria um novo objeto a partir dos dados armazenados anteriormente e deve receber como parâmetro a chave para localizar o objeto persistido.

Considere novamente a classe Conta (figura 6.4), como código de negócio. O código do metacomponente de persistência, responsável pela recuperação do estado de um objeto no meio de armazenamento, é mostrado na figura 6.7, entre as linhas 30 e 41. E o *weave* entre negócio e persistência, no código da figura 6.14, tendo sua árvore representada em 6.15.

```
1 public class Persistence
2 {
3     void executeWeave (Node noClasseNegocio)
4     {
5        Node noInsercao = noClasseNegocio;
6
7        // entrelaça código para gerar identificador único para novo objeto
8        retornoMetodo = noInsercao.addMethod (arvorePersistencia,
9                                                  "createtOid");
10       // cria novo atributo para identificador único
11       noInsercao.addAttribute("oid", Double, retornoMetodo);
12
13       // entrelaça código para criar objeto no meio de armazenamento
14       noInsercao = noClasseNegocio.getNodeConstructor();
15       noInsercao.addMethod (arvorePersistencia, "createObject");
16
17       // entrelaça código para ler objeto no meio de armazenamento
18       noInsercao = noClasseNegocio.getNodeConstructorIncarnation();
19       noInsercao.addMethod (arvorePersistencia, "readObject");
20
21       // entrelaça código para atualizar objeto no meio de armazenamento
22       noInsercao = noClasseNegocio.getNodeStatementsChange();
23       noClasseNegocio.addMethod (arvorePersistencia, "writeObject");
24    }
25
26    // gera identificador único para novo objeto
27    Double createOid()
28    {
29       return ultimoOid++;
30   }
31    ...
32 }
```

Figura 6.2: Exemplo de código de *weave* comum a todas as políticas

```
1 public class PersistenceByStateChange extends Persistence
2 {
3      boolean writeConditionIsOK()
4      {
5          return true;
6      }
7 }
8
9 public class PersistenceByOperationCall extends Persistence
10 {
11    boolean writeConditionIsOK()
12    {
13        if (counter == callQuantity)
14           return true;
15        return false;
16    }
17 }
```

Figura 6.3: Exemplo de condições de escrita para políticas *PersistenceByStateChange* e *PersistenceByOperationCall*

```
1 public class Conta
2 {
3    Double numero;
4    Double  saldo;
5
6    Conta (Double num, Double sdo)
7    {
8       this.numero = num;
9       this.saldo = sdo;
10   }
11   public void setSaldo (Double novoSaldo)
12   {
13      this.saldo = novoSaldo;
14   }
15 }
```

Figura 6.4: Exemplo de código de negócio: classe Conta

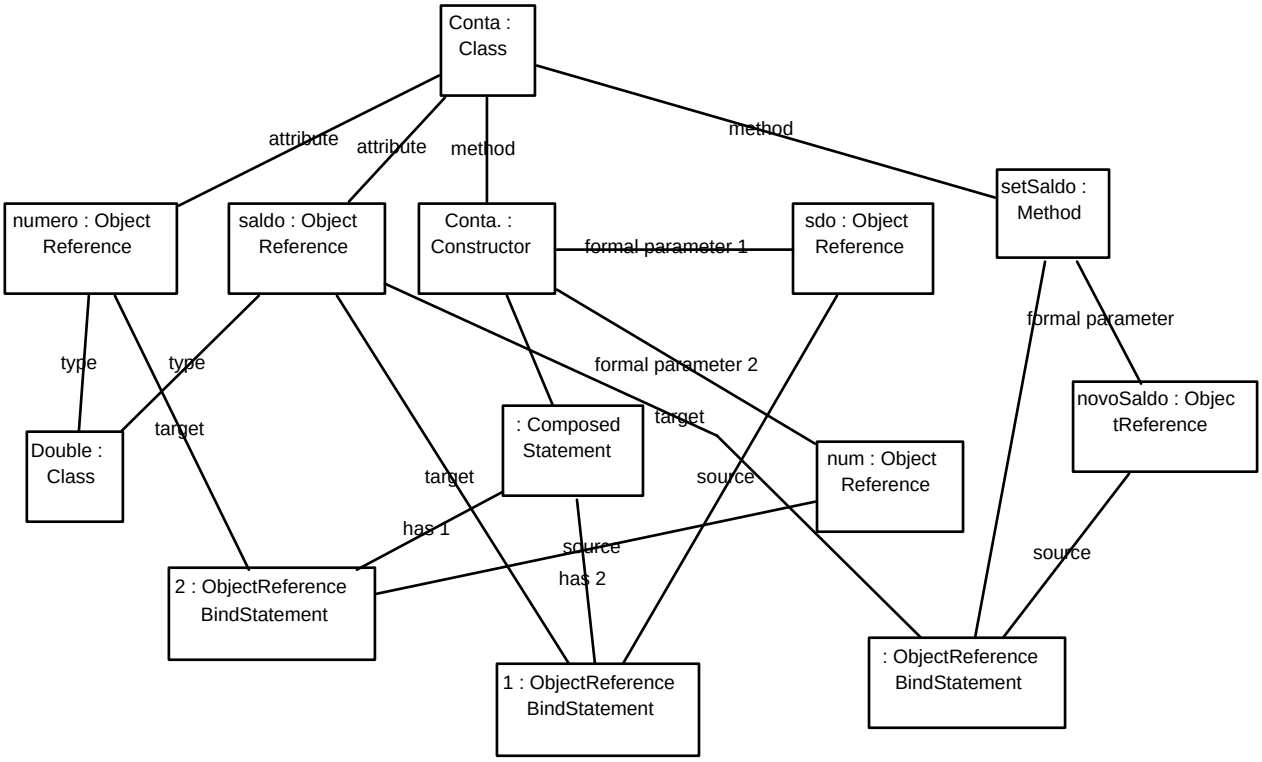

Figura 6.5: Árvore correspondente ao código da Classe Conta

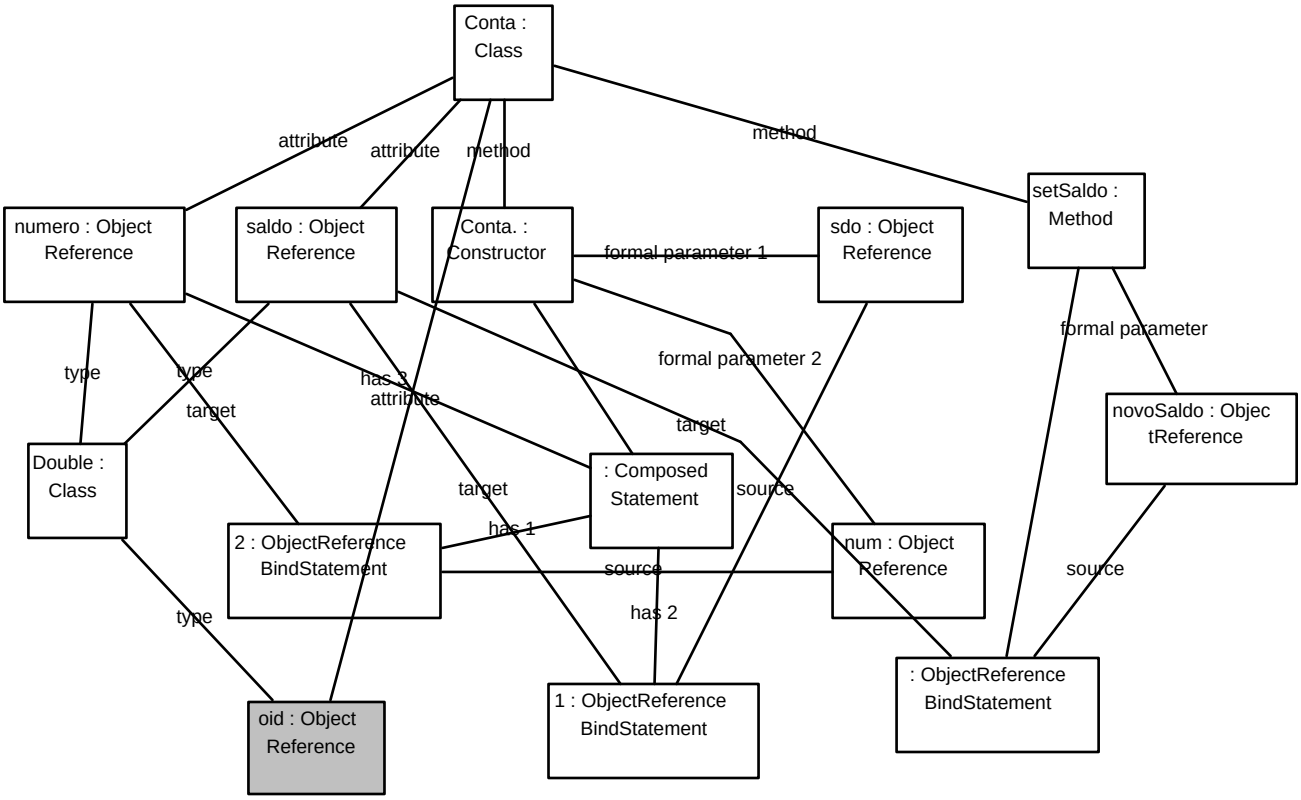

Figura 6.6: Exemplo de criação do Oid

```
1 public class Persistence
2 { ...
3     void createObject()
4     {
5         // estabelece conexão com meio de armazenamento
6         DataStorageAdapter ds = new RelationalDBMS();
7         ds.openConnection();
9         // executa criação do objeto
10        ds.createObjectOnDS(oid);
12        // encerra conexão com meio de armazenamento
13        ds.closeConnection();
14    }
15    void writeObject()
16    {
17        if (writeConditionisOK()) {
19           // estabelece conexão com meio de armazenamento
20           DataStorageAdapter ds = new RelationalDBMS();
21           ds.openConnection();
23           // executa atualização do objeto
24           ds.writeObjectToDS(oid);
26           // encerra conexão com meio de armazenamento
27           ds.closeConnection();
28        }
29    }
30    Double readObject(Key key)
31    {
32       // estabelece conexão com meio de armazenamento
33       DataStorageAdapter ds = new RelationalDBMS();
34       ds.openConnection();
36       // executa recuperação do estado do objeto
37       oid = ds.readObjectToDS(key);
39       // encerra conexão com meio de armazenamento
40       ds.closeConnection();
41    }
42 }
```

Figura 6.7: Exemplo de código de persistência

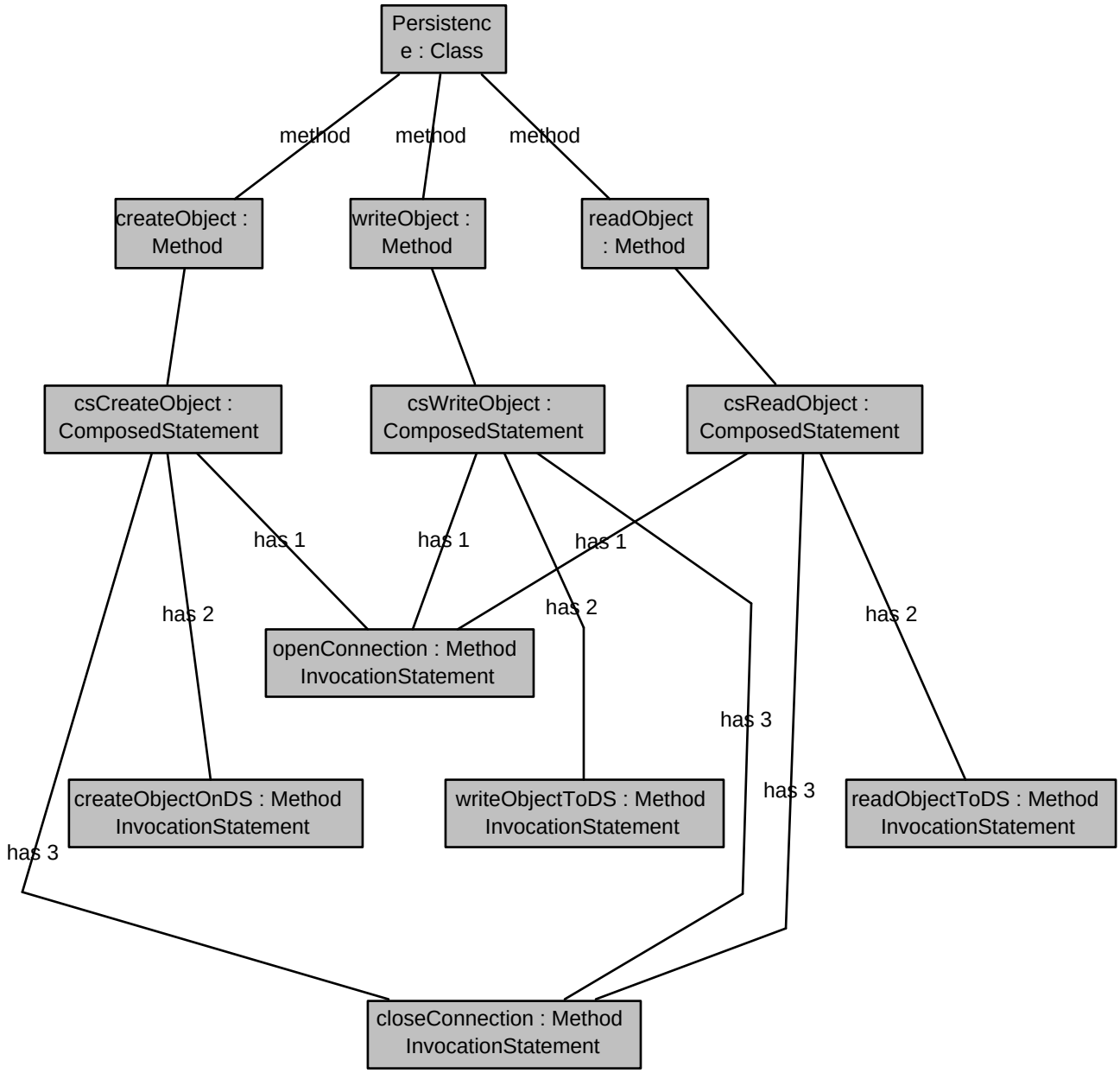

Figura 6.8: Exemplo de árvore de persistência

```
1 public class Conta
2 {
3    Double numero;
4    Double  saldo;
5    Double oid; // é criado um atributo identificador para o objeto
6
7    Conta (Double num, Double sdo)
8    {
9       this.numero = num;
10      this.saldo = sdo;
11
12      // gera identificador único para novo objeto
13      oid  = pers.createOid ();
14
15      // neste ponto é inserido o código correspondente
16      // ao método createObject()
17
18      // estabelece conexão com meio de armazenamento
19      DataStorageAdapter ds = new RelationalDBMS();
20      ds.openConnection();
21
22      // executa criação do objeto
23      ds.createObjectOnDS(oid);
24
25      // encerra conexão com meio de armazenamento
26      ds.closeConnection();
27    }
28 }
```

Figura 6.9: Resultado do *weave* entre negócio e persistência (*createObjectStore*)

```
1 public class Conta
2 {
3    Double numero;
4    Double  saldo;
5    Double oid; // é criado um atributo identificador para o objeto
6
7    Conta (Double num, Double sdo)
8    {
9       this.numero = num;
10      this.saldo = sdo;
11
12      // gera identificador único para novo objeto
13      oid  = pers.createOid ();
14
15      // neste ponto é inserido o código correspondente
16      // ao método createObject()
17
18      // estabelece conexão com meio de armazenamento
19      try
20      {
21          Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
22          conexaoJDBC = DriverManager.getConnection
23                   ("jdbc:odbc:teste", usuario, senha);
24          stmt = conexaoJDBC.createStatement();
25      }
26      // executa criação do objeto
27      nomeAtt = pers.getAttributesName(this);
28
29      String comando = "insert into TB_" + pers.getClassName(this) +
30        " (" + nomeAtt(1) + ", " + nomeAtt(2) + ") values
31        (" + num + "," + sdo + ") where oid = " + oid;
32
33      try
34      { stmt.executeQuery(comando); }
35     ...
36 }
```

Figura 6.10: Resultado do *weave* entre negócio e persistência (*createObjectStore* detalhado)

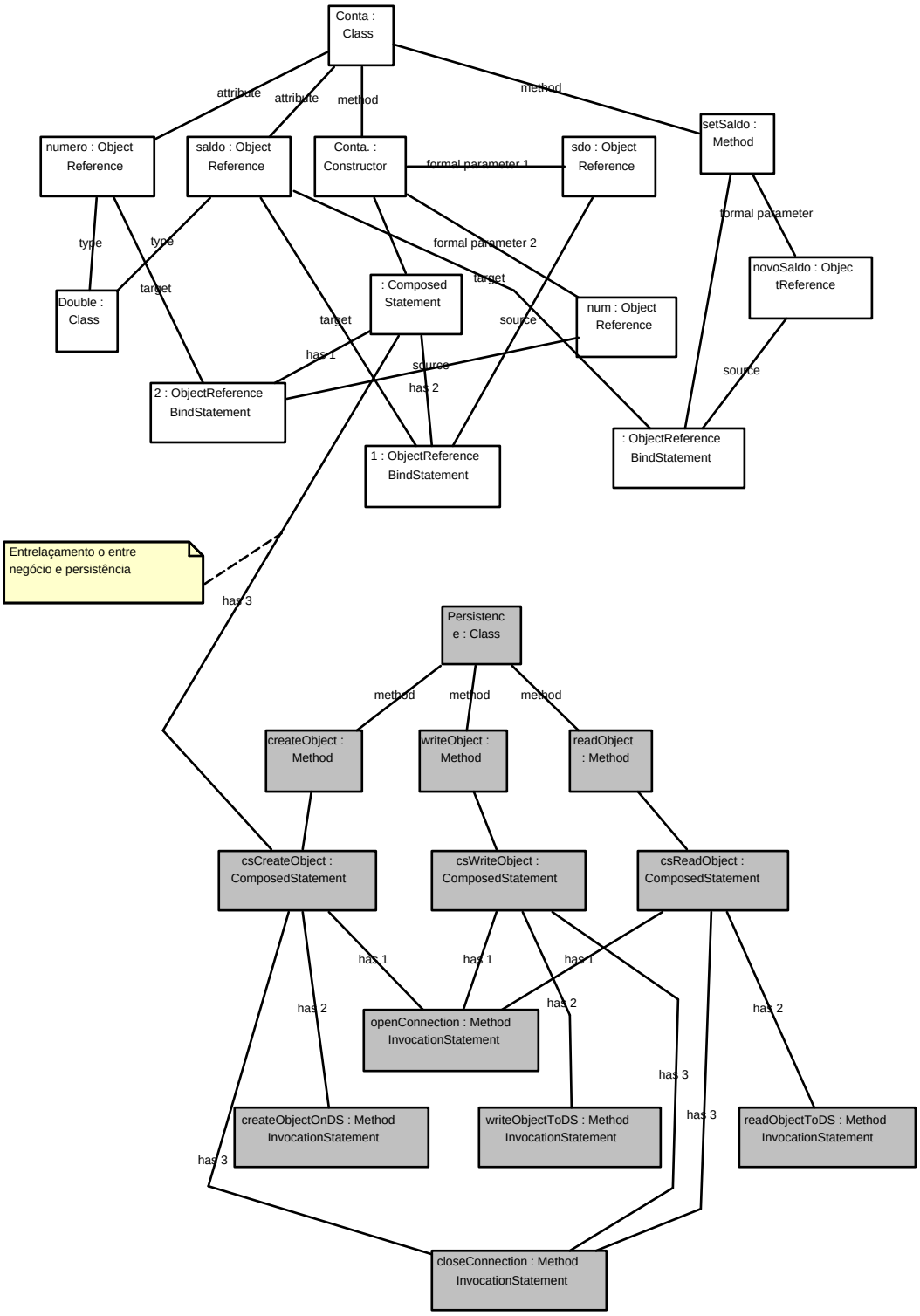

Figura 6.11: Árvore correspondente ao código resultante do *weave*

```
1 public class Conta
2 {
3    Double numero;
4    Double  saldo;
5    Double oid;
6
7    Conta (Double num, Double sdo)
8    {
9       this.numero = num;
10      this.saldo = sdo;
11      ...
12   }
13   public void setSaldo (Double novoSaldo)
14   {
15      this.saldo = novoSaldo;
16
17      // neste ponto é inserido o código correspondente
18      // ao método writeObject
19
20      if (writeConditionisOK())
21      {
22           // estabelece conexão com meio de armazenamento
23           DataStorageAdapter ds = new RelationalDBMS();
24           ds.openConnection();
25
26           // executa atualização do objeto
27           ds.writeObjectToDS(oid);
28
29           // encerra conexão com meio de armazenamento
30           ds.closeConnection();
31      }
32   }
```

Figura 6.12: Resultado do *weave* entre negócio e persistência(*writeObject*)

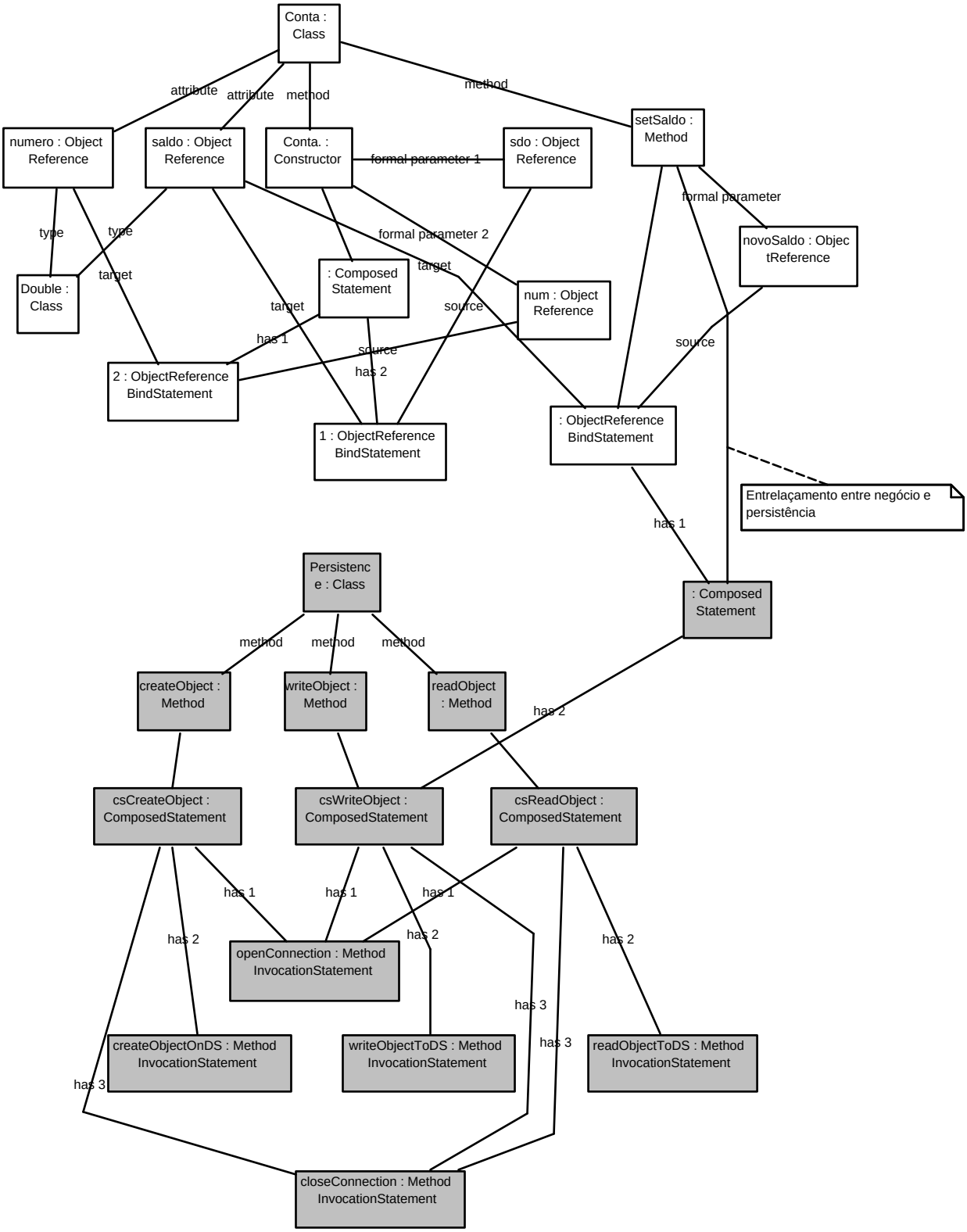

Figura 6.13: Árvore correspondente ao resultado do *weave* para atualização

```
1 public class Conta
2 {
3    Double numero;
4    Double  saldo;
5    Double oid;
6
7    Conta (Double num, Double sdo)
8    {
9      ...
10   }
11
12   // Construtor especial para recuperação do estado
13   Conta (Double num)
14   {
15      this.numero = num;
16
17      // estabelece conexão com meio de armazenamento
18      DataStorageAdapter ds = new RelationalDBMS();
19      ds.openConnection();
20      // executa recuperação do objeto
21      oid = ds.readObjectToDS(num);
22      // encerra conexão com meio de armazenamento
23      ds.closeConnection();
24   }
25 }
```

Figura 6.14: Resultado do *weave* entre negócio e persistência(*readObject*)

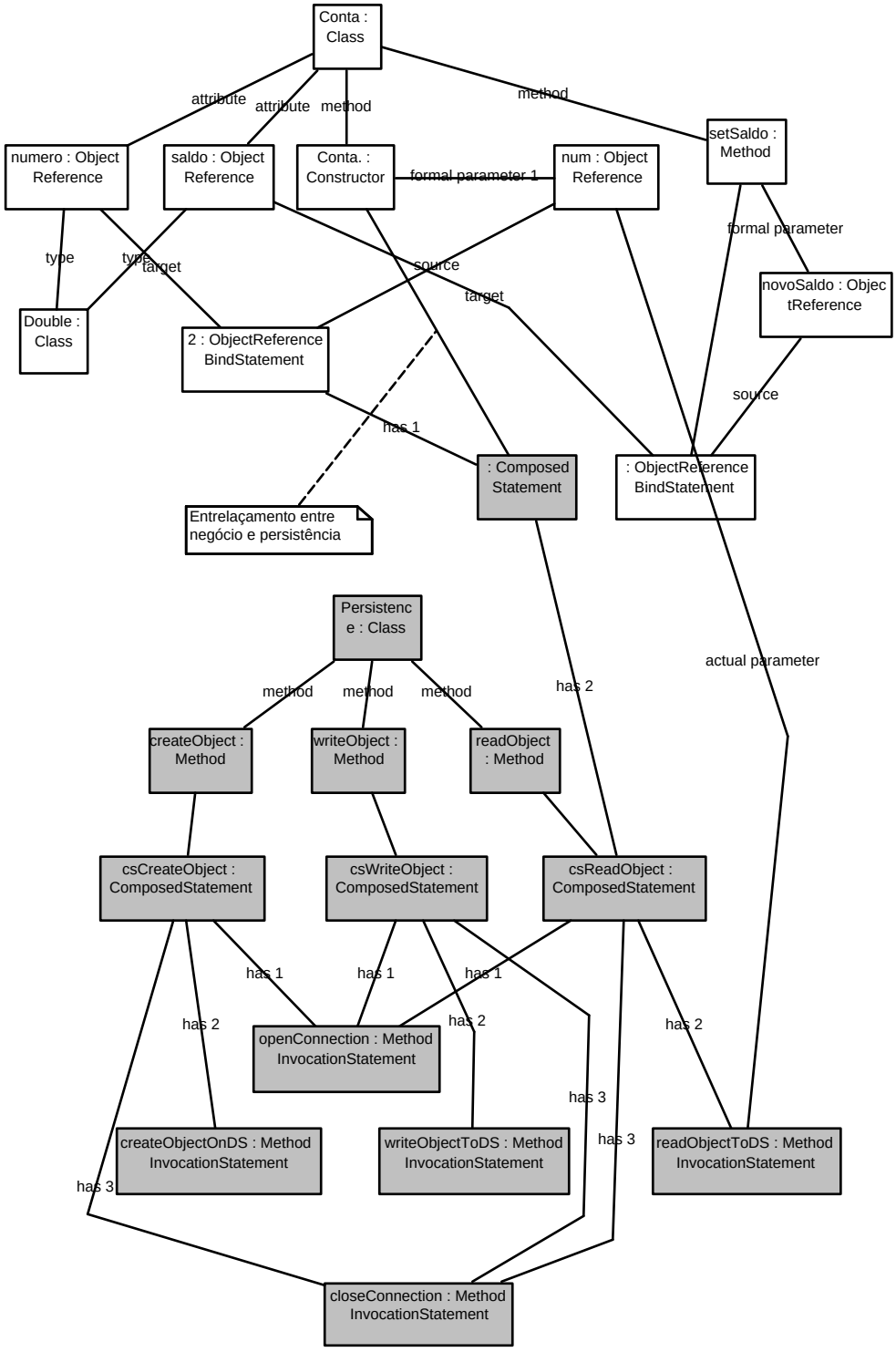

Figura 6.15: Árvore correspondente ao resultado do *weave* para recuperação

Capítulo 7

Conclusões e Trabalhos Futuros

O mecanismo de persistência proposto possibilita o desenvolvimento de aplicações com maior grau de produtividade porque libera o programador da implementação e manutenção dessas atividades, permitindo que ele se concentre em atividades de negócio da aplicação. Isto se deve ao fato de adotar características recomendadas pelos projetos estudados no capítulo 2.

Todos os aspectos do mecanismo foram definidos buscando-se flexibilidade, transparência e facilidade de uso. No que diz respeito as diversas plataformas de armazenamento, concluiu-se que a idéia de usar adaptadores para permitir independência dessas plataformas realmente condiz com as necessidades identificadas pelo estudo realizado e sintetizado no capítulo 2.

O uso de reflexão computacional, de forma diferenciada das abordagens tradicionais, contribuiu para que a persistência fosse provida da forma mais transparente possível. Além disso, propõe o uso de reflexão computacional sem o desvio na execução presente em vários projetos estudados no capítulo 3. Mesmo comparado a soluções que também evitam desvios nosso mecanismo apresenta vantagens. Em *Slim Binaries* [FK97], reflexão é obtida na compilação do código fonte e geração de *bytecodes*. Em Javassist[Chi00], o código fonte é preservado e as alterações para reflexão são introduzidas diretamente nos *bytecodes*. Nós propomos o entrelaçamento entre árvores de programa do negócio e de persistência. Esta forma de representação(árvore de programa) é mais adequada que os *bytecodes* porque preserva todas as informações semânticas em tempo de execução.

Reflexão computacional possibilita executar alterações na persistência de objetos de forma dinâmica, permitindo alterar a política de persistência de um certo objeto (ou classe) em tempo de execução, tornar persistente um objeto transiente, e vice-versa.

O mecanismo de persistência incorporado ao ambiente de execução[Cal00],

fornece portabilidade às aplicações, por ser baseado em máquinas virtuais, ambiente distribuído e orientado a objetos, como visto no capítulo 7.

A principal contribuição do trabalho é propor um mecanismo de persistência que utiliza técnicas para aumentar o grau de modularidade das aplicações, interferindo o mínimo possível no código de negócio para facilitar o re-uso de funcionalidades e futuras manutenções. Isto é obtido através do uso de reflexão computacional sem desvios, aplicando conceitos de entrelaçamento(weaving) entre árvores de programas, semelhantes aos usados em programação orientada a aspectos.

As políticas de persistência propostas por este trabalho poderiam ser oferecidas de forma mais flexível, permitindo a combinação de várias delas, atuando no mesmo sistema. Isto iria requerer um maior controle sobre seu uso. Por exemplo, poderia-se definir uma política para cada unidade do sistema, tais como classe, método e atributo.

Tolerância a falhas é um requisito importante em sistemas de missão crítica, e poderia ser introduzido no mecanismo de persistência, para aumentar a robustez do sistema. Uma forma de implementar tolerância a falhas é através de réplicas de objetos persistentes, de acordo com estudos realizados em Arjuna.

Uma ferramenta gráfica para definição de classes persistentes, poderia ter uma forma automática para carregar as classes de negócios. Uma forma de implementar esta facilidade, poderia ser interagindo com ferramentas case, ou linguagens de definições padrão.

O mecanismo de persistência apresentado por este trabalho baseia-se em árvores de programas e não está atrelado a nenhuma linguagem de programação. Então, uma possível extensão para este trabalho seria a definição de um mecanismo para interpretar árvores de programas e gerar código objeto para linguagens de programação específicas.

Cada meio de armazenamento requer formatação específica para salvar e recuperar dados. Ou seja, é possível definir e implementar adaptadores responsáveis pelo mapeamento entre formatos de objetos e diversos meios de armazenamentos, tais como bancos de dados relacionais, bancos de dados objeto-relacional e outros. Uma característica interessante seria a criacão de adaptadores genéricos, que não incluíssem dependências específicas de fornecedores.

Mecanismos de persistência têm forte vínculo com ciclos de vida de objetos porque precisam controlar objetos que não são mais referenciados. Normalmente esses objetos são eliminados por *garbage collection*. Para que os objetos também sejam removidos do meio de armazenamento, é preciso definir uma

forma de interação entre esse serviço e o mecanismo de persistência.

A construção de protótipos e experimentos sobre a forma de reflexão computacional proposta pode fundamentar uma análise mais profunda a respeito do impacto no desempenho das aplicações, principalmente se comparada ao seu uso tradicional, contendo desvios em tempo de execução.

A aplicação do mecanismo proposto em sistemas de grande porte. Porém, isto depende da implementação do núcleo da Virtuosi.

A formalização dos processos de weaving através de algoritmos.

Experimentos para outras políticas de persistência: ação atômica, por período, em horário pré-determinado, ou outras que possam ser definidas futuramente.

Referências Bibliográficas

[ABD+89] M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, and S. Zdonik. The object-oriented database system manifesto. In *In Proceedings of the First International Conference on Deductive and Object-Oriented Databases*, pages 223–40, Kyoto, Japan, December 1989.

[ADJ+96] M. P. Atkinson, L. Daynès, M. J. Jordan, T. Printezis, and S. Spence. An orthogonally persistent java. *ACM SIGMOD Record*, 25(4):68–75, December 1996.

[Amb97] Scott W. Ambler. The design of a robust persistence layer for relational databases. Technical report, http://www.ambysoft.com/persistenceLayer.pdf, 1997.

[Amb00] Scott W. Ambler. Mapping objects to relational databases. *IBM DeveloperWorks*, July 2000.

[BOS+] Bob Bretl, Allen Otis, Marc San Soucie, Bruce Schuchardt, and R. Venkatesh. Persistent java objects in 3 tier architectures. Technical report, GemStone System, Inc.

[BRL98] J. P. Briot, R.Guerraqui, and K. Lohr. Concurrency and distribution in object-oriented programming. *ACM Computing Surveys*, 30(3), September 1998.

[Cal96] Alcides Calsavara. *Constructing Highly-Available Distributed Metainformation Systems.* PhD thesis, University of Newcastle Upon Tyne, May 1996.

[Cal00] Alcides Calsavara. Virtuosi: Máquinas virtuais para objetos distribuídos. Technical report, Pontifícia Universidade Católica do Paraná, Brasil, 2000.

[Chi98a] Shigeru Chiba. Javassist - a reflection-based programming wizard for java. In *In Proceedings of the ACM OOPSLA'98 Workshop on Reflective Programming in C++ and Java*, October 1998.

[Chi98b] Shigeru Chiba. Macro processing in object-oriented languages. In *In Proceedings of Technology of Object-Oriented Languages and Systems*, Australia, November 1998.

[Chi00] Shigeru Chiba. Load-time structural reflection in java. In *ECOOP -European Conference on Object-Oriented Programming, LNCS 1850, Spring Verlag*, pages 313–336, Sophia Antipolis and Cannes, France, June 2000.

[CM93] Shigeru Chiba and Takashi Masuda. Designing an extensible distributed language with a meta-level architecture. In *In Proceedings of European Conference on Object Oriented Programming (ECOOP), LNCS 707*, pages page 482–501, Germany, July 1993.

[Com01] Computer Associates and Fujitsu. *Jasmine ODB 2.02 Database Design and Implementation*, 2001.

[Cop02] James O. Coplien. Software patterns - a pattern definition. Technical report, Bell Laboratories, Naperville, Illinois, 2002.

[CWH00] Mary Campione, Kathy Walrath, and Alison Huml. *The Java Tutorial: A Short Course on the Basics.* Addison-Wesley, 2000.

[Dat95] C. J. Date. *Introduction to Database Systems.* Addison Wesley Longman, Inc, 1995.

[ED97] Huw Evans and Peter Dickman. Drastic: A run-time architecture for evolving, distributed, persistent systems. In *Proceeding of the ECOOP'97 - Object-Oriented Programming*, pages 243–275, June 1997.

[EEB01] Tzilla Elrad, Robert E.Filman, and Atef Bader. Aspect-oriented programming. *Communication of the ACM*, 44(10):29–32, October 2001.

[Eke00] Bruce Ekel. *Thinking in Java.* Prentice-Hall, 2000.

[ES98] Huw Evans and Susan Spence. Porting a distributed system to pjama: Orthogonal persistence for java. In *Proceedings for the*

Third Workshop on Persistence and Java, Tiburon, California, September 1998.

[FK97] Michael Franz and Thomas Kistler. Slim binaries. Technical Report 40(12), University of California at Irvine, 1997.

[Fus97] Mark L. Fussell. Foundations of object relational mapping. Technical Report mlf-970703, ChiMu Corporation, Sunnyvale, CA, July 1997.

[GBCV96] Gowing, Brendan, Cahill, and Vinny. Meta-object protocols for c++: The iguana approach. In *Proceedings of reflection' 96*, 1996.

[GHRV95] E. Gamma, R. Helm, R.Johnson, and J Vlissides. *Design Patterns: Elements Of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[Gro99] Object Management Group. Persistent state service 2.0. Technical Report OMG Document orbos/99-07-07, Object Management Group (OMG), Inc, August 1999.

[HS98] Arthur H.Lee and Ho-Yun Shin. Building a persistent object store using the java reflection api. In *OOPSLA 98 - Workshop 13: Reflexive Programming in C++ and Java*, Vancouver, Canada, October 1998.

[JA98] Mick Jordan and Malcolm P. Atkinson. Orthogonal persistence for java - a mid-term report. In *Third International Workshop on Persistence and Java*, Tiburon, CA, September 1-3 1998.

[JK96] Michael Golm Jürgen Kleinöder. Metajava: An efficient run-time meta architecture for java. In *Proceedings of the International Workshop on Object Orientation in Operating Systems - IWOOOS '96, IEEE*, Seattle, Washington, October 1996.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP),Springer-Verlag*, pages 220–242, Finland, June 1997.

[KRDB91] Umeshwar Dayal Klaus R Dittrich and Alejandro P Buschmann. *On Object-Oriented Database Systems.* Springer Verlag, 1991.

[KVH02] Muna Matar Koenraad Vandenborre and Ghislain Hoffman. Orthogonal persistence using aspect oriented programming. In *AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software*, Enschede, The Netherlands, April 2002.

[Les02] Nicholas Lesiecki. Improve modularity with aspect-oriented programming. *IBM developerWorks*, January 2002.

[LS98] M.C. Little and S.K. Shrivastava. Understanding the role of atomic transactions and group communications in implementing persistent replicated objects. In *The 8th International Workshop on Persistent Object Systems: Design Implementation and Use*, pages 307–316, Tiburon, California, August 1998.

[Mae87] Patie Maes. Concepts and experiments in computational reflexion. In *OOPSLA 87 Proceedings*, Orlando, FL, October 1987.

[Man94] Frank Manola. An evaluation of object-oriented dbms developments. Technical Report TR-0263-08-94-165, GTE Laboratories Incorporated, Waltham, MA, August 1994.

[MH96] J. Eliot B. Moss and Antony L. Hosking. Approaches to adding persistence to java. In *First International Workshop on Persistence and Java*, Drymen, Scotland, September 1996.

[Mic97] Sun Microsystems. Jdbc guide: Getting started. Technical report, Sun Microsystems, Inc, 1997.

[MLS93] D.L. McCue M.C. Little and S.K. Shrivastava. Maintaining information about persistent replicated object in a distributed system. Technical report, Cornell University Computer Science Department, October 1993.

[MRB98] R. Martin, D. Riehle, and F. Buschmann. *Pattern Language Of Program Design 3*, pages 291–312. Addison-Wesley, 1998.

[MTI00] Marc-Olivier Killijian Michiaki Tatsubori, Shigeru Chiba and Kozo Itano. Openjava: A class-based macro system for java. In *Lecture Notes in Computer Science 1826, Reflection and Software Engineering, Walter Cazzola, Robert J. Stroud, Francesco Tisato (Eds.), Springer-Verlag*, pages pp.117–133, Ibaraki, Japan, 2000.

[NC01] Leonardo R. Nunes and Alcides Calsavara. Estudos sobre a concepção de uma linguagem de programação reflexiva e correspondente ambiente de execução. In *V Simpósio Brasileiro de Linguagens de Programação*, Maio 2001.

[OGB98] Alexandre Oliva, Islene Calciolari Garcia, and Luiz Eduardo Buzato. The reflexive architecture of guaraná. Technical report, Instituto de Computação Universidade Estadual de Campinas, Campinas, September 1998.

[PF98] et al. Paulo Ferreira, Marc Shapiro. Perdis: design, implementation, and use of a persistent distributed store. Technical Report QMW TR 752, CSTB ILC/98-1392, INRIA RR-3525 and INESC RT/5/98, QMW, CSTB, INRIA, INESC, October 1998.

[PGG01] A. Popovici, Th. Gross, and G.Alonso. Dynamic homogenous aop with prose. Technical report, ETH Zurich - Departament of Computer Science, Zurich, Switzerland, March 2001.

[Rot98] Bill Roth. An introduction to enterprise javabeans tm technology. Technical report, Sun Microsystems, Inc, October 1998.

[SGG91] S.K. Shrivastava, Dixon G.N., and Parrington G.D. An overview of the arjuna programming system. *IEEE Software*, 8(1):66–73, Jan 1991.