Um Arcabouço de Software Reflexivo para Persistência de Objetos em Bases de Dados Heterogêneas

Nádia Leão Pereira Quadros

Dissertação de Mestrado

Um Arcabouço de Software Reflexivo para Persistência de Objetos em Bases de Dados Heterogêneas

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por Nádia Leão Pereira Quadros e aprovada pela Banca Examinadora.

Campinas, 26 de julho de 1999.

Prof. Dr. Luiz Eduardo Buzato (Orientador)

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

Instituto de Computação
Universidade Estadual de Campinas

Um Arcabouço de Software Reflexivo para Persistência de Objetos em Bases de Dados Heterogêneas

Nádia Leão Pereira Quadros[1]

Julho de 1999

Banca Examinadora:

- Prof. Dr. Luiz Eduardo Buzato (Orientador)
- Profª Drª Maria Lúcia Blanck Lisbôa
  Instituto de Informática - Universidade Federal do Rio Grande do Sul
- Profª Drª Eliane Martins
  Instituto de Computação - Universidade Estadual de Campinas
- Profª Drª Cecília Mary Fischer Rubira (Suplente)
  Instituto de Computação - Universidade Estadual de Campinas

[1]**Auxílios concedidos pelo CNPq e pela CAPES para Nádia Leão Pereira Quadros, e pela FAPESP, processo 96/1532-9, para o Laboratório de Sistemas Distribuídos do IC-UNICAMP.**

Agradecimentos

A minha família pelo incentivo e apoio incondicional e em especial aos meus pais pelo sacrifício que fizeram para que minha dedicação aos estudos pudesse ser integral.

Ao meu marido Elbson pelo carinho e compreensão demonstrados nos momentos mais difíceis desta caminhada.

Ao meu orientador Luiz Eduardo Buzato pela sabedoria com que guiou este trabalho e pela paciência ao lidar com minhas dificuldades.

Aos amigos do LSD pela troca de experiências e idéias, essenciais no desenvolvimento deste trabalho. Obrigado também pelo companheirismo de todas as horas.

A Alexandre Oliva por ter implementado Guaraná e pela infinita paciência com que sempre atendeu as minhas dúvidas e dificuldades.

Agradeço também a Capes, ao CNPq e a meus pais pelo apoio financeiro, imprescindível para a concretização deste trabalho.

Resumo

Esta dissertação descreve uma abordagem inovadora em relação às encontradas na literatura para o problema de persistência de objetos. Nossa abordagem combina reflexão computacional e arcabouços de software para a construção de um *arcabouço de software reflexivo para persistência de objetos*. Com essa abordagem apresentamos uma solução para duas questões que, na nossa opinião, ainda não foram contempladas de maneira satisfatória em outros sistemas de persistência: (i) persistência totalmente transparente e (ii) acesso a bases de dados heterogêneas. Essa solução favorece ainda a construção de componentes de software altamente reutilizáveis e o aumento de produtividade nas etapas de construção e manutenção das aplicações.

Os benefícios do uso do arcabouço são demonstrados na dissertação em duas instâncias: primeiro com uma discussão sobre o seu uso como uma ferramenta de apoio ao processo de migração incremental de sistemas legados e segundo no uso do arcabouço num protótipo de sistema de comércio pela Internet.

Esta dissertação também apresenta uma avaliação de desempenho de reflexão computacional. Os resultados dessa avaliação surpreendem porque indicam que os custos associados ao uso dessa técnica podem ser maiores que os verificados inicialmente. Nossos resultados ainda não oferecem evidência definitiva sobre a questão de custo, mas representam mais um passo na direção de uma avaliação mais abrangente dessa questão.

Abstract

This dissertation describes an alternative approach to allow applications to take profit from persistent programming. Our approach combines computational reflection and frameworks techniques to build a *reflexive object persistence framework*. The approach proposed addresses two questions that have not yet answered in a satisfactory manner: (i) total transparency of persistence to the application programmer and (ii) access to heterogeneous database. The solution also encourages software reuse and increases productivity in the application development and maintenance stages.

The framework benefits are demonstrated through a discussion about the use of the framework as an aid to the incremental migration of legacy systems and also through the implementation of object persistence for an electronic commerce system for the Internet.

This dissertation also describes a performance evaluation of the computational reflection mechanisms. The evaluation results demonstrate that computational reflection costs can be greater then those that have been reported before.

Conteúdo

Lista de Tabelas

Lista de Figuras

Lista de Programas e Classes

Lista de Siglas

Sigla	*Significado*	*Página*
ANSI	American National Standards Institute	12
JDBC	Java DataBase Connectivity	12
JDK	Java Development Kit	18
JUB	Java Universal Binding	15
JVM	Java Virtual Machine	5
MOP	MetaObject Protocol	35
OID	Unique Object Identifier	40
ODBC	Open DataBase Connectivity	12
ODMG	Object Data Management Group	19
PSE	Persistent Store Engine	18
PSE PRO	Persistent Store Engine Professional	18
SCI	Sistema de Comércio pela Internet	74
SQL	Structured Query Language	12
UML	Unified Modeling Languague	3

Capítulo 1

Introdução

A programação de software tem tradicionalmente acompanhado o princípio de que a interface de um módulo deve expor sua funcionalidade e esconder sua implementação. O principal problema desse princípio é o fato de que, depois que os sistemas são construídos, é difícil adaptá-los para atender às necessidades de novos clientes. Essa dificuldade de adaptação pode ter conseqüências indesejáveis no desempenho do sistema, na sua portabilidade e interoperabilidade [20].

Engenheiros de sistemas de software têm procurado novas formas de estruturação/construção de software com o objetivo de “abrir” os componentes de seus sistemas, para torná-los mais adaptáveis. Uma técnica de estruturação de software que tem se mostrado promissora para a construção de componentes abertos é a técnica de reflexão computacional [26, 41]. Nessa abordagem, o sistema tem acesso a informações sobre sua estrutura interna e sobre sua computação e usa essas informações para modificar o seu comportamento.

No passado, as aplicações de reflexão computacional restringiam-se em geral a aplicações em inteligência artificial e em linguagens de programação. Mais recentemente, é possível notar o uso de técnicas reflexivas em áreas tão variadas como sistemas operacionais [48], sistemas distribuídos [42], tolerância a falhas [12, 25, 47] e banco de dados [24, 35].

Dois fatos comprovam essa tendência de diversificação no uso de reflexão computacional. Primeiro, conferências importantes como OOPSLA e ECOOP têm dedicado seções inteiras à reflexão computacional e suas aplicações. No Brasil, o interesse pode ser notado pela publicação de trabalhos que se apoiam em reflexão computacional nos Simpósios Brasileiros de Linguagens de Programação (SBLP), de Redes de Computadores (SBRC), de Computação Tolerante a Falhas (SCTF) e de Engenharia de Software (SBES). Segundo, há companhias utilizando princípios de reflexão computacional para estruturar os seus sistemas de software.

Dentro desse contexto, esta dissertação representa um esforço adicional para estender a aplicação de reflexão computacional em área associada a software de apoio (*middleware*) e bancos de dados. O campo escolhido para a pesquisa sobre a aplicação de reflexão computacional foi o da persistência de dados no paradigma de objetos.

A escolha desse campo de pesquisa se deu por dois motivos. Primeiro, a persistência é um mecanismo simples o bastante para permitir implementações bem estruturadas dentro do tempo previsto para um mestrado, ao mesmo tempo que permite o exercício de Guaraná [33], uma arquitetura de software reflexiva baseada em Java[1], construída no IC/UNICAMP. Pretendemos, com esta dissertação, contribuir para os testes e a validação dessa arquitetura reflexiva.

Segundo, várias soluções têm sido propostas para permitir a persistência dos objetos de uma aplicação, mas nenhuma se apoia em reflexão computacional. As diferentes soluções implementam a persistência através da modificação de uma linguagem de programação e/ou de seu compilador ou interpretador [5, 17], definição de uma interface de programação[2] para persistência [28, 46] ou implementam persistência através de herança [10, 23].

Nessa dissertação apresentamos uma abordagem alternativa para persistência de dados, que combina as técnicas de reflexão computacional e arcabouços de software[3] para a construção de um *arcabouço de software reflexivo para persistência de objetos*. Um arcabouço de software [19, 45] é um conjunto de classes abstratas e concretas que provê uma infra-estrutura genérica de soluções para um conjunto de problemas. Em outras palavras, um arcabouço de software padroniza a arquitetura de software de um domínio específico.

Com essa abordagem, pretendemos atingir dois objetivos: (i) prover persistência transparente para a aplicação que manipula os dados persistentes e (ii) prover uma interface flexível de acesso às bases de dados, que permita à aplicação manipular dados armazenados em bases de dados heterogêneas. Vamos mostrar como a combinação das técnicas de reflexão computacional e arcabouços de software pode produzir componentes de software transparentes e altamente reutilizáveis, inclusive em contextos não imaginados de antemão.

Esta dissertação representa também uma reflexão sobre o desempenho de reflexão computacional. A literatura sobre reflexão computacional contém muitos trabalhos sobre o uso dessa na estruturação de sistemas de software [5, 12, 22, 42] mas há poucas avaliações do custo adicional introduzido pelo seu uso [9, 47]. Além disso, essas poucas avaliações foram feitas utilizando aplicações muito simples, que podem não refletir o custo do uso de reflexão computacional em aplicações de médio e grande porte.

Nessa dissertação apresentamos uma avaliação de desempenho de reflexão computacional em aplicações de escala maior que as normalmente encontradas na literatura especializada. Os resultados obtidos surpreendem porque indicam que os custos associados ao uso dessa técnica podem ser maiores que os verificados inicialmente. Nossos resultados ainda não oferecem evidência definitiva sobre a questão de custo, mas representam um passo na direção de uma avaliação mais abrangente dessa questão.

[1] Java, Sun, JDBC e JDBC `COMPLIANT` são marcas registradas pela Sun Microsystems, Inc.

[2] Em Inglês: *Application Program Interface* (API).

[3] Em Inglês: *frameworks*.

A dissertação está organizada da seguinte forma:

Capítulo 2 - Persistência em Java: Este Capítulo apresenta um conjunto de sistemas de persistência para a linguagem Java. Durante toda a dissertação usaremos o termo "sistema de persistência" para denotar qualquer sistema, aplicação ou interface de programação construído para implementar a persistência dos dados de uma aplicação. O objetivo da coletânea apresentada neste Capítulo é analisar diferentes abordagens para persistência sob aspectos que consideramos importantes em um sistema que provê persistência, tais como transparência, heterogeneidade no acesso às bases de dados e facilidade de uso.

Os Capítulos 3 e 4 contém dois artigos sobre o arcabouço reflexivo para persistência de objetos.

Capítulo 3 - Um Arcabouço Reflexivo para Persistência de Objetos: Este artigo apresenta a arquitetura de software definida pelo arcabouço, uma descrição dos pontos fixos e adaptáveis dessa arquitetura e alguns exemplos de uso do arcabouço.

Capítulo 4 - Uso de um Arcabouço Reflexivo para Persistência de Objetos como Ferramenta de Apoio à Migração de Sistemas Legados: Este artigo descreve o arcabouço e discute o seu uso como uma ferramenta de apoio ao processo de migração incremental de sistemas legados.

Capítulo 5 - Uma Avaliação de Desempenho de Reflexão Computacional: Este Capítulo descreve a construção de um exemplo de uso do arcabouço e uma avaliação de desempenho de reflexão computacional, realizada a partir do exemplo construído.

Capítulo 6 - Conclusão: Este Capítulo descreve as contribuições dessa dissertação e os planos para continuidade do trabalho.

Apêndice A - Sistema de Persistência Reflexivo do SCI: Este Apêndice contém a definição de algumas classes usadas na construção do exemplo de uso do arcabouço, que não foram definidas no Capítulo 5.

Notação

Os diagramas de classe apresentados ao longo da dissertação seguem a notação proposta por Booch, Rumbaugh e Jacobson denominada Unified Modeling Language (UML). As definições de classes, programas e algoritmos estão escritos na linguagem de programação Java [2].

Capítulo 2

Persistência em Java

Um sistema computacional pode ter diferentes tipos de componentes de armazenamento, de memórias cache ultra-rápidas e voláteis (tempo de acesso $\sim 10^{-9}$s) a discos magnéticos (tempo de acesso $\sim 10^{-2}$s) algumas ordens de grandeza mais lentos que essas memórias. A existência desse espectro de componentes de armazenamento não é, na maioria das vezes, transparente para um programador, isto é, a linguagem de programação não trata todos esses componentes de forma uniforme. Para verificar a validade dessa afirmação basta observar que linguagens de programação em geral exibem um conjunto de construções para manipulação de dados armazenados na memória principal e um outro conjunto distinto de construções para a manipulação de dados armazenados em disco. Consequentemente, durante a implementação de um programa, o programador é obrigado a mapear dados voláteis em dados persistentes. Muitas vezes esse mapeamento requer um entendimento sólido do sub-sistema responsável pela persistência, desviando a atenção do programador de sua tarefa principal e gerando oportunidades para a introdução de falhas na aplicação. A eliminação da distinção entre dados voláteis e dados persistentes têm ocupado pesquisadores há algum tempo.

A pesquisa na área de persistência foi iniciada no final da década de 60 por um grupo da Universidade de Saint Andrews, na Escócia. Atkinson [3, 11], um dos lideres desse grupo, define persistência como "o período de tempo pelo qual um dado existe e é útil, utilizável". Portanto, persistência é uma propriedade que depende da aplicação, algumas podem precisar de seus dados por apenas alguns segundos, outras podem querer reutilizá-los durante um século. Em geral, em sistemas modernos de computação, persistência está associada a existência de um mecanismo que pode ser acionado pela aplicação para salvar seus dados em memória estável quando ela tem necessidade de reutilizá-los em ativações posteriores à atual.

Quase 30 anos de pesquisa foram suficientes para produzir resultados importantes. PS-Algol [4] e Napier [27] representam linguagens de programação que foram estendidas para apresentar persistência como um mecanismo ortogonal as primitivas de entrada e saída (E/S) das linguagens. A ortogonalidade de persistência em relação a tipos de dados e operadores de E/S permite que programadores tornem persistentes estruturas de dados complexas, sem se preocupar com o mapeamento da representação volátil para a persistente e vice-versa.

Mais recentemente, linguagens de programação orientadas a objetos persistentes foram projetadas e implementadas, como exemplificado por linguagens como Galileo [1], Trellis/Owl [30] e E [37, 38]. No âmbito de ambientes de programação, alguns sistemas preferem introduzir persistência via métodos específicos, disponíveis através de uma interface de programação. Esse é o caso de NIH [15].

A grande maioria dos desenvolvimentos mais recentes no campo de persistência têm ocorrido em relação a persistência em Java. Várias abordagens têm sido propostas para implementar persistência de dados para essa linguagem: modificação do interpretador da linguagem [5, 36], definição de uma interface de programação para persistência [23, 28, 46] e persistência através de herança [23].

Este Capítulo descreve alguns dos principais trabalhos no campo de persistência em Java. O foco em Java não representa a relegação a segundo plano de outras soluções, porque a pesquisa sobre persistência nessa linguagem tem sido guiada pelos resultados obtidos anteriormente.

O objetivo dessa coletânea é mostrar as diferentes abordagens usadas para implementar persistência em Java e apresentar uma avaliação dessas soluções. Ao final do Capítulo, apresentamos um conjunto de características desejáveis para um sistema de persistência, elaborado a partir de uma análise das diferentes abordagens para persistência descritas. Em seguida, introduzimos nossa solução para a construção de um sistema de persistência que contempla essas características. O conteúdo deste Capítulo também pode servir como um guia para programadores Java, que pretendem desenvolver aplicações que manipulam dados persistentes.

Nessa coletânea descrevemos os seguintes trabalhos: Linearização em Java [44], Persistência e Java Core Reflection [23], JDBC [16], Java Universal Binding [46], ObjectStore [28, 29], PSE/PSE PRO [28, 29] e PJama [5, 36]. Linearização e JDBC fazem parte da interface de programação de Java, PJama introduz a persistência a partir de modificações na máquina virtual de Java (Java Virtual Machine—JVM) e os demais implementam a persistência via uma interface de programação para Java. A descrição dos trabalhos contém exemplos de uso e uma avaliação dos mesmos. Essa avaliação analisa os trabalhos mencionados acima sob os seguintes aspectos:

- ***Transparência*** dos mecanismos para persistência em relação a aplicação que manipula os dados persistentes;

- ***Facilidade de uso*** do sistema de persistência;

- ***Heterogeneidade no acesso às bases de dados***: capacidade do sistema de persistência em prover acesso a mais de um tipo de base de dados, por exemplo, banco de dados relacional e sistema de arquivos, permitindo a aplicação manipular, de maneira uniforme, dados armazenados em bases de dados de tipos diferentes;
- ***Mapeamento do modelo de objetos no modelo relacional***, quando os dados persistentes da aplicação são armazenados em bancos de dados relacionais;
- ***Unidade de persistência***: quais são os tipos de dados da linguagem que podem ser persistentes, por exemplo, tipos primitivos, objetos;
- ***Atributo de persistência***: como a semântica de persistência é associada aos tipos de dados da linguagem, por exemplo, atribuição de persistência por herança ou através da criação de raízes de persistência;
- ***Persistência transitiva***: se um objeto é persistente então todos os objetos referenciados através desse objeto serão automaticamente promovidos a objetos persistentes.

Exemplo de Referência

Para facilitar a comparação entre os sistemas que serão descritos, utilizamos um exemplo de referência na construção de todos os programas que ilustram o funcionamento desses sistemas. O exemplo de referência é muito simples, sendo composto por duas classes `Person` e `Student`. `Student` é uma especialização de `Person` (Classes 2.1).

Não foi possível construir um programa ilustrando cada sistema de persistência descrito porque, em alguns casos, a bibliografia de que dispúnhamos sobre o sistema não forneceu subsídios suficientes para isto. Nesses casos, procuramos apresentar representações esquemáticas e descrições mais detalhadas dos sistemas, para que o entendimento de seu funcionamento e uso não fosse prejudicado.

2.1 Linearização em Java

Linearização[1] é uma técnica utilizada para transformar a representação volátil de um objeto em uma representação neutra em termos de arquitetura de hardware, protocolo de comunicação e armazenamento em disco. Em Java, a estrutura volátil de um objeto é mapeada para uma estrutura neutra utilizando objetos denominados *streams*. A interface de linearização de Java [44] define um conjunto de classes para escrita e leitura de objetos e tipos primitivos em *streams*.

[1] Em Inglês: *Serialization*.

Classes 2.1 Person e Student

```
public class Person{
    String name;
    int age;
    public Person(String name, int age){  // construtor da classe
        this.name = name;
        this.age = age;
    }
}

public class Student extends Person{
    Person responsible;
    public Student(String name, int age, Person resp){  // construtor da classe
        super(name, age);  // invoca construtor da classe base
        responsible = resp;
    }
}
```

Todas as classes cujos objetos podem ser linearizados em *streams* devem implementar a interface `Serializable`. Essa interface não tem métodos ou atributos e serve unicamente para associar a semântica de linearização à classe. Todos os subtipos de uma classe *Serializable* são automaticamente *Serializable*.

O Programa 2.2 mostra como armazenar um objeto Java da classe `Student` em um arquivo, usando a interface de linearização de Java. Primeiro, é necessário rescrever a classe `Person`, para que esta implemente a interface `Serializable`. Para escrever o objeto `std` em arquivo é preciso criar um objeto *stream* de saída para arquivo e um objeto que escreve dados nesse *stream*. O objeto `fos` da classe `FileOutputStream` é um *stream* de saída criado para escrever os dados no arquivo “record”. O objeto `s` da classe `ObjectOutputStream` escreve o objeto `std` e os objetos por ele referenciados no *stream* de saída `fos`, usando o método `writeObject`.

O Programa 2.3 mostra como recuperar o objeto linearizado no Programa 2.2. O procedimento é análogo ao descrito acima. Para ler um objeto de um arquivo é preciso criar um objeto *stream* de entrada e um objeto para ler os dados desse *stream*. O objeto `fis` da classe `FileInputStream` é o *stream* de entrada criado para ler os dados do arquivo “record”. O objeto `s` da classe `ObjectInputStream` usa o método `readObject` para recuperar o objeto `Student` e todos os objetos por ele referenciados do *stream* de entrada `fis`, realizando uma restauração em cadeia.

Programa 2.2 Serialization_Example1

```
public class Person implements Serializable {...}

import java.io.*; // Importa as classes da interface de linearização.

public class Serialization_Example1{
    public static void main(String[] args){
        Person p = new Person("Peter Smith", 50);
        Student std = new Student("John Smith", 14, p);
        try{
            // Cria um objeto stream de saída para escrever no arquivo "record".
            FileOutputStream fos = new FileOutputStream("record");
            // Cria o objeto que escreverá os dados no stream de saída "fos".
            ObjectOutputStream s = new ObjectOutputStream(fos);
            // Escreve o objeto "std" no stream de saída.
            s.writeObject(std);
            // A escrita dos objetos no arquivo é efetivada.
            s.flush();
        } catch(Exception e){ ... }
    }
}
```

Programa 2.3 Serialization_Example2

```
import java.io.*; // Importa as classes da interface de linearização.

public class Serialization_Example2{
    public static void main(String[] args){
        try{
            // Cria um objeto stream de entrada para ler os dados do arquivo "record".
            FileInputStream fis = new FileInputStream("record");
            // Cria o objeto que irá ler os dados do stream de entrada "fis".
            ObjectInputStream s = new ObjectInputStream(fis);
            // Lê um objeto do stream de entrada.
            Student std = (Student)s.readObject();
        } catch(Exception e){ ... }
    }
}
```

Avaliação

A interface de linearização de Java provê um mecanismo simples para implementação da persistência em arquivos. Objetos e tipos primitivos podem ser escritos em arquivos usando essa interface de programação, desde que a classe do objeto linearizado, ou uma superclasse do mesmo, implemente a interface `Serializable`. Todos os tipos primitivos de Java são linearizáveis.

A linearização dos objetos e tipos primitivos em *streams* e a restauração desses dados é feita internamente pelo sistema, de forma transparente para a aplicação, e o programador só precisa construir uma implementação específica por classe se forem necessárias adaptações especiais. Por outro lado, o programador precisa manter o registro dos dados persistentes e invocar as operações de escrita e leitura desses dados em arquivos, o que dificulta a utilização dessa técnica em aplicações que requerem um grande número de operações de escrita e leitura de dados persistentes, além de dificultar também a reutilização das classes dessas aplicações devido ao grande volume de código extra inserido para implementar a persistência.

É importante ressaltar que a interface `Serializable` não atribui a classe uma semântica de persistência, mas apenas indica que instâncias de classes que implementam essa interface podem ser linearizadas e, eventualmente, escritas em arquivos. A interface de linearização de Java não associa uma semântica de persistência aos tipos de dados da linguagem. Isto significa que os dados da aplicação não são rotulados como persistentes ou voláteis e o sistema de persistência não mantém nenhuma ligação entre o dado da aplicação e a instância persistente desse dado na base de dados. Essa ligação é mantida pelo programador no código da aplicação. Alguns sistemas de persistência associam a semântica de persistência aos tipos de dados através de herança ou criando raízes de persistência. Nas próximas Seções descreveremos sistemas de persistência com essas características.

A linearização de um objeto Java é transitiva, ou seja, se um objeto é linearizado, todo o grafo de objetos que compõe seu estado é automaticamente linearizado. Mas, se algum desses objetos não possui a semântica de linearização (implementa a interface `Serializable`), o sistema levantará uma exceção e a linearização do objeto não será completada com sucesso.

A linearização é um mecanismo lento quando o número de objetos linearizados é grande e quando as atualizações são freqüentes. Isto acontece porque a leitura dos objetos linearizados num mesmo *stream* de saída é seqüencial, ou seja, deve ocorrer na mesma ordem em que os objetos foram escritos. Além disso, a recuperação de um objeto linearizado ocorre na sua totalidade, ou seja, só termina quando todo o estado do objeto é recuperado, incluindo o estado de objetos nele contidos ou por ele referenciados.

Quanto a confiabilidade dos dados, a interface de linearização de Java preserva o tipo e as propriedades de segurança dos objetos Java, mas não provê um armazenamento de dados confiável [29]. Se a aplicação cair durante uma operação de escrita, o conteúdo do arquivo pode ser perdido.

2.2 Persistência e Java Core Reflection

Lee e Shin [23] descrevem um sistema de persistência que implementa a persistência de objetos Java através de uma interface de programação. O principal objetivo desse sistema é prover persistência transparente para a aplicação que manipula os dados persistentes. Para alcançar essa transparência eles utilizam Java Core Reflection.

Java Core Reflection [40] faz parte da interface de programação de Java. Através da interface de programação de Java Core Reflection o programador pode obter, em tempo de execução, informação sobre a estrutura interna de objetos, classes, métodos e atributos, pode invocar métodos e obter ou modificar o valor de atributos, desde que não quebre o encapsulamento da classe.

O sistema de persistência descrito em [23] é composto pela classe `PersistentRoot` e pelos componentes Index Manager e Storage Manager. Esse sistema de persistência associa a semântica de persistência às classes através de herança, sendo `PersistentRoot` a classe base de todas as classes com instâncias persistentes. Index Manager mantém um registro dos objetos persistentes da aplicação e Storage Manager faz a conexão entre os objetos persistentes e o repositório de objetos. Storage Manager não é um componente visível pela aplicação.

As principais operações implementadas na classe `PersistentRoot` são executadas pelo seu construtor e pelos métodos `save` e `load`. O construtor da classe registra no Index Manager o novo objeto persistente instanciado. O método `save` salva o estado corrente do objeto no repositório de objetos e o método `load` restaura o estado do objeto desse repositório. A interface com o repositório de objetos é feita através do componente Storage Manager.

Para escrever ou recuperar um objeto persistente do repositório de objetos de forma transparente para a aplicação, o sistema de persistência usa a interface de Java Core Reflection. Na escrita, Java Core Reflection é usada para determinar o nome e o valor dos atributos do objeto persistente. Todos os objetos referenciados pelo objeto persistente são automaticamente promovidos a objetos persistentes. Na restauração, uma instância volátil da mesma classe do objeto persistente é criada. O sistema de persistência usa a interface de Java Core Reflection para escrever nos atributos da nova instância, preenchendo-os com os dados do objeto persistente recuperado do repositório de objetos.

A operação de escrita dos objetos persistentes no repositório de objetos é disparada pelo método `saveAll`, e a restauração desses objetos persistentes pelo método `loadAll`. Esses métodos salvam e restauram todos os objetos persistentes previamente registrados em Index Manager. Ambos são métodos invocados pelo programador no código da aplicação.

A Figura 2.1 mostra a seqüência de operações para escrita do objeto `p` da classe `Person` no repositório de objetos. Primeiro, a classe `Person` foi rescrita como uma subclasse de `PersistentRoot`. Quando o objeto `p` é criado na aplicação (1), o construtor da classe

`Person` invoca a operação `register` no componente IndexManager (2). Essa operação irá registrar o objeto como um novo objeto persistente. Quando o programador invoca o método `saveAll` (3) no código da aplicação, IndexManager invoca o método `save` (4) em todos os objetos nele registrados. O método `save`, no objeto `p`, obtém todos os atributos da classe `Person` (5) e os valores desses atributos (6)(7). Finalmente, o estado do objeto é escrito no repositório de objetos através do componente Storage Manager (8). Os objetos `name` e `age` são a materialização[2] dos atributos do objeto `p`. A materialização desses atributos é implementada por Java Core Reflection.

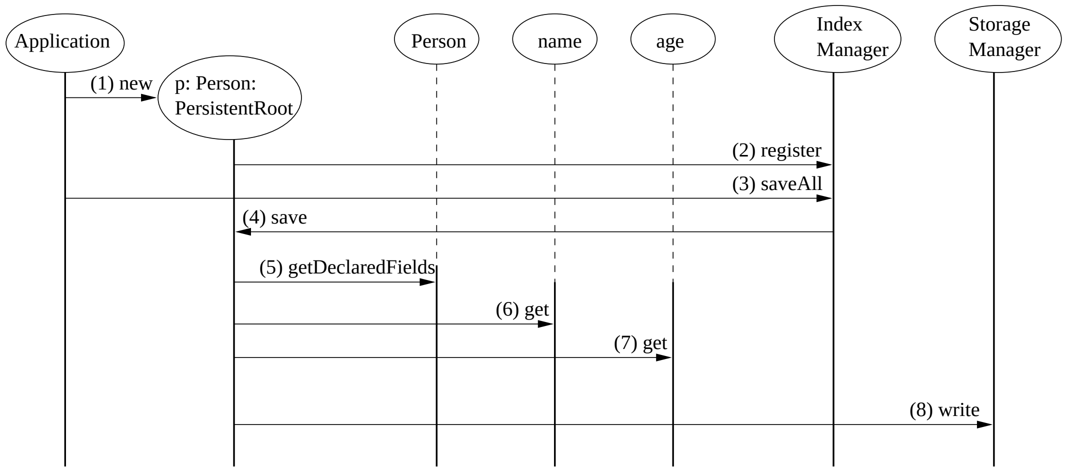

Figura 2.1: Persistência no sistema proposto por Lee e Shin

Avaliação

O sistema de persistência descrito implementa persistência em Java através de uma interface de programação, sem modificar a linguagem.

As classes de objetos persistentes são subclasses de `PersistentRoot` que implementa um conjunto de métodos para ler e escrever nos atributos desses objetos. Dessa forma, `PersistentRoot` implementa as operações de leitura e escrita dos objetos na base de dados de forma transparente para o programador. A transparência dessas operações de leitura e escrita é possível porque Java Core Reflection permite acesso a estrutura interna dos objetos. Esse acesso, porém, é restrito a interface pública e protegida do objeto. O acesso aos atributos privados que, evidentemente, também precisam ser preservados, não é permitido.

Uma solução para esse problema consiste na adição de dois métodos extras para cada classe persistente, implementados pelo programador: um para salvar os atributos privados

[2]Materialização é a representação na forma de objetos de informações sobre a estrutura das classes e objetos. Mais informações sobre materialização podem ser obtidas na Seção 3.2.1.

do objeto e outro para restaurar o valor desses atributos. Esses métodos devem escrever e obter os atributos privados de um vetor de objetos declarado na interface pública da classe `PersistentRoot`. O sistema de persistência, por sua vez, usa esse vetor de objetos para obter o valor dos atributos privados do objeto nas operações de escrita no repositório de objetos e escreve nesse vetor o valor dos mesmos atributos privados nas operações de leitura desses objetos do repositório de objetos. Essa solução, entretanto, compromete o objetivo inicial de se criar um serviço de persistência transparente.

No sistema de persistência descrito a semântica de persistência é associada às classes da aplicação por herança da classe `PersistentRoot`. Automaticamente, todas as instâncias de subclasses da classe `PersistentRoot` serão persistentes e o sistema de persistência irá manter o registro de todas as instâncias persistentes da aplicação. A persistência é transitiva, mas se um objeto persistente possui uma referência para um objeto cuja classe não é persistente, então esse objeto é interpretado como uma instância volátil da aplicação.

A associação de persistência por herança restringe os tipos de dados da linguagem que podem ser persistentes. Tipos primitivos e classes que não herdam de `PersistentRoot` não podem ser persistentes. Os tipos primitivos, no entanto, podem ser persistentes quando fazem parte do estado de objetos persistentes. Outra restrição dessa abordagem é o fato de não ser possível criar instâncias persistentes e voláteis de uma mesma classe. Isto é garantido pelo construtor da classe `PersistentRoot`, que registra toda nova instância da classe no IndexManager.

As operações de leitura e escrita dos objetos persistentes são disparadas pelos métodos `saveAll` e `loadAll`. Esse mecanismo minimiza o trabalho de codificação do programador, que não precisa manter o registro das instâncias persistentes e nem se preocupar em invocar operações para escrita e leitura dessas instâncias. Por outro lado, cria um serviço de persistência menos flexível, pois não existe um mecanismo para escrita e leitura de objetos persistentes sob demanda.

2.3 Java Database Connectivity

Java Database Connectivity (JDBC) [16] é uma interface de programação de Java que facilita a conexão e a interface com bancos de dados relacionais. JDBC é similar ao Microsoft ODBC (Open DataBase Connectivity), que é uma interface de programação para C++ usada para conexão e interface com bancos de dados relacionais.

JDBC define uma interface uniforme para uma variedade de módulos de conexão com bancos de dados relacionais diferentes, denominados *drivers*. São os *drivers* que, efetivamente, realizam a interface com um sistema de banco de dados específico. Para serem compatíveis com o JDBC (JDBC `COMPLIANT`), os *drivers* devem implementar pelo menos a interface definida pelo padrão ANSI SQL 92 (ANSI—American National Standards Institute, SQL—Structured Query Language).

Existem várias maneiras de se implementar um *driver* JDBC. A Figura 2.2 apresenta um esquema com os tipos de *drivers* que podem ser construídos.

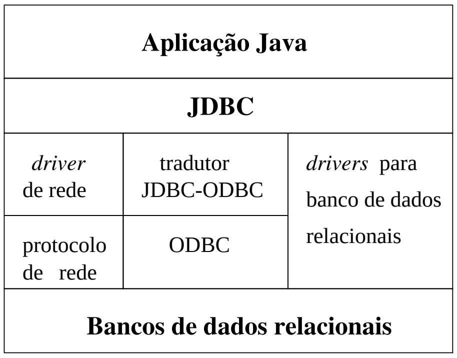

Figura 2.2: Tipos de *drivers* JDBC

O tipo mais simples é aquele construído como uma ponte para o ODBC (tradutor JDBC-ODBC). Com esse *driver* é possível fazer acesso a todos os bancos de dados conhecidos pelo ODBC. Essa abordagem é mais recomendada para aplicações que não pedem um alto desempenho, pois a conexão com o ODBC introduz mais um nível de indireção na comunicação com o banco de dados.

Um outro tipo de *driver* faz acesso ao banco de dados através de um protocolo de rede independente de banco de dados (*driver* de rede). Esse *driver* traduz as chamadas JDBC para o protocolo de rede, que é então traduzido em chamadas para a interface específica do banco de dados. Esse tipo de *driver* também introduz mais um nível de indireção na interface com o banco de dados.

O terceiro tipo de *driver* é aquele que faz acesso diretamente a interface do banco de dados. Esse é o tipo de *driver* mais eficiente. Por outro lado, esse *driver* só provê acesso a um sistema de banco de dados específico ou a um conjunto restrito de sistemas de banco de dados.

As classes mais importantes da interface de programação de JDBC são:

- `java.sql.Driver.Manager`: responsável pela carga/ligação com os *drivers* que serão usados pela aplicação;
- `java.sql.Connection`: representa uma conexão com um banco de dados particular;
- `java.sql.Statement`: os objetos dessa classe são usados para executar uma sentença SQL no banco de dados e obter o resultado produzido por ela;
- `java.sql.ResultSet`: essa classe provê acesso a um conjunto de dados gerados por um objeto `Statement`;

Programa 2.4 JDBC_Example

```
import java.sql.*; // Importa as classes da interface de JDBC.

public class JDBC_Example{
    public static void main(String[] args) {
        try{
            // Abre uma conexão com o banco de dados "record"
            // usando a ponte JDBC-ODBC.
            Connection con = DriverManager.getConnection("jdbc:odbc:record");
            // Cria e executa uma consulta neste banco de dados.
            Statement stmt = con.createStatement();
            ResultSet rs = stmt.executeQuery(
                        "SELECT person_name, person_age FROM people");
            // Obtém e imprime os valores do resultado da consulta.
            while( rs.next() ){
                // Obtém os valores da linha corrente.
                String name = rs.getString("person_name");
                int age = rs.getInt("person_age");
                System.out.println("Name: "+ name +", Age: "+ age);
            }
        } catch(SQLException e) { ... }  // Trata as exceções levantadas pelo JDBC.
    }  //main
}
```

- `java.sql.DatabaseMetaData` e `java.sql.ResultSetMetaData`: essas classes provêem meta-informação sobre o banco de dados e sobre o resultado de um consulta submetida ao banco de dados, respectivamente.

O Programa 2.4 mostra um exemplo de uso do JDBC. Nesse exemplo, o banco de dados "record" é manipulado usando a ponte JDBC-ODBC. Uma consulta é submetida a esse banco de dados para obter o nome e a idade de todos os registros armazenados na tabela `people`. Em seguida, o resultado dessa consulta é impresso na tela.

Avaliação

Muitas aplicações Java requerem acesso a bancos de dados relacionais pré-existentes e JDBC define uma maneira uniforme de acesso a esses bancos de dados. Dos sistemas de persistência já descritos, JDBC é o primeiro a prover heterogeneidade no acesso à base de dados, tornando possível às aplicações manipularem dados armazenados em diferentes tipos de bancos de dados relacionais, por exemplo, banco de dados Access e Oracle.

A interface de acesso aos bancos de dados usa SQL nas operações de adição, modificação e consulta aos dados do banco de dados, o que facilita o aprendizado de JDBC por programadores familiarizados com SQL.

O problema de JDBC está no mapeamento dos objetos Java para o modelo relacional. Pode ser necessária uma tradução complexa para mapear um modelo de objetos complexo para um modelo de dados relacional, podendo introduzir um custo adicional significativo no desempenho do sistema. JDBC não provê nenhuma facilidade para esse mapeamento, que deve ser feito pelo programador da aplicação.

Como na interface de linearização de Java (Seção 2.1), em JDBC a semântica de persistência não é associada aos tipos de dados de Java e qualquer objeto ou tipo primitivo da linguagem pode ser escrito em um banco de dados relacional, desde que o programador se encarregue do mapeamento de todos os dados persistentes para o modelo relacional. A persistência não é transitiva, o programador é quem explicitamente determina quais são os dados persistentes e armazena esses dados no banco de dados.

Com essas características, JDBC é um sistema de persistência que exige um esforço considerável de codificação, por parte do programador da aplicação, para que a persistência dos dados seja implementada. Isto traz o mesmo problema apontado na avaliação da interface de linearização de Java: um grande volume de código extra inserido na aplicação para implementar a persistência, dificultando a reutilização das classes dessa aplicação. JDBC ainda tem um agravante que é o mapeamento do modelo de objetos no modelo relacional. Na interface de linearização de Java, pelo menos a linearização dos dados e a propagação da persistência eram transparentes para o programador. Por outro lado, JDBC possui uma grande vantagem, que é o acesso uniforme a diferentes bancos de dados relacionais, permitindo a aplicação manipular de forma transparente dados armazenados em bases de dados heterogêneas.

2.4 Java Universal Binding

Java Universal Binding (JUB) [46] implementa a persistência de objetos Java através de uma interface de programação que provê acesso uniforme a bancos de dados de objetos e bancos de dados relacionais (Figura 2.3). Com JUB é possível armazenar e recuperar objetos de bancos de dados O_2, Oracle, Sybase e de todos os bancos de dados relacionais que se pode fazer acesso através do JDBC (Seção 2.3).

Para implementar a persistência usando JUB, as classes de objetos persistentes da aplicação devem implementar a interface `PersistentObject`. Essa interface declara um conjunto de métodos, e.g. `markModify`, `access` e `activate`, usados pelo sistema de persistência para gerenciar a persistência dos objetos. As classes de objetos persistentes precisam ainda ser pré-processadas (processadas antes de sua compilação) por JUB para que a persistência seja

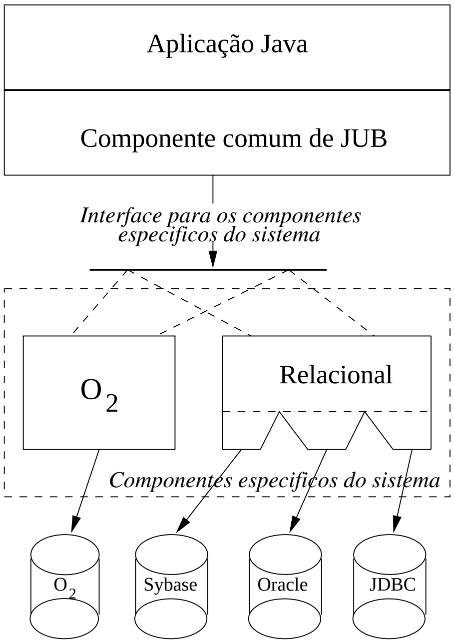

Figura 2.3: Arquitetura de JUB

implementada. O pré-processamento das classes gera uma implementação para os métodos da interface de `PersistentObject` (`markModify`, `access`, `activate`) e cria na base de dados as estruturas de dados necessárias para armazenar os objetos dessas classes. No banco de dados orientado a objetos O_2, é criada uma classe que corresponderá a classe Java. No banco de dado relacional, o pré-processamento de uma classe Java de nome `C` gera as seguintes operações [39]:

1. Uma tabela de nome `C` é criada, onde os atributos locais da classe `C` serão mapeados nas colunas dessa tabela, seguindo o mapeamento de tipos pré-definido no banco de dados.

2. Uma tabela de nome `C_S` é criada para armazenar as variáveis estáticas da classe, se houver alguma.

3. Para cada vetor `v` do tipo `T[ ]` é gerado uma tabela `T_A` e uma tabela `T_A_E` para armazenar os vetores e seus elementos, respectivamente.

4. As referências entre objetos são mapeadas nas tabelas como chaves estrangeiras.

Existem duas maneiras de armazenar um objeto na base de dados usando JUB. Na primeira, o programador requisita explicitamente que objetos sejam adicionados e removidos da base de dados. Nesse caso, a semântica de persistência não é associada ao objeto.

Na segunda maneira, o programador cria raízes de persistência. Uma raiz de persistência é um nome, a partir do qual pode-se recuperar um objeto mesmo depois que a aplicação que o criou terminar. Os objetos da aplicação adquirem um comportamento persistente quando associados a uma raiz de persistência. O programador cria suas raízes de persistência no código da aplicação e requisita que essas raízes sejam adicionadas e recuperadas da base de dados. Implicitamente, os objetos referenciados pela raiz de persistência também são promovidos a objetos persistentes e adicionados à base de dados. Nesse caso, a semântica de persistência é associada a raiz de persistência e aos objetos referenciados pela raiz.

JUB introduz o conceito de transações atômicas nas aplicações: o programador usa primitivas de sincronização de bancos de dados como `transaction`, `commit` e `abort` quando faz acesso aos objetos persistentes.

Antes de fazer acesso a um objeto persistente, o programador precisa informar ao JUB suas intenções. Isto é feito invocando-se o método `access` do objeto em questão. Essa invocação faz com que o objeto persistente seja carregado da base de dados para o espaço de endereçamento da aplicação.

Quando um objeto persistente é modificado na aplicação, JUB deve ser notificado para que as modificações sejam propagadas para a base de dados quando a transação terminar. Essa notificação é feita invocando-se o método `markModify` no objeto persistente modificado.

Os atributos do objeto persistente podem ser marcados com o modificador `transient` de Java, para indicar que o atributo não é persistente. Esses atributos nunca são armazenados na base de dados. Quando um objeto persistente é restaurado da base de dados, os atributos `transient` são inicializados pelo método `activate`. Esse método faz parte da interface de `PersistentObject`. O papel desse método é limitado à iniciação de atributos não persistentes.

Avaliação

JUB implementa a persistência de objetos Java combinando uma interface de programação para Java e um mecanismo de pré-processamento de classes.

JUB provê transparência no acesso a bases de dados heterogêneas. Essa funcionalidade pode ser bastante útil em sistemas que necessitam desse tipo de acesso, podendo diminuir o tempo necessário para o desenvolvimento desses sistemas, uma vez que dados armazenados em bases de dados heterogêneas são manipulados de maneira uniforme na aplicação. Além disso, a necessidade de ferramentas para acesso a bases de dados heterogêneas vem crescendo principalmente com a difusão da Internet, e um mecanismo com essas funcionalidades é especialmente importante em Java. JUB provê ainda um mapeamento implícito do modelo de objetos Java no modelo relacional.

A unidade de persistência em JUB é o objeto. Os objetos podem ser explicitamente adicionados e removidos da base de dados pelo programador ou pode-se associar a semântica de persistência aos objetos criando-se raízes de persistência. O uso de raízes de persistência, entretanto, facilita a referência aos objetos persistentes nas aplicações. Implicitamente, a semântica de persistência é propagada para os objetos referenciados a partir da raiz de persistência.

A interface uniforme para bases de dados heterogêneas e o uso de raízes de persistência são características que facilitam a implementação da persistência de objetos Java. Por outro lado, JUB introduz o conceito de transações atômicas nas aplicações, obrigando o programador a usar primitivas de sincronização de bancos de dados no acesso aos objetos persistentes. Esse conceito contribui negativamente para a transparência e facilidade no uso de JUB mas, por outro lado, introduz grandes benefícios como integridade dos dados, que certamente valem a pena para muitas aplicações. Outro ponto negativo com relação a transparência e facilidade no uso de JUB é a necessidade de pré-processar todas as classes de objetos persistentes da aplicação.

2.5 Produtos da Object Design

A Object Design criou uma família de produtos projetados para satisfazer os requisitos de gerência de banco de dados de um grande número de aplicações Java. Esses produtos são: ObjectStore, ObjectStore PSE e ObjectStore PSE PRO [28, 29].

PSE (Persistent Store Engine) e PSE PRO (Persistent Store Engine Professional) são sistemas de persistência 100% Java para armazenamento persistente, o que significa que podem ser usados em qualquer plataforma que tenha um compilador e máquina virtual compatíveis com o Java Development Kit (JDK) da Sun. Ambos são sistemas mono-usuário, dão suporte para transações atômicas com controle de concorrência baseado em *locks* de leitura e escrita nos bancos de dados e suporte para múltiplos leitores e um escritor.

PSE PRO é capaz de recuperar bancos de dados corrompidos por transações que abortaram em virtude de uma falha no sistema, mas as classes responsáveis por essa funcionalidade não estão presentes na versão de PSE. PSE PRO pode ainda manipular múltiplos bancos de dados de pequeno e médio porte (até 100MB), cada um com seu próprio contexto de transação, enquanto PSE só é capaz de manipular um banco de dados e um contexto de transação por vez. Essas são as principais diferenças entre esses dois produtos.

Já ObjectStore possui as seguintes funcionalidades adicionais, com relação aos dois primeiros:

- Sistema multi-usuário;
- Manipula múltiplos bancos de dados em um único contexto de transação;
- Suporte para agregação de objetos (coleções de objetos);

- Applets[3] e aplicações podem comunicar-se com bancos de dados em máquinas diferentes;
- Aplicações Java e C++ podem operar sobre os mesmo dados;
- Alto desempenho e armazenamento confiável de objetos, incluindo funcionalidades como *backup* e recuperação por retrocesso.

ObjectStore, PSE e PSE PRO implementam a persistência através de uma interface de programação para Java[4]. A interface de programação de ObjectStore é um superconjunto da interface de PSE PRO que, por sua vez, é um superconjunto da interface de PSE. Dessa forma, o programador pode migrar de um produto para outro, aumentando a funcionalidade de seu sistema de gerência de banco de dados com pequenas modificações na aplicação. A interface dos produtos é compatível com o padrão de interface Java para bancos de dados orientados a objetos proposta pelo Object Data Management Group (ODMG).

Usando a interface de programação de ObjectStore, PSE ou PSE PRO o programador pode: criar, abrir e fechar bancos de dados, iniciar e finalizar sessões, iniciar e finalizar transações, armazenar e recuperar objetos persistentes dos bancos de dados.

Para que os objetos possam ser escritos e lidos dos bancos de dados, as classes dos objetos persistentes e classes que usam esses objetos precisam ser pós-processadas (processadas depois da compilação). O pós-processador extrai informação sobre a definição da classe e insere código nas mesmas para que os objetos persistentes sejam automaticamente restaurados, quando ativados pela primeira vez, e escritos no banco de dados, quando modificados.

O Programa 2.5 mostra como criar um banco de dados e inserir um objeto da classe `Student` nesse banco de dados. Para que o programa funcione corretamente, as classes `Person`, `Student` e `ObjectStore_Example1` devem ser pós-processadas por ObjectStore, PSE ou PSE PRO antes do programa ser executado.

A navegação pelo modelo de objetos é a forma mais eficiente de acesso aos objetos, mas há certas situações que requerem um acesso associativo. ObjectStore apoia o processamento de consultas associativas através de coleções de objetos. Coleções de objetos são uma forma de armazenar e manipular um grande número de objetos persistentes. Os objetos podem ser recuperados usando seus atributos, o que pode poupar tempo e esforço de acesso à grupos de objetos.

O Programa 2.6 mostra um exemplo de uso de coleções. O programa consulta um banco de dados de estudantes para encontrar o conjunto de estudantes com idade inferior a 12 anos e, em seguida, imprime o nome e a idade desses estudantes.

---

[3]Applets são programas Java que podem ser executados a partir de uma página web.

[4]A ObjectDesign também oferece uma versão para C++ das interfaces de programação de ObjectStore, PSE e PSE PRO.

Programa 2.5 ObjectStore_Example1

```
import COM.odi.*; // Importa as classes da interface de ObjectStore, PSE ou PSE PRO.

public class ObjectStore_Example1{
    public static void main(String[] args){
        // Inicia ObjectStore, PSE ou PSE PRO
        ObjectStore.initialize(null, null);
        // Cria e abre um banco de dados de nome "record.odb" para escrita ou leitura.
        Database db = Database.create( "record.odb", ObjectStore.UPDATE);
        // Cria uma transação atômica para escrita de dados.
        Transaction tr = Transaction.begin(ObjectStore.UPDATE);
        Person p = new Person("Peter Smith", 50);
        Student st = new Student("John Smith",14, p);
        // Uma raiz de persistência é criada e associada ao objeto "st".
        // O objeto "st" e todos os objetos alcançados através dele são
        // armazenados no banco de dados.
        db.createRoot("Student1", st);
        // Termina a transação atômica.
        tr.commit();
        // Fecha o banco de dados "record.odb".
        db.close();
    }
}
```

Programa 2.6 ObjectStore_Example2

```
import COM.odi; // Importa as classes da interface de ObjectStore.

public class ObjectStore_Example2{
    public class void main(String[] args){
        //Inicia ObjectStore.
        ObjectStore.initialize(null, null);
        // Abre o banco de dados "record.db" para escrita ou leitura.
        Database db = Database.open("record.db", ObjectStore.UPDATE);
        // Cria uma transação atômica para leitura de dados.
        Transaction tr = Transaction.begin(ObjectStore.READONLY);
        // Obtem a raiz de persistência "AllStudents".
        // Esta raiz é uma coleção de objetos da classe Student.
        SetOfStudents students = (SetOfStudents) db.getRoot("AllStudents");
        // Obtem todos os objetos Student de idade menor que 12,
        // consultando o atributo "age" destes objetos.
        SetOfStudents children = students.query("age < 12");
        // Gera uma enumeração dos elementos da coleção "children".
        StudentsEnumeration se = children.elements();
        // Imprime o nome e a idade dos estudantes.
        while( se.hasMoreElements() ){
            Student s = (Student)se.nextElement();
            System.out.println("Student: "+ s.name +", age:"+ s.age);
        }
        // Termina a transação atômica.
        tr.commit();
        // Fecha o banco de dados.
        db.close();
    }
}
```

Avaliação

Os produtos da ObjectDesign descritos implementam a persistência de objetos Java combinando uma interface de programação para Java e um mecanismo de pós-processamento de classes. Esses produtos incluem um sistema de gerência de banco de dados de objetos. Em PSE e PSE PRO esse sistema de gerência é bastante limitado. Já ObjectStore possui um sistema de gerência de banco de dados mais robusto.

A abordagem desses produtos para implementação da persistência possui várias semelhanças com a abordagem de JUB (Seção 2.4). A unidade de persistência de ambos é o objeto, mas os sistemas de persistência descritos nessa Seção só possuem uma forma de armazenar os objetos persistentes na base de dados, criando raízes de persistência. A semântica de persistência é então associada aos objetos criando raízes de persistência ou por transitividade: objetos referenciados a partir de uma raiz de persistência também serão objetos persistentes. Como em JUB, os produtos da ObjectDesign precisam inserir código nas classes para implementar a persistência. A diferença é que JUB realiza um pré-processamento dessas classes e os outros um pós-processamento. A transparência dos mecanismos que implementam a persistência também é semelhante. Ambos introduzem a semântica de transações nas aplicações e provêem uma interface para armazenamento e recuperação de raízes de persistência.

ObjectStore, PSE e PSE PRO não implementam uma interface para base de dados heterogêneas. Os objetos persistentes são escritos e recuperados de bancos de dados de objetos.

O desempenho de PSE e PSE PRO começa a cair quando os bancos de dados começam a exceder centenas de MBs [29]. Além disso, esses dois produtos não foram projetados para um grande número de usuários concorrentes: PSE e PSE PRO dão suporte a múltiplas leituras, mas as operações de escrita são enfileiradas e os *locks* de escrita acontecem a nível de banco de dados. Se uma aplicação trabalha com um grande volume de atualizações e um bom desempenho é fundamental, é mais aconselhável usar um sistema de gerência de banco de dados mais robusto, como ObjectStore.

2.6 PJama

PJama[5] [5, 36] é um sistema de persistência que implementa a persistência de objetos Java de forma ortogonal a linguagem. Persistência ortogonal significa que qualquer objeto pode ser persistente, independente de seu tipo [5].

Em PJama, a persistência é implementada a partir de modificações na máquina virtual de Java. A Figura 2.4 mostra a JVM modificada. O *Interpreter* e a *Heap* são componentes da JVM original. O *Interpreter* aloca, modifica e lê objetos de/para a *Heap*.

[5]PJama foi inicialmente denominado PJava, mas esse nome foi registrado pela Sun para denotar PersonalJava.

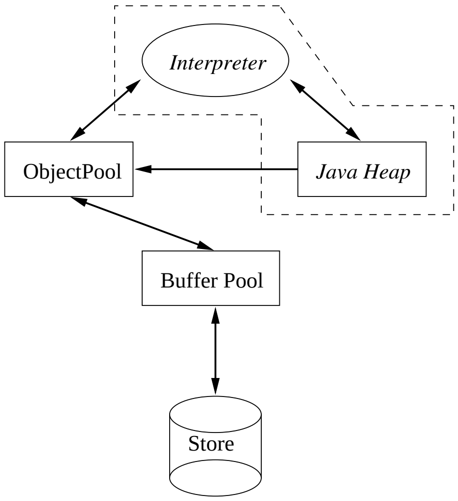

Figura 2.4: JVM modificada por PJama

Para dar suporte a persistência, três novos componentes foram adicionados à JVM: *Store*, *Buffer Pool* e *Object Pool*. *Store* é um repositório de objetos mantido no disco para armazenar os objetos persistentes. *Buffer Pool* é um cache dos objetos armazenados em *Store*. Antes de serem manipulados pelo interpretador, todos os objetos persistentes são copiados de *Buffer Pool* para *Object Pool* num formato similar àquele usado para os objetos na *Heap*. Dessa forma, o código usado para operar objetos na *Heap* também pode operar os objetos em *Object Pool*, com poucas mudanças.

Os novos objetos criados na aplicação são alocados na *Heap*. Quando esses objetos tornam-se persistentes, eles migram para *Object Pool* e são copiados para *Store* através de *Buffer Pool*. Finalmente, todas as modificações em objetos persistentes são propagadas de *Object Pool* para *Store*, novamente através de *Buffer Pool*.

PJama é implementado sem modificar a linguagem, as classes ou o compilador Java. PJama possui uma interface de programação para Java, usada pelo programador para criar as raízes de persistência da aplicação. Essa interface de programação é predominantemente formada de métodos da classe `PJavaStore`.

O Programa 2.7 cria e armazena um objeto persistente da classe `Student` no repositório de objetos. PJama implicitamente inicia uma transação atômica quando o método `main` é ativado. O método `getStore` obtém um objeto que representa o repositório de objetos. Esse objeto será usado para as operações de escrita e leitura no repositório de objetos. O método `newPRoot` cria uma nova raiz de persistência no repositório de objetos `pjs`. Quando o método `main` termina, a transação associada a `main` irá terminar. Se uma exceção ocorrer dentro do método `main` e não for tratada, a transação associada a `main` irá abortar e o repositório de objetos será restaurado para seu estado antes da execução do método `main`.

Programa 2.7 PJama_Example1

```
// Importa as classes PJavaStore e PJSException da API de PJama.
import PJavaStore;
import PJSException;

public class PJama_Example1{
    public static void main( String[ ] args ){  // Transação iniciada.
        Person p = new Person("Peter Smith", 50);
        Student st = new Student("John Smith", 14, p);
        try{
            // Obtém o objeto que representa o repositório de objetos.
            PJavaStore pjs = PJavaStore.getStore();
            // Cria uma nova raiz de persistência, associando
            // o nome "Student1" ao objeto "st".
            pjs.newRoot("Student1", st);
        } catch(PJSException e){ ... }  // Trata as exceções levantadas por PJama.
    }  // Transação finalizada.
}
```

O Programa 2.8 lê e modifica um objeto persistente. Neste exemplo, o método `getPRoot` obtém a raiz de persistência "Student1" do repositório de objetos. Quando um objeto persistente é restaurado, ele pode ser usado como um objeto qualquer da aplicação, como se não fosse persistente. Quando esta transação termina, as mudanças feitas no objeto persistente serão automaticamente preservadas.

Avaliação

PJama é um sistema de persistência que implementa a persistência de objetos Java sem precisar modificar as classes da aplicação. As classes não precisam herdar de uma classe base e nem ser pré ou pós-processadas para que instâncias dessas classes sejam persistentes. Em PJama, persistência e tipo são ortogonais e qualquer classe pode ter instâncias persistentes.

A unidade de persistência é o objeto e a semântica de persistência é associada aos mesmos criando-se raízes de persistência. Também em PJama a persistência é transitiva, sendo propagada para todos os objetos referenciados a partir de uma raiz de persistência.

Para implementar a persistência o programador utiliza a interface de programação de PJama para criar e restaurar as raízes de persistência. Este é praticamente o único esforço de codificação que precisa ser feito para implementar a persistência usando PJama. Com isso, o programador pode concentrar seus esforços na construção do modelo de objetos da aplicação, sendo a persistência introduzida com pequenas modificações no código desta.

Programa 2.8 PJama_Example2

```
// Importa as classes PJavaStore e PJSException da API de PJama.
import PJavaStore;
import PJSException;

public class PJama_Example2{
    public static void main( String[ ] args ){  // Transação iniciada.
        try{
            // Obtém o objeto que representa o repositório de objetos.
            PJavaStore pjs = PJavaStore.getStore();
            // Restaura a raiz de persistência "Student1".
            Student st = (Student) pjs.getPRoot("Student1");
            // Modifica o atributo "age" do objeto persistente.
            st.age = 15;
        } catch(PJSException e){ ... }  // Trata as exceções levantadas por PJama.
    }  // Transação finalizada.
}
```

Essa transparência das operações persistentes foi alcançada a partir de modificações na JVM. Essas modificações foram feitas de forma a minimizar as modificações nos componentes da máquina virtual (*Interpreter* e *Heap*). Isto é necessário para que PJama possa acompanhar, com mais facilidade, as novas versões da JVM.

2.7 Conclusão

Descrevemos um conjunto de sistemas de persistência que ilustram as diferentes abordagens para a implementação da persistência em Java e, de certa forma, para persistência em geral. A Tabela 2.1 apresenta um resumo das características dos sistemas de persistência descritos e a Tabela 2.2 apresenta uma avaliação desses sistemas de persistência quanto a transparência dos mecanismos para persistência e a facilidade de uso dos mesmos. As duas tabelas representam um resumo das informações contidas na Seção de Avaliação de cada sistema de persistência.

Pela Tabela 2.2 observa-se que a avaliação da transparência e da facilidade de uso foi a mesma em quase todos os sistemas de persistência. A partir dessa avaliação, concluímos que a maior transparência dos mecanismos que implementam a persistência dos dados é fator determinante na facilidade de uso do sistema de persistência e, consequentemente, na maior eficiência no desenvolvimento das aplicações.

Produtos	*Hetero-geneidade*	*Mapeamento OO/Relacional*	*Unidade de Persistência*	*Atributo de Persistência*	*Transiti-vidade*
Linearização em Java	—	—	tipo primitivo/ objeto	—	$\surd$
Persistência e Java Core Reflection	—	—	objeto	herança	$\surd$
JDBC	$\surd$	—	tipo primitivo/ objeto	—	—
JUB	$\surd$	$\surd$	objeto	raiz de persistência	$\surd$
ObjectStore/ PSE/PSE PRO	—	—	objeto	raiz de persistência	$\surd$
PJama	—	—	objeto	raiz de persistência	$\surd$

Tabela 2.1: Resumo dos sistemas de persistência descritos

Legenda

$\surd$ Este símbolo indica que o sistema de persistência possui a característica determinada pelo aspecto avaliado.
— Este símbolo indica que o sistema de persistência não possui a característica determinada pelo aspecto avaliado ou tal aspecto não é pertinente ao sistema de persistência.

Produtos	*Transparência*	*Facilidade de uso*
Linearização em Java	⋆⋆	⋆⋆
Persistência e Java Core Reflection	⋆⋆⋆	⋆⋆⋆
JDBC	⋆⋆	⋆
JUB	⋆⋆⋆	⋆⋆⋆
ObjectStore/ PSE/PSE PRO	⋆⋆⋆	⋆⋆⋆
PJama	⋆⋆⋆⋆	⋆⋆⋆⋆

Tabela 2.2: Avaliação dos sistemas de persistência descritos

Legenda

⋆ ruim
⋆⋆ regular
⋆⋆⋆ bom
⋆⋆⋆⋆ muito bom

Analisando as diferentes abordagens para persistência descritas, elaboramos um conjunto de características desejáveis em sistemas de persistência de objetos:

• ***Ortogonalidade entre tipo e persistência*** - Qualquer objeto pode ser persistente, independente do seu tipo [5], sendo possível criar instâncias persistentes e voláteis de uma mesma classe.

Ortogonalidade entre tipo e persistência é implementada por PJama e JDBC, onde as classes não precisam ser previamente rotuladas como persistentes para que instâncias persistentes dessas classes sejam criadas. Usando a interface de linearização de Java, Persistência e Java Core Reflection ou JUB, as classes com instâncias persistentes precisam herdar de uma classe base, sendo que, em JUB as classes com instâncias persistentes ainda precisam ser pré-processadas. Já com ObjectStore, PSE e PSE PRO essas classes precisam ser pós-processadas para que o comportamento persistente de suas instâncias seja implementado. Dessa forma, as classes da aplicação ficam divididas em 2 grupos: o grupo das classes cujas instâncias podem ser persistentes e o grupo das classes que não podem ter instâncias persistentes. Com Persistência e Java Core Reflection a flexibilidade é ainda menor. Não é possível criar instâncias persistentes e voláteis de uma mesma classe. Se uma classe pode ter instâncias persistentes, então todas as instâncias da classe serão persistentes.

- ***Transparência total dos mecanismos para persistência*** - A persistência dos objetos da aplicação é implementada de forma ortogonal ao programa da aplicação e sem alterar o código do mesmo. Os objetos persistentes e voláteis são manipulados da mesma forma pelo programador, que pode focalizar seus esforços na implementação da funcionalidade básica da aplicação.

  A transparência total dos mecanismos para persistência não é alcançada em nenhum dos sistemas de persistência descritos. Todos expõem os mecanismos para a persistência através de uma interface de programação, que é usada pelo programador para implementar a persistência dos objetos.

  Com o uso de reflexão computacional pode-se conseguir a transparência desejada dos mecanismos que implementam a persistência [24, 35]. A persistência é implementada em um meta-nível cuja computação é transparente para a aplicação de objetos persistentes. Esta separação de conceitos favorece a reutilização de código da aplicação e pode aumentar a eficiência no desenvolvimento das aplicações.

- ***Interface para bases de dados heterogêneas*** - Objetos persistentes de uma mesma aplicação podem ser armazenados e recuperados de bases de dados de tipos diferentes, como por exemplo, bancos de dados relacionais, bancos de dados de objetos e sistemas de arquivos.

  Um sistema de persistência com interface para bases de dados heterogêneas só foi implementado por JDBC e JUB, mesmo assim, com restrições sobre os tipos de base de dados que podem ser acessadas. O ideal, seria um sistema de persistência que pudesse ser adaptado para comunicar-se com qualquer tipo de base de dados. Isto seria particularmente útil para integrar bases de dados já existentes.

Para a construção de um sistema de persistência com essas características utilizamos uma abordagem que combina as técnicas de reflexão computacional e arcabouços de software. No próximo Capítulo, descrevemos esse sistema de persistência e mostramos como essas características são contempladas e os benefícios que a nossa abordagem pode trazer.

Lista de URLs

A Tabela 2.3 contém uma lista das URLs de alguns produtos e padrões citados nesse Capítulo, onde mais informações sobre os mesmos podem ser encontradas.

Produto	*URL*
ANSI	http://web.ansi.org/
Java Core Reflection	http://java.sun.com/products/jdk/1.1/docs/guide/reflection/index.html
JDBC	http://splash.javasoft.com/jdbc/
JDK	http://java.sun.com/products/jdk/1.1/
Linearização em Java	http://chatsubo.javasoft.com/products/jdk/rmi/serial/index.html
Padrão ODMG	http://www.odmg.org/
PJama	http://www.dcs.gla.ac.uk/pjava/
ObjectStore, PSE e PSE PRO	http://www.odi.com/

Tabela 2.3: Lista de URLs

Capítulo 3

Um Arcabouço Reflexivo para Persistência de Objetos

Este artigo será submetido ao SBLP'2000—IV Simpósio Brasileiro de Linguagens de Programação—, evento a ser realizado de 17-19 de maio de 2000, em Recife, PE.

Um Arcabouço Reflexivo para Persistência de Objetos

Nádia Leão Pereira Quadros — nadia.pereira@dcc.unicamp.br
Luiz Eduardo Buzato — buzato@dcc.unicamp.br

Laboratório de Sistemas Distribuídos
Instituto de Computação - UNICAMP

Resumo

Várias abordagens têm sido propostas para permitir a persistência dos objetos de uma aplicação: modificação de uma linguagem de programação e/ou de seu compilador ou interpretador, definição de uma interface de programação para persistência, persistência através de herança.

Neste artigo propomos uma abordagem alternativa para persistência de objetos, que combina as técnicas de reflexão computacional e arcabouços de software. Com reflexão computacional, propomos a separação de aspectos da computação realizada por uma aplicação: aspectos gerenciais não diretamente ligados à função primordial da aplicação, como persistência de objetos, são implementados em módulos de software transparentes e ortogonais à aplicação. O módulo de software que implementa a persistência é construído na forma de um arcabouço de software, como uma infra-estrutura genérica e flexível que pode ser adaptada de acordo com os requisitos de persistência de cada aplicação.

São várias as vantagens dessa abordagem: construção de um sistema de persistência flexível e adaptável, que favorece a reutilização de software; implementação da persistência de forma totalmente transparente para a aplicação que manipula os objetos persistentes; persistência incorporada sem que o código da aplicação sofra alterações; aumento da produtividade nas etapas de construção e manutenção das aplicações.

Abstract

Many different approaches have being proposed to allow object persistence: extension of a programming language, of its compiler or its interpreter, definition of an application programming interface for persistence, and persistence by inheritance.

This article describes an alternative approach to object persistence, based on computational reflection and frameworks. Computational reflection provides separation of concerns: management functions, like object persistence, are implemented in orthogonal and transparent modules independent of the application. The module that implements persistence is a generic framework that can be adapted to satisfy persistent service requirements of different applications.

The benefits of our approach are: a flexible and easily adaptable persistent service which aids software reuse; persistent operations orthogonal and transparent to the application; implementation of object persistence without changing the application code, with increased productivity in application development and maintenance.

3.1 Introdução

Várias abordagens têm sido propostas para permitir a persistência dos objetos de uma aplicação: modificação de uma linguagem de programação e/ou de seu compilador ou interpretador [5, 17, 36], definição de uma interface de programação[1] para persistência [16, 28, 46], persistência através de herança [10, 23].

O sistema descrito em [5, 36], denominado PJama, implementa a persistência dos objetos Java modificando a máquina virtual da linguagem. Dessa forma, PJama consegue tornar transparente para a aplicação que manipula os objetos persistentes, algumas operações necessárias para a implementação da persistência. Ainda assim, é preciso invocar operações para criação e restauração de objetos persistentes no código da aplicação.

Com PSE (Persistent Store Engine) [28] e JDBC (Java Database Connectivity) [16] a persistência dos objetos é implementada através de uma interface de programação. O primeiro provê uma interface de programação para o armazenamento e a recuperação de objetos Java ou C++ em bancos de dados orientados a objetos. O segundo é uma interface de programação de Java que provê acesso uniforme para dados armazenados em diferentes bancos de dados relacionais. Nos dois casos, para implementar a persistência é preciso invocar no código da aplicação um número muito grande de métodos das interfaces de programação. Esse grande volume de código extra inserido para implementar a persistência dificulta a reutilização das classes da aplicação.

Arjuna [10] implementa a persistência de objetos C++ através de herança: todas as classes cujas instâncias são persistentes herdam de uma classe base que implementa os mecanismos para persistência. Alguns métodos da classe base precisam ser sobrecarregados nas subclasses para permitir a escrita e leitura do estado do objeto persistente. Dessa forma, se uma aplicação deseja incorporar persistência usando Arjuna, todas as classes cujas instâncias são persistentes precisam ser rescritas, para que a semântica de persistência seja introduzida nessas classes.

Nas diferentes abordagens apresentadas, existe a preocupação em esconder do programador detalhes da implementação do sistema de persistência, com o objetivo de facilitar a construção de aplicações persistentes. Entretanto, em nenhuma dessas abordagens a persistência é implementada de forma totalmente transparente para o programador da aplicação e, em todas, o código da aplicação precisa ser alterado para incorporar a persistência.

Neste artigo propomos uma abordagem alternativa para persistência de objetos. Nossa proposta baseia-se na separação de aspectos da computação realizada por uma aplicação: aspectos gerenciais não diretamente ligados à função primordial da aplicação, como persistência de objetos, são implementados de forma transparente e ortogonal à aplicação. Essa é uma forma diferente de estruturação de sistemas, alcançada com o uso de reflexão compu-

[1] Em Inglês: *Application Program Interface* (API).

tacional [26, 41]. Com essa técnica, o sistema tem acesso a informações sobre sua estrutura interna e sobre sua computação e usa essas informações para implementar as funções gerenciais da aplicação de forma transparente para a mesma. Acreditamos que esta separação de aspectos da computação aumenta a produtividade nas etapas de construção e manutenção de aplicações e facilita a reutilização das classes dessas aplicações.

Além do uso de reflexão computacional, nossa abordagem se difere das demais porque não construímos um sistema de persistência particular, mas um arcabouço de software[2] [19, 45] para a construção desse sistema de persistência. O arcabouço para persistência é uma infra-estrutura genérica e flexível que pode ser adaptada de acordo com os requisitos de persistência de cada aplicação. Um dos pontos fortes do arcabouço é a existência de uma interface genérica para operações de escrita e leitura de dados em bases de dados, a partir da qual pode-se construir interfaces de acesso a quaisquer tipos de base de dados. Esta flexibilidade permite às aplicações manipularem dados de bases de dados heterogêneas.

O artigo está organizado da seguinte forma: a Seção 3.2 descreve as técnicas de reflexão computacional e arcabouços de software; a Seção 3.3 descreve nossa solução para o problema de persistência de objetos: um *arcabouço reflexivo para persistência de objetos*; a Seção 3.4 apresenta vários exemplos de uso do arcabouço e a Seção 3.5 conclui o trabalho.

3.2 Reflexão Computacional e Arcabouços

Esta Seção descreve as técnicas de reflexão computacional e arcabouços de software. Em cada descrição enumeramos os benefícios que essas técnicas podem trazer aos sistemas de software.

3.2.1 Reflexão Computacional

Reflexão computacional é uma técnica de estruturação de sistemas que permite a um sistema manter informações a seu respeito e usar essas informações para modificar o seu comportamento. Um sistema computacional que incorpora reflexão possui pelo menos dois níveis de abstração: o nível base e o meta-nível [26].

A tarefa do nível base é a de implementar aspectos funcionais da aplicação, enquanto o meta-nível cuida da implementação dos requisitos gerenciais tais como persistência [24, 35], tolerância a falhas [12, 25, 47], replicação [22] e distribuição [42]. Além disso, a implementação do meta-nível é feita de forma não intrusiva—a aplicação, no nível base, não tem que mudar sua aparência (código) ou comportamento (execução) se for incorporado ao sistema um novo mecanismo, por exemplo, o de persistência.

[2] Em Inglês: *frameworks*.

Os elementos do meta-nível são chamados meta-objetos e são responsáveis pelo comportamento reflexivo dos objetos do nível base. Os meta-objetos guardam informações e estendem o comportamento de um objeto do nível base, dito seu **referente** [25]. Para que um meta-objeto possa refletir sobre seus objetos referentes são necessárias informações sobre a estrutura interna desses objetos, tais como que métodos são implementados pelos objetos e quais são os atributos desses objetos. Tais informações são chamadas meta-informações estruturais e o processo de materialização consiste em representá-las na forma de objetos.

Interações entre objetos também devem ser materializadas para que o meta-nível possa inspecioná-las e possivelmente alterá-las. Isto é alcançado interceptando operações do nível base, criando objetos que representem essas operações (materialização) e transferindo o controle para o meta-nível. Recebendo operações do nível base materializadas, os meta-objetos podem modificar o efeito dessas operações, implementando de forma transparente os requisitos gerenciais da aplicação.

Por exemplo, para um determinado objeto O pode existir um meta-objeto M_O associado a ele que cuida da implementação de persistência no sistema. Cada mensagem s de acesso à base de dados enviada para O é interceptada, materializada e dirigida ao seu meta-objeto M_O. De posse desta meta-informação, o meta-objeto M_O realiza as operações necessárias para a implementação da persistência e depois retorna o controle para o nível base. O objeto emissor da mensagem s não toma conhecimento da computação reflexiva: ele envia a mensagem solicitando serviços de um objeto e recebe o resultado esperado, sem saber que a mensagem foi desviada para o meta-objeto M_O (Figura 3.1).

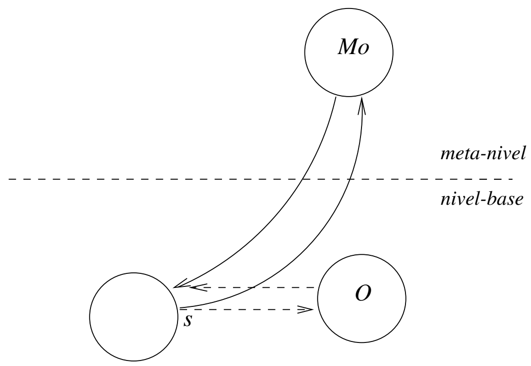

Figura 3.1: Interceptação de mensagens

Uma linguagem de programação possui uma arquitetura reflexiva se provê maneiras de conduzir a reflexão computacional explicitamente [26]. O interpretador da linguagem deve dar ao sistema em execução acesso aos dados que representam aspectos deste mesmo sistema. Sistemas implementados em tais linguagens podem executar a computação reflexiva incluindo código que descreva como os dados do sistema de execução devem ser manipulados.

Um dos aspectos mais importantes de uma arquitetura reflexiva orientada a objetos é seu protocolo de meta-objetos. O protocolo de meta-objetos[3] (MOP) define a interação entre objetos e meta-objetos.

Open C++ [8, 9], MetaXa [14, 21, 22] e Guaraná [31, 33] são exemplos de arquiteturas reflexivas que implementam reflexão através de um MOP. Open C++ foi construído a partir de modificações no compilador da linguagem C++, enquanto MetaXa e Guaraná foram construídas a partir de modificações no interpretador da linguagem Java.

Segundo Kleinöder [21], um ambiente de programação que incorpora uma arquitetura reflexiva tem as seguintes vantagens:

- ***Aumento na produtividade*** - A programação em meta-nível leva à construção de programas mais estruturados e de fácil manutenção.
- ***Separação de conceitos*** - O uso de um meta-espaço separado permite ao programador da aplicação focalizar no domínio desta. A funcionalidade adicional é fornecida por uma biblioteca de meta-componentes ou é desenvolvida juntamente com o programa da aplicação, mas de forma ortogonal ao mesmo.
- ***Configurabilidade*** - O meta-nível estabelece uma arquitetura aberta. Novas políticas podem ser implementadas, como por exemplo persistência ou distribuição, sem mudar o código da aplicação.
- ***Transparência*** - A computação do meta-nível é transparente à aplicação que reside no nível base.

3.2.2 Arcabouços de Software

Um dos maiores desafios da engenharia de software moderna está na criação de sub-sistemas autônomos e reutilizáveis que fatoram funcionalidades úteis a muitas aplicações ou a um domínio de aplicações. Nesse sentido, orientação a objetos forneceu aos engenheiros uma ferramenta poderosa pois uma classe abstrata pode ser vista como o projeto de um sub-sistema reutilizável para um objeto. Entretanto, tem sido observado que a reutilização de classes ou mesmo de bibliotecas de classes não provê um grau satisfatório de reutilização. Classes e objetos possuem granularidade pequena para alcançar o nível de reutilização desejado [13].

O conceito de arcabouços de software orientados a objetos [19, 45] tem sido proposto como alternativa para alcançar reutilização de software em grande escala. Um arcabouço de software é uma arquitetura de software que materializa o projeto para um conjunto de objetos que cooperam para implementar uma certa funcionalidade ou responsabilidade. Arcabouços

[3] Em Inglês: *MetaObject Protocol*

permitem reutilizar não apenas componentes isolados mas toda a arquitetura de um domínio específico. Esta reutilização de projeto em larga escala é a principal característica que diferencia os arcabouços das bibliotecas de classes.

A arquitetura de um arcabouço caracteriza-se por pontos fixos, que não podem ser mudados e por pontos adaptáveis, que se destinam a acomodar mudanças. Os pontos fixos de um arcabouço definem sua estrutura geral, sua divisão em classes e as responsabilidades e colaborações entre essas classes, bem como o fluxo de controle mantido entre suas instâncias. Assim, o programador pode concentrar-se nos aspectos específicos de sua aplicação, que são definidos completando-se os pontos adaptáveis do arcabouço. Dessa forma, os programadores implementam código que será chamado pelo arcabouço (pontos adaptáveis), ao invés de simplesmente implementarem código que invoca bibliotecas de classes.

Dentre as principais vantagens da utilização de arcabouços no desenvolvimento de aplicações podemos citar [34]: (i) infra-estrutura pré-fabricada que reduz tempo e recursos gastos com codificação, rastreamentos e testes, (ii) abstração de serviços: o arcabouço implementa a complexidade da aplicação e (iii) redução de custos em manutenção.

3.3 Um Arcabouço Reflexivo para Persistência de Objetos

O arcabouço reflexivo para persistência de objetos foi construído com base em três decisões de projeto: (i) prover um sistema de persistência transparente para a aplicação, (ii) prover um sistema de persistência que permita o acesso a dados armazenados em bases de dados heterogêneas e (iii) oferecer persistência ortogonal a tipo. A seguir, mostramos como cada uma dessas decisões de projeto influenciou a construção da arquitetura de software do arcabouço.

3.3.1 Transparência

A preocupação com a transparência das operações persistentes é um assunto recorrente em todos os sistemas para persistência. PJama [5, 36] é um exemplo de sistema de persistência que alcança um alto grau de transparência. Mas, mesmo em PJama, é preciso inserir código para a persistência no código da aplicação.

Nossa pretensão é alcançar transparência total das operações persistentes, implementando a persistência dos objetos de uma aplicação de forma ortogonal a mesma. Para atingir esses objetivos usamos a técnica de reflexão computacional. O arcabouço foi construído no meta-nível da aplicação que manipula os objetos persistentes de forma que sistema de persistência e aplicação possam ser desenvolvidos independentemente, por programadores diferentes se necessário.

A ligação entre o sistema de persistência e a aplicação é feita associando-se um meta-objeto de persistência a cada objeto persistente da aplicação. A Figura 3.2 mostra essa associação. O meta-objeto (MO) é o componente do sistema de persistência que coordena as operações para implementação da persistência do objeto associado a ele. Os meta-objetos implementam a persistência interceptando as operações de escrita e leitura no estado do objeto associado a ele e propagando essas operações para a base de dados. Este mecanismo de interceptação de operações torna a persistência transparente para a aplicação, uma vez que não há invocação de operações persistentes no código que implementa a funcionalidade da aplicação.

A associação entre objetos e meta-objetos também é feita no meta-nível da aplicação. Com isso, garantimos transparência total do sistema de persistência. Nas Seções 3.4.2 e 3.4.3 mostramos como essa associação é feita no meta-nível e como é possível associar um comportamento persistente aos objetos de uma aplicação modificando apenas o meta-nível dessa aplicação.

A forma como é feita a associação e a interação entre objeto e meta-objeto depende do MOP utilizado para implementar o arcabouço. O arcabouço foi implementado na linguagem Java utilizando Guaraná [33], uma arquitetura de software reflexiva e um sistema de execução baseado em Java, que não requer mudança na linguagem de programação Java—um programa criado e compilado em qualquer compilador Java irá rodar sob Guaraná. A medida que descrevemos o arcabouço, descreveremos também algumas particularidades do MOP de Guaraná.

3.3.2 Heterogeneidade

São poucas as implementações de persistência que provêem acesso a dados armazenados em bases de dados heterogêneas. JUB (Java Universal Binding) [46] é um exemplo de sistema de persistência para Java que provê acesso a bases de dados heterogêneas. JUB provê uma interface uniforme de acesso a bancos de dados O2, Oracle, Sybase e a todos os bancos de dados relacionais que podem ser acessados via JDBC [16], escondendo o mapeamento do modelo de objetos de Java no modelo de dados da base de dados usada.

Nosso objetivo é construir um sistema de persistência que além de transparente para a aplicação também permita a mesma manipular dados armazenados em bases de dados heterogêneas. Não pretendemos construir um conjunto fixo de interfaces para bases de dados específicas, como faz JUB, por dois motivos: (i) não queremos criar restrições quanto as bases de dados que podem ser utilizadas pelo sistema de persistência e (ii) não pretendemos determinar como deve ser o mapeamento do modelo de objetos Java no modelo de dados da base de dados, deixamos isso a cargo do programador que melhor sabe como esse mapeamento deve ser feito. Quando for construir seu sistema de persistência, vamos dar ao programador a opção de reutilizar uma interface para base de dados preexistente ou construir uma nova interface que atenda às suas necessidades.

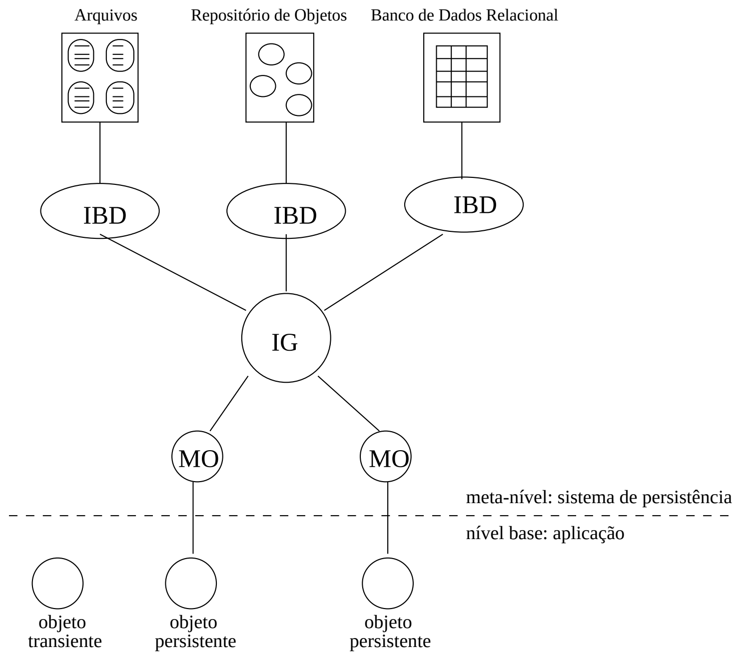

Figura 3.2: Visão geral do arcabouço

Para alcançar esse objetivo, construímos nosso sistema de persistência na forma de um arcabouço de software, com uma Interface Genérica (IG) de acesso às bases de dados (Figura 3.2). MO delega todas as operações de escrita e leitura nas bases de dados para IG, que se encarrega de rotear essas operações para uma Interface de Base de Dados específica (IBD). O programador, quando for construir seu sistema de persistência, terá que construir/reutilizar os IBDs que precisa para recuperar e armazenar dados nas bases de dados que pretende usar. Os IBDs implementam as operações de leitura e escrita de dados traduzindo essas operações para a sintaxe própria do sistema de banco de dados associado a eles. Por exemplo, se uma operação de leitura é requisitada a uma IBD para banco de dados relacional, uma sentença SQL do tipo `Select <campos> from <tabela>` será gerada e enviada ao banco de dados.

3.3.3 Ortogonalidade entre Tipo e Persistência

Ortogonalidade entre tipo e persistência significa que qualquer objeto pode ser persistente, independente do seu tipo [5]. PJama [5, 36] é um exemplo de sistema de persistência com essa característica.

No sistema de persistência que estamos propondo, qualquer classe da aplicação pode ter instâncias persistentes, basta que essas instâncias estejam associadas a um meta-objeto de persistência (MO—Figura 3.2). Essa associação é feita no sistema proposto criando-se raízes de persistência.

Uma raiz de persistência é um nome a partir do qual pode-se recuperar um objeto mesmo depois que a aplicação que o criou tiver terminado. Os objetos da aplicação tornam-se persistentes quando associados a uma raiz de persistência. Quando um objeto é associado a uma raiz de persistência, ele é automaticamente associado a um meta-objeto (MO) que implementa a persistência. Os objetos referenciados a partir do objeto raiz também adquirem comportamento persistente por transitividade e, consequentemente, são associados a um meta-objeto que implementará a sua persistência. Na Seção 3.4.1 mostramos como criar as raízes de persistência da aplicação usando o arcabouço reflexivo construído.

Na próxima Seção, descrevemos a arquitetura de software do arcabouço em termos de suas classes e relacionamentos entre essas classes, apontando os pontos fixos e adaptáveis dessa arquitetura.

3.3.4 Descrição da Arquitetura do Arcabouço

O arcabouço é uma infra-estrutura genérica para a construção de um sistema de persistência específico. A Figura 3.3 apresenta um diagrama de classes com as principais classes do arcabouço. Esse diagrama segue a notação proposta por Booch, Rumbaugh e Jacobson denominada Unified Modeling Language (UML) [6].

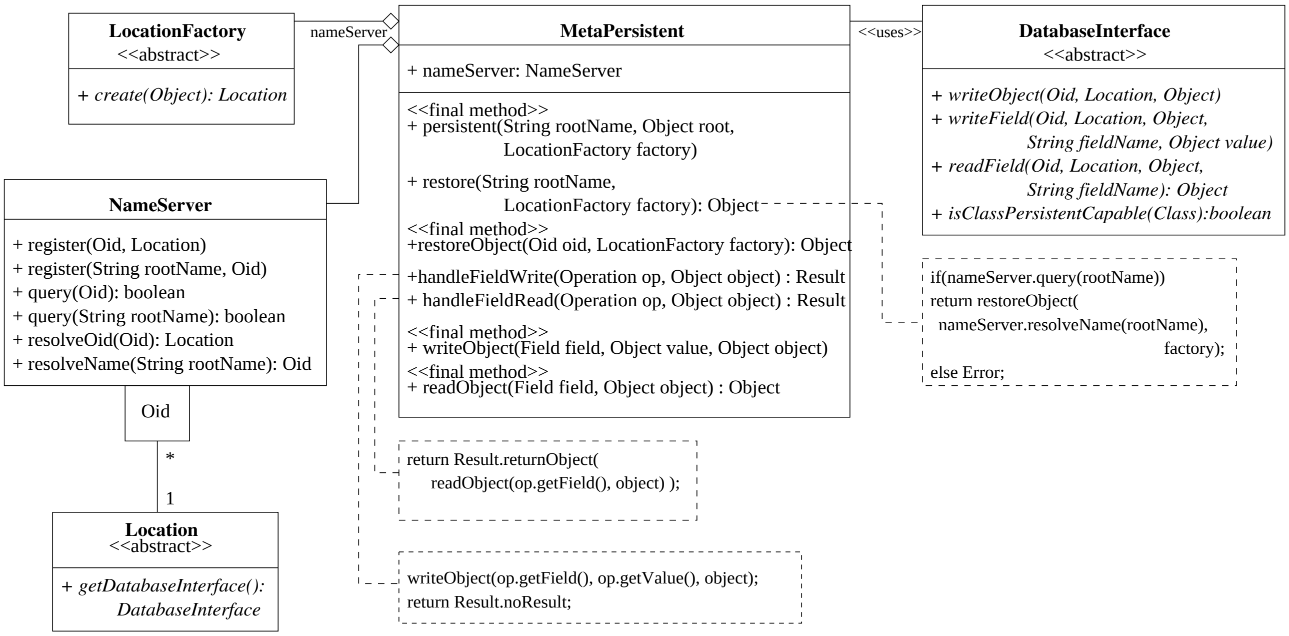

Figura 3.3: Diagrama de classes do arcabouço

A classe `MetaPersistent` (Figura 3.3) desempenha o papel de um gerente de persistência, coordenando a colaboração entre as classes do arcabouço para a implementação da persistência. As instâncias dessa classe são os meta-objetos, descritos na Seção 3.3.1, responsáveis pelo comportamento persistente dos objetos da aplicação.

A classe `DatabaseInterface` (Figura 3.3) provê para a classe `MetaPersistent` uma interface para escrita e leitura de objetos independente da base de dados onde o objeto está armazenado. Os meta-objetos da classe `MetaPersistent` implementam a persistência interceptando as operações de escrita e leitura ao estado dos objetos associados a eles e delegando essas operações para instâncias de subclasses de `DatabaseInterface` .

As classes `NameServer`, `Oid` e `Location` (Figura 3.3) implementam o subsistema de identificação e localização de objetos persistentes. A classe `Location` representa a localização de um objeto persistente, a classe `Oid` representa a identificação única[4] (OID) desse objeto e a classe `NameServer` armazena a identificação e a localização dos objetos persistentes.

A classe `LocationFactory` (Figura 3.3) é responsável por determinar a base de dados onde serão armazenados os objetos persistentes da aplicação. Toda instância da classe `MetaPersistent` está associado a um objeto dessa classe e requisita suas funções sempre que um objeto da aplicação é promovido a objeto persistente.

Os sistemas de persistência construídos a partir do arcabouço fazem parte do meta-nível da aplicação. No entanto, convencionamos chamar de meta-objetos apenas instâncias da classe `MetaPersistent`, pois são elas que estão diretamente associadas aos objetos persistentes, estabelecendo a ligação entre a aplicação e o sub-sistema de meta-nível que implementa a persistência.

Para utilizar o arcabouço é preciso conhecer e entender os pontos fixos e adaptáveis de sua arquitetura. Descrevemos primeiro os pontos fixos do arcabouço e em seguida os seus pontos adaptáveis.

Pontos Fixos

Os pontos fixos do arcabouço para persistência estão todos definidos na classe `MetaPersistent`. São eles:

- ***Criação de raízes de persistência*** - As raízes de persistência são criadas usando o método `persistent` (Figura 3.3). Esse método associa o nome `rootName` ao objeto `root` e associa um meta-objeto de persistência a esse objeto e a todos os objetos alcançados através dele. O objeto `factory` gera um objeto de localização para cada objeto persistente associado à raiz, incluindo o objeto `root`. O método `persistent` não pode ser redefinido em subclasses da classe `MetaPersistent`.

[4] Em Inglês: *unique object identifier.*

- ***Propagação da persistência*** - Quando um objeto persistente faz referência a um objeto volátil este último é promovido a objeto persistente, a menos que o modificador `transient` da linguagem Java apareça na declaração do objeto, indicando que o mesmo é uma instância volátil da aplicação. A propagação da persistência é implementada no método `writeObject` da classe `MetaPersistent` (Figura 3.3). Esse método não pode ser redefinido em subclasses de `MetaPersistent`.

O método `writeObject` é invocado quando o atributo (`field`) de um objeto persistente (`object`) deve ser atualizado na base de dados. O novo valor desse atributo (`value`) pode ser um objeto persistente da aplicação. Neste caso, o OID do objeto persistente é escrito na base de dados no lugar do próprio objeto. Quando for necessário, o objeto persistente poderá ser restaurado da base de dados através de seu OID. Se, por outro lado, `value` não for um objeto persistente da aplicação, então é necessário fazê-lo persistente. Neste caso, invoca-se o método `isClassPersistentCapable` (Figura 3.3) da interface de base de dados usada para escrever/ler o objeto persistente `object`. Esse método é invocado pelo sistema de persistência sempre que um objeto precisa ser escrito na base de dados. Esse método determina, a partir da classe do objeto, se a persistência deve ser propagada para o mesmo. Em caso afirmativo, o objeto `value` é associado a um meta-objeto de persistência e o seu OID é escrito na base de dados. Caso contrário, o objeto `value` é escrito diretamente na base de dados. Na Seção 3.4.1 apresentamos um exemplo que ilustra o papel do mecanismo implementado no método `isClassPersistentCapable`.

- ***Restauração de objetos persistentes*** - O método `restoreObject` (Figura 3.3) restaura o objeto persistente de identificação `oid`, retornando o objeto restaurado. A restauração é rasa, ou seja, cria apenas a casca do objeto persistente. Esse objeto é associado a um meta-objeto de persistência e seus atributos são restaurados sob demanda. Os atributos do objeto persistente declarados `transient` são iniciados pelo interpretador de Java. A função do objeto `factory` é a mesma descrita anteriormente, ou seja, gerar objetos de localização para cada objeto promovido a persistente a partir do objeto restaurado.

O método `readObject` (Figura 3.3) obtém da base de dados o valor do atributo `field` do objeto persistente `object`. O valor obtido pode ser um objeto ou um OID representando o objeto persistente.

Os métodos `restoreObject` e `readObject` não podem ser redefinidos em subclasses da classe `MetaPersistent`.

Pontos Adaptáveis

Alguns pontos adaptáveis do arcabouço possuem uma implementação pré-definida, que é usada caso o programador não queira, ou não precise, definir uma implementação específica para eles. Assim, não é preciso adaptar todos esses pontos para construir um sistema de persistência a partir do arcabouço. Isto deve ser feito se a implementação pré-definida não satisfizer os requisitos do sistema de persistência que se quer construir. A descrição dos pontos adaptáveis do arcabouço é acompanhada da implementação pré-definida do mesmo, caso exista uma. São eles:

• ***Interface de base de dados*** - Sistemas de persistência que provêem acesso a bases de dados heterogêneas precisam implementar várias interfaces de acesso às base de dados, uma para cada tipo de base de dados usada. Isto acontece porque bases de dados de tipos diferentes possuem uma interface de acesso diferente. A classe abstrata `DatabaseInterface` (Figura 3.3) define uma interface para leitura e escrita de objetos e atributos de objetos nas bases de dados. As diferentes interfaces de base de dados são construídas a partir de especializações dessa classe. Cada interface de base de dados construída deve implementar os métodos abstratos definidos na interface de `DatabaseInterface`, usando a sintaxe de um banco de dados particular.

Implementação pré-definida: O arcabouço implementa três interfaces de base de dados específicas: `PSEInterface`, `SerializationInterface` e `JDBCInterface`. Essas interfaces serão descritas na Seção 3.4.1.

• ***Localização dos objetos persistentes*** - A existência de bases de dados heterogêneas requer diferentes formas de localização de um objeto persistente, pois para cada base de dados pode ser necessário um conjunto de informações diferentes para localizar o objeto persistente. A classe abstrata `Location` (Figura 3.3) representa a localização de um objeto persistente. Essa classe deve ser especializada para a criação dos objetos de localização necessários.

Todo objeto de localização deve conhecer a interface de base de dados usada no acesso à base de dados referenciada por ele. O método abstrato `getDatabaseInterface` (Figura 3.3) deve ser redefinido em subclasses da classe `Location` e retornar essa interface.

Implementação pré-definida: O arcabouço implementa um objeto de localização na classe `OAddress`. As interfaces de base de dados pré-definidas do arcabouço usam objetos dessa classe para localizar os objetos persistentes na base de dados. A classe `OAddress` é descrita na Seção 3.4.1.

• ***Alocação de objetos persistentes nas bases de dados*** - Quando um objeto torna-se persistente é preciso determinar a base de dados onde esse objeto será armazenado. Esta função é atribuída à classe abstrata `LocationFactory` (Figura 3.3). O método abstrato `create(Object)` deve ser redefinido nas subclasses e gerar objetos de localização (instâncias de subclasses da classe `Location`) para os novos objetos persistentes.

• ***Servidor de nomes*** - A função de um servidor de nomes é guardar a identificação e a localização dos objetos persistentes da aplicação. Entretanto, um servidor de nomes não é um componente exclusivo do sistema de persistência, suas funções também são necessárias em outras áreas como distribuição e replicação de objetos. No arcabouço, construímos um servidor de nomes específico para persistência na classe `NameServer` (Figura 3.3). A semântica do servidor de nomes pode ser estendida especializando-se a classe `NameServer`.

Implementação pré-definida: As raízes de persistência e os objetos persistentes são registrados no servidor de nomes através dos métodos `register(String rootName, Oid)` e `register(Oid, Location)`, respectivamente. Consultas são submetidas ao servidor de nomes para determinar se um objeto é uma raiz de persistência ou se é um objeto persistente (métodos `query(String rootName)` e `query(Oid)`, respectivamente), para determinar a localização de um objeto a partir de seu OID (método `resolveOid(Oid)`) e para determinar o OID de uma raiz de persistência (método `resolveName(String rootName)`).

- ***Política de leitura e escrita de objetos persistentes*** - Quando um objeto é promovido a objeto persistente, ele tem seu estado armazenado na base de dados. A partir daí, toda leitura e escrita em atributos desse objeto é interceptada pelo meta-objeto associado a ele, que pode ou não propagar a operação interceptada para a base de dados, ou seja, obter ou modificar o valor do atributo na base de dados. A política de leitura e escrita de objetos persistentes determina quando as operações interceptadas devem ser propagadas para a base de dados. Essa política é implementada nos métodos `handleFieldRead` e `handleFieldWrite` da classe `MetaPersistent` (Figura 3.3). Esses métodos são invocados sempre que uma operação de leitura ou escrita em atributos do objeto persistente é interceptada. O parâmetro `Operation` representa a operação interceptada do nível base e o parâmetro `Object` é uma referência para o objeto persistente onde a operação foi invocada. Esses métodos retornam um objeto `Result`, que representa um resultado para a operação interceptada. Esse resultado pode ser vazio (`Result.noResult`), indicando que a operação interceptada não será alterada, ou pode conter um objeto (`Result.returnObject(Object)`). Nesse último caso, a operação interceptada não é executada no nível base e o objeto retornado representa o resultado dessa operação. Os objetos `Operation` e `Result` são elementos definidos no MOP de Guaraná.

O arcabouço possui uma implementação pré-definida para a política de leitura e escrita de objetos persistentes. Para modificar essa política, os métodos `handleFieldRead` e/ou `handleFieldWrite` precisam ser redefinidos em subclasses da classe `MetaPersistent`.

Implementação pré-definida: A política de leitura e escrita implementada no arcabouço propaga toda operação de leitura ou escrita em atributo interceptada, para a base de dados (Figura 3.3). Se a operação é de leitura, então o atributo é obtido da base de dados e retornado como resultado da operação. Se a operação é de escrita, então o atributo é modificado na base de dados. As operações de leitura e escrita de atributos nas bases de dados são sempre feitas através dos métodos `readObject` e `writeObject` da classe `MetaPersistent`.

- ***Política de restauração das raízes de persistência da aplicação*** - Esta política determina como deve ser feita a restauração das raízes de persistência. Os dois extremos são:

restauração rasa e restauração em profundidade. Na restauração rasa, somente a casca do objeto raiz é restaurado, os atributos do objeto são restaurados sob demanda. Na restauração em profundidade, o objeto raiz e todos os objetos persistentes referenciados através dele são restaurados. O método `restore` da classe `MetaPersistent` implementa a política de restauração das raízes de persistência da aplicação. O arcabouço possui uma implementação pré-definida para a política de restauração das raízes de persistência da aplicação.

Implementação pré-definida: O arcabouço implementa uma política de restauração de raízes de persistência rasa (Figura 3.3).

3.4 Exemplos de Uso do Arcabouço

Nessa Seção apresentamos exemplos de como construir sistemas de persistência específicos a partir do arcabouço. Além de mostrar como o arcabouço pode ser adaptado, vamos mostrar como o uso de reflexão computacional, e em particular da arquitetura reflexiva de Guaraná, pode gerar soluções simples e elegantes para o problema de persistência de objetos.

As Seções 3.4.1, 3.4.2 e 3.4.3 descrevem uma seqüência de passos para incorporar persistência a uma aplicação de forma totalmente transparente e sem alterar uma linha de código da mesma. A Seção 3.4.1 descreve como adaptar o arcabouço para a construção de um sistema de persistência que atende aos requisitos de persistência de uma aplicação específica. O sistema de persistência é construído no meta-nível da aplicação, de forma ortogonal a mesma. A Seção 3.4.2 mostra como criar as raízes de persistência de uma aplicação no meta-nível da mesma, usando meta-classes. Meta-classe é o nome dado ao meta-objeto associado a uma classe. A Seção 3.4.3 mostra como configurar o meta-nível de uma aplicação com o código desenvolvido nas Seções 3.4.1 e 3.4.2, sem alterar o código dessa aplicação.

A Seção 3.4.4 mostra um exemplo de como adaptar a política de leitura e escrita do arcabouço para a construção de um sistema de persistência que utiliza uma cache de dados persistentes.

A Seção 3.4.5 discute a utilização do arcabouço na implementação de mecanismos para redirecionar eventos de entrada e saída em objetos e mostra como implementar um mecanismo de replicação de objetos a partir do arcabouço.

3.4.1 Persistência em Bases de Dados Heterogêneas

Nesta Seção mostramos como adaptar o arcabouço para construir um sistema de persistência para uma aplicação que manipula objetos persistentes distribuídos em bases de dados heterogêneas.

A aplicação usada como exemplo é uma aplicação bancária, escrita em Java, que manipula objetos das classes `Client` e `Account` (Classes 3.1). Os objetos da classe `Client` são

Classes 3.1 Client e Account

```
class Client{
    String name;
    int age;
    Account account; }

class Account{
    long number;
    int branch;}
```

objetos persistentes armazenados num banco de dados relacional denominado Record. Os objetos da classe `Account` referenciados pelo atributo `account` da classe `Client`, também são persistentes por transitividade. Esses objetos, por sua vez, são armazenados num banco de dados orientado a objetos denominado Accounts. Dessa forma, a aplicação manipula objetos armazenados em dois bancos de dados diferentes: Record e Accounts.

O primeiro passo para construir um sistema de persistência para a aplicação descrita acima é identificar as interfaces de base de dados necessárias.

O arcabouço implementa três interfaces de base de dados específicas, que podem ser usadas se satisfazem os requisitos do sistema de persistência que se quer construir. Essas interfaces são implementadas pelas classes `PSEInterface`, `SerializationInterface` e `JDBCInterface` (Figura 3.4).

A classe `PSEInterface` implementa uma interface para escrita e leitura de objetos em bancos de dados PSE. PSE (Persistent Store Engine) [28] é uma ferramenta 100% Java para gerência de bancos de dados orientados a objetos. Essa ferramenta possui uma interface de programação Java para o armazenamento e a recuperação de objetos persistentes.

A classe `SerializationInterface` implementa uma interface para escrita e leitura de objetos em arquivos. Essa interface foi implementada usando a interface de linearização de Java [44].

A classe `JDBCInterface` implementa uma interface para banco de dados relacional que utiliza JDBC para fazer acesso aos bancos de dados. JDBC (Java Database Connectivity) [16] é uma interface de programação Java que provê uma maneira uniforme de acesso a vários bancos de dados relacionais. Todos os bancos de dados que podem ser conectados via JDBC, também podem ser conectados através da classe `JDBCInterface`.

Neste exemplo vamos usar a classe `JDBCInterface` para interface com o banco de dados Record e a classe `PSEInterface` para interface com o banco de dados Accounts. O programador do sistema de persistência pode optar por construir suas próprias interfaces de base de dados, caso as implementações pré-definidas no arcabouço não satisfaçam os requisitos de acesso às bases de dados que ele pretende usar.

As classes `PSEInterface` e `JDBCInterface` usam objetos da classe `OAddress` para localizar os objetos persistentes na base de dados (Figura 3.4). O construtor dessa classe espera duas cadeias de caracteres como parâmetros: a primeira representa o nome do banco de dados e a segunda o nome da classe de interface de base de dados que deve ser usada no acesso ao banco de dados especificado no primeiro parâmetro. O método `getDatabaseInterface` retorna uma instância da classe de interface de base de dados especificada no segundo parâmetro do construtor.

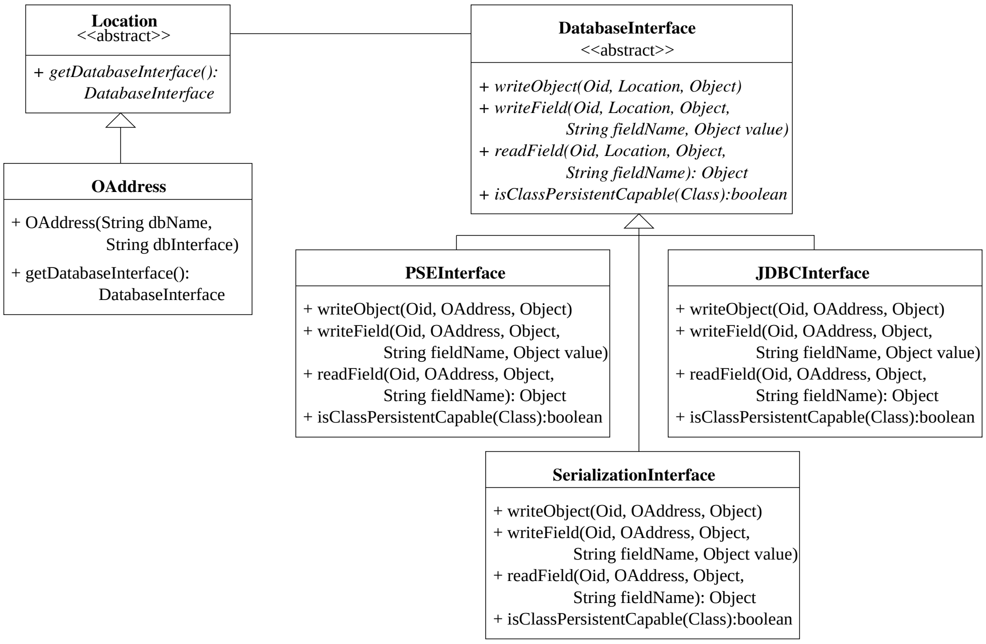

Figura 3.4: Interfaces de base de dados pré-definidas no arcabouço

Definidas as interfaces de base de dados e o objeto de localização, o próximo passo na construção do sistema de persistência é gerar objetos de localização para os objetos persistentes da aplicação bancária. Isto é feito na classe `TLocationFactory` (Classe 3.2). O método `create` cria um objeto de localização para o objeto passado por parâmetro: objetos persistentes da classe `Client` são alocados no banco de dados Record e objetos persistentes da classe `Account` são alocados no banco de dados Accounts.

Finalmente é preciso garantir que os objetos `Account`, referenciados pelos objetos `Client`, não sejam armazenados no banco de dados Record, juntamente com os outros atributos de `Client`. Os objetos da classe `Account` devem ser armazenados no banco de dados Accounts. Para garantir isso, o método `isClassPersistentCapable` da interface de base de dados `JDBCInterface` precisa ser construído corretamente. Esse método é sempre invocado antes de um atributo do objeto `Client` ser escrito na base de dados. Esse método determina se a persistência deve ser propagada para um objeto. Os atributos `name` e `age` da classe `Client`

Classe 3.2 TLocationFactory

```
// TLocationFactory é subclasse da classe LocationFactory.
class TLocationFactory extends LocationFactory{
    // Cria um objeto de localização para o objeto passado por parâmetro.
    public Location create(Object object){
        // Se for um objeto da classe Client, então aloca o objeto no banco de dados "Record"
        // usando a interface JDBCInterface.
        if( object.getClass().equals("Client") )
            return new OAddress("JDBCInterface", "Record");
        // Se for um objeto da classe Account, então aloca o objeto no banco de dados
        // "Accounts" usando a interface PSEInterface.
        else if( object.getClass().equals("Account") )
            return new OAddress("PSEInterface", "Accounts");
    }
}
```

podem ser escritos diretamente no banco de dados Record, mas o atributo `account` não. A persistência é então propagada para o objeto referenciado por esse atributo, esse objeto é associado a um meta-objeto de persistência e é corretamente alocado no banco de dados Accounts de acordo com a implementação de `TLocationFactory` (Classe 3.2).

Os demais pontos adaptáveis do arcabouço não precisam ser redefinidos neste exemplo, podemos usar suas implementações pré-definidas.

O sistema de persistência da aplicação bancária está pronto para ser usado. O último passo é criar as raízes de persistência da aplicação e assim estabelecer a ligação entre sistema de persistência no meta-nível e aplicação bancária no nível base. As raízes de persistência são criadas invocando o método `persistent` em um objeto da classe `MetaPersistent` (Figura 3.3), passando a política de alocação de objetos `TLocationFactory` por parâmetro para esse método. O método `persistent` deve ser invocado para todas as instâncias da classe `Client`. Os objetos da classe `Account` são automaticamente promovidos a objetos persistentes, por transitividade.

Apesar do sistema de persistência permanecer transparente no meta-nível, as raízes de persistência são criadas no código da aplicação de nível base. Na próxima Seção, mostramos como criar as raízes de persistência da aplicação bancária no meta-nível da mesma.

3.4.2 Meta-Classes

Em Java temos a representação das classes na forma de objetos em tempo de execução. Dessa forma, é possível criar meta-classes que tornam persistentes a classe referenciada por elas

Classe 3.3 PersistentClass

```
class PersistentClass extends MetaObject{
    // Este método é invocado sempre que uma mensagem chega a meta-classe.
    public void handle(Message msg, Object object){
        // Verifica se a mensagem é NewObject.
        if( msg.getClass().getClassName().equals("NewObject") ){
            // Obtém a referência para a nova instância criada.
            Object newObject = ((NewObject)message).getObject();
            // Obtém o nome da nova raiz de persistência.
            String rootName = getObjectName(newObject);
            // Cria uma nova raiz de persistência.
            MetaPersistent mp = new MetaPersistent();
            mp.persistent(rootName, newObject, new TLocationFactory());
        } //if
    } //handle
}
```

e/ou todas as suas instâncias. Esta Seção ilustra o caso em que a semântica de persistência é associada à classe através de meta-classes. Neste caso, a classe é utilizada como origem de persistência para todas as suas instâncias.

Em Guaraná, o comportamento reflexivo dos objetos pode ser determinado pela classe do objeto através de mensagens `NewObject`. Durante a criação de um novo objeto, uma mensagem `NewObject` é enviada para a meta-classe associada a classe do novo objeto instanciado, informando a criação de uma nova instância da classe. Neste momento, a meta-classe pode reconfigurar o meta-nível das novas instâncias, tornando-as persistentes por exemplo. O envio e recebimento de mensagem no meta-nível é mais uma funcionalidade de Guaraná. Qualquer instância de uma classe que implementa a interface `Message` pode ser enviada para o meta-nível de qualquer objeto.

A classe `PersistentClass` implementa a meta-classe (Classe 3.3). A associação de `PersistentClass` à classe `Client` fará com que toda instância dessa classe seja persistente. O método `handle(Message msg, Object object)` é invocado sempre que uma mensagem é recebida pela meta-classe. Se a mensagem recebida for `NewObject`, então uma nova instância da classe `Client` foi criada. Neste momento, é obtida uma referência para a nova instância e uma nova raiz de persistência da aplicação é criada invocando-se o método `persistent` e passando um objeto da classe `TLocationFactory` por parâmetro para esse método.

A associação entre a classe `Client` e a meta-classe `PersistentClass` é feita invocando-se o método `reconfigure` da interface de programação de Guaraná (Classe 3.4). Esse método associa o meta-objeto passado no terceiro parâmetro ao objeto passado no primeiro parâmetro.

Classe 3.4 Application

```
// Aplicação bancária.
class Application{
    // Inicia a aplicação bancária.
    public void static main(String[] arg){
        ...
        // Associa a meta-classe PersistentClass à classe Client.
        Guarana.reconfigure(Client.class, null, new PersistentClass());
        // Client.class é uma referência para a classe Client e null significa que
        // essa classe não está previamente associada a nenhuma meta-classe.
        ...
    }
}
```

Dessa forma, criamos as raízes de persistência da aplicação bancária no meta-nível, de forma transparente para a mesma. Entretanto, o método `reconfigure` da interface de programação de Guaraná ainda é invocado no código da aplicação bancária, fazendo com que a transparência ainda não seja total.

3.4.3 Persistência Transparente

Esta Seção descreve a terceira e última etapa para implementar a persistência dos objetos da aplicação bancária sem modificar o código da mesma: primeiro construímos o sistema de persistência (Seção 3.4.1), em seguida usamos meta-classes para criar as raízes de persistência no meta-nível (Seção 3.4.2) e agora falta retirar do código da aplicação a invocação do método `reconfigure` da interface de programação de Guaraná.

Para isso criamos um meta-programa que inicia o meta-nível da aplicação e em seguida inicia a aplicação. A iniciação do meta-nível cria as configurações necessárias para que o comportamento reflexivo seja associado aos objetos da aplicação de forma transparente para a mesma. No nosso exemplo, a iniciação do meta-nível deve associar a meta-classe `PersistentClass` à classe `Client`.

A classe `MetaMain` (Classe 3.5) implementa o meta-programa. O método `main` é invocado quando o meta-programa é executado. Esse método associa a meta-classe `PersistentClass` à classe `Client`, e só depois inicia a aplicação. Dessa forma, atingimos o nosso objetivo: implementar a persistência dos objetos de uma aplicação sem modificar o código da mesma.

Classes 3.5 MetaMain e Application

```
// Meta-Programa
class MetaMain{
    public void static main(String[] arg){
        // Associa um meta-objeto da classe PersistentClass à classe Client.
        Guarana.reconfigure( Client.class, null, new PersistentClass() );
        // Inicia a aplicação.
        Application.main(null); }
}

// Aplicação bancária.
class Application{
    public void static main(String[] arg){ ... }
}
```

3.4.4 Cache de Dados

Neste exemplo vamos mostrar como implementar um sistema de persistência que utiliza uma cache dos objetos persistentes. O objetivo deste exemplo é mostrar como a política de leitura e escrita de objetos persistentes pode ser adaptada.

Todas as operações de leitura e escrita em atributos do objeto persistente interceptadas por `MetaPersistent` devem ser enviadas à cache de objetos que determina, segundo sua política interna, o momento em que essas operações devem efetivamente ocorrer na base de dados.

A Figura 3.5 mostra as modificações realizadas na política de leitura e escrita para incorporar a cache de objetos. Os métodos `handleFieldRead` e `handleFieldWrite` foram sobrecarregados na classe `MetaCaching`. Esses métodos delegam as operações de leitura e escrita nos atributos dos objetos persistentes para o componente que implementa a cache de objetos. Esse componente determina quando os métodos `writeObject` e `readObject` da classe `MetaPersistent` devem ser invocados.

Para que a nova política de leitura e escrita entre em vigor, as raízes de persistência devem ser criadas invocando-se o método `persistent` da classe `MetaCaching`. Assim, os objetos persistentes são corretamente associados a meta-objetos da classe `MetaCaching`. O mesmo é válido para o método `restore` que deve ser invocado em uma instância de `MetaCaching`.

3.4.5 Replicação de Objetos

Analisando o arcabouço sob uma perspectiva mais genérica, concluímos que sua arquitetura de software pode ser adaptada para implementar outros mecanismos além de persistência.

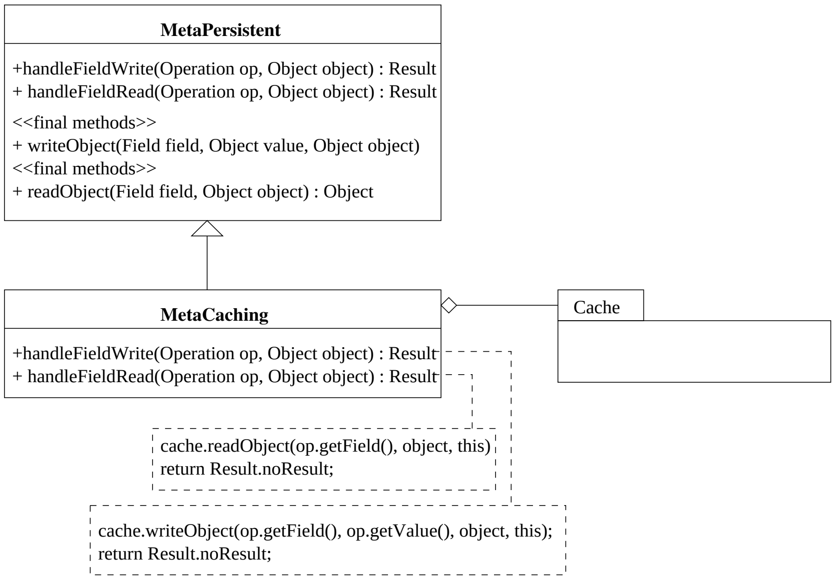

Figura 3.5: Sistema de persistência com cache de dados

A arquitetura definida pelo arcabouço, sua divisão em classes e a colaboração entre essas classes, implementa um mecanismo para redirecionar eventos de entrada e saída (E/S) em objetos. Sob esse ponto de vista, persistência pode ser descrito como um mecanismo para redirecionar dados para uma memória não-volátil.

São vários os mecanismos cujo princípio básico de funcionamento é o redirecionamento de eventos de E/S: distribuição, replicação de objetos, *caching*, migração de dados. Nesta Seção mostramos como o mecanismo de replicação de objetos pode ser implementado a partir do arcabouço.

Replicação [43] é um mecanismo usado para implementar tolerância a falhas que consiste em manter várias réplicas de um mesmo objeto no sistema. Existem duas alternativas para a implementação de replicação: na primeira, somente as operações de escrita em atributos do objeto são redirecionadas para as réplicas; na segunda, as invocações de métodos e acesso a atributos do objeto também são redirecionadas para as réplicas e um algoritmo de eleição seleciona o melhor resultado para a operação.

A Figura 3.6(a) mostra como a primeira alternativa pode ser implementada usando o arcabouço. Todo objeto replicado (`Robject`) é associado a uma instância da classe `MetaReplication`, subclasse de `MetaPersistent`. A política de leitura e escrita implementada por `MetaReplication` propaga toda operação de escrita no objeto para a base de dados e nunca propaga as operações de leitura. Logo, o método `handleFieldRead` precisa ser redefinido em `MetaReplication` para não propagar as operações de leitura e o método `handleFieldWrite` não precisa ser modificado. Foi construída uma nova interface de base de dados denominada `ReplicationInterface`. Essa interface conhece as réplicas do objeto e propaga as operações de escrita para as mesmas.

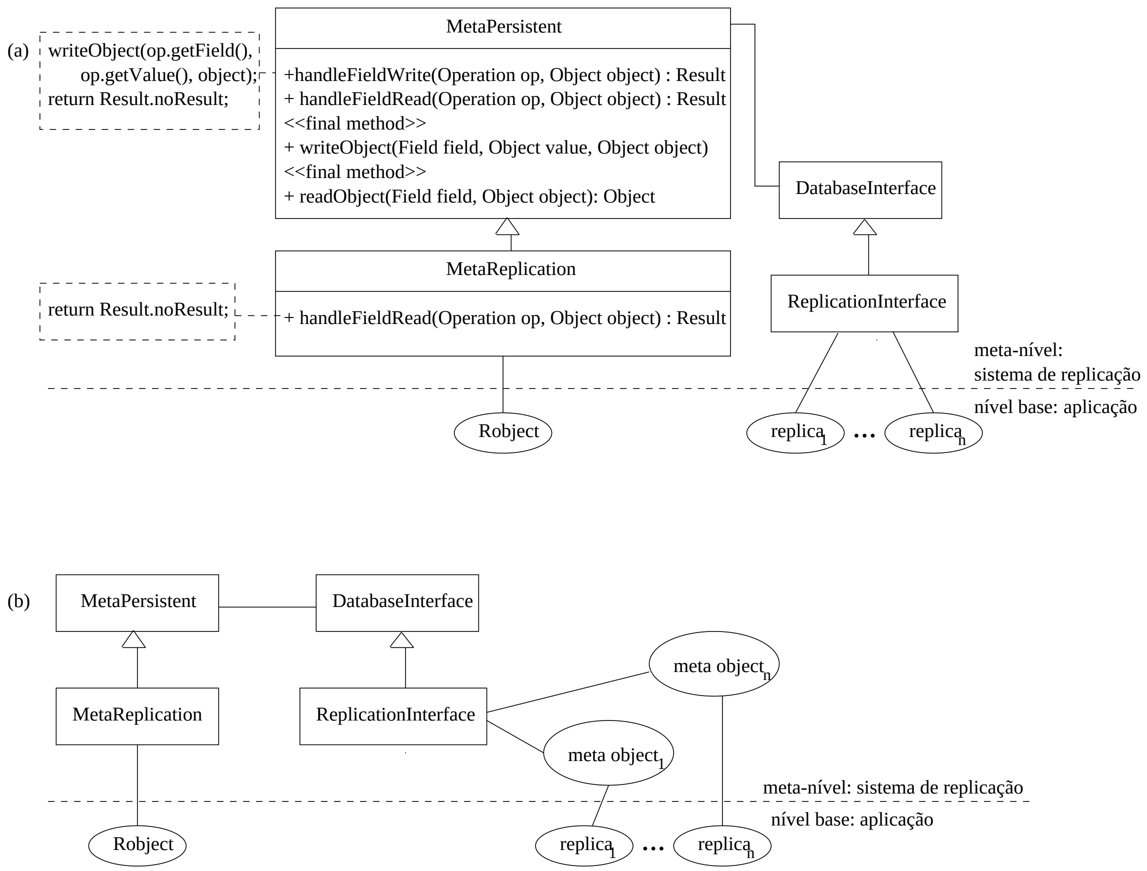

Figura 3.6: Implementando replicação a partir do arcabouço

As réplicas de `Robject` são criadas e iniciadas quando o objeto de localização de `Robject` é gerado. Esse objeto de localização guarda as referências para as réplicas criadas, para que `ReplicationInterface` possa encontrá-las mais tarde. O algoritmo que decide que objetos devem ser replicados é implementado no método `isClassPersistentCapable` da classe `ReplicationInterface`. Quando o objeto replicado `Robject` alcança um objeto `O` que não está replicado, o método `isClassPersistentCapable` é invocado. Se esse método retorna verdadeiro, o objeto `O` será o mais novo objeto replicado da aplicação.

Para implementar a segunda alternativa de replicação, o arcabouço precisaria ser adaptado para redirecionar invocações de métodos para as réplicas. O algoritmo de eleição poderia ser implementado em `ReplicationInterface` que, neste caso, delegaria as operações para um conjunto de meta-objetos, cada um associado a uma réplica (Figura 3.6(b)). Esses meta-objetos executam as operações nas réplicas e retornam o resultado para `ReplicationInterface`, que seleciona o resultado para a operação.

3.5 Conclusão

Esse artigo descreveu uma abordagem alternativa para o problema de persistência de objetos. Essa abordagem combina as técnicas de reflexão computacional e arcabouços de software para a construção de um arcabouço reflexivo para persistência de objetos, que atende a três requisitos básicos: (i) transparência total das operações persistentes, (ii) acesso a bases de dados heterogêneas e (ii) ortogonalidade entre tipo e persistência.

O primeiro requisito foi alcançado utilizando a técnica de reflexão computacional. Usando essa técnica criamos um sistema de persistência no meta-nível da aplicação e iniciamos essa aplicação a partir de um meta-programa. O meta-programa primeiro inicia o meta-nível da aplicação, criando as configurações necessárias para implementar a persistência e só depois inicia a aplicação. Dessa forma, mostramos que é possível implementar persistência sem modificar o código das aplicações.

O acesso a bases de dados heterogêneas foi alcançado construindo o sistema de persistência na forma de um arcabouço de software, com uma interface de base de dados genérica e flexível que pode ser adaptada para prover acesso a quaisquer tipos de base de dados. Dessa forma, objetos persistentes de uma mesma aplicação podem ser armazenados e recuperados de bases de dados diferentes, através de uma interface de base de dados uniforme. Além disso, se uma aplicação precisa buscar/armazenar dados de um banco de dados diferente, não é preciso adaptar todo o sistema de persistência da aplicação, basta construir uma interface de base de dados específica para o novo banco de dados e acoplá-la ao sistema de persistência.

O requisito de ortogonalidade entre tipo e persistência também foi contemplado de maneira satisfatória em nossa solução. Qualquer objeto pode ser persistente desde que esteja associado a um meta-objeto de persistência. Esse meta-objeto pode ser associado a qualquer objeto, inclusive as classes, que também são objetos em Java.

Capítulo 4

Uso de um Arcabouço Reflexivo para Persistência de Objetos como Ferramenta de Apoio à Migração de Sistemas Legados

Este artigo foi submetido ao SBBD'99—XIV Simpósio Brasileiro de Banco de Dados—, evento a ser realizado de 11-13 de outubro de 1999, em Florianópolis, SC.

Uso de um Arcabouço Reflexivo para Persistência de Objetos como Ferramenta de Apoio à Migração de Sistemas Legados

Nádia Leão Pereira Quadros
nadia.pereira@dcc.unicamp.br

Luiz Eduardo Buzato
buzato@dcc.unicamp.br

Laboratório de Sistemas Distribuídos
Instituto de Computação - UNICAMP

Resumo

As razões para a construção de um arcabouço para persistência são: (i) aumento de produtividade e (ii) redução de custos de manutenção. A independência entre os objetos da aplicação e os repositórios de dados que armazenam seu estado permite o desenvolvimento e a manutenção de aplicações mais complexas enquanto alavanca a migração das estruturas de gerência de dados já existentes. Este artigo descreve um arcabouço reflexivo para persistência de objetos e discute o seu uso como uma ferramenta de apoio ao processo de migração incremental de sistemas legados. A partir do arcabouço podem ser construídas incrementalmente várias *pontes* de software que implementam a mediação entre componentes do sistema legado e componentes do novo sistema. O arcabouço foi construído utilizando a técnica de reflexão computacional.

Abstract

The motivations for building an object persistence framework are: (i) increased productivity and (ii) reduced maintenance costs. Independence between object-oriented applications and databases allows enterprises do develop and maintain more complex applications and still leverage legacy database applications. This article describes a reflexive object persistence framework and discusses its use as an aid in the incremental migration of legacy data management infrastructures. The framework can be used to build gateways that act as proxies and govern the dialog between legacy and new (target) software components. The framework was built using computation reflection.

4.1 Introdução

Sistemas legados são reconhecidos como um dos maiores empecilhos para a adaptação de empresas a sistemas computacionais com melhor relação custo/benefício. Um sistema legado [7] é um sistema que resiste a modificações, isto é, não pode ser facilmente modificado para adaptar-se à evolução de funções que a empresa requer.

A migração de sistemas é um processo que é iniciado com um sistema legado e que termina com um sistema destino que é significativamente diferente do original, mas contém uma quantidade substancial dos dados e da funcionalidade do sistema legado. A migração precisa ser bem planejada, para que não seja tão ou mais complexa e custosa que a operação do sistema legado.

No processo de migração incremental de software, pontes[1] desempenham um papel fundamental [7]. Pontes são introduzidas entre componentes legados e componentes destino para mediar o diálogo entre o sistema legado e o sistema destino até que o componente legado seja substituído. Pontes devem ser componentes transparentes, pouco intrusivos, caso contrário podem dificultar o processo de migração incremental.

Este artigo descreve o projeto e implementação de um *arcabouço reflexivo para persistência de objetos* e discute o seu uso como ferramenta de apoio ao processo de migração incremental de sistemas legados. O arcabouço é uma infra-estrutura genérica e flexível para a persistência de objetos que podem residir em bases de dados heterogêneas. Reflexão computacional foi adotada na sua construção porque permite que a persistência seja implementada de forma totalmente transparente para a aplicação. O arcabouço apoia a migração incremental de sistemas legados porque permite a implementação transparente de pontes entre componentes destino e componentes legados, inclusive com a infra-estrutura de bancos de dados legada.

O artigo está organizado em 4 Seções principais. A Seção 4.2 diz sucintamente o que são arcabouços de software, reflexão computacional e a estratégia para migração incremental de sistemas legados. A Seção 4.3 discute persistência de objetos e descreve a arquitetura de software do arcabouço reflexivo para persistência de objetos. A Seção 4.4 discute o uso do arcabouço como ferramenta de apoio à migração de sistemas legados. Finalmente, a Seção 4.5 discute os resultados deste trabalho.

4.2 Arcabouços, Reflexão Computacional e Migração

Esta Seção descreve brevemente arcabouços de software, a técnica de reflexão computacional e a estratégia para migração incremental de sistemas legados proposta por Brodie e Stonebraker.

[1] Em Inglês: *gateways*.

4.2.1 Arcabouços de Software

Um arcabouço de software[2] [19, 45] é um conjunto de classes abstratas e concretas que provê uma infra-estrutura genérica de soluções para um conjunto de problemas. Em outras palavras, um arcabouço padroniza uma arquitetura de software que auxilia a solução de problemas em um domínio específico.

Durante a construção de um arcabouço, vários aspectos de uma aplicação não podem ser antecipados. Esses aspectos precisam ser genéricos para que possam ser facilmente adaptados para situações específicas. Reutilizar um arcabouço significa herdar uma estrutura de controle global e adaptar esses pontos genéricos segundo as necessidades específicas de cada aplicação. Essa reutilização de projeto em larga escala é a principal característica que diferencia os arcabouços das bibliotecas de classes orientadas a objetos.

Dentre as principais vantagens da utilização de arcabouços no desenvolvimento de aplicações podemos citar [34]: (i) infra-estrutura pré-fabricada que reduz tempo e recursos gastos com codificação, rastreamentos e testes, (ii) abstração de serviços: o arcabouço implementa a complexidade da aplicação e (iii) redução de custos em manutenção.

4.2.2 Reflexão Computacional

Reflexão computacional é uma técnica de estruturação de sistemas que permite a um sistema manter informações a seu respeito e usar essas informações para modificar o seu comportamento. Um sistema computacional que incorpora reflexão possui pelo menos dois níveis de abstração: o nível base e o meta-nível [26].

A tarefa do nível base é a de implementar aspectos funcionais da aplicação, enquanto o meta-nível cuida da implementação dos requisitos gerências tais como persistência [24, 35], tolerância a falhas [12, 25, 47], replicação [22] e distribuição [42]. Além disso, a implementação do meta-nível é feita de forma não intrusiva—a aplicação, no nível base, não tem que mudar sua aparência (código) ou comportamento (execução) se for incorporado ao sistema um novo mecanismo, por exemplo, o de persistência.

Os elementos do meta-nível são chamados meta-objetos. Os meta-objetos são os responsáveis pelo comportamento reflexivo dos objetos do nível base. Os meta-objetos guardam informações sobre a estrutura interna desses objetos tais como que métodos são implementados pelos objetos e quais são seus atributos. Os meta-objetos estendem o comportamento de um objeto do nível base interceptando operações invocadas nesse objeto. Dessa forma os meta-objetos podem modificar o efeito dessas operações, implementando de forma transparente as atividades administrativas do sistema.

[2]Em Inglês: *framework*.

Segundo Kleinöder [21], um ambiente de programação que incorpora uma arquitetura reflexiva tem as seguintes vantagens:

- ***Aumento na produtividade*** - A programação em meta-nível leva à construção de programas mais estruturados e de fácil manutenção.
- ***Separação de conceitos*** - O uso de um meta espaço separado permite ao programador da aplicação focalizar no domínio desta. A funcionalidade adicional é fornecida por uma biblioteca de meta componentes ou é desenvolvida juntamente com o programa da aplicação, mas de forma ortogonal ao mesmo.
- ***Configurabilidade*** - O meta-nível estabelece uma arquitetura aberta. Novas políticas podem ser implementadas, como por exemplo persistência ou distribuição, sem mudar o código da aplicação.
- ***Transparência*** - A computação do meta-nível é transparente à aplicação, que reside no nível base.

Um dos aspectos mais importantes de uma arquitetura reflexiva orientada a objetos é seu protocolo de meta-objetos. O protocolo de meta-objetos[3] (MOP) define a interface entre objetos e meta-objetos.

4.2.3 Migração Incremental de Sistema Legados

Brodie e Stonebraker [7] descrevem uma estratégia para migração incremental de sistemas legados, denominada Chicken Little. Essa estratégia consiste em migrar partes pequenas e específicas do sistema legado, denominadas incrementos, até que a migração desejada seja alcançada.

A arquitetura de um sistema legado possui três componentes, não necessariamente identificáveis: (i) interface com o usuário e com outros sistemas, (ii) aplicações e (iii) sistema de banco de dados [7]. A estratégia de migração incremental Chicken Little procura decompor o sistema legado nessas três camadas antes de iniciar a migração. A complexidade do processo de migração e a probabilidade de seu sucesso ou falha, será afetado pelo grau de organização do sistema legado, ou seja, pela capacidade de decomposição de sua arquitetura nessas três camadas. Se a decomposição for possível, então será possível a inserção de pontes que permitirão a migração incremental dos dados legados e da aplicação legada como um todo.

[3]Em Inglês: *MetaObject Protocol*

4.3 Um Arcabouço Reflexivo para Persistência de Objetos

A idéia de construir um arcabouço reflexivo para persistência de objetos surgiu da percepção de que as diferentes soluções para persistência não contemplavam duas características consideradas importantes para um sistema de persistência: (i) transparência dos mecanismos de persistência e (ii) apoio a infra-estrutura heterogênea de gerência de bancos de dados. Julgamos que essas características são fundamentais para a implementação de infra-estruturas para persistência pouco intrusivas e facilmente adaptáveis a sistemas já existentes.

Existem várias *especificações* de sistemas para objetos persistentes, incluindo o proposto pelo Object Management Group (OMG), pelo Object Database Management Group (ODMG) e Enterprise JavaBeans (EJB). Entretanto, nenhuma dessas especificações aborda a *implementação* real de um sistema de persistência. Neste artigo, descreveremos um arcabouço reflexivo para persistência de objetos em bases de dados heterogêneos e uma estratégia para a sua utilização como ferramenta para a migração incremental de sistemas legados.

Este artigo não aborda outras questões importantes relacionadas à persistência de objetos porque elas já foram endereçadas satisfatoriamente, na nossa opinião, por outros sistemas de persistência. Entre as questões não abordadas encontram-se as relacionadas ao problema do mapeamento bidirecional (semi-)automático entre modelos de objetos e equivalentes relacionais (entidade-relacionamento). Esse é um problema difícil, mas muito pesquisado. Outro problema não abordado neste artigo é o do acoplamento entre o sistema de persistência e um monitor de transações atômicas[4].

Este artigo concentra-se na descrição de uma solução para o problema da transparência e da heterogeneidade de bases de dados. O problema da transparência não foi resolvido satisfatoriamente mesmo por sistemas profissionais de persistência de objetos. Os sistemas existentes recorrem geralmente à introdução de palavras reservadas ou de construções especiais para indicar a persistência de objetos. O arcabouço resolve os problemas relacionados à transparência com o emprego de reflexão computacional.

A heterogeneidade é contemplada pela inclusão no arcabouço de uma interface de base de dados genérica e adaptável. Normalmente, sistemas de persistência não contemplam heterogeneidade, restringindo o mapeamento de objetos persistentes a somente um modelo de dados.

4.3.1 Visão Geral da Arquitetura do Arcabouço

A Figura 4.1 traz uma representação dos principais componentes do arcabouço, permitindo a visualização em alto nível da sua arquitetura de software.

[4]Em Inglês: *transactional service*.

O Gerente de Persistência (GP) (Figura 4.1) coordena os mecanismos de persistência. As funções do GP incluem: (i) determinação de quando e de que operação persistente deve ser executada e (ii) determinação de onde, no espaço de endereçamento persistente, deve ser armazenado ou recuperado um objeto da aplicação. Cada instância do GP é responsável pela persistência de um objeto da aplicação.

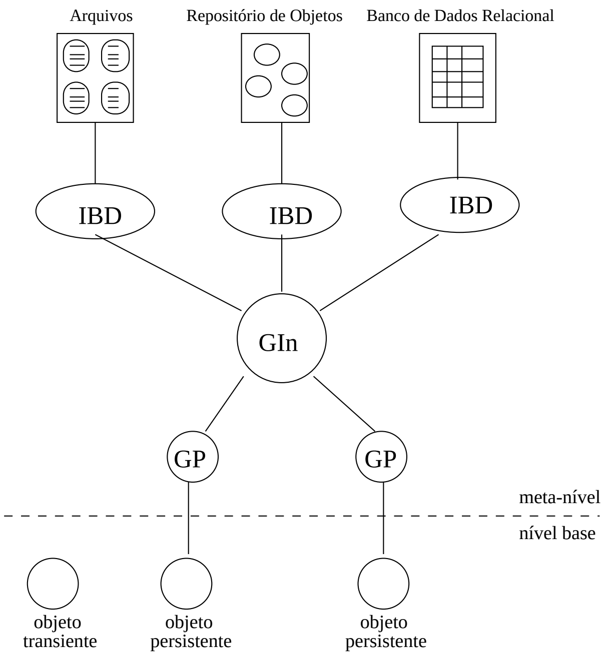

Figura 4.1: Visão geral do arcabouço

O Gerente de Interface (GIn) é uma interface de operações persistentes independente de implementação, ou seja, a interface para operações persistentes é a mesma independente do tipo de base de dados onde os objetos estejam armazenados (Figura 4.1). O GP delega todas as operações de escrita e de leitura realizadas nas bases de dados para o GIn, que se encarrega de endereçá-las para a interface de base de dados específica, como por exemplo uma interface para banco de dados relacional.

As Interfaces de Base de Dados (IBDs) implementam as operações de leitura e escrita de dados mapeando essas operações para o modelo de dados do sistema de gerência de banco de dados ao qual está acoplada. Assim, se uma operação de leitura é requisitada a uma IBD para banco de dados relacional, uma sentença SQL do tipo `Select <campos> from <tabela>` será gerada e enviada ao banco de dados. A resposta a esse comando é processada pelo IBD que se encarrega de mapear os resultados para objetos, de acordo com o modelo objeto-para-dado que ele implementa, e em seguida retorna o resultado para GIn.

A arquitetura contempla componentes para encapsulamento dos seguintes modelos:

- modelo de objetos (MO): utilizado na aplicação orientada a objetos. O modelo de objetos é implementado pela aplicação.
- modelo de dados (MD): responsável pela descrição do esquema de dados adotado para a persistência. Esse modelo é função do sistema de banco de dados adotado para a implementação da persistência. Esse modelo é implementado e descrito (esquema) pelo banco de dados utilizado para a persistência
- modelo de mapeamento entre o MO e o MD: esse modelo determina a correspondência entre o MO e o MD. A implementação desse mapeamento fica a cargo do IBD.

A reflexão computacional permite a inserção transparente dos objetos do arcabouço em aplicações orientadas a objetos, isto é, o mapeamento entre o modelo de objetos e o modelo de dados ocorre sem a necessidade de modificação da aplicação. Os componentes do arcabouço—GP, GIn e IBD—residem no meta-nível e implementam a persistência de forma ortogonal a aplicação, que reside no nível base. Esta é uma das principais contribuições deste trabalho. Demonstra-se que é possível realizar a persistência sem a necessidade de modificação do código da aplicação ou do uso de um mecanismo explícito na linguagem, por exemplo herança, como ocorre em todos os outros sistemas para persistência.

4.3.2 Visão Detalhada da Arquitetura do Arcabouço

Passamos agora a descrever a arquitetura de software do arcabouço de forma mais detalhada, em função de seu modelo de objetos (Figura 4.2), representado em UML (Unified Modeling Language) [6].

Interface para operações persistentes heterogêneas

A classe abstrata `DatabaseInterface` (Figura 4.2) implementa as funções do Gerente de Interface (GIn) da Figura 4.1. Essa classe define uma interface para leitura e escrita de objetos e atributos de objetos nas bases de dados. Cada subclasse de `DatabaseInterface` deve implementar os métodos abstratos definidos na interface dessa classe usando a sintaxe de um sistema de banco de dados particular. Dessa forma, a classe `DatabaseInterface` provê uma interface para escrita e leitura de dados independente do tipo da base de dados onde o objeto está armazenado. As classes `PSEInterface` , `SerializationInterface` e `JDBCInterface` (Figura 4.2) são exemplos de classes concretas que implementam interfaces com um sistema de banco de dados específico. Essas classes implementam os componentes de interfaces de base de dados (IBDs) da Figura 4.1.

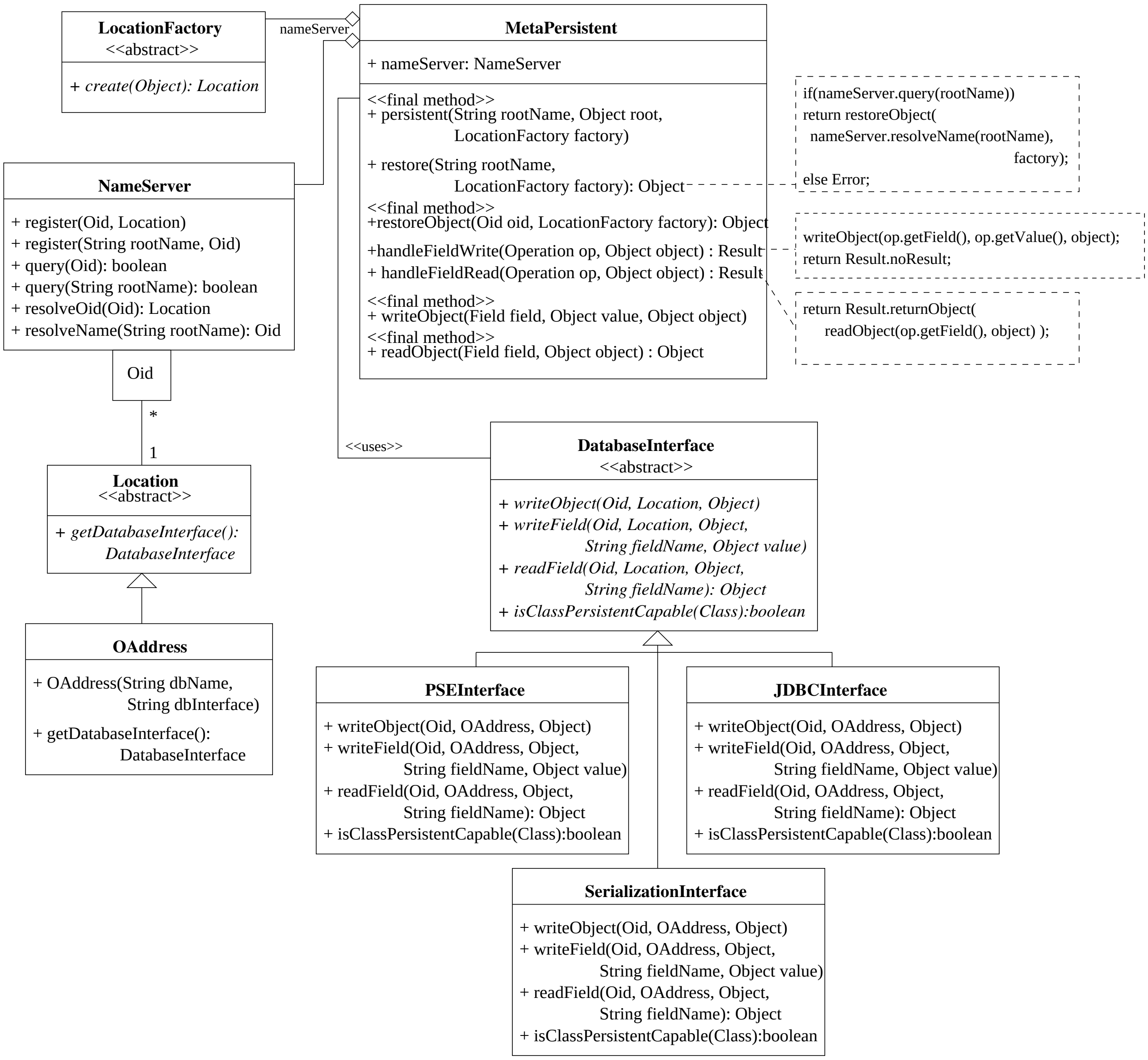

Figura 4.2: Diagrama de classes do arcabouço

Implementação da persistência via protocolo de meta-objetos

A classe `MetaPersistent` (Figura 4.2) implementa a funcionalidade do Gerente de Persistência (GP) da Figura 4.1. Essa classe coordena a colaboração entre as classes do arcabouço para a implementação da persistência.

Cada objeto persistente da aplicação é associado a uma instância de `MetaPersistent` que cuidará de salvar e recuperar o estado desse objeto da base de dados. Dessa forma, a semântica de persistência é atribuída a objetos e não a classes de objetos, permitindo que uma classe tenha instâncias persistentes e instâncias voláteis.

As instâncias de `MetaPersistent` implementam a persistência interceptando as operações de escrita e leitura nos atributos do objeto associado a elas. Quando a operação interceptada é de escrita de atributo, o método `handleFieldWrite` é invocado para propagar a escrita à base de dados. Quando a operação interceptada é de leitura de atributo, o método `handleFieldRead` é invocado para obter o dado da base de dados e retorná-lo como resultado da operação de leitura.

`MetaPersistent` consulta a base de dados através da interface de escrita e leitura de objetos definida pela classe `DatabaseInterface`. A abstração criada por `DatabaseInterface` permite que as consultas às bases de dados realizadas por `MetaPersistent` sejam independentes do tipo de base de dados onde o objeto está armazenado.

Como as operações persistentes são disparadas pela interceptação de leituras e escritas no estado de um objeto persistente, a persistência dos objetos é transparente para a aplicação. Deste modo, não há invocação de operações persistentes em meio ao código que implementa a funcionalidade da aplicação.

A forma como é feita a associação entre objetos e meta-objetos depende do MOP usado. Usamos na implementação deste arcabouço o MOP de Guaraná [33]. Esse MOP associa dinamicamente objetos à meta-objetos, o que permite adiar a determinação do estado de um objeto (persistente ou volátil) para o tempo de execução.

Registro de Objetos Persistentes

Para que `MetaPersistent` armazene e, mais tarde, possa recuperar os objetos persistentes é necessário identificá-los unicamente e localizá-los nas bases de dados. As classes `Oid` e `Location` (Figura 4.2) servem para este propósito, mantendo o registro de sua localização.

A classe `Oid` representa o identificador único[5] (OID) de um objeto persistente da aplicação. O sistema de persistência cria um OID para cada objeto persistente. O OID de um objeto também guarda a informação do nome da classe desse objeto. Essa informação é necessária para reincarnar os objetos persistentes.

[5] Em Inglês: *unique object identifier.*

A classe abstrata `Location` representa a localização de um objeto nas bases de dados. Especializações dessa classe devem guardar as informações necessárias para a localização da base de dados onde o objeto está armazenado. A informação sobre o tipo da base de dados também é necessária para a seleção da interface que deve ser usada para ler e escrever o objeto na base de dados, uma vez que, bases de dados de tipos diferentes possuem uma sintaxe e semântica de consulta dos dados diferentes. O método `getDatabaseInterface` (que deve ser redefinido em subclasses da classe `Location`) deve retornar a interface que deve ser usada para consulta a base de dados onde o objeto persistente está armazenado.

O OID e a localização dos objetos são armazenados num servidor de nomes implementado pela classe `NameServer` (Figura 4.2). Todo objeto persistente é registrado no servidor de nomes passando seu OID e sua localização (método `register(Oid, Location)`). Consultas são submetidas ao servidor de nomes para determinar se um objeto é persistente (método `query(Oid)`) e para determinar a localização de um objeto a partir de seu OID (método `resolveOid(Oid)`).

Este subsistema de identificação e localização de objetos persistentes, composto das classes `NameServer`, `Oid` e `Location`, pode ser especializado pelo programador para a construção de um serviço mais sofisticado, desde que preserve a semântica e a interface de programação das classes do arcabouço.

Criação e restauração de objetos persistentes

No arcabouço, um objeto pode se tornar persistente direta ou indiretamente. A persistência direta é conseguida através da associação do atributo de persistência a um objeto. Uma raiz de persistência é criada no meta-nível da aplicação e associada transparentemente a um objeto da aplicação. A realização da associação empresta o atributo de persistência ao objeto. A persistência indireta ocorre se o objeto persistente se associar a outros objetos da aplicação. Nesse momento, o objeto associado é promovido a objeto persistente, se já não o for.

Raízes de persistência são criadas através do método `persistent(String name, Object object, LocationFactory factory)` da classe `MetaPersistent`. Esse método associa o nome `name` ao objeto `object`, registra esse objeto como uma nova raiz de persistência no servidor de nomes (método `register(String name, Oid oid)`) e torna-o um objeto persistente da aplicação, associando a ele uma instância da classe `MetaPersistent`. O comportamento persistente da raiz será propagado a todo objeto que faz parte do estado dessa raiz. Esses novos objetos persistentes, por sua vez, propagam o comportamento persistente aos objetos que fazem parte do seu estado, e assim por diante. Dessa forma, temos persistência indireta ou por transitividade.

Quando um objeto torna-se persistente é preciso determinar a base de dados onde esse objeto será armazenado. Esta função é atribuída a classe abstrata `LocationFactory` (Figura 4.2). Essa classe implementa a política de alocação de objetos persistentes nas bases de dados. O método abstrato `create(Object)` deve ser redefinido em subclasses e gerar objetos de localização (instâncias de subclasses da classe `Location`) para os novos objetos persistentes.

Cada raiz de persistência é associada, no momento de sua criação, a uma política de alocação de objetos persistentes implementada no objeto `factory`. Esse objeto gera um objeto de localização para o objeto raiz e é propagado para os objetos persistentes criados a partir dessa raiz. Para cada um dos novos objetos persistentes, `factory` também gera um objeto de localização. No entanto, os objetos persistentes criados a partir de uma mesma raiz de persistência não precisam ser alocados na mesma base de dados. De maneira geral, objetos distintos, da mesma classe ou não, podem ser armazenados em bases de dados diferentes. Isto deve estar determinado em `factory` e requer ainda a implementação de mapeadores entre o modelo de objetos e os diversos modelos de dados usados pelas bases de dados.

O que diferencia os objetos raiz de persistência dos demais objetos persistentes é apenas o fato de uma raiz de persistência possuir um nome, e portanto, poder ser localizada e restaurada a partir desse nome. Para restaurar o estado de uma raiz de persistência usa-se o método `restore(String name, LocationFactory factory)` da classe `MetaPersistent`. A restauração dos demais objetos persistentes é feita internamente pelo sistema de persistência.

Na restauração também é associado um objeto `factory` à raiz de persistência. Esse objeto não influirá na localização dos objetos que compõe o estado do objeto raiz, uma vez que a localização desses objetos já foi estabelecida. O objeto `factory` será usado para gerar objetos de localização para novos objetos persistentes criados a partir da raiz de persistência.

Persistência por transitividade: casos excepcionais

A Seção anterior descreveu um mecanismo simples para a propagação de persistência, mas há exceções que justificam o refinamento desse mecanismo.

Considere as classes Java, `Client` e `Account` (Classes 4.1). Suponha que um objeto persistente *OP* da classe `Client` precisa ser escrito em um arquivo. Em Java, tipos primitivos, Strings e instâncias de classes que implementam a interface `Serializable` [44] podem ser escritos diretamente em arquivos. Logo *OP* poderia ser escrito em arquivo, uma vez que seus atributos são tipos primitivos (`age`), Strings (`name`) ou instâncias de classes que implementam a interface `Serializable` (`account`). Por outro lado, se *OP* precisa ser escrito em um banco de dados relacional, o objeto referenciado pelo atributo `account` não poderia ser escrito diretamente numa tabela relacional, sem um mapeamento de dados definido previamente. Neste caso, a persistência seria propagada para o objeto `account` para que seu estado possa ser escrito na base de dados separadamente.

Classes 4.1 Client e Account

```
class Client implements Serializable{
    String name;
    int age;
    Account account; }

class Account implements Serializable{
    long number;
    int branch;}
```

Logo, para cada tipo de base de dados existe um conjunto de classes cujos objetos podem ser escritos/lidos diretamente na base de dados. Se a classe de um objeto não pertence a esse conjunto, então esse objeto não pode ser escrito diretamente na base de dados. Neste caso, o objeto torna-se persistente e o OID desse objeto é armazenado no lugar da referência para o objeto. Em seguida, o estado desse objeto é escrito na base de dados. Quando um objeto precisar ser restaurado, os objetos persistentes que fazem parte de seu estado serão encontrados a partir de seus OIDs. O método `query(Oid)` da classe `NameServer` retorna a localização de um objeto a partir de seu OID. O problema de referência circular é tratado com uma consulta ao servidor de nome antes de tornar o objeto persistente.

O método `isClassPersistentCapable(Class class)` (Figura 4.2) deve ser implementado para cada interface de base de dados específica, definindo o conjunto de classes cujos objetos podem ser escritos na base de dados a partir dessa interface.

A Figura 4.3 apresenta o algoritmo para propagação da persistência, seguindo a sintaxe da linguagem Java [2]. Cada atributo do objeto persistente (`object`) é inspecionado para verificar se o mesmo pode ser escrito diretamente na base de dados. Para inspecionar o estado do objeto persistente é necessário meta-informação estrutural (Seção 4.2.2) sobre o objeto. A meta-informação é obtida através da interface de programação de Java Core Reflection [40].

De acordo com a abordagem proposta para propagação da persistência, o estado de um objeto pode ser distribuído em bases de dados heterogêneas. Para isso, basta definir uma política de alocação de objetos que garanta essa distribuição. Suponha uma política de alocação de objetos persistentes que aloca objetos da classe `Client` em uma base de dados relacional e objetos da classe `Account` em uma base de dados orientada a objetos. Uma vez que objetos da classe `Account` não podem ser armazenados diretamente numa base de dados relacional (isto precisa ser garantido no método `isClassPersistentCapable` da interface que escreve objetos na base de dados relacional), o objeto `account` torna-se persistente e é alocado numa base de dados orientada a objetos.

```
/**
 *  objectClass: classe do objeto persistente
 *  object     : objeto persistente
 *  dbInterface: interface de base de dados usada para escrever e ler o objeto (object)
 *               da base de dados
 */
void inspectObjectState(Class objectClass, Object object, DatabaseInterface dbInterface){

    // Obtém o conjunto de atributos(públicos, privados e protegidos)
    // declarados na  classe “objectClass”.
    Fields[] fields = objectClass.getDeclaredFields();
    // Verifica se cada atributo pode ou não ser escrito diretamente na base de dados.
    for( int i=0; i<fields.length; i++){
        // Obtém o valor do atributo “i” no objeto “object”.
        Object fieldValue = fields[i].getFieldValue(object);
        // Se o objeto “fieldValue” não pode ser escrito diretamente na base de dados,
        // então este objeto  é promovido a objeto persistente.
        if( !dbInterface.isClassPersistentCapable(fieldValue.getClass()) )
            // Propaga persistência para fieldValue.
            doPersistence(fieldValue);
    }
    // Para inspecionar todo o estado do objeto é preciso percorrer as superclasses
    // do mesmo para verificar os atributos herdados destas superclasses.
    // java.lang.Object é a classe base de todo objeto java.
    if( !objectClass.getSuperclass().getName().equals(“java.lang.Object”) )
        inspectObjectState(objectClass.getSuperclass(), object, dbInterface);
}
```

Figura 4.3: Algoritmo de propagação da persistência

Implementação

O arcabouço foi implementado utilizando Guaraná [33], uma arquitetura de software reflexiva e um sistema de execução baseado em Java, que não requer mudança na linguagem de programação Java; um programa criado e compilado em qualquer compilador Java irá rodar em Guaraná. Isto significa que aplicações Java já existentes podem incorporar persistência sem modificações no seu código.

Foram implementadas três interfaces para bases de dados distintas para que pudesse ser construído um exemplo prático de uso do arcabouço. A escolha das ferramentas utilizadas na construção das interfaces levou em conta 2 aspectos: (i) facilidade de interface com aplicação Java e (ii) viabilidade para aquisição da ferramenta.

A primeira interface construída foi uma interface para banco de dados PSE. PSE (Persistent Store Engine) [28] é uma versão de distribuição livre bastante simplificada do ObjectStore. PSE possui uma interface de programação para Java através da qual pode-se armazenar e recuperar objetos Java de um repositório de dados. Objetos persistentes e transientes são manipulados pela aplicação da mesma forma.

A segunda interface construída foi uma interface para banco de dados relacional que utiliza JDBC [16] para realizar as operações de escrita e leitura nos bancos de dados. A comunicação de JDBC com os bancos de dados relacionais é feita través de *drivers* que traduzem consultas JDBC para a sintaxe própria do banco de dados relacional. O banco de dados relacional utilizado foi MiniSQL (mSQL) [18], também de distribuição livre. Utilizamos o *driver* JDBC-mSQL da Imaginary Software (também de distribuição livre) para conexão com os bancos de dados mSQL.

A terceira é uma interface para um sistema de arquivos. Utilizamos a interface de linearização de Java[6] para a escrita e recuperação dos objetos dos arquivos.

4.4 Uso do Arcabouço para Migração Incremental de Sistemas Legados

Podem ser construídos dois tipos de pontes a partir do arcabouço: (i) ponte entre um módulo de aplicação migrado e um banco de dados legado e (ii) ponte para auxiliar o processo de migração de dados do banco de dados legado para o banco de dados destino. A construção dessas pontes será comentada a seguir, logo após a descrição do modo de uso genérico do arcabouço para persistência. Este método servirá como base para o desenvolvimento das pontes.

[6] Em Inglês: *serialization interface.*

4.4.1 Uso Genérico do Arcabouço

Para usar o arcabouço para persistência um programador deve executar duas etapas relativamente simples: (i) construção do sistema de persistência e (ii) criação ou restauração das raízes de persistência da aplicação.

Para construção do sistema de persistência, as partes genéricas do arcabouço são especializadas de acordo com as necessidades de cada aplicação. É preciso criar as interfaces de base de dados (Seção 4.3.2) necessárias, o esquema para registro dos objetos persistentes e a política de alocação desses objetos (Seção 4.3.2). Só é preciso implementar as partes adaptáveis do arcabouço, pois o restante do código será reutilizado.

O sistema de persistência construído é um programa ortogonal a aplicação de objetos persistentes. A ligação entre esse sistema de persistência e a aplicação é estabelecida com a criação ou restauração das raízes de persistência da aplicação (Seção 4.3.2). Para isso, cria-se uma instância da classe `MetaPersistent` e invoca-se o método `persistent`, para criar uma nova raiz de persistência, ou o método `restore`, para restaurar uma raiz que já existe. Deve ser criada uma instância de `MetaPersistent` para cada raiz. As instâncias de `MetaPersistent` são meta-objetos encarregados de implementar a persistência do objeto raiz associado a eles.

4.4.2 Ponte entre Módulo da Aplicação e Sistema de Banco de Dados

A construção deste tipo de ponte a partir do arcabouço dá-se em três passos: (i) construção da interface de base de dados, (ii) iniciação do servidor de nomes e (iii) reincarnação dos objetos.

Primeiro é preciso construir a interface de base de dados que converte o modelo de objetos da aplicação no modelo de dados do banco de dados legado e vice-versa. Essa interface é construída como uma subclasse da classe `DatabaseInterface` (Figura 4.2). Essa interface será usada sempre que um objeto da aplicação precisar ser recuperado ou escrito no banco de dados legado.

O segundo passo é registrar os objetos persistentes no servidor de nomes e criar objetos de localização para esses objetos persistentes, com informações sobre como localizá-los no banco de dados legado. Os objetos de localização são instâncias de subclasses da classe `Location` (Figura 4.2). O método `getDatabaseInterface` desses objetos de localização deve retornar a classe de interface de base de dados construída no primeiro passo.

Depois do segundo passo, a ponte já está pronta para ser usada. O terceiro passo associa a ponte aos objetos persistentes da aplicação.

No terceiro passo são inseridas chamadas para o método `restore` da classe `MetaPersistent` (Figura 4.2) no código da aplicação, para reincarnar os objetos persistentes.

Esses objetos são reincarnados a partir dos dados obtidos do banco de dados legado. Esses dados são localizados usando o objeto de localização construído no segundo passo e obtidos usando a interface de base de dados do primeiro passo. Ainda no método `restore`, os objetos persistentes reincarnados são associados a um meta-objeto da classe `MetaPersistent` e a ligação da ponte com a aplicação está estabelecida.

4.4.3 Migração de Dados

A ponte para migração de dados é uma extensão da ponte definida na Seção anterior. Para possibilitar a migração de dados, a ponte anterior é modificada da seguinte maneira. Primeiro é construída uma interface de base de dados para o novo banco de dados, ou seja, uma interface para recuperar e escrever os objetos persistentes no banco de dados destino. Durante a migração, a ponte irá usar a interface para o banco de dados legado e a interface para o banco de dados destino.

Em seguida, é definida a política de migração dos dados. Essa política determina quando e para qual banco de dados os objetos da aplicação devem ser migrados. Essa política pode estar implementada em uma subclasse de `MetaPersistent` (Figura 4.2) ou pode estar implementada em uma classe distinta, conhecida por `MetaPersistent`.

Finalmente, os métodos `handleFieldRead` e `handleFieldWrite` da classe `MetaPersistent` precisam ser rescritos para consultar a política de migração sempre que for feito um acesso a um objeto persistente armazenado no banco de dados legado. Neste caso, a política de migração determina se o objeto será migrado ou não. Em caso afirmativo, é criada uma localização para o objeto persistente no banco de dados destino (criação de novos objetos `Location` (Figura 4.2)), a antiga localização é substituída no servidor de nomes e a escrita do objeto no banco de dados destino é autorizada. Essa escrita já utilizará a nova localização do dado. A partir daí, todo acesso ao objeto será direcionado para o banco de dados destino.

4.4.4 A Função das Pontes Após o Processo de Migração

As pontes são necessárias durante o processo de migração para realizar a mediação entre componentes migrados e componentes legados. Por outro lado, a utilidade das pontes vai além do auxílio durante a migração, elas podem ser incorporadas ao sistema destino no final do processo. Isto pode acontecer quando, por motivos técnicos ou econômicos, um componente legado é mantido no sistema ou quando a ponte pode servir a futuras migrações. Utilizando as idéias apresentadas nesse artigo, a funcionalidade das pontes pode ser alterada à medida que o processo de migração avança, como mostra a Figura 4.4.

Numa primeira etapa, uma ponte é introduzida para realizar a mediação entre os módulos da aplicação destino e o banco de dados legado (Figura 4.4(a)). Na fase seguinte, a ponte continua implementando a interface entre a aplicação e banco de dados, mas agora também participa da migração dos dados do banco de dados legado para o banco de dados destino, que ocorre sob demanda: os dados são migrados na medida em que são demandados pela aplicação; a ponte obtém dados do componente legado e os transfere para o componente destino. Isto é possível porque as pontes construídas a partir do arcabouço, podem fazer acesso a bases de dados heterogêneas (Figura 4.4(b)).

Terminada a migração, a ponte é incorporada ao sistema e sua função será a de implementar a persistência dos objetos da aplicação (Figura 4.4(b)), mesma função desempenhada no início, mas agora ela escreve e recupera os dados do banco de dados destino.

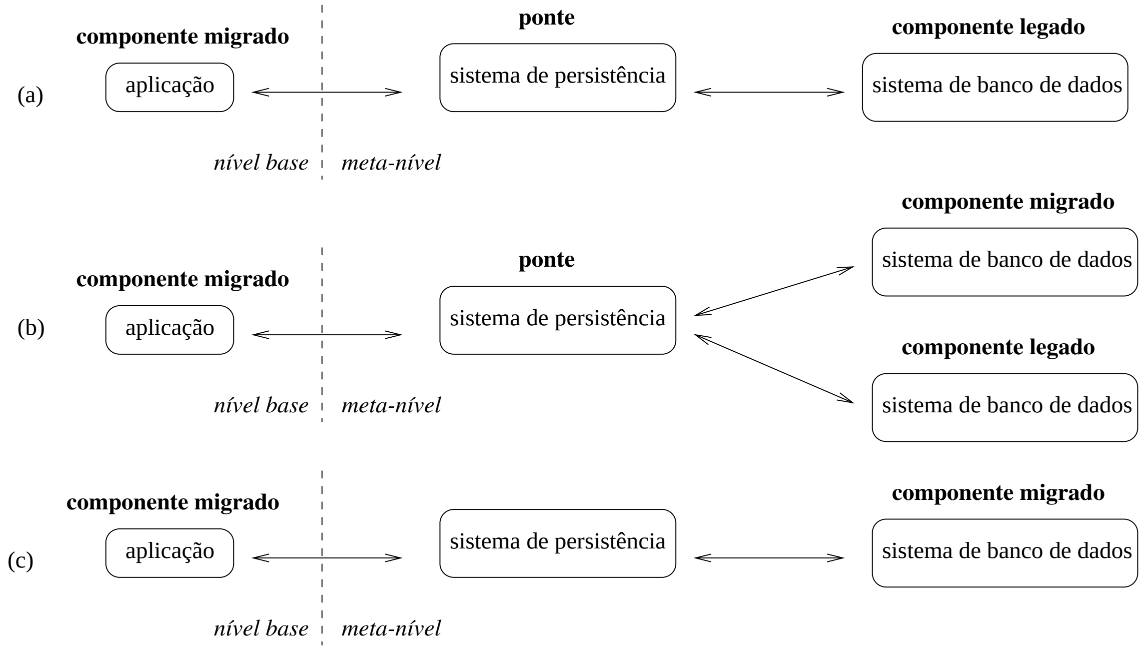

Figura 4.4: (a) Utilizando pontes para interface entre aplicação destino e banco de dados legado (b) Utilizando pontes para migração de dados (c) Ponte incorporada ao sistema destino

4.5 Conclusão

Esse artigo descreveu um arcabouço reflexivo para persistência de objetos em bases de dados heterogêneas que pode ser usado como ferramenta de auxílio no processo de migração incremental de sistemas legados. Entretanto, a utilidade do arcabouço não se restringe a construção de pontes para uso no processo de migração incremental de sistemas legados. O arcabouço representa uma abordagem alternativa para o problema de persistência de objetos em bases de dados heterogêneas. Nossa abordagem difere das demais por propor uma solução simples que resolve os problemas de transparência e heterogeneidade de bases de dados.

O sistema foi totalmente implementado e testado para a construção de um protótipo de um sistema de comércio pela Internet. Dados preliminares sobre o custo para a implementação reflexiva do arcabouço parecem indicar que a sua adoção simplifica o processo de implementação de persistência de objetos.

Esse trabalho mostra que é possível implementar persistência de objetos de uma forma totalmente transparente para o programador da aplicação, em contraste com as soluções não transparentes oferecidas por outros sistemas para persistência.

Capítulo 5

Uma Avaliação de Desempenho de Reflexão Computacional

Este Capítulo traz uma avaliação de desempenho de reflexão computacional, realizada a partir do arcabouço para persistência de objetos. Assim que tomamos a decisão de realizarmos essa avaliação de desempenho, nos deparamos com um problema interessante: a inexistência de *benchmarks* para sistemas reflexivos.

A estratégia adotada pela maioria dos trabalhos que avaliam o desempenho de reflexão computacional, consiste em medir o custo adicional imposto pelo uso de reflexão localmente, isto é, para uma operação da aplicação medem apenas o custo da interceptação e materialização da operação. Além disso, esses trabalhos consideram aplicações onde o número de objetos reflexivos e de interações entre eles é relativamente pequeno. Com essas restrições de localidade e escala, esses testes podem não refletir o custo do uso de reflexão computacional em aplicações de médio e grande porte.

A estratégia que adotamos para medir o desempenho de reflexão computacional tenta minimizar esses problemas de localidade e escala. Nossa estratégia consiste no projeto de uma aplicação A, que manipula objetos persistentes e de um sistema de persistência P, que implementa a persistência dos objetos de A. A partir daí são construídas duas versões dessa aplicação: a primeira reflexiva, com P implementado no meta-nível de A e a segunda não-reflexiva. Para medir o desempenho selecionamos um conjunto de tarefas de A, medimos o tempo de execução dessas tarefas nas duas versões e comparamos os resultados.

A aplicação A utilizada nos testes é um Sistema de Comércio pela Internet desenvolvido em Java. O mecanismo P para a versão reflexiva de A é construído utilizando o arcabouço reflexivo para persistência de objetos. Para a versão não-reflexiva de A implementamos uma versão não-reflexiva do arcabouço, substituindo os mecanismos de interceptação e materialização de operações pela invocação explícita das operações persistentes no código da aplicação.

Este Capítulo está organizado da seguinte forma: a Seção 5.1 descreve o Sistema de Comércio pela Internet em termos de sua interface com o usuário e seu modelo de dados; a Seção 5.2 descreve a construção das versões reflexiva e não-reflexiva do Sistema de Comércio pela Internet, bem como as modificações realizadas no arcabouço para a construção da segunda versão; a Seção 5.3 descreve os testes de desempenho realizados, apresenta os resultados obtidos e uma discussão desses resultados; a Seção 5.4 descreve alguns trabalhos relacionados e a Seção 5.5 apresenta algumas considerações finais.

5.1 O Sistema de Comércio pela Internet

O principal objetivo do Sistema de Comércio pela Internet (SCI) é proporcionar aos usuários da Internet um ambiente gráfico para pesquisa de preço e compra de produtos em diferentes fornecedores[1], enquanto proporciona aos fornecedores mais uma forma de colocar seus produtos à venda pela rede.

Os usuários fazem acesso ao SCI pela Internet através de duas interfaces homem-computador (IHC) distintas, adequadas aos dois tipos de usuários do sistema: fornecedores e clientes. Os fornecedores fazem acesso ao SCI pela Interface do Fornecedor (Figura 5.1). Através dessa interface, os fornecedores podem cadastrar-se no sistema, alterar seu cadastro ou se desligarem do sistema. Os clientes fazem acesso ao SCI pela Interface de Vendas (Figura 5.1). Através dessa interface, os clientes podem fazer pesquisas de preço nos diferentes fornecedores cadastrados e, eventualmente, elaborar um pedido de compra em um fornecedor.

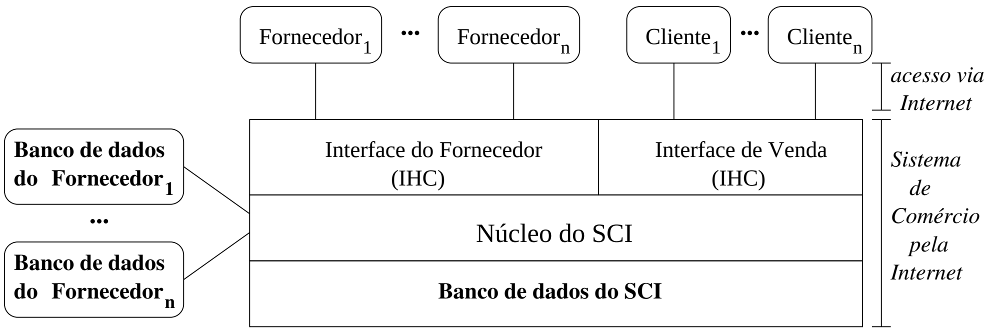

Figura 5.1: Arquitetura do SCI

O SCI possui um banco de dados local (Figura 5.1) onde são armazenados os cadastros dos fornecedores do sistema e faz acesso remotamente aos bancos de dados desses fornecedores, para buscar informações sobre seus produtos e enviar pedidos de compra de clientes. A não existência de um banco de dados que centraliza os dados dos produtos de todos os

[1]Um fornecedor no SCI representa uma empresa que vende produtos.

fornecedores cadastrados é uma característica que distingue o SCI de outros sistemas do mesmo tipo. Isto contribui para que os dados dos produtos estejam sempre atualizados, o que poderia não acontecer se o sistema mantivesse um banco de dados local com todos esses dados.

Foram duas as motivações para usar o SCI em nossos testes: (i) o SCI precisa de um sistema de persistência heterogêneo que permita a interface com os diferentes bancos de dados dos fornecedores cadastrados no sistema; (ii) o SCI não foi construído para ser um exemplo de uso do arcabouço, o que nos permitiu avaliar a facilidade de integração de um sistema de persistência construído a partir do arcabouço à uma aplicação já existente.

5.1.1 Interface do Fornecedor

Para tornar seus produtos disponíveis através do SCI, o fornecedor precisa primeiro se cadastrar no sistema via Interface do Fornecedor. O fornecedor, ao se cadastrar, fornece informações sobre sua empresa, tais como nome, endereço e telefone, informações sobre como localizar o seu banco de dados de produtos, tais como nome e URL do banco de dados e informações sobre o esquema de seu banco de dados, tais como tipo de dados disponível sobre os produtos (preço, tamanho, cor, etc), chaves de pesquisa, etc.

Os fornecedores estão agrupados no sistema pelo ramo de suas atividades comerciais, chamado no sistema de *categoria do fornecedor*. O SCI possui uma lista de categorias pré-definidas: supermercado, loja de departamentos, livraria, etc. O fornecedor informa a categoria a que pertence no cadastramento.

Os fornecedores cadastrados possuem um login e uma senha de acesso ao SCI. O login de um fornecedor é o CGC de sua empresa e a senha é determinada pelo fornecedor no momento do cadastro. Para alterar os dados de seu cadastro ou se desligar do sistema, o fornecedor precisa usar seu login e sua senha. Dessa forma garantimos que só o próprio fornecedor tem acesso aos dados de seu cadastro.

As operações da Interface do Fornecedor envolvem acesso apenas ao banco de dados local do SCI.

5.1.2 Interface de Venda

Enquanto a consulta de um fornecedor envolve basicamente as operações de cadastro, alteração e desligamento, a consulta de um cliente ao sistema envolve uma série de operações:

- ***Pesquisa de fornecedor*** - O cliente faz acesso ao SCI pela Internet através da Interface de Vendas e consulta a lista de fornecedores cadastrados na categoria onde ele deseja realizar uma pesquisa de produto, por exemplo a lista de fornecedores cadastrados na categoria supermercado.

Essa operação envolve acesso apenas ao banco de dados local do SCI, para buscar a lista de categorias e de fornecedores cadastrados no sistema.

- ***Conecta fornecedor*** - De posse da lista de fornecedores da categoria selecionada, o cliente seleciona um fornecedor e solicita uma conexão com esse fornecedor. O sistema exibe as chaves de pesquisa e os tipos de dados disponíveis sobre os produtos do fornecedor e conecta o banco de dados do mesmo (banco de dados remoto). Exemplos de tipos de dados dos produtos de um fornecedor da categoria supermercado, seriam nome do produto, marca e preço. A chave de pesquisa é o tipo de informação do produto, por exemplo nome, usado para selecionar um conjunto de produtos.

Essa operação envolve acesso ao banco de dados local do SCI e ao banco de dados do fornecedor. As chaves de pesquisa e os tipos de dados dos produtos são informações obtidas da ficha de cadastro do fornecedor, armazenada no banco de dados local do SCI.

- ***Pesquisa de produto*** - A pesquisa de produto é realizada através de chaves de pesquisa. Suponha que um cliente deseja conhecer as marcas e o preço do arroz comercializados pelo supermercado contatado por ele. Para isso, ele seleciona a chave de pesquisa "nome" e digita a palavra a ser pesquisada "arroz". O SCI realiza então uma consulta no banco de dados do supermercado para selecionar todos os produtos cujo "nome" contém a string "arroz" e, em seguida, mostra o resultado da consulta.

Essa operação envolve acesso apenas ao banco de dados do fornecedor.

- ***Elaboração de uma cesta de produtos*** - A cesta é um conjunto de produtos selecionados pelo cliente num determinado fornecedor. A cada pesquisa de produto, o cliente seleciona os itens que deseja adicionar à cesta, bem como a quantidade que deseja de cada item. A cesta exibe o preço parcial e total de seus itens e o nome do seu fornecedor. O cliente pode elaborar várias cestas em fornecedores diferentes para comparar preços e depois escolher aquela que deseja comprar. Não existe um banco de dados de cestas: as cestas associadas a um pedido de compra são armazenadas no banco de dados do fornecedor, juntamente com o pedido de compra. As demais são perdidas.

A operação de elaborar uma cesta de produtos não envolve acesso a bancos de dados.

- ***Submete pedido de compra*** - O pedido de compra sempre é feito com base em uma cesta de produtos elaborada pelo cliente. Para submeter um pedido, o cliente precisa primeiro preencher uma ficha com informações como: endereço para entrega da compra e cartão de crédito para cobrança. O segundo passo é determinar a forma de pagamento: a vista, a prazo, número e valor das parcelas e data do pagamento. As opções de pagamento também são definidas pelo fornecedor no seu cadastro. Em seguida, o cliente pode submeter o pedido, que será enviado e armazenado no banco de dados do fornecedor.

Essa operação envolve acesso ao banco de dados do SCI para obter algumas informações do cadastro do fornecedor e acesso ao banco de dados do fornecedor para armazenar o pedido de compra.

5.1.3 Modelo de Objetos do SCI

A Figura 5.2 mostra um diagrama parcial de classes do SCI. Todas as instâncias das classes representadas nesse diagrama são objetos persistentes da aplicação ou são instâncias que referenciam objetos persistentes da aplicação.

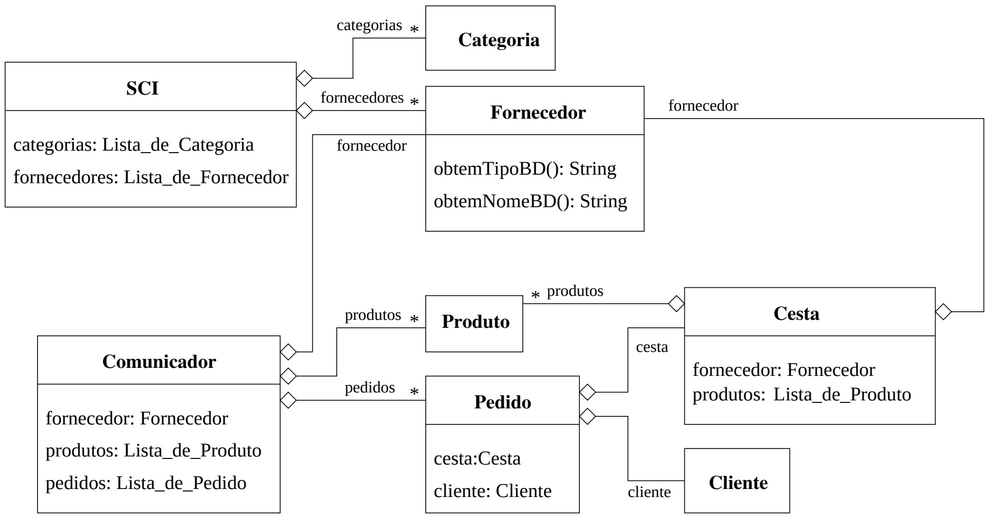

Figura 5.2: Diagrama parcial de classes do SCI

A classe `SCI` contém a lista de categorias e a lista de fornecedores cadastrados no sistema, representados pelos atributos `categorias` e `fornecedores`, respectivamente. A lista de categorias é um conjunto de objetos da classe `Categoria`. Essa classe representa uma categoria do fornecedor pré-definida no SCI, como supermercado ou loja de departamentos. A lista de fornecedores é um conjunto de objetos da classe `Fornecedor`. Essa classe representa um fornecedor cadastrado no SCI e contém todos os dados do cadastro desse fornecedor. A lista de categorias e a lista de fornecedores são dados persistentes da aplicação armazenados no banco de dados local do SCI.

A classe `Comunicador` representa a comunicação entre o SCI e o banco de dados de um fornecedor específico. Essa classe contém a lista de produtos comercializados pelo fornecedor e os pedidos de compra submetidos à esse fornecedor. O atributo `produtos` é a lista de produtos comercializados pelo fornecedor, o atributo `pedidos` é a lista de pedidos de compra submetidos ao mesmo e o atributo `fornecedor` representa o fornecedor contatado. A lista de produtos é representada no sistema por um conjunto de objetos da classe `Produto`. Essa classe representa um produto comercializado no SCI. A lista de produtos é um dado persistente do sistema obtido do banco de dados do fornecedor. A lista de pedidos é representada por um conjunto de objetos da classe `Pedido`. Essa classe representa um pedido de compra submetido por um cliente à um fornecedor. A lista de pedidos é armazenada no banco de dados do fornecedor.

A classe `Pedido` está associada a uma cesta de compras e a um cliente, representados pelas classes `Cesta` e `Cliente`, respectivamente. A classe `Cesta` representa uma cesta de produtos elaborada no SCI. Toda cesta contém uma lista de produtos inseridos nessa cesta e o fornecedor desses produtos. Nem toda instância dessa classe é um objeto persistente da aplicação. As únicas instâncias persistentes dessa classe são aquelas associadas a um pedido de compra (instância da classe `Pedido`). Isso significa que várias cestas de produtos podem ser elaboradas por um cliente no SCI, mas apenas aquelas que o cliente efetivamente comprou serão dados persistentes armazenados no banco de dados do fornecedor, juntamente com um pedido de compra.

A classe `Cliente` representa um cliente que efetuou uma compra no SCI. Essa classe contém todos os dados desse cliente. As instâncias dessa classe também são armazenadas no banco de dados do fornecedor juntamente com um pedido de compra.

5.2 Persistência de Objetos no SCI

A persistência dos objetos no SCI foi inicialmente implementada utilizando JDBC [16] para interface com os bancos de dados do SCI e dos fornecedores. Para usar o SCI em nossos testes de desempenho, o sistema de persistência que usa JDBC foi substituído por um sistema de persistência construído a partir do arcabouço.

Para os testes de desempenho precisamos de duas versões do SCI: a primeira com um sistema de persistência construído a partir do arcabouço reflexivo para persistência de objetos e a segunda com um sistema de persistência construído usando uma versão não-reflexiva do arcabouço. Na próxima Seção descrevemos como a primeira versão do SCI foi construída. Em seguida, descrevemos a implementação da versão não-reflexiva do arcabouço e a construção da segunda versão do SCI.

5.2.1 Construindo um Sistema de Persistência Reflexivo

Nesta Seção descrevemos a criação de um sistema de persistência para o SCI a partir do arcabouço reflexivo para persistência de objetos. O sistema de persistência é implementado no meta-nível da aplicação de forma transparente para a mesma e sem precisar modificar seu código.

Para a construção desse sistema de persistência, seguimos os mesmos passos descritos na Seção 3.4.1. O primeiro passo é definir as interfaces de bases de dados necessárias. Em

Classe 5.1 SCI_LocationFactory

```
public class SCI_LocationFactory implements LocationFactory{
    public Location create(Object object){
        // Aloca o novo objeto persistente no banco de dados "bdSCI.odb".
        // PSEInterface é a interface usada para fazer acesso a esse banco de dados.
        return new OAddress("PSEInterface", "bdSCI.odb");
    }
}
```

nossos testes, vamos usar as três interfaces pré-definidas no arcabouço, implementadas nas classes `PSEInterface`, `JDBCInterface` e `SerializationInterface` (Figura 3.4). O SCI manipula objetos persistentes armazenados no seu banco de dados local e nos bancos de dados dos fornecedores, logo é preciso definir as interfaces de base de dados usadas para acessar cada um desses bancos de dados. Para o banco de dados local do SCI decidimos usar a interface de base de dados implementada pela classe `PSEInterface`. Para o banco de dados dos fornecedores a decisão fica a cargo do próprio fornecedor. O fornecedor deve definir, no momento do cadastro, qual das três interfaces implementadas no arcabouço deve ser usada para fazer acesso ao seu banco de dados.

O próximo passo é gerar os objetos de localização para os objetos persistentes, ou seja, criar as classes que alocam os objetos persistentes nas bases de dados. A classe `SCI_LocationFactory` (Classe 5.1) aloca objetos persistentes no banco de dados local do SCI e a classe `Fornec_LocationFactory` (Classe 5.2) aloca objetos persistentes no banco de dados de um fornecedor específico, definido no parâmetro do construtor da classe. Como as três interfaces de base de dados do sistema de persistência usam objetos de localização da classe `OAddress` (Figura 3.4), todos os objetos de localização criados são instâncias dessa classe.

Com a definição das interfaces de bases de dados que serão usadas, dos objetos de localização e das classes que criam esses objetos, o sistema de persistência está pronto para ser usado. Falta agora associar o sistema de persistência à aplicação de objetos persistentes. Isso é feito criando ou restaurando as raízes de persistência da aplicação.

As raízes de persistência do SCI podem ser identificadas no diagrama da Figura 5.2. Os atributos `fornecedores` e `categorias` da classe `SCI`, e `produtos` e `pedidos` da classe `Comunicador` são as raízes de persistência da aplicação. Objetos das classes `Fornecedor`, `Categoria`, `Produto`, `Pedido`, `Cesta` e `Cliente` referenciados pelas raízes de persistência, serão promovidos a objetos persistentes.

As raízes de persistência são criadas/restauradas no meta-nível da aplicação, por meta-objetos associados as instâncias das classes `SCI` e `Comunicador`. A classe `MO_SCI` (Classe A.1) implementa o meta-objeto associado a instância da classe `SCI` e a classe `MO_Comunicador`

Classe 5.2 Fornec_LocationFactory

```
public class Fornec_LocationFactory implements LocationFactory{
    String dbType;
    String dbName;
    // O tipo e o nome da interface de base de dados são definidos pelo fornecedor.
    public Fornec_LocationFactory(Fornecedor f){
        // Obtém o tipo da interface de base de dados do fornecedor.
        dbType =f.obtemTipoBD();
        // Obtém o nome do banco de dados do fornecedor.
        dbName=f.obtemNomeBD();
    }
    public Location create(Object object){
        // Aloca o novo objeto persistente no banco de dados definido pelo fornecedor.
        return new OAddress(dbType, dbName);
    }
}
```

(Classe A.2) implementa o meta-objeto associado as instâncias da classe `Comunicador`. O código dessas duas classes está no Apêndice A.

As raízes de persistência são criadas/restauradas invocando-se os métodos `persistent/restore` da classe `MetaPersistent` (Figura 3.3). As raízes `fornecedores` e `categorias` devem ser associadas a um objeto de alocação de objetos persistentes da classe `SCI_LocationFactory` (Classe 5.1) e as raízes `produtos` e `pedidos` associadas a um objeto de alocação da classe `Fornec_LocationFactory` (Classe 5.2). Dessa forma, os objetos persistentes serão corretamente alocados nos bancos de dados do SCI e dos fornecedores.

O momento certo para a criação/restauração das raízes de persistência é imediatamente após a execução do construtor dessas classes, antes que qualquer acesso as raízes de persistência seja feito. Se a restauração dessas raízes acontecesse antes da invocação do construtor, as raízes de persistências (`fornecedores`, `categorias`, `produtos` e `pedidos`) seriam iniciadas pelo construtor da classe, perdendo o seu estado restaurado.

Para associar os meta-objetos às instâncias das classes `SCI` e `Comunicador` sem modificar o código da aplicação, é preciso usar dois artifícios: (i) criar meta-classes para as classes `SCI` e `Comunicator` e (ii) criar um meta-programa. O papel das meta-classes é associar meta-objetos (`MO_SCI` e `MO_Comunicador`) à toda nova instância das classes `SCI` e `Comunicador`. Já o meta-programa associa as meta-classes às classes da aplicação e inicia a mesma. A classe `MC_SCI` (Classe 5.3) implementa a meta-classe associada a classe `SCI`, a classe `MC_Comunicador` (Classe 5.4) implementa a meta-classe associada a classe `Comunicador` e a classe `MetaMain` (Classe 5.5) implementa o meta-programa.

Classe 5.3 MC_SCI

```
public class MC_SCI extends MetaObject{
    public void handle(Message message, Object object){
        // Se a mensagem recebida é NewObject, então uma instância da
        // classe SCI foi criada.
        if( message.getClass().getName().equals("NewObject") )
            // Associa uma meta-configuração a nova instância da classe SCI.
            Guarana.reconfigure( ((NewObject)message).getObject(), null, new MO_SCI() );
    }
}
```

Classe 5.4 MC_Comunicador

```
public class MC_Comunicador extends MetaObject{
    public void handle(Message message, Object object){
        // Se a mensagem recebida é NewObject, então uma instância da
        // classe Comunicador foi criada.
        if( message.getClass().getName().equals("BR.unicamp.Guarana.NewObject") )
            // Associa uma meta-configuração a nova instância da classe Comunicador.
            Guarana.reconfigure( ((NewObject)message).getObject(), null,
                                  new MO_Comunicador() );
    }
}
```

Classe 5.5 MetaMain

```
// Meta Programa
public class MetaMain{
    public static void main(String[] args){
        // Assoia meta-configuração a classe SCI.
        Guarana.reconfigure(SCI.class, null, new MC_SCI() );
        // Assoia meta-configuração a classe Comunicador.
        Guarana.reconfigure(Comunicador.class, null, new MC_Comunicador() );
        // Inicia a aplicação.
        SCI.main();
    }
}
```

5.2.2 Construindo um Sistema de Persistência Não-Reflexivo

Nesta Seção descrevemos a versão não-reflexiva do arcabouço e a criação de um sistema de persistência para o SCI a partir dessa versão.

As únicas modificações realizadas no arcabouço foram feitas para substituir os mecanismos de interceptação e materialização de operações e para modificar a forma como a persistência é associada aos objetos, já que não existe mais o mecanismo de associação de meta-objetos aos objetos persistentes.

Para a versão não-reflexiva do arcabouço foram construídas duas novas classes: `PersistentObject` (Figura 5.3(a)) e `ObjectState` (Figura 5.3(b)). Nessa versão, a persistência é implementada através de herança: todas as classes de objetos persistentes são subclasses de `PersistentObject`. Essa classe implementa a associação e a interface com o sistema de persistência fazendo as vezes de um MOP, mas sem a transparência alcançada pelo mesmo. O atributo `mp` representa a associação entre um objeto persistente e o sistema de persistência. Se o objeto é persistente, então `mp` possui uma referência para uma instância de `MetaPersistent` (Figura 3.3). Se o objeto não é persistente, então `mp` é `null`.

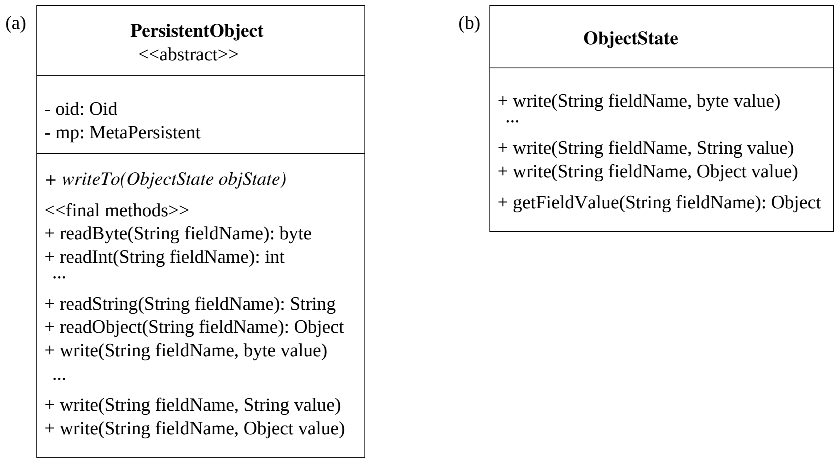

Figura 5.3: (a) Classe `PersistentObject` (b) Classe `ObjectState`

Os métodos `write` e `read` delegam para `mp` as operações de escrita e leitura em atributos do objeto persistente. Existe um método `read` e um método `write` para cada tipo primitivo de Java e para os tipos `String` e `Object`. A invocação desses métodos é feita no código da aplicação sempre que uma operação de escrita ou leitura de atributo na base de dados é necessária. O método invocado depende do tipo do atributo que será lido/escrito. Para ler um atributos do tipo `byte`, por exemplo, deve-se invocar o método `readByte`.

O método `writeTo` é um método abstrato que precisa ser redefinido nas subclasses de `PersistentObject`. Esse método é invocado pelo sistema de persistência para obter o estado do objeto persistente e em seguida salvá-lo na base de dados. A classe ObjectState (Figura 5.3(b)) guarda o estado de um objeto persistente. O método writeTo deve escrever o estado do objeto no parâmetro `objectState` para que esse objeto possa ser salvo na base de dados. O estado de um objeto é escrito através dos métodos `write` da classe `ObjectState`. Existe um método `write` para cada tipo primitivo de Java e para os tipos `String` e `Object`. O sistema de persistência usa o método `getFieldValue` para ler o estado do objeto.

Na versão reflexiva o estado interno do objeto persistente é obtido através de objetos `OperationFactory` [33] de forma transparente. Com `OperationFactory` o meta-objeto associado a um objeto de nível base pode criar operações de leitura e escrita em atributos desse objeto, independente da visibilidade desses atributos. Esta é mais uma funcionalidade provida pelo MOP de Guaraná.

Em resumo, o arcabouço sofreu as seguintes modificações para a construção da versão não-reflexiva: adição de duas novas classes, `PersistentObject` e `ObjectState`, e pequenas modificações em algumas classes do arcabouço, para eliminar os mecanismos de reflexão, como substituição dos objetos `OperationFactory` por objetos `ObjectState`.

O sistema de persistência para o SCI construído a partir do arcabouço não-reflexivo, possui as mesmas características daquele definido a partir da versão reflexiva do arcabouço, ou seja, mesmas interfaces de base de dados, objetos de localização e objetos de alocação de objetos persistentes. Por outro lado, a aplicação de objetos persistentes precisa sofrer algumas modificações para utilizar o sistema de persistência construído a partir do arcabouço não-reflexivo, ao contrário do que aconteceu na primeira versão do SCI, onde a persistência foi implementada sem modificações no código da aplicação.

Primeiro, as classes de objetos persistentes da aplicação foram rescritas para herdar da classe `PersistentObject` e o método `writeTo` foi implementado em cada uma dessas classes.

Segundo, foram implementados métodos nas classes de objetos persistentes para obter e modificar os atributos dessas classes. Os atributos passaram a ser manipulados pela aplicação somente através desses métodos. Isto foi necessário para garantir o mesmo número de operações de escrita e leitura nas bases de dados da versão reflexiva. Os métodos que obtêm um atributo, fazem isso invocando um método `read` (da classe base `PersistentObject`), e os métodos que modificam o valor de um atributo, fazem isso invocando o método `write`. Isto claro, só acontece se a instância for persistente. Dessa forma, garante-se que todo acesso a atributos de objetos persistentes leva a um acesso ao banco de dados, como na versão reflexiva.

Com essas duas modificações, está pronta a segunda versão do SCI. Na próxima Seção descrevemos os testes de desempenho realizados a partir das duas versões do SCI.

5.3 Experimento

O testes mediram o tempo de execução das duas versões do SCI em 11 operações típicas desse sistema. As operações foram executadas 5 vezes em cada versão e a comparação foi realizada com base em uma média desses tempos.

Todas as operações testadas envolvem interface com um usuário externo, que pode ser um fornecedor ou um cliente do sistema. Logo, tomamos o cuidado de computar apenas o tempo de processamento das operações, eliminando o tempo de interação do usuário com a interface do sistema.

Como o arcabouço precisa de Guaraná para ser executado e este, por sua vez, utiliza uma JVM modificada para incorporar reflexão, não tivemos como executar nossos testes em um browser, que utiliza a JVM padrão. Os testes foram então executados com acesso local ao SCI. Os bancos de dados utilizados nos testes também eram locais.

Para que o acesso aos bancos de dados não influencie as medidas de tempo foram tomados alguns cuidados. Em cada operação:

- O número de objetos persistentes criados/restaurados é o mesmo nas duas versões.
- O número de operações de leitura e escrita em bancos de dados também é o mesmo nas duas versões.
- Os bancos de dados usados pelas duas versões têm a mesma estrutura e conteúdo.
- Os dados de entrada da operação são os mesmos para as duas versões.

5.3.1 Plataforma

Os testes foram realizados em um Pentium 100MHz, 32MB memória RAM, com RedHat Linux 5.2. Usamos Guaraná v.1.5.2 nos testes com a versão reflexiva do SCI. Guaraná v.1.5.2 modifica a máquina virtual de Kaffe v.1.1 para incorporar reflexão a Java. Kaffe é uma implementação da interface de programação e da máquina virtual de Java, de distribuição livre. A versão não-reflexiva do SCI foi testada usando Kaffe v.1.1, mas sem as modificações introduzidas por Guaraná para incorporar reflexão.

5.3.2 Cenários de Teste

Os testes foram realizados com as seguintes operações:

Interface do Fornecedor:

1. *Inicia cadastro*: inicia a Interface do Fornecedor para cadastro de um novo fornecedor no SCI.

2. *Cadastro*: cadastra um novo fornecedor no SCI.

3. *Exclui cadastro*: exclui o cadastro de um fornecedor do SCI.

4. *Recupera cadastro*: recupera o cadastro de um fornecedor do SCI para uma eventual alteração nos dados desse cadastro.

5. *Altera cadastro*: altera o cadastro de um fornecedor do SCI.

Interface de Venda:

6. *Inicia pesquisa*: inicia a Interface de Venda.

7. *Pesquisa fornecedor*: realiza uma pesquisa por categoria na lista de fornecedores cadastrados no SCI.

8. *Conecta fornecedor*: mostra informações sobre os produtos do fornecedor selecionado e conecta o banco de dados desse fornecedor.

9. *Pesquisa produto*: realiza uma pesquisa por produtos no banco de dados de um fornecedor específico.

10. *Submete pedido*: envia um pedido de compra a um fornecedor.

Cenário de uso:

11. *Compra de produtos*: esta operação engloba a seqüência de operações executadas por um cliente para realizar uma compra de produtos, o que corresponde a execução das operações 6 a 10.

As operações da Interface do Fornecedor fazem acesso ao banco de dados do SCI. As operações 6 e 7 também fazem acesso ao banco de dados do SCI para carregar a lista de categorias e selecionar fornecedores de acordo com a categoria selecionada pelo cliente. Já as operações 8 e 9 fazem acesso a dois bancos de dados diferentes: banco de dados do SCI e banco de dados do fornecedor selecionado. A operação 10 faz acesso ao banco de dados do fornecedor, para submeter o pedido de compra do cliente. Por fim, a operação 11 é um cenário de uso do sistema que envolve o acesso aos bancos de dados do SCI e do fornecedor.

Os testes das operações de 1 a 10 foram realizados sem fazer acesso a bases de dados heterogêneas.. Foram realizados duas baterias de testes com essas operações. Na primeira, os bancos de dados do SCI e do fornecedor eram bancos de dados de objetos manipulados através da interface `PSEInterface`. Na segunda, os dois bancos de dados eram sistemas de arquivos acessados através da interface `SerializationInterface`. Em cada bateria de testes, as operações foram executadas 5 vezes, e a média desses tempos foi calculada separadamente.

Os testes com a operação 11 foram diferentes, por se tratar de um cenário de uso do sistema. Com essa operação foram feitos dois testes. No primeiro, o banco de dados do SCI e do fornecedor eram bancos de dados de objetos acessados através da interface `PSEInterface`. No segundo teste, o banco de dados do SCI continuou o mesmo, mas o banco de dados do fornecedor era um sistema de arquivos acessado através da interface `SerializationInterface`.

Todas as operações descritas foram selecionadas porque envolvem interface com bancos de dados e logo, contam com a participação do sistema de persistência.

5.3.3 Resultados

As Tabelas 5.1, 5.2, 5.3, 5.4, 5.5 e 5.6 mostram o resultado dos testes de desempenho. Os dados das tabelas são exibidos em termos percentuais para facilitar a comparação.

Operação	*Overhead*
Inicia cadastro	+ 97,46%
Cadastro	+ 40,16%
Exclui cadastro	+ 62,12%
Recupera cadastro	+ 78,77%
Altera cadastro	+ 48,44%
Inicia pesquisa	+ 65,75%
Pesquisa fornecedor	+ 67,97%
Conecta fornecedor	+ 83,33%
Pesquisa produto	+ 113%
Submete pedido	+ 65,96%

Tabela 5.1: *Overhead* no tempo de execução da versão reflexiva em relação a versão não-reflexiva, fazendo acesso a bancos de dados de objetos

As Tabelas 5.1 e 5.2 mostram o *overhead* no tempo de execução da versão reflexiva em relação a versão não-reflexiva, fazendo acesso a bancos de dados de objetos e sistemas de arquivos, respectivamente. Por exemplo, se a operação Inicia cadastro na versão não-reflexiva e usando banco de dados de objetos gastou 10s para ser executada, então a mesma operação,

Operação	*Overhead*
Inicia cadastro	+ 79,17%
Cadastro	+ 27,78%
Exclui cadastro	+ 77,78%
Recupera cadastro	+ 78,26%
Altera cadastro	+ 39,13%
Inicia pesquisa	+ 58,82%
Pesquisa fornecedor	+ 70%
Conecta fornecedor	+ 82,35%
Pesquisa produto	+ 126,67%
Submete pedido	+ 78,95%

Tabela 5.2: *Overhead* no tempo de execução da versão reflexiva em relação a versão não-reflexiva, fazendo acesso a sistemas de arquivos

Operação	*Overhead*
Inicia cadastro	+ 883,33%
Cadastro	+ 577,78%
Exclui cadastro	+ 633,33%
Recupera cadastro	+ 995,65%
Altera cadastro	+ 734,78%
Inicia pesquisa	+ 758,82%
Pesquisa fornecedor	+ 665%
Conecta fornecedor	+ 888,24%
Pesquisa produto	+ 1000%
Submete pedido	+ 889,47%

Tabela 5.3: *Overhead* no tempo de execução das operações que fazem acesso a bancos de dados de objetos em relação as operações que fazem acesso a sistemas de arquivos, na versão não-reflexiva

Operação	*Overhead*
Inicia cadastro	+ 983,72%
Cadastro	+ 643,48%
Exclui cadastro	+ 568,75%
Recupera cadastro	+ 998,78%
Altera cadastro	+ 790,63%
Inicia pesquisa	+ 796,30%
Pesquisa fornecedor	+ 655,88%
Conecta fornecedor	+ 893,55%
Pesquisa produto	+ 933,82%
Submete pedido	+ 817,65%

Tabela 5.4: *Overhead* no tempo de execução das operações que fazem acesso a bancos de dados de objetos em relação as operações que fazem acesso a sistemas de arquivos, na versão reflexiva

Operação	*Overhead*
Compra de produtos	+ 62,61%

Tabela 5.5: *Overhead* no tempo de execução da versão reflexiva em relação a versão não-reflexiva fazendo acesso a bancos de dados homogêneos

Operação	*Overhead*
Compra de produtos	+ 54,67%

Tabela 5.6: *Overhead* no tempo de execução da versão reflexiva em relação a versão não-reflexiva usando bancos de dados heterogêneos

usando o mesmo banco de dados, gastou aproximadamente 19,7s para ser executada na versão reflexiva.

As Tabelas 5.3 e 5.4 mostram o *overhead* no tempo de execução de operações que fazem acesso a bancos de dados de objetos em relação as operações que fazem acesso a sistemas de arquivos, na mesma versão da aplicação. As operações da Tabela 5.3 foram executadas na versão não-reflexiva e as operações da Tabela 5.4 foram executadas na versão reflexiva. Por exemplo, o tempo gasto pela operação Inicia cadastro na versão não-reflexiva, fazendo acesso a um banco de dados de objetos, é 883,33% maior que o tempo gasto pela mesma operação, na mesma versão, mas fazendo acesso a um sistema de arquivos. Já o tempo gasto pela operação Inicia cadastro na versão reflexiva fazendo, acesso a um banco de dados de objetos, é 983,72% maior que o tempo gasto pela mesma operação, na mesma versão, mas fazendo acesso a um sistema de arquivos.

As Tabelas 5.5 e 5.6 mostram o *overhead* no tempo de execução da versão reflexiva em relação a versão não-reflexiva na operação Compra de produtos, nas duas configurações de bancos de dados testadas. A Tabela 5.5 mostra que no teste com bancos de dados homogêneos (banco de dados de objetos/ banco de dados de objetos), o tempo de execução da versão reflexiva é 62,61% maior que o tempo gasto pela versão não-reflexiva. A Tabela 5.6 mostra que no teste com bancos de dados heterogêneos (banco de dados de objetos/ sistema de arquivos), o tempo de execução da versão reflexiva é 54,67% maior que o tempo gasto pela versão não-reflexiva.

5.3.4 Discussão

Nesse experimento tomamos todos os cuidados para que os testes de desempenho refletissem apenas o custo adicional do uso de reflexão: construímos a versão não-reflexiva do arcabouço com pequenas modificações no mesmo, feitas de modo a substituir os mecanismos de reflexão (Seção 5.2.2) e tomamos uma série de cuidados para que as operações com banco de dados, que geralmente são operações lentas, não influenciassem os testes de desempenho (Seção 5.3).

Diante desse fato, concluímos, com base nos dados das Tabelas 5.1 a 5.5, que os mecanismos introduzidos para oferecer reflexão computacional são os responsáveis pela queda de até 50% no desempenho da versão reflexiva do SCI com relação a sua versão não-reflexiva.

Por outro lado, esse experimento serviu para comprovar as vantagens da programação reflexiva e em especial do uso do arcabouço reflexivo para persistência de objetos. Usando o arcabouço reflexivo construímos um sistema de persistência ortogonal e totalmente transparente para a aplicação, que pode ser usado sem modificar o código da mesma. Já com o arcabouço não-reflexivo, o código de cada classe teve que ser modificado para incorporar a persistência.

Outra conclusão desse experimento foi que, uma vez conhecida a técnica de reflexão computacional e o MOP de Guaraná, é mais fácil construir um sistema de persistência usando a versão reflexiva do arcabouço do que usando a versão não-reflexiva do mesmo.

Os testes também serviram para comparar o desempenho da interface para banco de dados PSE com o desempenho da interface para sistema de arquivos, que utiliza linearização de Java. Pelas Tabelas 5.3 e 5.4 podemos verificar o quanto a interface para sistema de arquivos é mais rápida que a interface para banco de dados PSE, em todas as operações.

Essa diferença tão grande de desempenho não era esperada. Entretanto, existem dois fatos que podem explicar isto. O primeiro diz respeito a forma como a interface para sistema de arquivos foi implementada. Essa interface foi construída para servir de exemplo de uso da interface de linearização de Java. Não houve uma preocupação maior com o projeto do sistema de arquivo. A interface foi construída para armazenar um objeto por arquivo, o que torna a recuperação de objetos muito mais rápida do que no banco de dados PSE, apesar de ser uma solução inviável para sistemas de médio e grande porte. O segundo, diz respeito a versão do ObjectStore usada. O arcabouço utiliza a ferramenta ObjectStore PSE, que é uma versão de livre distribuição bem simples do ObjectStore, e que não possui o mesmo desempenho do mesmo.

O ótimo desempenho dos sistemas de arquivos em relação ao banco de dados de objetos, levou a decisão de usar bancos de dados homogêneos nos testes das operações 1 a 10. Se isso não fosse feito, o bom desempenho de um tipo de banco de dados poderia alterar as medidas de tempo total, o que foi observado nos testes com bancos de dados heterogêneos da Tabela 5.6. Na operação de Compra de produtos, o custo de reflexão foi menor do que o custo medido nas operações 5 a 10, apesar da operação de Compra de produtos não ser nada mais que uma seqüência de execução dessas cinco operações.

Em resumo, esse experimento forneceu mais evidências de que o custo de reflexão computacional pode ser muito alto e que é necessário ponderar se as vantagens dessa técnica pagam o preço desse custo. Por outro lado, existe um grande número de aplicações onde os benefícios da programação reflexiva, tais como separação de conceitos, transparência e configurabilidade, compensam o preço dessa perda de desempenho.

Os testes foram feitos usado o MOP de Guaraná. Para tirar conclusões definitivas sobre reflexão, é preciso realizar os mesmos tipos de testes descritos nesse Capítulo com outros protocolos de meta-objetos como, por exemplo, Open C++ [8] e MetaXa [14]. Assim, poderemos dizer se o MOP de Guaraná foi o responsável pelo desempenho alcançado, ou se a técnica de reflexão computacional, por si só, tem um custo muito alto.

5.4 Trabalhos Relacionados

Na literatura, existem poucos trabalhos sobre experimentos para medir o custo do uso de reflexão computacional, especialmente se nos restringimos a protocolos de meta-objetos [9, 47]. Além disso, os poucos trabalhos existentes não fornecem evidências suficientes de que os custos introduzidos pela implementação dos mecanismos utilizados para permitir reflexão computacional são aceitáveis.

Nesta Seção, vamos descrever dois trabalhos que também buscaram medir o custo do uso de reflexão computacional através de uma análise comparativa de versões reflexivas e não-reflexivas de aplicações. Os dois trabalhos usaram Open C++ em seus testes.

Ji Xu, Brian Randell e Avelino F. Zorzo [47] realizaram experimentos para medir o custo da introdução de reflexão computacional em uma aplicação distribuída que ordena uma lista de inteiros e tolera falhas de software. Os experimentos foram realizados usando GNU C++ versão 2.6.3 e Open C++ versão 1.2 como linguagens de programação.

As vantagens da implementação em GNU C++ foram o desenvolvimento rápido das aplicações devido a não modificação do compilador, e alguma redução no tempo extra para o projeto de tolerância a falha. Entretanto, as conclusões foram de que a utilização de objetos não foi suficiente para prover uma separação clara entre o código da aplicação e o código extra para a tolerância a falha, criando um ambiente propício ao aparecimento de erros de implementação.

A implementação em Open C++ apresentou, com relação a primeira implementação, um *overhead* no tempo de execução de razão 2, causado pela diferença no tempo de execução entre chamadas de operações C++ e chamadas de operações reflexivas em Open C++. Entretanto, a conclusão do trabalho foi de que quando o custo de comunicação através de uma rede de computadores é considerado, o *overhead* das chamadas reflexivas não tem contribuição significativa no tempo de execução da aplicação e, portanto, o *overhead* no tempo de execução, causado pela introdução de um meta-nível, é aceitável devido a separação clara e estruturada de conceitos nos estágios de desenvolvimento e operacional.

Shigeru Chiba e Takashi Masuda [9] realizaram experimentos para medir o *overhead* no tempo de execução causado pelas chamadas de operações reflexivas, com base no número de argumentos dos procedimentos. Não foram usadas implementações específicas para os testes e os métodos invocados não realizavam nenhuma operação, para que a computação se resumisse ao processo de chamada de função. Foram usados os compiladores Sun C++ versão 2.1 e Open C++[2]. Os experimentos compararam os tempos das chamadas de funções C++, métodos virtuais e métodos reflexivos. Os dois primeiros executado tanto por

[2]O compilador Open C++ é um pré-processador do compilador C++. E uma vez que nenhuma modificação é adicionada ao compilador C++, um programa em Open C++ é traduzido em um código C++.

C++ quanto por Open C++ e o terceiro executado somente por Open C++.

Os experimentos mostraram que a razão entre o tempo de execução da chamada de um método reflexivo e o tempo de execução da chamada de um método virtual cresce com o aumento do número de argumentos do método, uma vez que o *overhead* da chamada reflexiva é atribuído principalmente ao processo de materialização da lista de argumentos. O trabalho mostrou que a chamada de um método reflexivo é 6 a 8 vezes mais lenta do que a chamada de um método virtual. Entretanto, esse *overhead* é desprezível se Open C++ for usado para computação distribuída, pois a latência média das chamadas de métodos do experimento é de poucos milisegundos, enquanto a latência de uma rede está entre centenas de milisegundos e microsegundos.

Os dois trabalhos concluíram que o custo de reflexão computacional é desprezível em aplicações distribuídas, devido a latência de uma rede. Entretanto, nenhum dos trabalhos usou aplicações práticas em seus experimentos. Para que estas conclusões sejam confirmadas é necessário realizar testes com aplicações de médio ou grande porte (centenas de objetos).

5.5 Conclusão

O objetivo desse Capítulo foi construir um exemplo de uso do arcabouço e realizar uma avaliação de desempenho de reflexão computacional a partir desse exemplo, buscando fornecer mais evidências do custo do uso dessa técnica.

Através de experimentos práticos constatamos que os mecanismos para introduzir reflexão computacional nas aplicações podem ter um alto custo. Por outro lado, demonstramos mais uma vez os benefícios da programação reflexiva, e em particular, do MOP de Guaraná.

O conteúdo desse Capítulo é só parte de um trabalho de avaliação de reflexão computacional que pretende-se fazer. Todo o trabalho baseia-se na construção e comparação de versões reflexivas e não-reflexiva de uma aplicação usando MOPs diferentes.

Capítulo 6

Conclusão

As principais contribuições dessa dissertação são:

- Projeto de um *arcabouço reflexivo para persistência de objetos*. O arcabouço é uma infra-estrutura genérica e flexível para a construção de um sistema de persistência transparente e que permite à aplicação manipular objetos persistentes armazenados em bases de dados heterogêneas.

- Implementação do arcabouço reflexivo para persistência de objetos na linguagem Java, usando a arquitetura reflexiva de Guaraná. O arcabouço define uma interface de programação Java que pode ser usada para implementar um sistema de persistência para uma aplicação Java qualquer. As aplicações que utilizam um sistema de persistência construído a partir do arcabouço devem ser executadas usando Guaraná, uma vez que esse introduz modificações na máquina virtual de Java.

- Discussão sobre o uso do arcabouço como ferramenta de apoio ao processo de migração incremental de sistemas legados. Descrevemos como, a partir do arcabouço, é possível (i) implementar pontes transparentes entre um módulo de aplicação migrado e um banco de dados legado e (ii) implementar pontes que auxiliam o processo de migração de dados do banco de dados legado para o banco de dados destino.

- Uma avaliação de desempenho de reflexão computacional que forneceu mais evidências sobre o custo do uso de reflexão computacional. Os resultados de nossos experimentos não reafirmaram o bom desempenho das aplicações reflexivas, descrito em outros trabalhos relacionados. Para nós, isso é um forte indício de que ainda há muito trabalho a ser feito, antes de afirmar que reflexão computacional é uma técnica vantajosa sobre o ponto de visto de custo.

- Contribuição para os testes e a validação da arquitetura reflexiva de Guaraná. Durante a implementação do arcabouço e do exemplo de uso do mesmo foram encontradas e reportadas várias falhas dessa arquitetura. A construção do arcabouço e do exemplo de uso do mesmo serviram ainda para demonstrar as vantagens de algumas funcionalidades do MOP de Guaraná, como o mecanismo de associação dinâmica de objetos e meta-objetos, troca de mensagens no meta-nível e acesso a dados privados e protegidos dos objetos. Por outro lado, os experimentos para medida de desempenho revelaram um custo bastante alto para o uso de reflexão computacional. Mais testes precisam ser feitos para revelar se esse custo deve ser atribuído à técnica de reflexão computacional, à arquitetura reflexiva de Guaraná, ou à uma combinação de ambos.

Trabalhos Futuros

Podemos destacar duas linhas de pesquisa a partir do trabalho apresentado nessa dissertação: (i) continuação dos experimentos para avaliação do custo introduzido pelo uso de reflexão computacional e (ii) construção de um exemplo de uso do arcabouço como ferramenta de apoio ao processo de migração incremental de sistemas legados.

No primeiro caso, a proposta é definir um *benchmark* para sistemas reflexivos e testá-lo em diferentes arquiteturas reflexivas, como Guaraná, Open C++ [8] e MetaXa [14]. Os testes em cada arquitetura reflexiva seguem a mesma filosofia comparativa dos testes realizados nessa dissertação, ou seja, construir versões reflexivas e não-reflexivas das aplicações e comparar os resultados obtidos. Também é possível fazer uma comparação dos resultados entre as diferentes arquiteturas reflexivas testadas. Uma proposta de *benchmark* para sistemas reflexivos seria aplicações que envolvem persistência de objetos utilizando o arcabouço descrito.

A segunda linha de pesquisa é interessante para demonstrar a validade ou não das idéias discutidas no Capítulo 4.

O arcabouço reflexivo para persistência de objetos é ainda parte de um projeto maior, denominado MOLDS [32]. O objetivo de MOLDS é definir uma biblioteca de meta-objetos que implementam funções básicas como persistência, distribuição e migração, mas que podem ser compostas no meta-nível para atender aos diferentes requisitos de aplicação distribuídas.

Apêndice A

Sistema de Persistência Reflexivo do SCI

Este Apêndice contém as definições das classes `MO_SCI` e `MO_Comunicador`. Essas classes fazem parte do sistema de persistência reflexivo do SCI, descrito na Seção 5.2.1.

Classe A.1 MO_SCI

```
public class MO_SCI extends MetaObject{
    ...
    // Este método trata as operações (operation) interceptadas
    // no objeto de nível base (object).
    public Result handle(Operation operation, Object object){
        // Verifica se a operação é uma invocação do construtor da classe.
        if( operation.isConstructorInvocation() &&
            operation.getConstructor().getName().equals(object.getClass().getName()) )
             return Result.inspectResult; // Requisita inspeção do resultado da operação.
        else return Result.noResult; // Retorna um resultado vazio,
    } // indicando que a operação não será tratada.

    // Este método trata o resultado (result) das operações invocadas
    // no objeto de nível base (object).
    public Result handle(Result result, Object object){
        // Verifica se o resultado interceptado é o resultado do construtor da classe.
        if( result.getOperation().isConstructorInvocation() &&
            result.getOperation().getConstructor().getName().equals
                                        (object.getClass().getName()) ){
            try{ // Cria/restaura a raiz de persistência “categorias”.
                doPersistence(“categorias”, SCI.class.getDeclaredField(“categorias”).get(object));
            } catch(Exception e){}
            try{ // Cria/restaura a raiz de persistência “fornecedores”.
                doPersistence(“fornecedores”, SCI.class.getDeclaredField(“fornecedores”).get(object));
            } catch(Exception e){}
        }
        return Result.noResult; // Retorna um resultado vazio, indicando que
    } // o resultado da operação não será modificado.

    // Cria/restaura as raízes de persistência.
    private void doPersistence(String field, Object object){
        // Cria o meta-objeto de persistência.
        MetaPersistent mp = new MetaPersistent();
        // Verifica se field já é uma raiz de persistência.
        // O método instance retorna uma referência para o servidor de nomes.
        if( NameServer.instance().query(field) )
            // Restaura o estado persistente da raiz,
            // escrevendo o objeto restaurado no atributo “field”.
            try{ // factory é usado para criar operações na aplicação de nível base.
                factory.write( SCI.class.getDeclaredField(field),
                              (List)(mp.restore(field, new SCI_LocationFactory())) ).perform();
            } catch(Exception e){}
        else // O objeto ainda não é persistente. Cria uma nova raiz de persistência.
            mp.persistent( field, object, new SCI_LocationFactory() );
    }
} //class MO_SCI
```

Classe A.2 MO_Comunicador

```
public class MO_Comunicador extends MetaObject{
    ...
    // Este método trata as operações (operation) interceptadas
    // no objeto de nível base (object).
    public Result handle(Operation operation, Object object){
        // Verifica se a operação é uma invocação do construtor da classe.
        if( operation.isConstructorInvocation() &&
        if( operation.isConstructorInvocation() &&
            operation.getConstructor().getName().equals(object.getClass().getName()) ){
            return Result.inspectResult; // Requisita inspeção do resultado da operação.
        else return Result.noResult; // Retorna um resultado vazio,
    } // indicando que a operação não será tratada.

    // Este método trata o resultado (result) das operações invocadas
    // no objeto de nível base (object).
    public Result handle(Result result, Object object){
        // Verifica se o resultado interceptado é o resultado do construtor da classe.
        if( result.getOperation().isConstructorInvocation() &&
            result.getOperation().getConstructor().getName().equals
                                                    (object.getClass().getName()) ){
            // O primeiro parâmetro do construtor da classe Comunicador é o fornecedor que
            // será contactado. Obtém o fornecedor que está sendo contactado.
            final Fornecedor f = (Fornecedor)(result.getOperation().getArguments()[0]);
            try{ // Cria/restaura a raiz de persistência "produtos".
              doPersistence("produtos",Comunicador.class.getDeclaredField("produtos").get(object),f);
            } catch(Exception e){}
            try{ // Cria/restaura a raiz de persistência "pedidos".
              doPersistence("pedidos", Comunicador.class.getDeclaredField("pedidos").get(object),f);
            } catch(Exception e){}
        }
        return Result.noResult; // Retorna um resultado vazio, indicando que
    } // o resultado da operação não será modificado.

    // Cria/restaura as raízes de persistência.
    private void doPersistence(String field, Object object, Fornecedor f){
        // Cria o meta-objeto de persistência.
        MetaPersistent mp = new MetaPersistent();
        // Verifica se field já é mesmo uma raiz de persistência.
        // O método instance retorna uma referência para o servidor de nomes.
        if( NameServer.instance().query(field) )
            // Restaura o estado persistente da raiz,
            // escrevendo o objeto restaurado no atributo "field".
            try{ // factory usado para criar operações na aplicação de nível base.
              factory.write( Comunicador.class.getDeclaredField(field),
                            (List)(mp.restore(field, new Fornec_LocationFactory(f))) ).perform();
            } catch(Exception e){}
        else // O objeto ainda não é persistente. Cria uma nova raiz de persistência.
            mp.persistent( field, object, new Fornec_LocationFactory(f) );
    }
} //class MO_Comunicador
```

Bibliografia

[1] A. Albano, G. Ghelli, and R. Orsini. The Implementation of Galileo's Persistent Values. In M. P. Atkinson, P. Buneman, and R. Morrison, editors, *Data Types and Persistence*, pages 253–263. Springer-Verlag, 1988.

[2] K. Arnold and J. Gosling. *The Java Programing Language*. Addison-Weslley, 1996.

[3] M. P. Atkinson, P. J. Bailey, K. J. Chisholm, W. P. Cockshott, and R. Morrison. An Approach to Persistent Programming. *Computer Journal*, 26(4):360–365, dezembro de 1983.

[4] M. P. Atkinson, K. J. Chisholm, and W. P. Cockshott. PS-Algol: an Algol with a Persistent Heap. Technical Report CSR-94-81, University of Edinburgh, Department of Computer Science, dezembro de 1981.

[5] M. P. Atkinson, L. Daynès, M. J. Jordan, T. Printezis, and S. Spence. An Orthogonally Persistent Java. *ACM Sigmod Record*, 25(4), dezembro de 1996.

[6] G. Booch, J. Rambaugh, and I. Jacobson. *Unified Modeling Language*. Rational Software Corporation, janeiro de 1997. version 1.0.

[7] M. L. Brodie and M. Stonebraker. *Migrating Legacy Systems: Gateways, Interfaces & the Incremental Approach*. Morgan Kaufmann Publishers, Inc., 1995.

[8] S. Chiba. A Meta-Object Protocol for C++. In *OOPSLA'95*, pages 285–299. ACM SIGPLAN Notes, 1995.

[9] S. Chiba and T. Masuda. Designing and Extensible Distributed Language with a Meta-Level Architecture. In *ECOOP'93*, pages 482–501. Lecture Notes in Computer Science, 1993.

[10] G. N. Dixon, G. D. Parrington, S. K. Shrivastava, and S. M. Wheater. The Treatment of Persistent Object in Arjuna. Computing Laboratory, University of Newcastle upon Tyne, Newcastle upon Tyne, UK.

[11] M. P. Atkinson et al. Object-Oriented Database System Manifesto. In *First International Workshop on Deductive and Object-Oriented Databases*, Kyoto, Japan, dezembro de 1990. North-Holland.

[12] J. C. Fabre, V. Nicomette, T. Perennou, and Z. Wu. Implementing Fault Tolerant Application Using Reflective Object-Oriented Programming. In *25th Symposium on Fault Tolerant Computing Systems*, pages 291–311, junho de 1995.

[13] D. G. Firesmith. Frameworks: The Golden Path to Object Nirvana. *JOOP–Journal of Object-Oriented Programming*, outubro de 1993.

[14] M. Golm. Design and Implementation of a meta architecture for Java. Master's thesis, Institut für Mathematische Maschinen und Datenverarbeitung, Friedrich-Alexander-Universität Erlangen-Nürnberg, 1997.

[15] K. E. Gorlen, S. M. Orlow, and P. Plexico. *Data Abstraction and Object-Oriented Programming in C++*. John Wiley and Sons, 1989.

[16] G. Hamilton and R. Cattell. *JDBC: A Java SQL API.* Sun Microsystems Computer Corporation, janeiro de 1997. versão 1.20.

[17] E. N. Hanson, T. M. Harvet, and M. A. Roth. Experiences in Database System Implementation Using a Persistent Programming Language. *Software-Practice and Experience*, 23(12):1361–1377, dezembro de 1993.

[18] Hughes Technologies. *MiniSQL 2.0: User Guide*, julho de 1997. versião 2.0.

[19] R. E. Johnson and B. Foote. Designing Reusable Classes. *JOOP–Journal of Object-Oriented Programming*, 1(2):22–35, junho/julho de 1988.

[20] G. Kiczales. Towards a New Model of Abstraction in Software Enginnering. In *IMSA' 92 Workshop on Reflection and Meta-Level Architectures*, 1992.

[21] J. Kleinnőder and M. Golm. MetaJava: An Efficient Run-Time Meta Architecture for Java. In Washington Seatle, editor, *IWOOS'96–International Workshop on Object Orientation in Operating Systems.* IEEE, outubro de 1996.

[22] J. Kleinnőder and M. Golm. Transparent and Adaptable Object Replication using a Reflective Java. Technical Report TR-14-96-07, Computer Science Department, Friedrich-Alexander-University, Erlangen-Numberg, Germany, setembro de 1996.

[23] A. H. Lee and H. Shin. Building a Persistent Object Store using the Java Reflection API. In *Workshop on Reflective Programming in C++ and Java*, outubro de 1998.

[24] A. H. Lee and J. L. Zachary. Reflection on Metaprogramming. *IEEE Transactions on Software Engineering*, 21(11):883–893, novembro de 1995.

[25] M. L. B. Lisboa, C. M. F. Rubira, and L. E. Buzato. Arquitetura Reflexiva para o Desenvolvimento de Software Tolerante a Falhas. In *XVI Seminário da Sociedade Brasileira de Computação*, pages 155–166, 1996.

[26] P. Maes. Concepts and Experiments in Computational Reflection. In *OOPSLA'87*, pages 147–155. ACM SIGPLAN Notices, outubro de 1997.

[27] R. Morrison, A. L. Brown, R. C. H. Connor, and A. Dearle. Napier88 Reference Manual. Technical Report PPRR-77-89, University of St. Andrews, 1989.

[28] P. O'Brien. Java Data Management Using ObjectStore and PSE. An Object Design White Paper, novembro de 1996.

[29] P. O'Brien. Making Java Objects Persistent. SIGS Publication, Inc. New York, NY, USA, 1997.

[30] P. O'Brien, D. Halbert, and M. Kilian. The Trellis Programming Environment. In *OOPSLA'87*, Orlando, Flórida, USA, outubro de 1987.

[31] A. Oliva. Guaraná: Uma Arquitetura de Software para Reflexão Computacional Implementada em Java. Master's thesis, Instituto de Computação, Universidade Estadual de Campinas, setembro de 1998.

[32] A. Oliva and L. E. Buzato. An Overview of MOLDS: a Meta-Object Library for Distributed Systems. In *WoSID'98–Segundo Workshop em Sistemas Distribuídos*, junho de 1998.

[33] A. Oliva, L. E. Buzato, and I. C. Garcia. The Reflective Architecture of Guaraná. Technical Report IC-98-14, Instituto de Computação, Universidade Estadual de Campinas, abril de 1998.

[34] H. E. Orfali. *The Essential Distributed Objects Survival Guide.* Wiley, 1996.

[35] A. Paepcke. PCLOS: A Flexible Implementation of CLOS Persistence. In *ECOOP'88*, pages 374–389, agosto de 1988.

[36] T. Printezis, M. Atkinson, L. Daynès, S. Spence, and P. Bailey. The Design of a New Persistent Object Store for PJama. In *PJW2–The Second International Workshop on Persistence and Java*, San Francisco, California, agosto de 1997.

[37] J. E. Richardson and M. J. Carey. Persistence in the E Language. *Software-Practice and Experience*, 19(2):1115–1150, dezembro de 1988.

[38] J. E. Richardson and M. J. Carey. Implementing Persistence in E. In *Workshop on Persistent Object Systems*, pages 302–319, Newcastle, Australia, janeiro de 1989.

[39] C. S. Santos and E. Theroude. Persistent Java. In *PJW1–First International Workshop on Persistence and Java*, Glasgow, Scotland, setembro de 1996.

[40] N. Saraiya and B. Joy. *Java Core Reflection - API and Specification*. Sun Microsystems Computer Corporation, Mountain View, CA, USA, fevereiro de 1997.

[41] B. C. Smith. Prologue to "Reflection and Semantics in a Procedural Languge". PhD Thesis Prologue, 1985.

[42] R. J. Stroud. Transparency and Reflection in Distributed Systems. *ACM Operating Systems Review*, 22(2):99–103, abril de 1993.

[43] R. J. Stroud and Z. Wu. Using Meta-Objects to Adapt a Persistent Object System to Meet Applications Needs. In *6th SIGOPS European Conference Workshop on Matching Operating Systems to Applications Needs*, 1994.

[44] Sun Microsystems Computer Corporation, Moutain View, CA, USA. *Java API Documentation*, dezembro de 1996. versão 1.1.

[45] R. J. Wirfs-Brock and R. E. Johnson. Surveying Current Research in Object-Oriented Design. *Communications of the ACM*, 33(9), setembro de 1990.

[46] F. Xhumari, C. S. dos Santos, and M. Skubiszewski. Java Universal Binding: Storing Java Objects in Relational and Object-Oriented Databases. In *PJW2–The Second International Workshop on Persistence and Java*, San Francisco, California, agosto de 1997.

[47] J. Xu, B. Randell, and A. F. Zorzo. Implementing Software-Fault Tolerance in C++ and Open C++: An Object-Oriented and Reflective Approch. Technical report, Department of Computing Science, University of Newcastle upon Tyne, Newcastle upon Tyne, UK, 1997.

[48] Y. Yokote. The Apertos Reflective Operating System: The Concept and Its Implementation. In *OOPSLA'92*, volume 27, pages 414–434. ACM SIGPLAN Notes, outubro de 1992.