Universidade Federal de Itajubá
Programa de Pós-Graduação em
Ciência e Tecnologia da Computaçao

Metaprogramação e Metadados de Persistência e Indexação para o *Framework Object-Injection*

Maurício Faria de Oliveira

Dezembro de 2012
Itajubá - MG

UNIVERSIDADE FEDERAL DE ITAJUBÁ
PROGRAMA DE PÓS-GRADUAÇÃO EM
CIÊNCIA E TECNOLOGIA DA COMPUTAÇAO

Maurício Faria de Oliveira

Metaprogramação e Metadados de Persistência e Indexação para o *Framework Object-Injection* [1]

Dissertação submetida ao Programa de Pós-Graduação em Ciência e Tecnologia da Computação como parte dos requisitos para obtenção do Título de Mestre em Ciências em Ciência e Tecnologia da Computação.

Área de Concentração: Sistemas de Computação

Orientador: Prof. Dr. Enzo Seraphim

Dezembro de 2012
Itajubá - MG

[1]Este trabalho conta com apoio financeiro da CAPES.

Ficha catalográfica elaborada pela Biblioteca Mauá –
Bibliotecária Margareth Ribeiro- CRB_6/1700

O48m

Oliveira, Maurício Faria
Metaprogramação e Metadados de Persistência e Indexação pa_
ra o Framework Object-Injection / Maurício Faria de Oliveira. --
Itajubá, (MG) : [s.n.], 2012.
93 p. : il.

Orientador: Prof. Dr. Enzo Seraphim.
Dissertação (Mestrado) – Universidade Federal de Itajubá.

1. Metaprogramação. 2. Framework. 3. Persistência. 4. Progra_
mação Orientada a Objetos. 5. Estruturas de Dados. I. Seraphim,
Enzo, orient. II. Universidade Federal de Itajubá. III. Título.

Dedicado a meus pais,
amorosa base.

Agradecimentos

Deus, por instâncias.

Meus pais, por herança de características e comportamento.

Meus irmãos, por implementação de interfaces.

Meus avós, por proteção de acesso.

Enzo Seraphim, por *educação* e exemplo.

Thatyana Seraphim, por depuração.

Edmilson Marmo, por coerência e consistência.

Gilson Carvalho, por ensinamento.

Arsinoé Wilche, por persistência.

Antonio Alves, por ponteiros.

Gustavo Walbon, por comunicação.

UNIFEI e IBM, por flexibilidade.

CAPES, por apoio.

Sumário

Resumo **ix**

Abstract **x**

1 Introdução **1**

1.1 Motivação 4

1.2 Objetivo 5

1.3 Organização 5

2 Revisão Bibliográfica **8**

2.1 Java 8

2.2 Persistência de Objetos 11

2.2.1 Persistência de Objetos em Java 13

2.3 Metaprogramação 15

2.3.1 Classificação de Metaprogramas 16

2.3.2 Metaprogramação em Java 18

3 *Framework Object-Injection* **31**

3.1 Módulos 32

3.1.1 Módulo Metaclasses 33

3.1.2 Módulo Armazenamento 36

3.1.3 Módulo Blocos 37

3.1.4 Módulo Dispositivos 39

3.2 Considerações Finais 40

4 Metaprogramação e Metadados para o *Framework Object-Injection* **41**

4.1 Metadados 42

4.2 Análise e Geração de Código 44
4.3 Classes de Vínculo 48
4.4 Primeiro Caso de Uso 48
4.5 Segundo Caso de Uso 53
4.5.1 Cenário 53
4.5.2 Diagrama de Classe UML 56
4.6 Considerações Finais 56

5 Conclusões 58

5.1 Trabalhos Futuros 59

A Segundo Caso de Uso 65

A.1 Pessoa 65
A.1.1 Classe `Pessoa` 65
A.1.2 Classe `EntityPessoa` 67
A.1.3 Classe `PrimaryKeyPessoa` 70
A.2 Sonho 71
A.2.1 Classe `Sonho` 71
A.2.2 Classe `EntitySonho` 75
A.2.3 Classe `PrimaryKeySonho` 79

Lista de Figuras

2.1 Java: Linguagem de Programação, Plataforma, Plataforma Subjacente 9
2.2 Java: Linguagem de Programação, Plataforma, Compilador 9
2.3 Java: Edição Java SE 11
2.4 Metaprogramação: Metaprogramas e Programas-Objeto 16
2.5 Metaprogramação: Reflexão 17
2.6 Instrumentação: Definição de Classe 22
2.7 Instrumentação: Transformação de Classe 23
2.8 Instrumentação: Redefinição de Classe 23
2.9 Instrumentação: Retransformação de Classe 24
2.10 Processador de Anotações: Acoplamento com Compilador 29

3.1 *Framework Object-Injection*: Módulos 33
3.2 *Framework Object-Injection*: Pacote Metaclasses 34
3.3 *Framework Object-Injection*: Pacote Armazenamento 36
3.4 *Framework Object-Injection*: Pacote Blocos 37
3.5 *Framework Object-Injection*: Pacote Dispositivos 39

4.1 Metodologia: Metadados, Analisador e Gerador de Código, Interfaces 41
4.2 Metodologia: Pacote Metaprogramação 42
4.3 Metodologia: Pacote Raiz 43
4.4 Metodologia: Mecanismos de Metaprogramação e Classes Auxiliares 47
4.5 Metodologia: Primeiro Caso de Uso 50
4.6 Metodologia: Segundo Caso de Uso 56

Lista de Listagens

1.1 Exemplo de Código Clichê: Método de Comparação de Atributos 3
1.2 Exemplo de Código Clichê: Método de Serialização de Atributos 6
1.3 Exemplo de Código Clichê: Método de Desserialização de Atributos 7
2.1 Tipos Genéricos: Declarações de classe, construtor e método com parâmetros de tipo 18
2.2 Tipos Genéricos: Uso de parâmetros de tipo em classe, construtor e método . . 19
2.3 Reflexão: Atributos (Campos), Métodos e Construtores de um Objeto 21
2.4 Instrumentação: Assinatura do Método `ClassFileTransformer.transform` . 25
2.5 Anotações: Anotação `@Override` 26
2.6 Anotações: Tipos-anotação com valor padrão, elemento `value` e marcador . . . 27
2.7 Anotações: Anotações com valor padrão, elemento `value` e marcador 27
2.8 Processadores de Anotações: Acoplamento ao compilador Java 28
4.1 Exemplo de Anotação `@Order`: Classe `Aluno` 45
4.2 Exemplo de Anotação `@StringMetric`: Classe `Proteina` 45
4.3 Exemplo de Anotação `@DoubleMatrixMetric`: Classe `Imagem` 45
4.4 Exemplo de Anotação `@PointMetric`: Classe `Imagem` 46
4.5 Primeiro Caso de Uso: Classe `MeuExemplo` 49
4.6 Primeiro Caso de Uso: Classe `EntityMeuExemplo` 51
4.7 Primeiro Caso de Uso: Classe `PrimaryKeyMeuExemplo` 52
4.8 Segundo Caso de Uso: Aplicação de teste 54
A.1 Segundo Caso de Uso: Classe `Pessoa` 65
A.2 Segundo Caso de Uso: Classe `EntityPessoa` 67
A.3 Segundo Caso de Uso: Classe `PrimaryKeyPessoa` 70
A.4 Segundo Caso de Uso: Classe `Sonho` 71
A.5 Segundo Caso de Uso: Classe `EntitySonho` 75

A.6 Segundo Caso de Uso: Classe `PrimaryKeySonho` 79

Lista de Abreviaturas

API *Application Programming Interface* – Interface de Programação de Aplicações.

CAD *Computer-Aided Design* – Projeto Auxiliado por Computador.

CAM *Computer-Aided Manufacturing* – Fabricação Auxiliada por Computador.

CRTP *Curiously Recurring Template Pattern* – Padrão com Gabarito (*Template*) Curiosamente Recursivo.

DRAM *Dynamic Random-Access Memory* – Memória de Acesso Aleátorio Dinâmica.

EJB *Enterprise JavaBeans* – JavaBeans Empresarial.

ISO *International Organization for Standardization* – Organização Internacional para Padronização.

Java EE *Java Platform, Enterprise Edition* – Edição Empresarial (Distribuída) da Plataforma Java.

Java FX *Java "Effects"* – Plataforma para multimídia sobre a Plataforma Java.

Java ME *Java Platform, Micro Edition* – Edição Micro (Móvel) da Plataforma Java.

Java SE *Java Platform, Standard Edition* – Edição Padrão da Plataforma Java.

JDK *Java Development Kit* – Kit de Desenvolvimento Java.

JDO *Java Data Objects* – Objetos de Dados Java.

JNI *Java Native Interface* – Interface Nativa para Java.

JPA *Java Persistence API* – API de Persistência Java.

JRE *Java Runtime Environment* – Ambiente de Execução Java.

JVM *Java Virtual Machine* – Máquina Virtual Java.

JVM TI *Java Virtual Machine Tool Interface* – Interface de Ferramentas para Máquina Virtual Java.

MOBD *Mapeamento Objeto-Banco de Dados*.

MOR *Mapeamento Objeto-Relacional*.

NPI *Native Programming Interface* – Interface de Programação Nativa.

ODBMS *Object Database Management System* – Sistema de Gerenciamento de Bancos de Dados de Objetos (SGBDO).

ODL *Object Definition Language* – Linguagem de Definição de Objetos.

ODM *Object-to-Database Mapping* – Mapeamento Objeto-Banco de dados (MOBD).

ODMG *Object Data Management Group* – Grupo de Gerenciamento de Dados de Objetos.

ODMS *Object Data Management System* – Sistema de Gerenciamento de Dados de Objetos (SGDO).

OID *Object Identifier* – Identificador de Objetos.

OIF *Object Interchange Format* – Formato de Intercâmbio de Objetos.

OQL *Object Query Language* – Linguagem de Consulta de Objetos.

ORM *Object-Relational Mapping* – Mapeamento Objeto-Relacional (MOR).

RDBMS *Relational Database Management System* – Sistema de Gerenciamento de Bancos de Dados de Relacional (SGBDR).

RMI *Remote Method Invocation* – Invocação Remota de Métodos.

RTTI *Run-Time Type Information* – Informação de Tipos em Tempo de Execução

SGBD *Sistema de Gerenciamento de Bancos de Dados*.

SGBDO *Sistema de Gerenciamento de Bancos de Dados de Objetos*.

SGBDR *Sistema de Gerenciamento de Banco de Dados Relacional*.

SGDO *Sistema de Gerenciamento de Dados de Objetos*.

UML *Unified Modeling Language* – Linguagem Unificada de Modelagem.

UUID *Universally Unique Identifier* – Identificador Universalmente Único.

XML *Extensible Markup Language* – Linguagem de Marcação Extensível.

Resumo

Este trabalho desenvolve uma metodologia fundamentada em metaprogramação para automatizar a implementação de interfaces de persistência e indexação do *framework Object-Injection* em classes de aplicações. Através da análise de metadados em classes de aplicações, são geradas classes auxiliares, compatíveis em interface e estado, contendo implementação de interfaces e autogerenciamento de persistência e indexação, de forma transparente e minimamente intrusiva para a aplicação. São implementados validadores e geradores de classes de tempo de compilação e execução baseados em processadores de anotações (tempo de compilação) e reflexão combinada à compilação (tempo de execução), considerando aspectos de desempenho e flexibilidade. São implementadas, também, classes de vínculo entre aplicação e *framework* simplificadas. Com este trabalho, a interação entre aplicação e *framework* é minimizada à anotações e um gerenciador de entidades, contribuindo para o desenvolvimento de aplicações adotando o *framework Object-Injection*.

Abstract

This study develops a metaprogramming-based methodology for automating the implementation of interfaces for persistence and indexing from the Object-Injection framework for application classes. Through analysis of metadata within application classes, auxiliary classes are generated, both being interface- and type-compatible, implementing interfaces and self-managed persistence and indexing, in a way that is transparent and minimally intrusive for the application. Compile-time and run-time class-validators and class-generators are implemented based on annotation processors (compile time) and reflection combined with compilation (run time), so to address the concerns performance and flexibility. Simplified application-framework link classes are also implemented. Through this study, the application-framework interaction is minimized to using annotations and an entity manager, contributing to the development process of applications adopting the Object-Injection framework.

CAPÍTULO

1 Introdução

Programas de computador, em geral, requerem armazenamento e recuperação de dados. Particularmente, alguns programas de computador requerem armazenamento de dados por um período de tempo superior a uma execução (por exemplo, editores de arquivos), ou necessitam alternar porções de dados em memória (por restrições de recursos ou eficiência de utilização; por exemplo, registros bancários). Esses programas interpõem armazenamento e recuperação de dados durante uma ou mais execuções.

Dados armazenados por um período inferior a uma execução de um programa de computador, descartados com a finalização do processo respectivo, são chamados transientes, ou transitórios. Inversamente, dados armazenados por um período de tempo superior a uma execução, preservados após a finalização do processo, são chamados persistentes, ou duráveis (em relação a armazenamento em meio persistente, como discos, e não volátil, como memória DRAM).

Dados são armazenados e recuperados através de estruturas de dados, em formato e operação definidos para um propósito (por exemplo, percurso, ordenação, dispersão, agrupamento). Estruturas de indexação definem índices – ligações entre dados armazenados – para localizar conjuntos de dados através de respectivos subconjuntos, ou chaves. Estruturas persistentes podem armazenar e recuperar dados em meio persistente. Estruturas persistentes de indexação definem índices para dados armazenados em meio persistente, com propósito de localizar conjuntos de dados em poucos acessos.

Em programação orientada a objetos, dados e operações são encapsulados em objetos. Um objeto é uma instância de uma classe. Uma classe define um modelo de dados com estado e comportamento através de atributos (dados) e métodos (operações), respectivamente. Diversas linguagens de programação desse paradigma contém estruturas de dados para objetos transientes; objetos persistentes, entretanto, envoltos em maior complexidade e discussões

(DARWEN; DATE, 1995), contam com menor suporte nativo em linguagens de programação.

Um *framework* contém definições e, opcionalmente, implementações de funcionalidades (por exemplo, operações e estruturas de dados), possivelmente genéricas, passíveis de implementação, especialização ou substituição. Um *framework* orientado a objetos oferece funcionalidades através de conceitos como classes abstratas, interfaces, sobrecarga de métodos, herança e polimorfismo. Um programa de computador pode utilizar um *framework* para exercer funcionalidades inexistentes em uma linguagem de programação, ou como camada de abstração para certas operações, com adaptações para restrições e necessidades específicas.

O *framework Object-Injection* (CARVALHO et al., 2013) é um *framework* de indexação e persistência de objetos escrito na linguagem de programação Java. Fundamentado em índices primários e secundários (para armazenar objetos e chaves, respectivamente), transfere às classes responsabilidades sobre atributos, como comparação, leitura e escrita em formato sequencial, requeridas pelos índices. Dessa forma, classes de objetos persistentes devem implementar uma interface de persistência com métodos de comparação, serialização e desserialização de atributos para armazenamento em índices primários. E, de forma semelhante, classes de chaves de objetos persistentes devem implementar uma interface de indexação similar sobre atributos-chave para armazenamento em índices secundários e, também, interfaces específicas de alguns desses índices.

Em resumo, um programa de computador pode utilizar o *framework Object-Injection* para armazenar e recuperar objetos, com suporte a persistência e indexação, através da implementação de interfaces com métodos de comparação, serialização e desserialização de atributos.

Métodos de comparação de atributos têm como finalidade verificar ou quantificar uma ou mais condições (normalmente, igualdade ou similaridade) entre atributos de objetos. Métodos de (des) serialização de atributos tem como finalidade (ler) escrever atributos de um objeto sequencialmente (de) para um vetor de bytes. Aliando classes auxiliares, ou ajudantes (*helpers*), de comparação e (des) serialização à sobrecarga de métodos, a implementação desses métodos consiste em repetições uniformes de código em função de cada atributo – padrão de projeto comum em *frameworks* de persistência de objetos (JSR-220, 2006).

Devido à evidente repetição e similaridade de código em métodos de comparação e (des) serialização de atributos em um programa de computador, esse código é caracterizado como *código clichê* (código padronizado, ou código *boilerplate*) (Listagens 1.1, 1.2 e 1.3). Algumas linguagens de programação dispõem de mecanismos que possibilitam reduzir ou eli-

minar esse tipo de código, como mecanismos de metaprogramação.

```
1  public boolean equalToEntity(EntityExemplo obj) {
2
3      // Atributos: campo1, campo2, ..., campo13.
4      return (this.isCampo1() == obj.isCampo1())
5          && (this.getCampo2() == obj.getCampo2())
6          && (this.getCampo3() == obj.getCampo3())
7          && (this.getCampo4().equals(obj.getCampo4()))
8          && (this.getCampo5().equals(obj.getCampo5()))
9          && (this.getCampo6() == obj.getCampo6())
10         && (this.getCampo7() == obj.getCampo7())
11         && (this.getCampo8() == obj.getCampo8())
12         && (this.getCampo9() == obj.getCampo9())
13         && (this.getCampo10() == obj.getCampo10())
14         && (this.getCampo11().equals(obj.getCampo11()))
15         && (this.getCampo12().equals(obj.getCampo12()))
16         && (this.getCampo13().equals(obj.getCampo13()));
17 }
```

Listagem 1.1: Exemplo de Código Clichê: Método de Comparação de Atributos

Metaprogramação permite análise e geração de programas por metaprogramas e transferência de operações de tempo de execução para tempo de compilação, através de uma metalinguagem de programação. Particularmente, uma linguagem de programação, ao ser a própria metaliguagem de programação, permite reflexão, ou reflexividade. Mecanismos de metaprogramação permitem redução de código e aumento de flexibilidade em um programa de computador.

O kit de desenvolvimento Java (particularmente, biblioteca de classes e compilador) dispõe de mecanismos de metaprogramação através da linguagem de programação Java:

Tipos genéricos permitem abstração sobre tipos de dados em tempo de execução, com garantias de tempo de compilação.

Reflexão permite introspecção sobre elementos de um programa de computador e intercessão em invocações de métodos de interfaces, em tempo de execução.

Instrumentação permite modificar classes em tempo de execução através de manipulação de representação binária (*Java bytecode*).

Anotações são metaelementos para anexar metadados a elementos, acessíveis em tempo de execução através de reflexão e em tempo de compilação através de processadores de anotações.

Processadores de anotações permitem introspecção e geração de código em tempo de compilação, acoplados ao compilador.

Compilação em execução permite a invocação de um compilador Java em tempo de execução.

Dentre esses mecanismos, a combinação de anotações com reflexão ou com processadores de anotações é apropriada para automatizar a implementação de interfaces de persistência e indexação do *framework Object-Injection*. Dessa forma, um programa de computador é preservado da escrita e manutenção de código clichê relativo à informações sobre elementos, implícitas em declarações ou explícitas em anotações. Uma vez que essa funcionalidade é fundamentada em mecanismos da linguagem de programação Java, é passível de incorporação ao *framework Object-Injection*, em contribuição à usabilidade e produtividade de desenvolvimento.

1.1 Motivação

A existência de código clichê é um efeito comum em diversos *frameworks* de persistência de objetos, causado pela necessidade de declarações explícitas sobre elementos e ausência de mecanismos para automatizar essas declarações em diversas linguagens de programação.

Normalmente, esse código é trivial e repetitivo, baseado em informações implícitas existentes (por exemplo, declarações de atributos e métodos) e adequado à especificações conhecidas pelo próprio *framework*. Dessa forma, sua escrita e manutenção tornam-se um processo redundante, dispendioso e passível de erros, frequentemente necessário em modificações de código em um programa de computador.

Através de mecanismos de metaprogramação é possível automatizar a geração de código relativo à declarações de elementos, reduzindo ou eliminando porções de código clichê. Consequentemente, são minimizadas a escrita e manutenção de código no desenvolvimento, atenuando a parcela de codificação puramente operacional, sem valor agregado.

O *framework Object-Injection* é um alicerce propício para o desenvolvimento e experimentos com essa abordagem. O *framework* é uma fonte de ocorrências de código clichê (Listagens 1.1, 1.2 e 1.3), o problema em questão, por requerer implementações de interfaces em função de elementos, e, sendo escrito na linguagem de programação Java, permite aplicar mecanismos de metaprogramação do kit de desenvolvimento Java.

Uma solução baseada na abordagem proposta pode ser incorporada como contribuição ao *framework*, beneficiando critérios de desenvolvimento como produtividade e usabilidade do *framework Object-Injection*.

1.2 Objetivo

O objetivo deste trabalho é automatizar a implementação de interfaces de persistência e indexação do *framework Object-Injection* para aplicações.

1.3 Organização

O restante deste trabalho é organizado da seguinte forma: a revisão bibliográfica é apresentada no capítulo 2, o *framework Object-Injection* é descrito no capítulo 3, o desenvolvimento do trabalho e casos de uso são detalhados no capítulo 4, e, finalmente, conclusões, contribuições e sugestões de trabalhos futuros são apresentadas no capítulo 5.

```
public void pushEntity(byte[] array, int position) {

    PushStream push = new PushStream(array, position);

    // Atributos: classId, uuid, campo1, campo2, ..., campo10
    push.pushUuid(classId);
    push.pushUuid(uuid);
    push.pushBoolean(this.isCampo1());
    push.pushByte(this.getCampo2());
    push.pushChar(this.getCampo3());
    push.pushCalendar(this.getCampo4());
    push.pushDate(this.getCampo5());
    push.pushDouble(this.getCampo6());
    push.pushFloat(this.getCampo7());
    push.pushInt(this.getCampo8());
    push.pushLong(this.getCampo9());
    push.pushShort(this.getCampo10());

    // Atributo: campo11
    for (MeuExemplo obj : this.getCampo11())
        if (obj instanceof Entity)
            push.pushEntity((Entity) obj);
        else
            push.pushEntity(new EntityMeuExemplo(obj));

    // Atributo: campo12
    if (this.getCampo12() == null)
        push.pushInt(0);
    else
        push.pushString(this.getCampo12().getBytes());

    // Atributo: campo13
    push.pushMatrix(getCampo13());
}
```

Listagem 1.2: Exemplo de Código Clichê: Método de Serialização de Atributos

```
1  public boolean pullEntity(byte[] array, int position) {
2
3      PullStream pull = new PullStream(array, position);
4
5      // Atributo: classId
6      Uuid storedClass = pull.pullUuid();
7
8      if (this.classId.equals(storedClass) == false)
9          return false;
10
11     // Atributos: uuid, campo1, campo2, ..., campo13
12     this.uuid = pull.pullUuid();
13     this.setCampo1(pull.pullBoolean());
14     this.setCampo2(pull.pullByte());
15     this.setCampo3(pull.pullChar());
16     this.setCampo4(pull.pullCalendar());
17     this.setCampo5(pull.pullDate());
18     this.setCampo6(pull.pullDouble());
19     this.setCampo7(pull.pullFloat());
20     this.setCampo8(pull.pullInt());
21     this.setCampo9(pull.pullLong());
22     this.setCampo10(pull.pullShort());
23     this.setCampo11(pull.pullEntity(this.getEntityStructure()));
24     this.setCampo12(pull.pullString());
25     this.setCampo13((double[][]) pull.pullMatrix());
26
27     return true;
28 }
```

Listagem 1.3: Exemplo de Código Clichê: Método de Desserialização de Atributos

Capítulo 2
Revisão Bibliográfica

Neste capítulo são apresentados tópicos relacionados à plataforma e linguagem de programação Java, persistência de objetos e metaprogramação.

2.1 Java

O termo Java pode referir-se à linguagem de programação Java ou à plataforma de computação [1] Java, originalmente desenvolvidas pela *Sun Microsystems, Inc.* em 1995, incorporada à *Oracle Corporation* em 2010.

A linguagem de programação Java é uma linguagem de programação de propósito geral, concorrente, baseada em classes, orientada a objetos, com sintaxe derivada das linguagens de programação C e C++ (GOSLING et al., 2012).

A plataforma de computação Java é uma plataforma de *software* executável sobre diversas plataformas de *software/hardware* (multiplataforma), consistindo de Java API e Máquina Virtual Java (Figura 2.1) (CAMPIONE; WALRATH, 2012; LINDHOLM et al., 2012).

A Java API (*Application Programming Interface*, Interface de Programação de Aplicações) é um conjunto de declarações e definições de funcionalidades disponível através de bibliotecas de interfaces e classes para a linguagem de programação Java (CAMPIONE; WALRATH, 2012).

A Máquina Virtual Java (*Java Virtual Machine*, JVM) (LINDHOLM et al., 2012) é uma especificação de máquina abstrata, com realizações (implementações) específicas para diversas plataformas de *software* e *hardware* – efetivamente uma camada de abstração sobre a plata-

[1] Uma plataforma de computação (ou plataforma) é um ambiente de *software* e/ou *hardware* para execução de programas de computador, normalmente descrita como combinação de programas, bibliotecas, sistema operacional (*software*) e arquitetura de computador (*hardware*) (CAMPIONE; WALRATH, 2012)

forma subjacente e detalhes de implementação (Figura 2.1) (CAMPIONE; WALRATH, 2012).

Programas escritos na linguagem de programação Java para a plataforma de computação Java adotam diversas funcionalidades da Java API, e são transformados por um compilador Java em *arquivos-classe* com formato e instruções executáveis sobre a Máquina Virtual Java (GOSLING et al., 2012; LINDHOLM et al., 2012) (Figura 2.2).

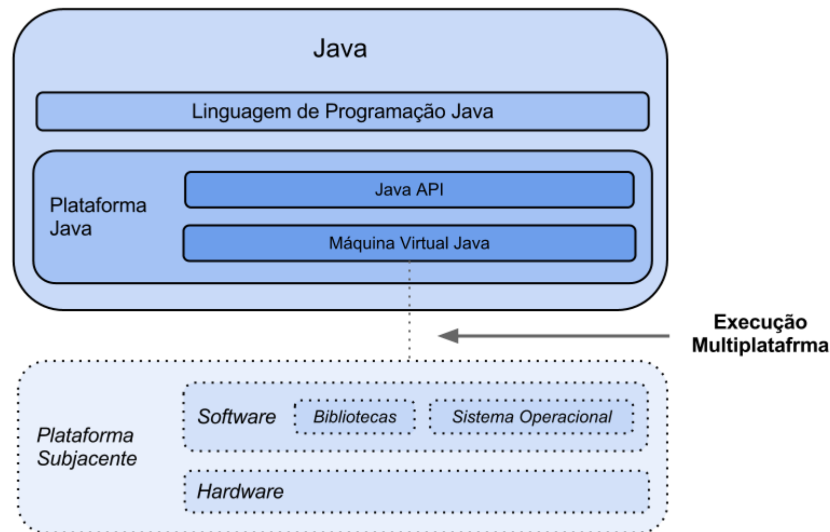

Figura 2.1: Java: Linguagem de Programação, Plataforma, Plataforma Subjacente

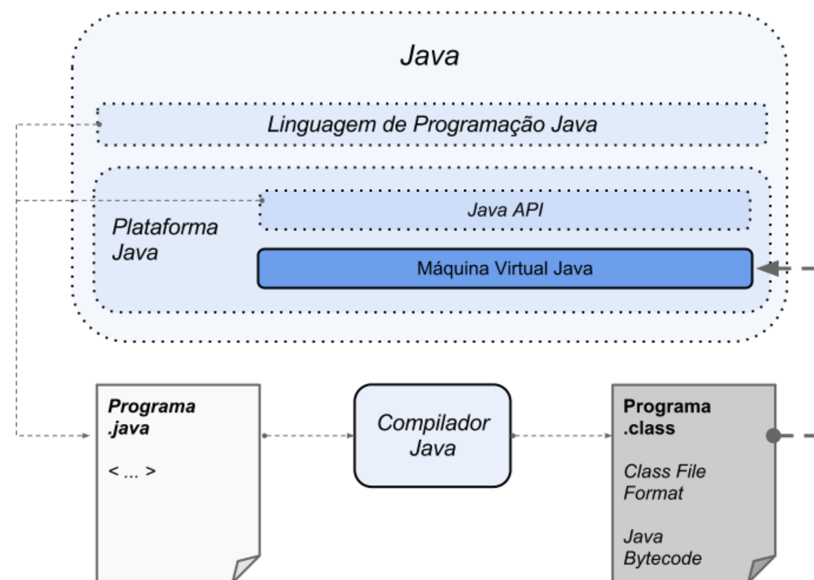

Figura 2.2: Java: Linguagem de Programação, Plataforma, Compilador

Para a Máquina Virtual Java, um *arquivo-classe* é uma representação binária no formato *Class File Format* (formato de arquivo-classe), descrevendo modelo de dados (atributos), blocos de instruções (métodos), e tipos (herança de classes e interfaces) (LINDHOLM et al., 2012).

O conjunto de instruções da Máquina Virtual Java é conhecido como *Java bytecode* (LINDHOLM et al., 2012). O formato das instruções são operações (*opcodes*) de 1 *byte* e, opcionalmente, parâmetros (*operandos*) de 1 ou mais *bytes*, resultando em instruções *multibyte*. Exemplos de instruções incluem instruções de leitura/escrita, aritmética, transferência de controle (testes, desvios), manipulação de pilhas e operandos, sincronização, conversão de tipos, criação e manipulação de objetos, invocação e retorno de métodos (LINDHOLM et al., 2012).

Outras linguagens de programação e compiladores também são empregados na geração de *Java bytecode* para execução na Máquina Virtual Java, com proveito de infraestrutura disponível e execução multiplataforma, como *Jython*[2] , *JRuby*[3], *Groovy*[4] e *Scala*[5].

Em função de áreas de mercado, diferentes *edições* da plataforma de computação Java estão em vigor, adequadas a nichos específicos, com diferentes recursos incorporados à Java API e a Máquina Virtual Java (EVANS, 2012):

- Java SE (*Java Platform, Standard Edition*): conjunto básico de funcionalidades para a linguagem de programação Java e execução de aplicações sobre a plataforma de computação Java (GOSLING et al., 2012; LINDHOLM et al., 2012).

- Java EE (*Java Platform, Enterprise Edition*), extensão de Java SE para aplicações distribuídas (JSR-316, 2009; EVANS, 2012).

- Java ME (*Java Platform, Micro Edition*): redução de Java SE para aplicações sobre dispositivos com menos recursos, com um subconjunto da Java API de Java SE e extensões específicas para esses dispositivos, e uma máquina virtual diferenciada.

- Java FX (em inglês, foneticamente similar à palavra *effects* – efeitos): plataforma para aplicações multimídia com suporte a aceleração por *hardware*.

Detalhando a edição Java SE (Figura 2.3 (ORACLE, 2012a)), em sua composição são encontrados 2 conjuntos de funcionalidades: ambiente de execução Java (*Java Runtime Environment*, JRE) – provendo Máquina Virtual Java, Java API e outros componentes para execução sobre a plataforma Java – e kit de desenvolvimento Java (*Java Development Kit*, JDK) – um superconjunto de JRE, com ferramentas para o desenvolvimento de aplicações Java, como compiladores e depuradores (*debuggers*).

[2] `http://jython.org`

[3] `http://jruby.org`

[4] `http://groovy.codehaus.org`

[5] `http://scala-lang.org`

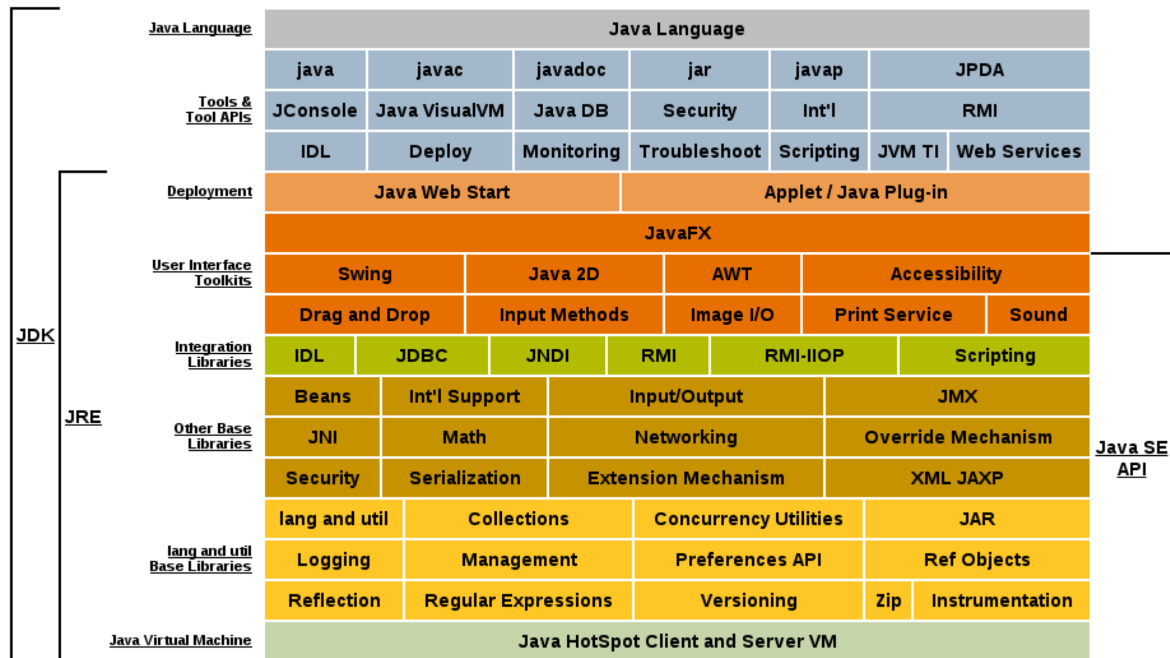

Figura 2.3: Java: Edição Java SE

2.2 Persistência de Objetos

Persistência é o armazenamento de dados para recuperação posterior (HARRINGTON, 2000), normalmente em meio de armazenamento persistente, ou não-volátil. Um objeto é classificado transiente ou persistente e gerenciado em memória de acordo com seu tempo de vida (*lifetime*) (BERLER et al., 2000).

Um objeto transiente é gerenciado em memória principal pelo ambiente de execução (linguagem de programação, bibliotecas, sistema operacional). Por exemplo, objetos de um método (parâmetros, variáveis locais) são alocados em um quadro do segmento de pilha (*stack frame*) em uma invocação e desalocados no retorno do método, e objetos estáticos de um programa (variáveis globais) são alocados no segmento de dados ou no segmento de *heap* e desalocados na finalização do programa. Esses objetos têm tempo de vida limitado a, respectivamente, uma invocação de método e uma execução do programa.

Um objeto persistente é gerenciado tanto em memória principal quanto em memória secundária (meio de armazenamento persistente). O ambiente de execução gerencia a transiência intrínseca a objetos (alocação e desalocação de memória) e um Sistema de Gerenciamento de Dados de Objetos (SGDO, ou *Object Data Management System*, ODMS) (BERLER et al., 2000) gerencia a persistência do objeto (armazenamento e recuperação), a partir de sua inicialização, referência ou atualização até sua desreferenciação ou eliminação.

Dependendo do formato de armazenamento de objetos, um Sistema de Gerenciamento de Dados de Objetos (SGDO) é classificado como Sistema de Gerenciamento de Bancos de Dados de Objetos (SGBDO, ou *Object Database Management System*, ODBMS), com armazenamento nativo de objetos, ou Mapeamento Objeto-Banco de Dados (MOBD, *Object-to-Database Mapping*, ODM), com armazenamento de objetos em um banco de dados não-objeto, como um Sistema de Gerenciamento de Banco de Dados Relacional (SGBDR, ou *Relational Database Management System*, RDBMS).

Enquanto SGBDRs demonstram-se especialmente apropriados para domínios de aplicações de negócios (DATE, 2003), outros domínios não são capturados de forma transparente ou eficiente; por exemplo, objetos.

A representação de conceitos de programação orientada a objetos através do modelo relacional apresenta empecilhos, devido à diferença entre modelos de dados dos paradigmas, descritos como incompatibilidade (ou ineficiência) objeto-relacional (*object-relational impedance mismatch*). Um processo de conversão entre representações, conhecido como mapeamento objeto-relacional (MOR; *object-relational mapping*, ORM), é necessário para representar conceitos como identificadores, tipos de dados, referências, herança, polimorfismo, generalidade e encapsulamento, através de conceitos como tabelas, colunas e linhas.

Enquanto MOBDs têm ampla adoção comercial, em correspondência à presença de SGBDRs, SGBDOs estão presentes em nichos específicos, normalmente compostos por aplicações com requisitos de modelagem de dados ou operações não atendidos por SGBDRs; por exemplo, CAD (*Computer-Aided Design*, Design Auxiliado por Computador), CAM (*Computer-Aided Manufacturing*, Fabricação Auxiliada por Computador), multimídia e Sistemas de Informação Geográfica. Exemplos de SGBDOs incluem *VelocityDB*[6], *InterSystems Caché*[7], *Versant db4o*[8] e *Eloquera DB*[9]. Geralmente, SGBDOs são transparentemente integrados à aplicação devido à compatibilidade de modelos de dados (objetos), dispensando conversão de representação.

Objetivando o desenvolvimento de aplicações portáveis entre diferentes SGDOs, o grupo ODMG (*Object Data Management Group*, Grupo de Gerenciamento de Dados de Objetos) desenvolveu um conjunto de especificações para persistência de objetos (BERLER et al., 2000). Nesse domínio, a portabilidade de aplicações envolve os seguintes fatores: esquema de dados

[6] `http://velocitydb.com`

[7] `http://intersystems.com/cache`

[8] `http://db4o.com`

[9] `http://eloquera.com/eloquera-db`

(*data schema*), ligações (*bindings*) com linguagens de programação, e linguagens de manipulação e consulta a dados.

A especificação ODMG 3.0, endereçando tais fatores, tem como principais componentes (BERLER et al., 2000):

- Modelo de objetos;
- Linguagens de especificação de objetos:
  - Linguagem de Definição de Objetos (*Object Definition Language*, ODL);
  - Formato de Intercâmbio de Objetos (*Object Interchange Format*, OIF);
- Linguagem de Consulta de Objetos (*Object Query Language*, OQL);
- Ligações para linguagens de programação C++, Smalltalk e Java.

Os direitos da especificação ODMG 3.0 sobre as ligações para a linguagem de programação Java foram cedidos para a especificação de persistência *Java Data Objects* 1.0 (JSR-12, 2003), de forma que ocorresse a aplicação, disseminação e desenvolvimento da especificação.

2.2.1 Persistência de Objetos em Java

Com a evolução da plataforma e linguagem de programação Java, diversas especificações e ferramentas de persistência foram desenvolvidas e aperfeiçoadas, direcionadas à determinadas tecnologias e edições da plataforma Java; por exemplo, as especificações *Java Data Objects*, *Enterprise JavaBeans*, *Java Persistence API* e os *frameworks Hibernate* e *TopLink*.

Java Data Objects

A especificação *Java Data Objects* (JDO) permite desacoplar o domínio da aplicação e os mecanismos de persistência através de arquivos XML e *enhancers* (ferramentas disponibilizadas por implementações, ou *produtos*, JDO).

Com JDO, arquivos XML armazenam definições de persistência de objetos, interpretadas por *enhancers*, que então realizam a inserção da funcionalidade de persistência em arquivos-classe Java. Dessa forma, a persistência de objetos ocorre transparentemente, isentando classes de aplicação de implementar interfaces e estender superclasses de mecanismos de persistência.

A especificação JDO 2.0 (JSR-243, 2006) constitui uma extensão à especificação original, com aperfeiçoamento de usabilidade, alinhamento com a edição Java EE e a padronização do suporte a banco de dados. Produtos JDO armazenam objetos tanto em SGBDOs quanto em MOBDs.

Enterprise JavaBeans

A especificação *Enterprise JavaBeans* (EJB; *JavaBeans* Empresarial), inicialmente desenvolvida pela *IBM Corporation* e posteriormente adotada na edição Java EE, baseada em componentes reusáveis (*JavaBeans*, essencialmente classes), define uma arquitetura para desenvolvimento e implantação (*deployment*) de aplicações distribuídas e orientadas a objetos contendo lógica de negócios (*business logic*).

Em síntese, são definidos contratos para um servidor de aplicações prover suporte a recursos como transações, concorrência, escalonamento, invocação remota de métodos (*remote method invocation*, RMI), segurança e persistência. O ambiente de execução provido a uma aplicação pelo servidor de aplicações é denominado container EJB (*EJB container*).

Em versões anteriores à especificação EJB 3.0 (JSR-220, 2006), a representação de objetos persistentes se dá através de *Entity Beans* (*Beans* Entidades) com configuração através de arquivos XML. Cada *entity bean* (classe) é associado a uma tabela em um SGBDR (Sistema de Gerenciamento de Banco de Dados Relacional), e cada instância do *entity bean* (objeto) corresponde a uma linha em tal tabela.

O gerenciamento de persistência pode ser realizado pelo próprio *entity bean* (*bean-managed persistence*) ou pelo container EJB (*container-managed persistence*). A segunda abordagem apresenta maior portabilidade entre SGBDRs, devido à transferência de responsabilidade para o servidor de aplicações, possivelmente compatível com diversos SGBDRs.

Java Persistence API

A especificação *Java Persistence API* (JPA) 1.0, para gerenciamento de persistência e mapeamento objeto-relacional (*object-relational mapping*), foi incluída na especificação EJB 3.0. Entretanto, diferente do restante da especificação EJB, direcionada à edição Java EE, a especificação JPA está disponível também para a edição Java SE.

Com a especificação JPA, *Entity Beans* são declarados obsoletos, e objetos persistentes são representados através de entidades (*entities*), configurados através de anotações ao invés de

arquivos XML (utilizado em versões anteriores de EJB e JDO) e armazenados em SGBDRs através de provedores JPA (implementações da especificação).

Posteriormente, a especificação JPA 2.0 (JSR-317, 2009) tornou-se independente, definida à parte da especificação EJB 3.1 (JSR-317, 2009), que passa a requerer suporte à especificação JPA 2.0 em servidores de aplicações. Dentre as justificativas para a separação das especificações está a significativa adoção de JPA com a edição Java SE, não somente Java EE (utilizando EJB), culminando no suporte a JPA em *frameworks* de persistência ORM, como *Hibernate*.

Frameworks ORM

Hibernate é um popular *framework* ORM para a linguagem de programação Java, compatível, a partir da versão 3.2, com a especificação JPA 2.0, disponibilizando um provedor JPA.

Nesse domínio, um *framework* ORM de destaque é *TopLink* da *Oracle Corporation*. Esse *framework* tornou-se a implementação de referência da especificação JPA 1.0 com a doação de seu código fonte para a *Sun Microsystems* em 2006, para o servidor de aplicações de código aberto *GlassFish*. De forma semelhante, ocorreu a doação de código fonte de uma nova versão para a *Eclipse Foundation* em 2007, culminando no projeto *EclipseLink*, implementação de referência da especificação JPA 2.0.

2.3 Metaprogramação

Metaprogramação é a manipulação de programas por metaprogramas (TAHA; SHEARD, 2000).

Um metaprograma é um programa de computador capaz de representar e manipular outros programas, programas-objeto, como dados (Figura 2.4), opcionalmente através de uma metalinguagem de programação (ATTARDI; CISTERNINO, 2001; TAHA; SHEARD, 2000; SKALSKI; MOSKAL; OLSZTA, 2004).

Um programa-objeto é, genericamente, um programa de computador comum, constituído por sentenças em uma linguagem de programação, linguagem-objeto. Um caso particular é um metaprograma ser seu próprio programa-objeto, caracterizando reflexão (Figura 2.5) (ATTARDI; CISTERNINO, 2001).

Exemplos de metaprogramas incluem analisadores, interpretadores, compiladores e geradores de programas – sendo, em comum, manipuladores de dados relacionados à sentenças de linguagens de programação (TAHA; SHEARD, 2000).

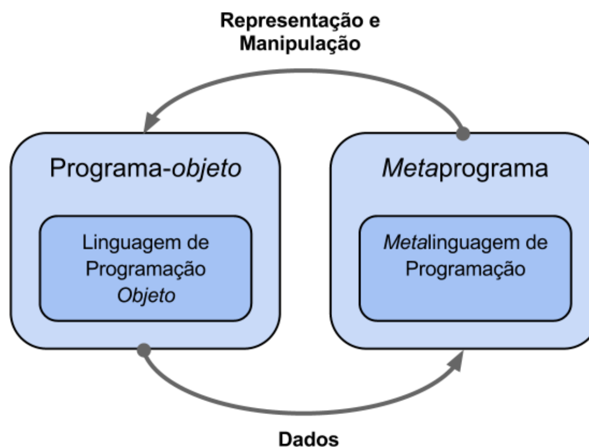

Figura 2.4: Metaprogramação: Metaprogramas e Programas-Objeto

2.3.1 Classificação de Metaprogramas

Metaprogramas são geralmente classificados como analisadores ou geradores de programas (TAHA; SHEARD, 2000).

Um analisador de programas efetua a observação de propriedades, estrutura e ambiente de execução do programa-objeto, de forma a produzir ações ou resultados; por exemplo, grafos de fluxo de dados e tracejamentos de execução (TAHA; SHEARD, 2000; ATTARDI; CISTERNINO, 2001).

Um gerador de programas constrói programas-objeto, geralmente soluções para problemas de um domínio. Frequentemente, cada programa-objeto é uma solução particular (ou especializada) para um problema de um conjunto de problemas relacionados, existentes no domínio (TAHA; SHEARD, 2000). Em diversos domínios, uma solução particular (gerada) requer menos recursos que a solução geral (não-gerada); por exemplo, operações para gráficos 3D especializadas para um certo *hardware* (ATTARDI; CISTERNINO, 2001).

Mesclas de analisadores e geradores de programas são frequentemente encontradas em ferramentas de linguagens de programação, como compiladores e montadores (*assemblers*). Por exemplo, a linguagem de programação C++ permite declarações e definições de classes e métodos com parâmetros de tipo (através de *templates*), e implementações específicas para os tipos utilizados nesses parâmetros são geradas em tempo de compilação (ATTARDI; CISTERNINO, 2001).

Outras variáveis consideradas na classificação de metaprogramas incluem o momento de execução do metaprograma e programa-objeto (tempo de compilação ou execução) e a identidade entre metalinguagem e linguagem-objeto (TAHA; SHEARD, 2000).

Um gerador de programas de tempo de compilação, ou estático, gera código para compilação (por exemplo, *bison*[10] e *yacc* (JOHNSON, 1979)), enquanto um gerador de programas de tempo de execução, ou dinâmico, gera código para execução, pronto para ser carregado. Particularmente, um programa-objeto gerado pode ser, também, um gerador de programas, caracterizando (meta) programação de múltiplos estágios (TAHA; SHEARD, 1999).

Sendo a metalinguagem e a linguagem-objeto uma mesma linguagem, é definido um sistema de metaprogramação homogêneo (ou heterogêneo, caso contrário), com capacidade de reflexão (Figura 2.5) (TAHA; SHEARD, 2000).

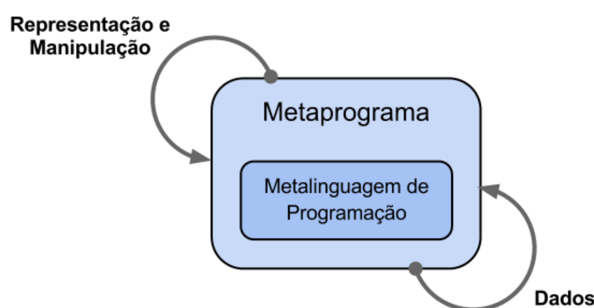

Figura 2.5: Metaprogramação: Reflexão

Um programa de computador com capacidade de reflexão, ou reflexivo, pode acessar e, possivelmente, manipular seu estado de execução (TAHA; SHEARD, 2000; ATTARDI; CISTERNINO, 2001). Nesse sentido, são definidos níveis de complexidade de reflexão: introspecção, que denota possibilidade de acesso ao estado de execução, como propriedades e estrutura de um programa, e intercessão, que denota possibilidade de manipulação desse estado, como alterações de propriedades e comportamento.

Reflexão requer a capacidade de reificação (*reification*), ou materialização, que consiste na exposição do estado de execução como dados, através de um modelo de dados para representação de conceitos ou abstrações (implícitos) de um programa de computador utilizando recursos de linguagens de programação (explícitos) (ATTARDI; CISTERNINO, 2001; OLIVÉ, 2007).

Suportar e permitir invocações de mecanismos de reflexão pode acarretar custos de desempenho (*performance overhead*) em execução (ATTARDI; CISTERNINO, 2001). Na linguagem de programação C++, por exemplo, o mecanismo de informação de tipos em tempo de execução (*Run-Time Type Information*, RTTI) requer código adicional para representar, registrar e localizar informação de tipos (STROUSTRUP, 1994).

[10]`http://www.gnu.org/software/bison`

Tradicionalmente, linguagens compiladas oferecem pouco ou nenhum suporte à reflexão, descartando informações relacionadas a tipos, abstrações e código fonte em tempo de compilação. Em contrapartida, linguagens interpretadas oferecem maior suporte à reflexão, mantendo tais informações para um sistema de interpretação e execução (ATTARDI; CISTERNINO, 2001).

2.3.2 Metaprogramação em Java

Diversos mecanismos de metaprogramação estão presentes no ambiente de execução Java e no kit de desenvolvimento Java, incluindo tipos genéricos, reflexão, instrumentação, anotações, processadores de anotações e compilação em execução.

Esses mecanismos definem um sistema de metaprogramação homogêneo, sendo, a linguagem de programação Java, tanto metalinguagem quanto linguagem-objeto.

Tipos Genéricos

O mecanismo de tipos genéricos (*generics*) permite a utilização de parâmetros de tipo, capazes de representar diversos tipos, com segurança de tipo (*type safety*) em tempo de compilação (GOSLING et al., 2012; BRACHA et al., 1998).

Diferentemente de parâmetros de função (métodos e construtores), argumentos para parâmetros de tipo são tipos (classes e interfaces), e não valores (literais e referências), e são aplicáveis em classes, interfaces, métodos, construtores e atributos (GOSLING et al., 2012), permitindo reuso de código (Listagens 2.1 e 2.2).

```java
1  // Classe com parametros de tipo T e S
2  class UmaClasse<T, S> {
3
4      // Campos com parametros de tipo T e S
5      T umCampo;
6      S umOutroCampo;
7
8      // Construtor com parametros de tipo T e S
9      UmaClasse(T parametroUmCampo, S parametroOutroCampo) {...}
10
11     // Metodo com parametro de tipo U
12     <U> void umMetodo(U umParametro) {...}
13 }
```

Listagem 2.1: Tipos Genéricos: Declarações de classe, construtor e método com parâmetros de tipo

```
1 // Parametros de tipo T e S: representam Integer e String.
2 UmaClasse<Integer, String> umObjeto =
3     new UmaClasse<Integer, String>(42, "Quarenta e dois");
4
5 // Parametro de tipo U: representa Float (automaticamente).
6 umObjeto.umMetodo(3.1415);
7
8 // Parametro de tipo U: representa Boolean (automaticamente).
9 umObjeto.umMetodo(true);
```

Listagem 2.2: Tipos Genéricos: Uso de parâmetros de tipo em classe, construtor e método

Anteriormente à introdução de tipos genéricos, um efeito semelhante era obtido através de conversão de tipos (*type casting*) envolvendo a classe `Object`, raiz da hierarquia de classes (GOSLING et al., 2012). Essa abordagem consiste em burlar a segurança de tipo em atributos, métodos e parâmetros através de conversão de tipos de/para `Object` em atribuições e invocações (BRACHA et al., 1998). Dessa forma, critérios de segurança de tipo são satisfeitos artificialmente, prevenindo a detecção de inconsistências de tipo em tempo de compilação. Consequentemente, falhas de desenvolvimento em porções de código relacionadas à conversão de tipos ocasionariam erros em tempo de execução, eventualmente provocando o término da aplicação.

Tipos genéricos foram introduzidos com a extensão da linguagem de programação Java na edição Java SE versão 5.0. Provendo compatibilidade com classes existentes (sem suporte a tipos genéricos), são empregadas técnicas de remoção de tipos (*type erasure*) em tempo de compilação e conversão de tipos (*type casting*) em tempo de execução; entretanto, são introduzidas garantias de segurança de tipo (*type safety*) no processo de compilação (BRACHA et al., 1998).

Consequentemente, tipos genéricos são retrocompatíveis com tipos não-genéricos e contam com segurança de tipo em tempo de compilação.

Reflexão

O mecanismo de reflexão permite introspecção e intercessão em aplicações através da linguagem de programação Java.

A API de Reflexão Java (*Java Reflection API*) consiste em metodologias de instrospecção sobre elementos da linguagem de programação Java e intercessão em invocações de mé-

todos de interfaces por classes de procuração (*proxy*). Essa API é especificada no pacote `java.lang.reflect`.

A classe `Class` é o vínculo entre a aplicação e a API de Reflexão Java, especificando métodos para examinar membros de classes (Listagem 2.3), como atributos, métodos e construtores, e também parâmetros e tipo de retorno desses, além de modificadores (por exemplo, proteção de acesso, imutabilidade, escopo de classe), parâmetros de tipo e hierarquia de classes (superclasses, classes internas, classes envoltórias, classes declarantes), bem como carregar classes e instanciar objetos.

Uma instância de `Class` é associada à cada tipo em uma aplicação, acessível através do método de escopo de classe `getClass`. Esse método é definido na classe `Object`, raiz da hierarquia de classes na linguagem de programação Java, e, portanto, presente em todos os objetos.

Diversos dos métodos para examinar membros permitem a distinção entre elementos *declarados* e elementos *públicos* (Listagem 2.3). Elementos declarados, em referência à declarações em código fonte, são os elementos declarados na classe em reflexão, com qualquer nível de proteção de acesso – *particular* (`private`), *protegido* (`protected`) e *público* (`public`). Elementos públicos, em referência ao nível de proteção de acesso público, são elementos acessíveis por outras classes em tempo de execução, advindos da classe em reflexão e por herança de sua hierarquia de classes. Essa distinção provê garantias de que acessos e invocações a elementos públicos não provocarão violações de encapsulamento.

Uma classe de procuração (*proxy class*) permite interceptar invocações de métodos de interfaces. Dessa forma, é possível efetuar alterações de comportamento, por exemplo, através da adição ou substituição de funcionalidades.

Adotar reflexão em uma aplicação exige considerações sobre desempenho, segurança, manutenção e portabilidade. Certas otimizações da Máquina Virtual Java são desabilitadas devido à resolução dinâmica de tipos, acarretando menor desempenho em porções de código com reflexão. Gerenciadores de Segurança (*Security Managers*; responsáveis por políticas de segurança de execução) podem negar permissões de execução necessárias à reflexão, prevenindo a execução em ambientes restritivos (por exemplo, *applets* ou dispositivos embarcados). Finalmente, quebras de abstração e exposição de detalhes internos (por exemplo, membros particulares de classes) através de reflexão podem acarretar comprometimento de funcionalidade e portabilidade entre diferentes versões de uma aplicação.

```java
public static void reflexao(Object objeto) {

    /* Classe */
    System.out.println("\nClasse:");
    System.out.println(" - " + objeto.getClass().getName());

    /* Atributos */
    System.out.println("\Atributos (publicos):");
    for (Field f : objeto.getClass().getFields())
        System.out.println(" - " + f.getName() + ": \t" + f);

    System.out.println("\Atributos (declarados):");
    for (Field f : objeto.getClass().getDeclaredFields())
        System.out.println(" - " + f.getName() + ": \t" + f);

    /* Metodos */
    System.out.println("\nMetodos (publicos):");
    for (Method m : objeto.getClass().getMethods())
        System.out.println(" - " + m.getName() + ": \t" + m);

    System.out.println("\nMetodos (declarados):");
    for (Method m : objeto.getClass().getDeclaredMethods())
        System.out.println(" - " + m.getName() + ": \t" + m);

    /* Construtores */
    System.out.println("\Construtores (publicos):");
    for (Constructor c : objeto.getClass().getConstructors())
        System.out.println(" - " + c.getName() + ": \t" + c);

    System.out.println("\nConstrutores (declarados):");
    for (Constructor c : objeto.getClass().getDeclaredConstructors())
        System.out.println(" - " + c.getName() + ": \t" + c);
}
```

Listagem 2.3: Reflexão: Atributos (Campos), Métodos e Construtores de um Objeto

Instrumentação

O mecanismo de instrumentação permite modificar arquivos-classe em tempo de execução, permitindo alterações de características e comportamento de classes.

Por definição, um *arquivo-classe* (*class file*) consiste em um vetor de *bytes* (não necessariamente um *arquivo*, apesar da nomenclatura) contendo a representação binária de uma classe segundo o formato *Class File Format*, com tabela de constantes, descrição de atributos, corpos de métodos (*Java bytecode*), informações de herança etc (LINDHOLM et al., 2012). O processo de instrumentação de arquivos classe é também conhecido por instrumentação de *Java bytecode*.

Instrumentação de *Java bytecode* consiste em processos de *definição* e *transformação* de classes na Máquina Virtual Java. É possível definir, transformar, redefinir e retransformar classes.

Definir uma classe (Figura 2.6) consiste em carregar e interpretar um arquivo-classe e disponibilizar a classe resultante para a aplicação (independentemente de instrumentação).

Esse processo é realizado normalmente pela Máquina Virtual Java durante a execução da aplicação.

A definição de uma classe pode ocorrer involuntariamente, com a primeira referência à classe na aplicação (sob demanda, de forma transparente), ou voluntariamente, com a invocação de um carregador de classes (*class loader*).

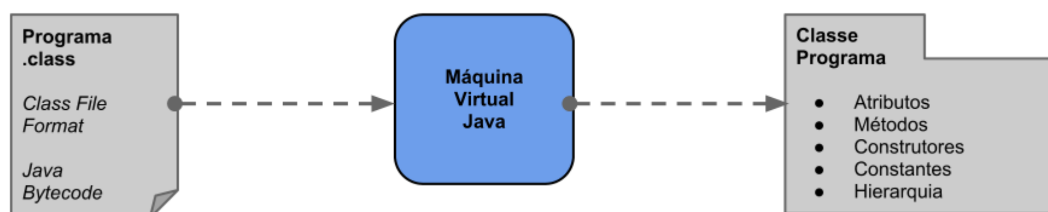

Figura 2.6: Instrumentação: Definição de Classe

Transformar uma classe (Figura 2.7) consiste em modificar um arquivo-classe durante o processo de definição, entre as fases de carregamento e interpretação, disponibilizando para a aplicação uma classe resultante diferente da que seria obtida com o arquivo-classe original, sem modificações.

Essa intervenção é realizada pela Máquina Virtual Java com a invocação de *transformadores*, agentes modificadores de arquivos-classe, registrados na Máquina Virtual Java em sua inicialização ou em tempo de execução e acoplados à aplicação em execução.

A invocação de transformadores pode ocorrer involuntariamente, em uma definição invo-

luntária, ou voluntariamente, em uma definição voluntária ou com a invocação de redefinição ou retransformação de classes.

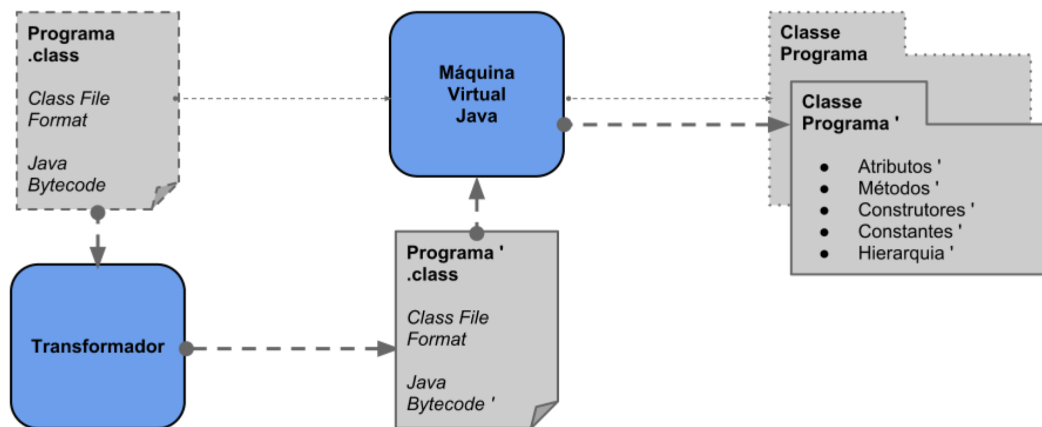

Figura 2.7: Instrumentação: Transformação de Classe

Redefinir uma classe (Figura 2.8) consiste em definir novamente uma classe, porém, com um arquivo-classe diferente, disponibilizando para a aplicação uma classe resultante diferente da classe atual (resultante da definição ou redefinição da classe).

A redefinição difere da transformação em dois aspectos: relação com o arquivo-classe original e momento de disponibilização da classe com modificações para a aplicação. Enquanto a transformação é baseada no arquivo-classe original, efetuando sua *modificação*, a redefinição não o é, efetuando sua *substituição*. Consequentemente, a disponibilização da classe com modificações por redefinição pode ocorrer somente *após* a definição da classe, enquanto, por transformação, ocorre *junto* à definição da classe.

A redefinição de classes somente ocorre voluntariamente, por invocação.

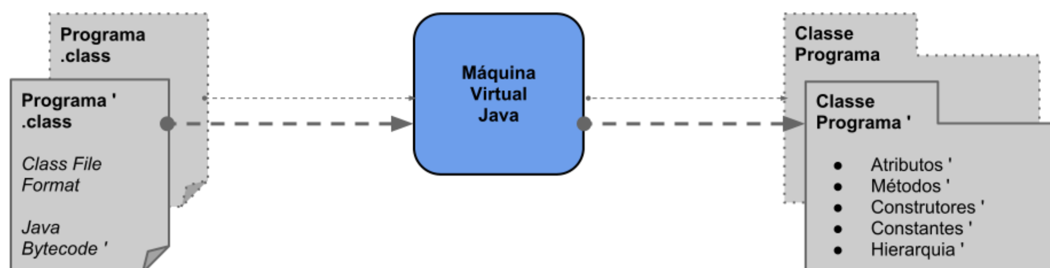

Figura 2.8: Instrumentação: Redefinição de Classe

Retransformar uma classe (Figura 2.9) consiste em transformar novamente uma classe, após sua definição (primeira transformação) ou em sua redefinição (demais transformações). Efetivamente, a retransformação de classes é outra invocação de transformadores sobre arquivos-classe em retransformação ou redefinição.

A retransformação de classes pode ocorrer voluntariamente, por invocação, ou involuntariamente, decorrente de uma redefinição (por sua vez, voluntária).

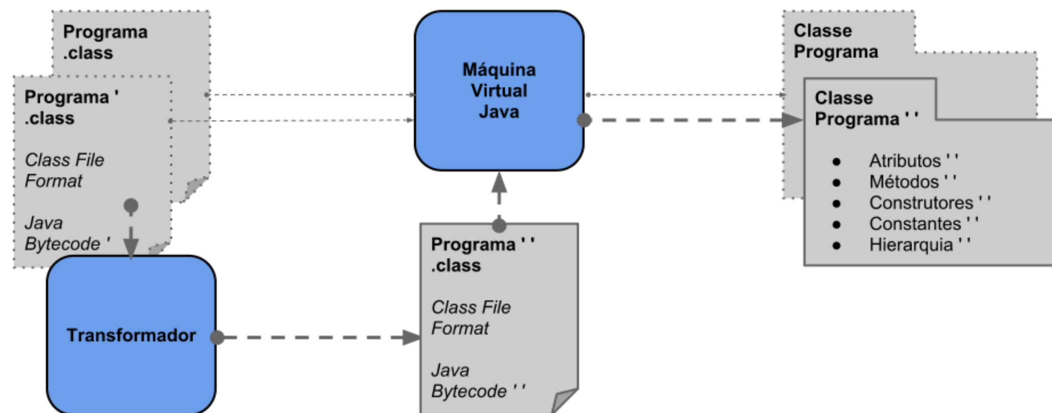

Figura 2.9: Instrumentação: Retransformação de Classe

Classes em redefinição ou retransformação devem ser *redefinições legais* da definição, segundo a *semântica de redefinição*. Dessa forma, são permitidas modificações em corpos de métodos, tabela de constantes e valores de atributos, porém não são permitidas modificações como adição, alteração de nomes ou remoção de atributos ou métodos, tampouco modificações em assinaturas de métodos ou informações de herança. Oportunamente, classes em definição podem ser modificadas de forma menos restritiva, através de transformação.

A instrumentação de *Java bytecode* é baseada em agentes, contando com duas interfaces de programação:

- Pacote `java.lang.instrument`: esse pacote especifica uma API de instrumentação de *Java bytecode*, através da linguagem de programação Java.

- Interface JVM TI (*Java Virtual Machine Tool Interface*; Interface de Ferramentas para Máquina Virtual Java) (ORACLE, 2012b): essa interface consiste em uma NPI (*Native Programming Interface*; Interface de Programação Nativa) (LIANG, 1999) para acesso e manipulação do estado de execução de aplicações, através de funções de instrumentação de *Java bytecode*, gerenciamento de memória, execução de *threads*, quadros de segmento de pilha (*stack frame*), membros de classes, entre outros, através de JNI (*Java Native Interface*; Interface Nativa Java).

Diversos compoentes (classes, interfaces) do pacote `java.lang.instrument`, mencionados a seguir, têm *equivalentes nativos* em JVM TI.

Um agente, ou transformador, deve implementar a interface `ClassFileTransformer` (transformador de arquivos-classe), com a definição do método `transform` (Listagem 2.4). Esse método tem como entrada (argumento) um arquivo classe inicial e como saída (retorno) um arquivo classe final (ou nulo, em ausência de modificações); outros argumentos determinam propriedades da classe: nome, carregador de classes (`ClassLoader`) e domínio de proteção (segurança) da classe, e instância `Class` em redefinição/retransformação (ou nulo, em definição), para reflexão.

```
1 byte[] transform(
2             ClassLoader loader,
3             String className,
4             Class<?> classBeingRedefined,
5             ProtectionDomain protectionDomain,
6             byte[] classfileBuffer)
7         throws IllegalClassFormatException
```

Listagem 2.4: Instrumentação: Assinatura do Método `ClassFileTransformer.transform`

Transformadores são registrados através da classe de serviços `Instrumentation`. Essa classe define métodos para registrar (`addTransformer`) e desregistrar (`removeTransformer`) transformadores, obter informações sobre classes carregadas (`getAllLoadedClasses`), e redefinir (`redefineClasses`) e retransformar (`retransformClasses`) classes. É facultado a um transformador suportar retransformação (parâmetro de registro `canRetransform`); em caso negativo, esse não é invocado em retransformações, sendo utilizado o resultado da transformação.

Na presença de vários transformadores, ocorre o encadeamento de invocações:

1. O primeiro transformador tem como entrada o arquivo classe inicial (argumento de `ClassLoader.defineClass` ou `Instrumentation.redefineClasses`);

2. Transformadores seguintes tem como entrada a saída do transformador anterior;

3. O último transformador tem como saída o arquivo de classe final para a Máquina Virtual Java.

A sequência de invocação de transformadores tem a seguinte ordem:

1. Transformadores sem suporte de retransformação não-nativos;

2. Transformadores sem suporte de retransformação nativos;
3. Transformadores com suporte de retransformação não-nativos;
4. Transformadores com suporte de retransformação nativos.

Assistentes de instrumentação de *Java bytecode* auxiliam a modificação de arquivos-classe, com operações apropriadas para o formato *Class File Format*, abrangendo adição, remoção e modificação de constantes, atributos, métodos e informações de herança; por exemplo, *JavaAssist* (CHIBA, 2000) e *Apache Commons BCEL* [11] (*Byte Code Engineering Library*).

Anotações

O mecanismo de anotações consiste em metaelementos para anexar metadados a elementos da linguagem de programação Java (GOSLING et al., 2012).

Metadados são acessíveis em tempo de execução através de reflexão e em tempo de compilação através de processadores de anotações. Por padrão, metadados não tem efeito sobre a operação de elementos (métodos e construtores).

Anexar metadados a elementos consiste em adicionar anotações a seus modificadores. Esse processo é conhecido como anotar elementos (*to annotate elements*) e os elementos são ditos anotados (*annotated*).

Anotação é um modificador (como `public` e `static`) associado a um tipo-anotação (*annotation type*), prefixado com o símbolo `@` (arroba – em inglês, *at*, como acrônimo para *annotation type*). Por exemplo, a anotação `@Override` é um modificador associado ao tipo-anotação `Override`. Convencionalmente, anotações e outros modificadores são posicionados em linhas distintas, com anotações antes de outros modificadores (Listagem 2.5).

```
1 @Override
2 public static void umMetodo() {...}
```

Listagem 2.5: Anotações: Anotação `@Override`

Tipo-anotação (*annotation type*) é um tipo especial de interface (GOSLING et al., 2012), com declaração semelhante à declaração de interfaces e prefixada com o símbolo `@`. Em contraste com interfaces, em tipos-anotação, métodos são elementos de tipo-anotação (*annotation type*

[11] `http://commons.apache.org/bcel`

elements), ou, simplesmente, elementos, aos quais podem ser atribuídos valores, contando com valores padrão (palavra-chave `default`).

A atribuição de valores a elementos ocorre em anotações através de pares elemento-valor, ou nome-valor (*element-value pairs, name-value pairs*). Adicionalmente, o nome de elemento `value` pode ser omitido caso seja o único elemento do tipo-anotação. Um tipo-anotação marcador (*marker*) não possui elementos (Listagens 2.6 e 2.7).

```
1  // Tipo-anotacao com elemento com valor padrao.
2  @interface UmTipoAnotacao {
3      int     primeiroElemento();
4      boolean segundoElemento();
5      String  terceiroElemento() default "Um valor padrao.";
6  }
7
8  // Tipo-anotacao com elemento 'value'.
9  @interface OutroTipoAnotacao {
10     String value();
11 }
12
13 // Tipo-anotacao marcador.
14 @interface TipoAnotacaoMarcador { }
```

Listagem 2.6: Anotações: Tipos-anotação com valor padrão, elemento `value` e marcador

```
1  @UmTipoAnotacao(
2      primeiroElemento = 42,
3      segundoElemento = true
4      // par elemento-valor omitido para terceiroElemento
5      // (uso de valor padrao).
6  )
7  public void umMetodo() {...}
8
9  @OutroTipoAnotacao("um valor")
10 public void outroMetodo() {...}
11
12 @TipoAnotacaoMarcador
13 public void umOutroOutroMetodo() {...}
```

Listagem 2.7: Anotações: Anotações com valor padrão, elemento `value` e marcador

Através de meta-anotações (anotações para tipos-anotação) é possível anexar metadados à tipos-anotação. Por exemplo, a linguagem de programação Java contém as anotações pré-definidas `@Target`, `@Retention`, `@Inherited` para configurar tipos-anotação:

`@Target` (alvo) permite restringir tipos de elementos aplicáveis a um tipo-anotação: campos, construtores, métodos, parâmetros, variáveis locais, pacotes, tipos (classes, enumerados e interfaces, incluindo anotações) e anotações (usado por meta-anotações).

`@Retention` permite especificar a política de retenção (disponibilidade) de anotações: descartar em compilação, reter em compilação (em arquivos-classe) mas não reter em execução, ou reter em compilação e execução (disponível para reflexão).

`@Inherited` (herdado) permite configurar herança automática de anotações de uma classe (superclasse) para suas subclasses.

Conceitos relacionados à anotações são especificados no pacote `java.lang.annotation`.

Processadores de Anotações

Um processador de anotações consiste em um programa Java capaz de processar elementos com anotações, acoplável (*pluggable*) ao compilador Java (Listagem 2.8, Figura 2.10), atuando tanto como analisador quanto gerador de programas.

```
1 javac
2     -processorpath /caminho/para/processadores_de_anotacoes/
3     -processor ClasseProcessadorDeAnotacoes
4     ArquivoDeCodigoFonte.java [*.java]
```

Listagem 2.8: Processadores de Anotações: Acoplamento ao compilador Java

Durante o processo de compilação, processadores de anotações são invocados pelo compilador para complementar a compilação de arquivos de código fonte contendo certas anotações, registradas pelo processador de anotações (Figura 2.10). Um processador de anotações pode efetuar diversas ações, baseado em tipos-anotação e pares elemento-valor.

Dentre as possíveis ações está a introdução de novos arquivos de código fonte no processo de compilação, caracterizando um processador de anotações como um gerador de programas estático. Os novos arquivos de código fonte podem ser novamente processados por (este ou outros) processadores de anotações, determinando rodadas (*rounds*) de processamento (Figura 2.10).

No pacote `javax.annotation.processing` são especificadas classes e interfaces para declaração de processadores de anotações (`AbstractProcessor`) e comunicação com um ambi-

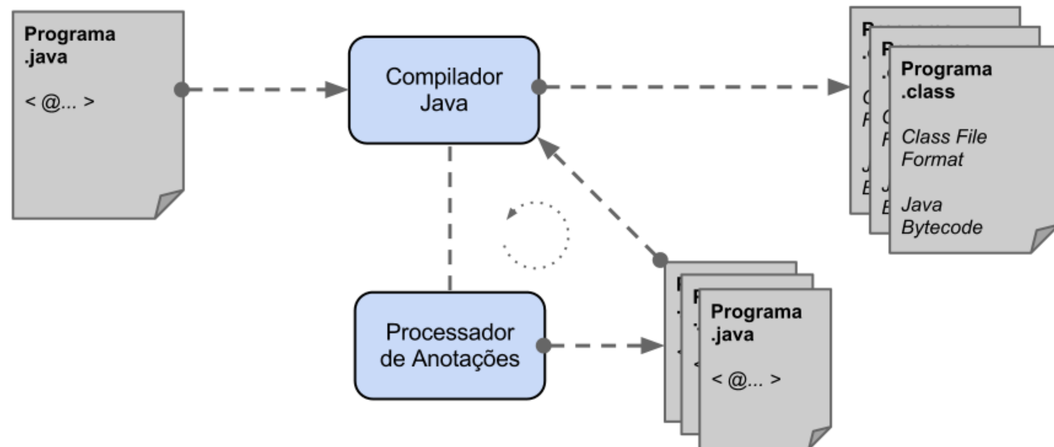

Figura 2.10: Processador de Anotações: Acoplamento com Compilador

ente de processamento de anotações, provendo suporte a mensagens de compilação (`Messager`) e leitura e escrita de arquivos (`Filer`), como código fonte e classes etc.

O pacote `javax.lang.model` contém a especificação de um modelo da linguagem de programação Java, baseado no conceito de *Mirrors* (espelhos) (BRACHA; UNGAR, 2004). Basicamente, esse modelo distingue entre elementos (`javax.lang.model.element`), construções estáticas da linguagem de programação, e tipos de elementos (`javax.lang.model.type`).

Nesse domínio, são definidos os seguintes elementos: pacotes, classes, construtores, métodos, parâmetros, campos, variáveis e tipos de dados; e os seguintes tipos: vetores, declarações (classes e interfaces), tipo nulo (`null`), tipos primitivos (`boolean, byte, short, int, long, char, float` e `double`), tipos de referência (classes, interfaces, vetores, parâmetros de tipo, e nulo) e tipos executáveis (construtores, métodos).

Dessa forma, um processador de anotações pode atuar como analisador e gerador de programas através, respectivamente, do modelo da linguagem de programação Java e recursos do ambiente de processamento de anotações.

Compilação em Execução

O mecanismo de compilação em execução consiste em invocar um compilador Java em tempo de execução, transformando programas-objeto resultantes de geradores de programas em arquivos-classe.

As ferramentas de compilação são especificadas na API de Compiladores da Linguagem de Programação Java (*Java Programming Language Compiler API*) (JSR-199, 2006), definida no pacote `javax.tools`. Essa API especifica interfaces padronizadas para invocação de ferra-

mentas de compilação, como um compilador Java com suporte a diagnósticos de compilação e arquivos de código fonte (BIESACK, 2007; SADZIAK, 2009).

A interface `JavaCompiler` especifica métodos para configurar uma tarefa, ou processo, de compilação (`CompilationTask`), com gerenciadores de arquivos (`JavaFileManager`), diagnósticos (`DiagnosticListener`), arquivos de código fonte (`JavaFileObject`) e opções de compilador. A interface (`JavaFileObject`) permite representar arquivos de código fonte em diferentes formas, por exemplo, *strings* de texto.

Enquanto classes e interfaces desse pacote são requeridas pela especificação da edição Java SE, não são requeridas implementações ou ferramentas para tal. Entretanto, em ambientes de execução Java com o kit de desenvolvimento Java presente, as ferramentas deste são utilizadas em implementações do pacote, acessíveis através da classe `ToolProvider`.

Discutivelmente, um gerador de programas baseado nesse mecanismo pode ser classificado como estático ou dinâmico. Enquanto um gerador de programas estático gera código para compilação e um gerador de programas dinâmico gera código para execução, aquele pode efetuar tanto a compilação dos programas-objeto quanto a execução dos arquivos-classe resultantes.

Essa discussão é esclarecida com a observação do processo sob diferentes granularidades. Sob granularidade grossa, um programa-objeto é gerado e executado, aparentando um gerador de programas dinâmico. Entretanto, sob granularidade fina, um programa-objeto é gerado e compilado em arquivo-classe, e então este (não o programa-objeto) é executado; caracterizando, portanto, um gerador de programas estático.

Considerações Finais

Neste capítulo, foram apresentados conceitos e técnicas sobre os tópicos plataforma Java, linguagem de programação Java, persistência de objetos e metaprogramação.

Conceitos sobre a plataforma Java, como Máquina Virtual Java, *Java bytecode*, Java API e a linguagem de programação Java, elucidam aspectos do desenvolvimento e funcionamento de aplicações Java.

Técnicas relacionadas às especificações de persistência de objetos e aos mecanismos de metaprogramação consistem em aplicações dos conceitos descritos neste capítulo à plataforma Java e linguagem de programação Java.

Vale lembrar que os mecanismos de metaprogramação, particularmente, são adotados no desenvolvimento deste trabalho.

Capítulo 3

Framework Object-Injection

Object-Injection é um *framework* de persistência e indexação de objetos escrito na linguagem de programação Java (CARVALHO et al., 2013). Suas funcionalidades são fundamentadas em dois níveis de índices: primários, para persistência, e secundários, para indexação.

Um índice primário é um índice sobre um conjunto de atributos que contém a chave primária. Um índice secundário, inversamente, não contém a chave primária (RAMAKRISHNAN; GEHRKE, 2003; GARCIA-MOLINA; ULLMAN; WIDOM, 2008). Em perspectiva funcional, índices primários determinam a localização de registros em arquivos de dados contendo os atributos do respectivo objeto (RAMAKRISHNAN; GEHRKE, 2003); índices secundários, em contrapartida, determinam a identificação de objetos por atributos, para recuperararação dos respectivos registros através de índices primários (RAMAKRISHNAN; GEHRKE, 2003).

As chaves primárias são definidas como identificadores UUID (*Universally Unique Identifier*; identificador universalmente único), valores de 128 bits (ZAHN; DINEEN; LEACH, 1990; LEACH; MEALLING; SALZ, 2005; ISO (International Organization for Standardization), 2004).

A garantia de unicidade de identificadores UUID é atrelada a algoritmos de geração. Três algoritmos de geração de identificadores UUID são especificados para identificação de objetos (ISO (International Organization for Standardization), 2004):

- **Algoritmo Temporal**: baseado em valores de relógio, para um computador, e, adicionalmente, endereços físicos de interfaces de rede, para vários computadores;
- **Algoritmo Textual**: baseado em *hash* criptográfico sobre nomes (*strings*) únicos em um contexto;
- **Algoritmo Aleatório**: baseado em geradores de números (pseudo)aleatórios.

Em conformidade com esses algoritmos, o esgotamento de possibilidades, considerando uma taxa de geração de UUIDs de 100 trilhões de UUID por nanosegundo, ocorreria em aproximadamente 3,1 trilhões de anos (ISO (International Organization for Standardization), 2004).

A funcionalidade de persistência é baseada em índices primários, responsáveis pelo armazenamento e recuperação de objetos em meio persistente. Identificadores de Objetos (*Object Identifier*, OID) são empregados como chaves primárias, definidos através do algoritmo aleatório de geração de identificadores UUID. São disponibilizadas implementações de índices primários baseados nas estruturas de dados *hash* extensível (FAGIN et al., 1979) e lista duplamente encadeada.

A funcionalidade de indexação é baseada em índices secundários, definidos por domínio de indexação e um conjunto de atributos-chave. Esses atributos e o identificador de objeto são replicados em objetos-chave, ou chaves, em uma estrutura de indexação apropriada para o domínio. São disponibilizadas implementações das estruturas de indexação Árvore B+ (domínio ordenável) (COMER, 1979), Árvore M (domínio métrico) (CIACCIA; PATELLA; ZEZULA, 1997) e Árvore R (domínio espacial) (GUTTMAN, 1984) .

As estruturas de indexação baseadas em árvores armazenam chaves de roteamento em nós internos e chaves, propriamente, em nós folha. Dessa forma, a partir de um valor de atributo-chave, através da navegação em chaves de roteamento, é alcançada uma chave, com um identificador de objeto. Caracterizando uma referência, esse identificador é aplicado à recuperação do objeto em um índice primário.

3.1 Módulos

O *framework Object-Injection* é organizado em 4 módulos (Figura 3.1):

- `Metaclasses`: define interfaces de entidades (objetos persistentes), chaves (objetos indexáveis contendo atributos-chave de um objeto persistente), domínios de indexação e classes ajudantes de (des)serialização.

- `Armazenamento`: define estruturas de dados para índices primários e secundários, responsáveis por gerenciar entidades e chaves em unidades de armazenamento do módulo `Blocos`.

- `Blocos`: define unidades de armazenamento para estruturas do módulo `Armazenamento`,

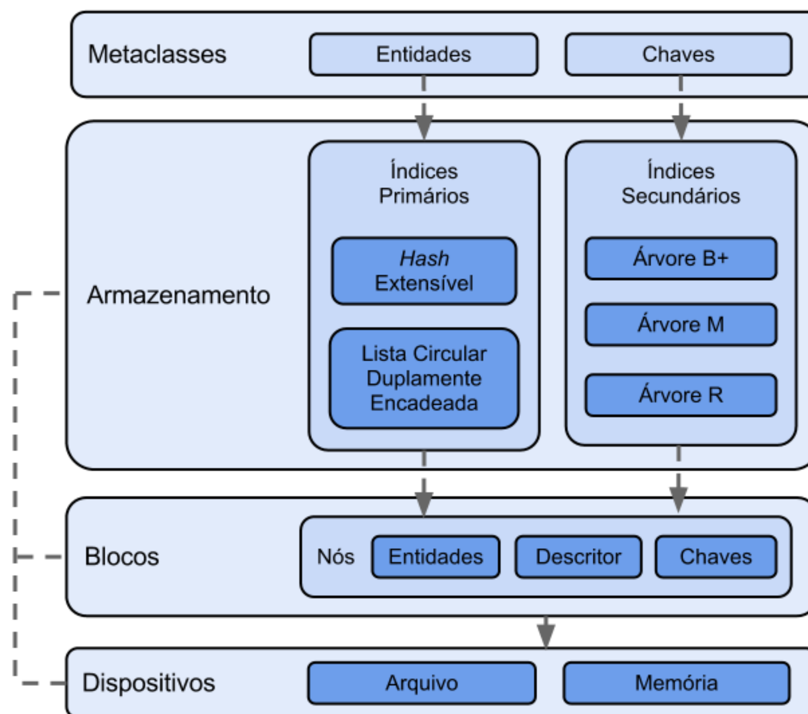

Figura 3.1: *Framework Object-Injection*: Módulos

possibilitando a divisão dos dados armazenados. Unidades de armazenamento são independentes de meio de armazenamento e efetivamente disponibilizadas pelo módulo `Dispositivos`.

- `Dispositivos`: define gerenciadores de recursos computacionais para unidades de armazenamento. Essa é uma camada de abstração sobre meios de armazenamento, responsável por gerenciar alocação, manipulação e liberação de recursos, disponibilizando unidades de armazenamento para estruturas do módulo `Armazenamento`.

Esses módulos são detalhados a seguir.

3.1.1 Módulo Metaclasses

O módulo `Metaclasses` define o vínculo entre aplicação e *framework* através de entidades e chaves, associadas a domínios de indexação.

Metaclasses adicionam significado para o *framework* em classes da aplicação, com resolução e segurança de tipos (*type safety*) garantidas em tempo de compilação. As metaclasses do *framework* são as interfaces `Entity`, `Key` e as interfaces relacionadas a domínios de indexação `Order`, `Metric`, `Point`, `Rectangle`, e `StringMetric`.

A Figura 3.2 ilustra classes do módulo relacionadas a persistência e indexação e exemplos de classes de aplicação: metaclasses de persistência (fundo claro), metaclasses de indexação (fundo escuro), um exemplo de classe de aplicação (fundo branco), além de suas especializações para entidade (persistência) e chaves (indexação). Todas essas classes são descritas a seguir.

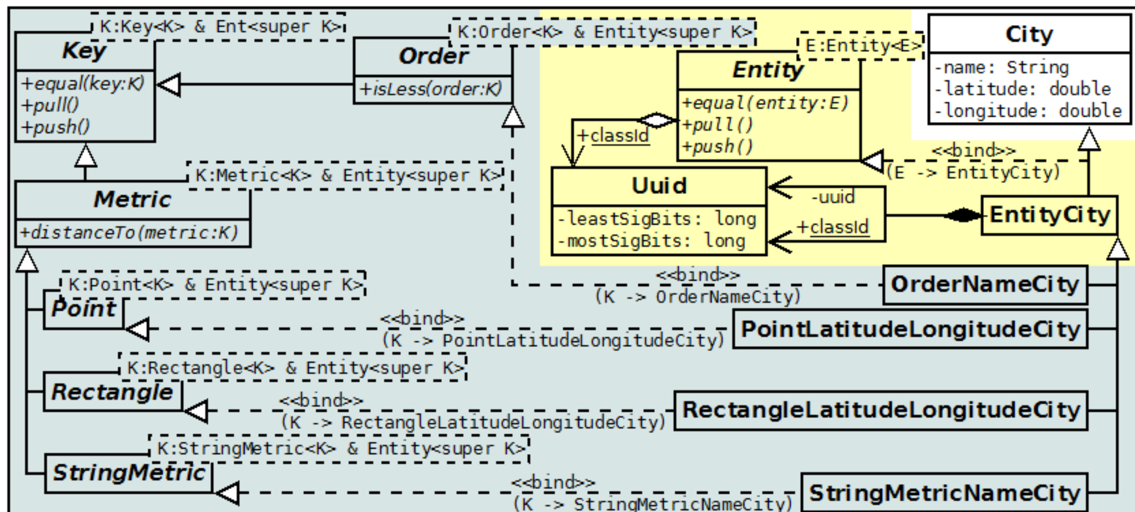

Figura 3.2: *Framework Object-Injection*: Pacote Metaclasses

Uma entidade é a representação de um objeto persistente, definida na interface `Entity`. Uma chave é a representação de um objeto persistente e indexável através de atributos-chave, definida na interface `Key`. Em uma aplicação, classes devem implementar a interface `Entity` para fazer uso da funcionalidade de persistência e, adicionalmente, a interface `Key` para a funcionalidade de indexação.

A interface `Entity` especifica métodos de identificação, comparação de igualdade e (des)serialização de entidades. A interface `Key` especifica métodos de comparação de igualdade e (des)serialização de chaves. Adicionalmente, a interface `Entity` especifica um atributo de escopo de classe para identificação e comparação de igualdade entre classes de entidades. Ainda, a interface `Key` é especializada em domínios de indexação, com especificações de métodos particulares à cada domínio, através das interfaces `Order` (domínio ordenável) e `Metric` (domínio métrico), especializada em `Point` e `Rectangle` (domínio métrico espacial) e `StringMetric` (domínio métrico textual).

Métodos de identificação permitem identificar unicamente cada entidade e classe de entidades em uma aplicação, através do respectivo identificador UUID. Métodos de comparação de igualdade permitem verificar se quaisquer duas entidades ou chaves são iguais (não idênticas), evitando duplicatas, ou se pertencem à mesma classe de entidades, assegurando coerência

durante recuperação de entidades. Por fim, métodos de (des)serialização permitem converter atributos de entidades e chaves em uma sequência (ou vetor; *array*, ou *stream*) de *bytes* e vice-versa, para unidades de armazenamento; classes ajudantes (*helpers*) de (des)serialização, `PullStream` e `PushStream` , definem métodos com esse propósito.

A interface `Order` especifica um método de comparação de ordem entre chaves. Esse método é empregado na ordenação de chaves, satisfazendo propriedades de relação de ordem total (transitividade, anti-simetria, totalidade) (ZEZULA et al., 2005).

A interface `Metric` especifica um método de comparação de distância entre chaves. Esse método é empregado em chaves com relação de (dis)similaridade, porém sem relação de ordem, satisfazendo propriedades de função de distância (não-negatividade, simetria, reflexividade) (ZEZULA et al., 2005).

As interfaces `Point` e `Rectangle` especificam métodos de acesso à coordenadas dimensionais; adicionalmente, `Rectangle` especifica métodos de acesso à coordenadas de origem e extensão em dimensões.

A interface `StringMetric` especifica métodos acessadores para o texto (*string*) de indexação.

Com adoção de funcionalidades especificadas em interfaces, são garantidas transparência e retrocompatibilidade para classes de uma aplicação. É possível eximir classes existentes da aplicação de modificações, através de classes auxiliares dedicadas ao vínculo com o *framework*.

Uma classe auxiliar de persistência ou indexação pode especializar uma classe da aplicação e implementar a interface `Entity` ou `Key`. Isso torna a classe auxiliar compatível, em tipos, atributos e métodos, com a classe da aplicação e interface.

Esse arranjo é transparente à classe da aplicação através de conversões de tipo entre classes de aplicação e classes auxiliares, realizadas automaticamente pelo *framework*.

Outro benefício é não serem necessárias modificações às classes da aplicação, por vezes indisponíveis em código fonte, tornando o *framework* retrocompatível com classes da aplicação através de classes auxiliares.

As interfaces `Entity` e `Key` adotam o padrão de projeto CRTP (*Curiously Recurring Template Pattern*; padrão com *template* curiosamente recursivo) a fim de garantir restrições de comparação entre entidades ou chaves. Nesse padrão de projeto, um parâmetro de tipo de uma classe recebe como argumento a própria classe e, logo, si próprio.

Aquelas interfaces requerem um parâmetro de tipo, empregado em métodos de comparação.

Dessa forma, a assinatura dos métodos de comparação restringe comparações de igualdade à mesma classe de entidades ou chaves. Por definição, a comparação de igualdade entre classes distintas tem resultado falso em qualquer caso.

Exemplificando o módulo, a Figura 3.2 apresenta o cenário de uma aplicação. `City` é uma classe da aplicação, representando uma cidade. `EntityCity` é uma classe auxiliar de persistência. São definidas as classes auxiliares de indexação `OrderNameCity` e `OrderMayorCity` (chaves para indexação em domínio ordenável, através dos atributos `name` e `mayor`, respectivamente), `PointLatitudeLongitudeCity` e `RectangleLatitudeLongitudeCity` (chaves para indexação em domínio métrico espacial, através dos atributos `latitude` e `longitude`, simultaneamente), e `StringMetricNameCity` e `StringMetricMayorCity` (chaves para indexação em domínio métrico textual, através dos atributos `name` e `mayor`, respectivamente).

3.1.2 Módulo Armazenamento

O módulo `Armazenamento` define a especificação de estruturas de dados no *framework*. A Figura 3.3 apresenta as classes do módulo, descritas a seguir.

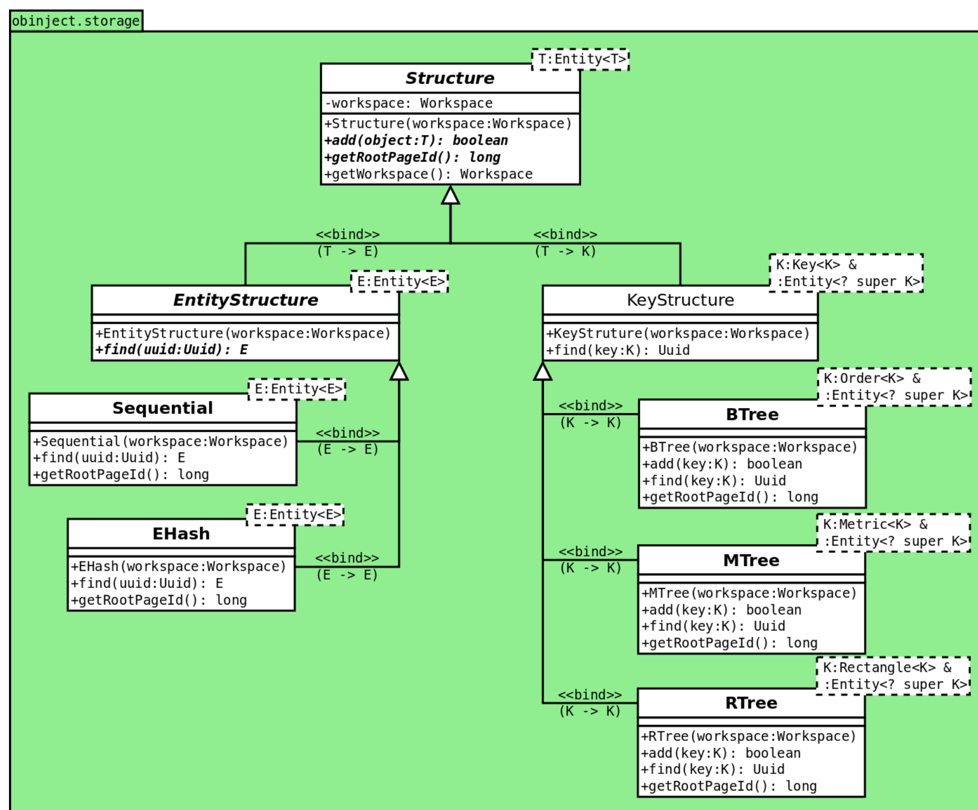

Figura 3.3: *Framework Object-Injection*: Pacote Armazenamento

Estruturas para índices primários, responsáveis por gerenciar entidades, são especificadas

na classe EntityStructure. Estruturas para índices secundários, responsáveis por gerenciar chaves, são especificadas na classe KeyStructure. Essas classes abstratas especificam métodos de gerenciamento (como inserção e busca) e atributos de classes de entidades e chaves, empregados em instanciação genérica de objetos.

Ambas as classes são especializações da classe abstrata Structure. Essa classe define a dependência em relação à interface Workspace. Estruturas de dados organizam os dados armazenados (entidades e chaves) em unidades de armazenamento, gerenciadas por um Workspace (espaço de trabalho). Essa organização permite compartilhamento de um espaço de entidades e chaves entre diferentes estruturas de dados.

Com o *framework*, são disponibilizadas as implementações de estruturas de dados Sequential (lista circular duplamente encadeada), EHash (*hash* extensível) – especializações da classe EntityStructure – e BTree (Arvore B+), MTree (Árvore M) e RTree (Árvore R) – especializações da classe KeyStructure.

3.1.3 Módulo Blocos

O módulo Blocos define unidades de armazenamento de estruturas de dados (Figura 3.4).

Figura 3.4: *Framework Object-Injection*: Pacote Blocos

Unidade de armazenamento é uma unidade funcional de uma estrutura de dados, consistindo de uma sequência de *bytes* em formato específico para armazenamento de um ou mais objetos. Normalmente, a sequência de *bytes* é associada a blocos de meios de armazenamento, tendo

tamanho definido em função do tamanho de um bloco.

A classe `Node` (nó) e suas especializações especificam formato de dados, métodos e classes funcionais de unidades de armazenamento, além de métodos de (des)serialização de atributos. Dentre as especializações da classe `Node` estão a classe `AbstractNode`, contendo métodos de identificação de estruturas de dados, e as classes funcionais `DescriptorNode`, para descrição de propriedades de estruturas de dados (por exemplo, nó raiz, nós livres, estatísticas de acesso), `EntityNode` e `KeyNode`, para armazenamento de entidades e chaves, respectivamente. Essas classes são especializadas, ainda, por estruturas de dados, segundo suas particularidades.

O formato de dados de um nó é descrito em 5 seções: cabeçalho (*header*), propriedades (*features*), índice de entradas (*entries*), objetos (*entities, keys*) e espaço disponível (*free space*). O formato da seção cabeçalho é idêntico entre todos os nós, e o formato das seções propriedades, índice de entradas e objetos é definido por estruturas de dados.

A seção cabeçalho contém atributos para identificação de estruturas de dados, prevenindo a manipulação de nós por estruturas de dados incorretas, e referências para nós adjacentes, permitindo a navegação em estruturas de dados.

A seção propriedades contém atributos de descrição de um nó; por exemplo, número de objetos e nível ou profundidade na estrutura de dados.

A seção índice de entradas contém atributos para localização de objetos; por exemplo, deslocamento (*offset*), nós filhos e identificadores de objetos representativos.

A seção objetos contém um vetor (*array*) de entidades ou chaves em formato serializado, ou sequencial.

A seção espaço disponível contém *bytes* disponíveis para acomodar a extensão de outras seções.

Suportando objetos de tamanho variável (logo, número variável de objetos), as seções são posicionadas da seguinte forma em unidades de armazenamento: seções de tamanho fixo na extremidade inicial (*bytes* mais significativos), seções de tamanho variável na extremidade final (*bytes* menos significativos), e espaço disponível entre extremidades.

Objetos são armazenados em espaço disponível no sentido da extremidade final para a extremidade inicial, mantendo espaço disponível entre extremidades. Em caso de insuficiência de espaço disponível, ocorre adição ou divisão (*split*) de nós, segundo as políticas das estruturas de dados.

Apesar do índice de entradas ser uma seção de tamanho variável, em função do número

de objetos armazenados, é considerada seção de tamanho fixo, devido à menor magnitude e frequência de variação de tamanho em relação aos objetos.

Algumas responsabilidades de Nós são delegadas a `Workspace`, como interações com meios de armazenamento e definição do comprimento da sequência de *bytes* manipulada (tamanho do nó) – arbitrário e constante, idêntico entre os nós em um `Workspace`.

3.1.4 Módulo Dispositivos

O módulo `Dispositivos` (Figura 3.5) define gerenciadores de dispositivos para estruturas de dados.

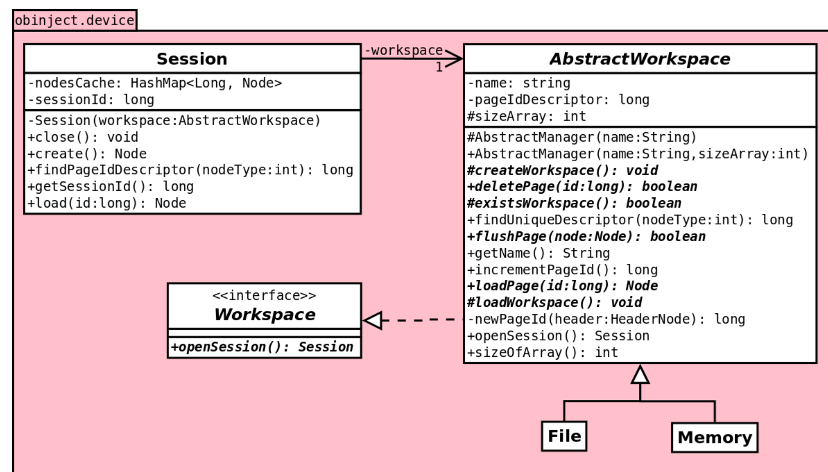

Figura 3.5: *Framework Object-Injection*: Pacote Dispositivos

Um gerenciador de dispositivo é responsável por gerenciar alocação, manipulação e liberação de unidades de armazenamento em meios de armazenamento para estruturas de dados, através de uma interface transparente, independente de detalhes de dispositivos.

Um *workspace* (espaço de trabalho) consiste em um conjunto de unidades de armazenamento disponível para acesso e compartilhamento por estruturas de dados através de operações de leitura e escrita, especificado na interface `Workspace`.

A classe abstrata `AbstractWorkspace` define construtores de inicialização e carregamento da interface `Workspace` e métodos de leitura, escrita e remoção de unidades de armazenamento.

As classes `File` e `Memory` especializam a classe `AbstractWorkspace` como gerenciadores de dispositivos sobre arquivos (meio de armazenamento persistente) e vetores em memória (meio de armazenamento volátil), respectivamente.

A classe `Session` (sessão) representa uma sessão de trabalho temporária sobre um *workspace*, mantendo unidades de armazenamento em *cache*. `Session` é, também, uma abs-

tração sobre dispositivos, especificando apenas métodos de criação e leitura de unidades de armazenando e finalização de sessão, momento em que ocorre escrita de unidades de armazenamento em *cache* para um *workspace*.

A interface `Workspace` é o vínculo entre estruturas de dados e gerenciadores de dispositivos, definido na classe `Structure`.

Uma vez que a interface `Workspace` especifica um método de obtenção de sessão, e é, ainda, implementada pela classe `AbstractWorkspace`, é possível realizar operações através de diferentes níveis de abstração: com sessões (mais abstrato) ou gerenciadores de dispositivo (menos abstrato), de forma flexível para desenvolvedores de aplicações.

3.2 Considerações Finais

Neste capítulo, foi apresentado o *framework Object-Injection*, abrangendo a descrição de conceitos e técnicas empregados em sua implementação, seus módulos de organização e as funcionalidades disponíveis para aplicações.

Dentre os módulos do *framework*, o módulo `Metaclasses`, com as interfaces `Entity`, `Key` e especializações de `Key` (`Order`, `Metric`, `Point`, `Rectangle`, e `StringMetric`), tem significativa relação com o desenvolvimento deste trabalho.

Capítulo 4

Metaprogramação e Metadados para o *Framework Object-Injection*

Este capítulo apresenta uma metodologia para automatizar a implementação de interfaces de persistência e indexação do *framework Object-Injection*, efetuando geração de código de classes auxiliares para aplicações adotando o *framework*.

A metodologia desenvolvida é fundamentada em metaprogramação, combinando metaclasses do *framework* e metadados em classes de aplicações para alimentação de analisadores e geradores de código (Figura 4.1).

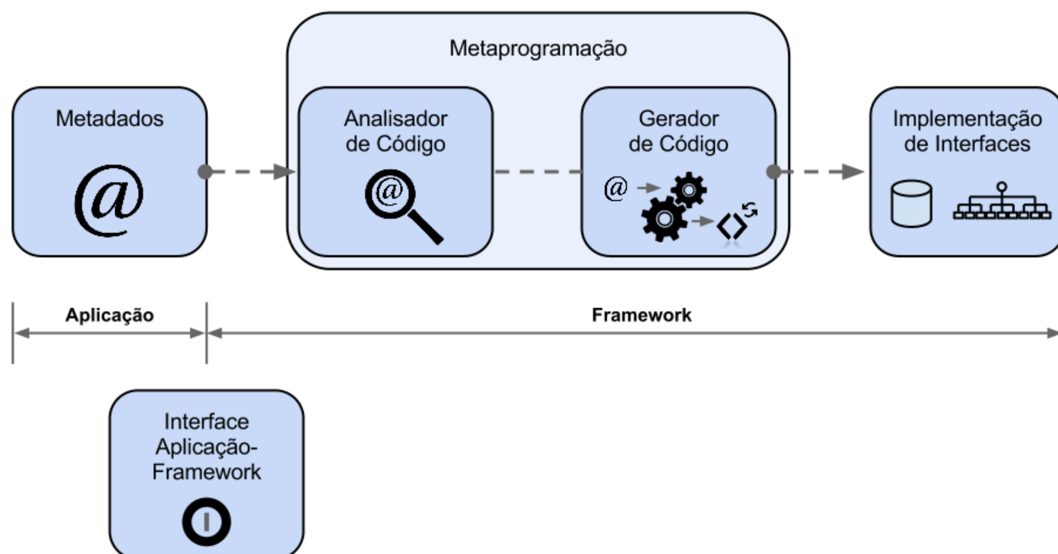

Figura 4.1: Metodologia: Metadados, Analisador e Gerador de Código, Interfaces

Essa metodologia foi incorporada ao *framework* com o módulo Metaprogramação, no pacote `obinject.metaprogramming` (Figura 4.2), contendo validadores e geradores de classes, e a adição de classes de vínculo aplicação-*framework* no pacote raiz `obinject`, contendo interfaces, anotações e gerenciador de entidades (Figura 4.3).

A validação das anotações, validadores e geradores de código, e classes de vínculo foi realizada através de casos de uso.

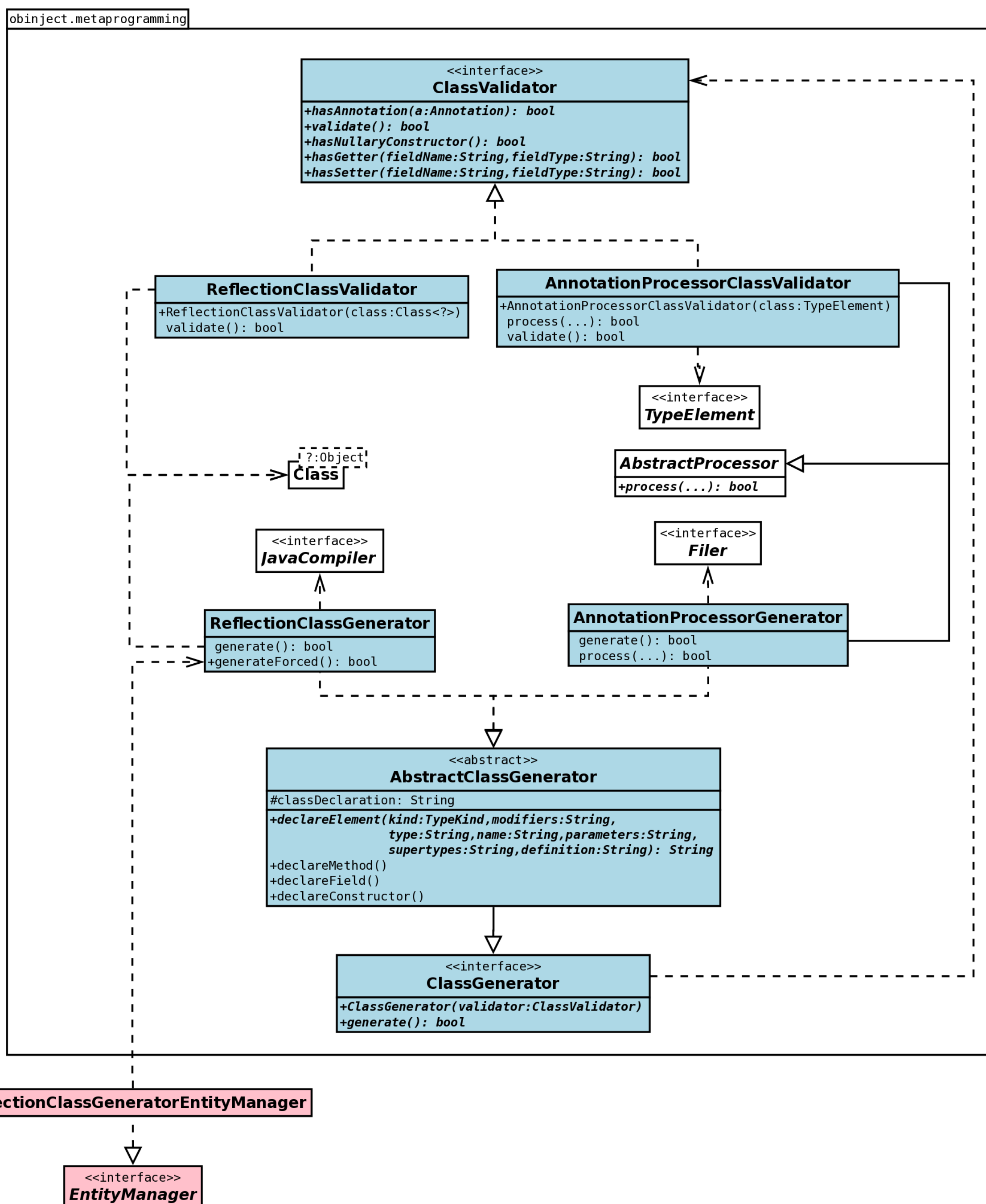

Figura 4.2: Metodologia: Pacote Metaprogramação

4.1 Metadados

Como metaclasses, metadados adicionam significado para o *framework* em classes de aplicações. Entretanto, metadados são menos restritivos e mais significativos que metaclasses, devido

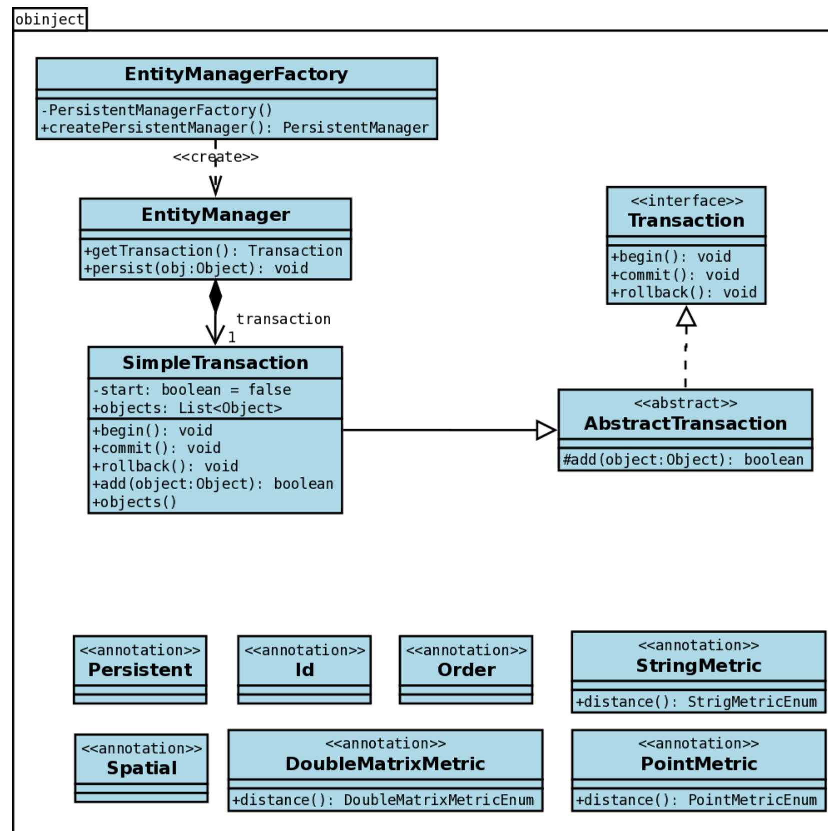

Figura 4.3: Metodologia: Pacote Raiz

à existência de construções de linguagem de programação especializadas à representação de metadados.

Na linguagem de programação Java, metadados são representados através de anotações e tipos-anotação, elementos de sintaxe conveniente para aplicações e acessíveis através da própria linguagem de programação Java.

Consequentemente, através da combinação de metaclasses e metadados, é possível representar propriedades significativas para o *framework* de forma conveniente para a aplicação.

Adicionalmente, através da combinação de analisadores e geradores de código, é possível efetuar geração de código baseado em análise de metadados, automatizando a implementação de classes da aplicação.

A metodologia desenvolvida especifica anotações de persistência e indexação, descritas a seguir.

A anotação de persistência é `@Persistent`. Essa é uma anotação marcadora (*marker*), aplicável a tipos (classes). Um gerador de classes, baseado em uma classe contendo essa anotação, gera uma classe auxiliar com a implementação da interface `Entity` e um índice primário. Exemplos nas Listagens 4.1, 4.2, 4.3 e 4.4.

As anotações de indexação, descritas a seguir, são aplicáveis a atributos (campos). Um gerador de classes, baseado em um atributo anotado com uma anotação de indexação, gera uma classe auxiliar com as implementações da interface `Key` e da especialização correspondente à anotação, bem como um índice secundário. Algumas anotações são marcadoras, outras contém elementos para configuração de indexação. As anotações de indexação são:

- `@Id`: o atributo é a chave primária da classe, com relação de ordem, indexado em árvore B+. Exemplos nas Listagens 4.1, 4.2, 4.3 e 4.4.
- `@Order`: o atributo tem relação de ordem, indexado em árvore B+. Exemplo na Listagem 4.1.
- `@StringMetric (distance={levenshtein, dameraulevenshtein, xuDamerau})`: o atributo tem relação de distância entre cadeias de caracteres, indexado em árvore M; as funções de distância podem ser `levenshtein` e variantes `damerauLevenshtein` (transposição) e `xuDamerau` (proteínas). Exemplo na Listagem 4.2.
- `@DoubleMatrixMetric (distance=veryLeastDistance)`: o atributo tem relação de distância matricial, indexado em árvore M; função de distância pode ser `veryLeastDistance`. Exemplo na Listagem 4.3.
- `@PointMetric (distance={euclidean, manhattan, earthSpherical})`: o atributo tem relação de distância, indexado em árvore M; função de distância pode ser `euclidean` (euclidiana), `manhattan`, e `earthSpherical` (esférica terrestre). Exemplo na Listagem 4.4.
- `@Spatial`: o atributo tem relação espacial, indexado em árvore R. Exemplo similar à anotação `@PointMetric`.

Anotações de persistência e indexação têm políticas de retenção de tempo de execução, permitindo acesso por reflexão para validadores e geradores de classe, e outras ferramentas.

4.2 Análise e Geração de Código

Mecanismos de metaprogramação em Java para análise e geração de código baseado em anotações incluem, em tempo de execução, reflexão (análise) e compilação em execução (geração), e, em tempo de compilação, processadores de anotações (análise e geração).

```java
1 @Persistent
2 public class Aluno {
3
4     @Id
5     private int matricula;
6
7     @Order
8     private String nome;
9
10    private Date datIngresso;
11
12    /* Construtores, Metodos de Acesso (omitidos por brevidade). */
13 }
```

Listagem 4.1: Exemplo de Anotação `@Order`: Classe `Aluno`

```java
1 @Persistent
2 public class Proteina {
3
4     @Id
5     private int codigo;
6
7     @StringMetric(
8         distance = StringMetricEnum.XuDamerau)
9     private String cadeia;
10
11    /* Construtores, Metodos de Acesso (omitidos por brevidade). */
12 }
```

Listagem 4.2: Exemplo de Anotação `@StringMetric`: Classe `Proteina`

```java
1 @Persistent
2 public class Imagem {
3
4     @Id
5     private String arquivo;
6
7     @DoubleMatrixMetric(
8         distance = DoubleMatrixMetricEnum.VeryLeastDistance)
9     private int caracteristicas[][];
10
11    /* Construtores, Metodos de Acesso (omitidos por brevidade). */
12 }
```

Listagem 4.3: Exemplo de Anotação `@DoubleMatrixMetric`: Classe `Imagem`

```
1  @Persistent
2  public class Cidade {
3
4      @Id
5      private int cep;
6
7      @PointMetric(
8          distance = PointMetricEnum.EarthSpherical)
9      private float pontoCentral[];
10
11     private String nome;
12
13     /* Construtores, Metodos de Acesso (omitidos por brevidade). */
14 }
```

Listagem 4.4: Exemplo de Anotação `@PointMetric`: Classe `Imagem`

Implementações provenientes de geração de código podem ser armazenadas em classes auxiliares, conhecidas somente pelo *framework*, evitando intrusão de código em classes originais, tornando o processo de geração de código transparente para a aplicação.

A adoção de metadados e classes auxiliares conduz à uma abordagem conveniente e pouco intrusiva para a aplicação. No contexto de *frameworks*, usabilidade e intrusão são critérios relevantes para adoção e desenvolvimento de aplicações (HOU; RUPAKHETI; HOOVER, 2008; CORRITORE; WIEDENBECK, 2001).

Considerações sobre o momento de execução – tempo de compilação ou execução – de analisadores e geradores de código (combinados para geração de código) conduzem a uma discussão sobre desempenho e flexibilidade.

Adotar geração de código em tempo de execução culmina em custos de desempenho (*performance overhead*), pois é necessário verificar a existência de classes auxiliares e, caso ainda não existam, invocar um processo de compilação em execução, antes de efetuar qualquer operação com a classe em questão.

Tal processo não é necessário com geração de código em tempo de compilação, pois as classes auxiliares são geradas durante a compilação. Consequentemente, qualquer operação com a classe em questão pode ocorrer imediamente, sem verificações, não incorrendo custos de desempenho em execução.

Em relação à flexibilidade, a geração de código em tempo de compilação é dependente do código fonte de classes de aplicações, dado que processadores de anotações são, essencialmente,

analisadores de código. Em algumas situações, o código fonte pode não estar disponível, como aplicações proprietárias, distribuições de pacotes binários etc.

De forma diferente, a geração de código em tempo de execução não depende do código fonte de classes de aplicação, dado que reflexão consiste somente de informações disponíveis em tempo de execução.

Adicionalmente, com geração de código em tempo de execução é possível introduzir classes de aplicação sem classes auxiliares a uma aplicação em tempo de execução, pois a verificação de classes auxiliares e invocação do gerador de classes são executados antes de efetuar operações com as classes em questão, criando as classes auxiliares. A geração de código em tempo de compilação não conta com essa flexibilidade, pois é necessário que todas as classes auxiliares sejam criadas antes da execução da aplicação.

Comparando as alternativas, geração de código em tempo de execução culmina em custos de desempenho devido à etapas de verificação de classes auxiliares e processos de compilação em execução, porém, permite maior flexibilidade, não dependendo do código fonte de classes de aplicações e permitindo introdução de classes de aplicação ainda sem classes auxiliares em tempo de compilação. De forma contrária, geração de código em tempo de compilação não incorre custos de desempenho em execução, porém, permite menos flexibilidade, dependendo do código fonte de classes de aplicações e requerendo a criação de classes auxiliares em tempo de compilação.

Endereçando ambos lados do balanço (*trade-off*) entre desempenho e flexibilidade, foram desenvolvidos geradores de classes de tempo de compilação e tempo de execução (Figura 4.4).

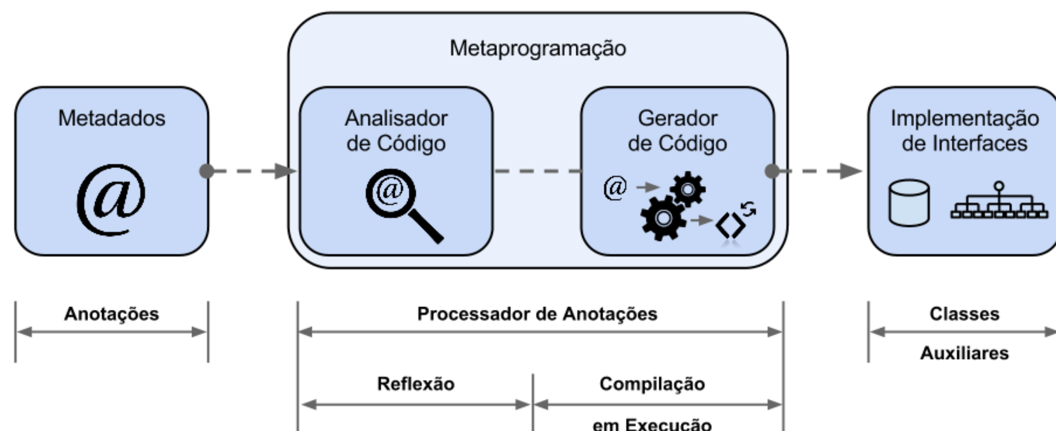

Figura 4.4: Metodologia: Mecanismos de Metaprogramação e Classes Auxiliares

Para garantir que classes da aplicação satisfaçam requisitos de persistência e indexação do

framework (por exemplo, construtor sem argumentos, métodos de acesso a atributos, etc), também foram desenvolvidos validadores de classes, invocados por geradores de classes e responsáveis por lançar erros de validação, abortando o processo de geração de classes.

4.3 Classes de Vínculo

Acrescentando à entidades funcionalidades além da implementação das interfaces de persistência e indexação, através de geradores de classes, é definida a responsabilidade de gerenciamento de índices primários (persistência) e secundários (indexação) para suas instâncias, criando índices gerenciados por entidades (*entity-managed indices*), semelhantemente à *JavaBeans* (*bean-managed persistence*).

A implementação dessa responsabilidade é fundamentada em atributos de escopo de classe representando índices primários e secundários da entidade, especificados através de anotações de indexação. Dessa forma, todas instâncias de uma entidade tem acesso a seus índices de persistência e indexação.

De forma a simplificar a interação aplicação-*framework*, considerando anotações, geradores de classes e entidades com gerenciamento de índices, é especificada a interface `EntityManager`, encapsulando geração de classes e gerenciamento de entidades.

Um gerenciador de entidades, em relação à geração de classes, pode, opcionalmente, realizar a verificação de existência de classes auxiliares em tempo de execução, e, em caso negativo, invocar um gerador de classes de tempo de execução, gerando a classe auxiliar; em relação a índices gerenciados por entidades, um gerenciador de entidades pode, opcionalmente, redirecionar operações de persistência e indexação para a classe de entidades em questão. Adicionalmente, essa interface também inclui métodos para transações, em suporte futuro a um mecanismo de transações com interface simples.

4.4 Primeiro Caso de Uso

Como primeiro caso de uso para apresentação dos conceitos deste trabalho é definida a classe `MeuExemplo` (Listagem 4.5), contendo 13 atributos de diversos tipos de dados (`boolean`, `byte`, `char`, `Calendar`, `Date`, `double`, `float`, `int`, `long`, `short`, `List<MeuExemplo>`, `String` e matriz de `double`), demostrando os *tipos primitivos* e alguns *tipos referência* para armazenamento de objetos no *framework*.

A classe `MeuExemplo` contém as anotações `@Persistent` e `@Id`. Devido à anotação `@Persistent`, uma entidade dessa classe se tornará persistente em operações do gerenciador de entidades (`EntityManager`). Devido à anotação `@Id`, o atributo `campo12` será a chave primária dessa entidade.

```
1  package obinject.runtime.test;
2
3  import java.util.*;
4  import obinject.Id;
5  import obinject.Persistent;
6
7  @Persistent
8  public class MeuExemplo
9  {
10     private boolean campo1;
11     private byte campo2;
12     private char campo3;
13     private Calendar campo4;
14     private Date campo5;
15     private double campo6;
16     private float campo7;
17     private int campo8;
18     private long campo9;
19     private short campo10;
20     private List<MeuExemplo> campo11 = new ArrayList<MeuExemplo>();
21     @Id
22     private String campo12;
23     private double[][] campo13;
24
25     /* Metodos de Acesso:
26        Getters e Setters.
27        (omitidos por brevidade) */
28 }
```

Listagem 4.5: Primeiro Caso de Uso: Classe `MeuExemplo`

Uma aplicação de teste gerou 1.000 objetos da classe `MeuExemplo`. Cada objeto teve valores aleatórios atribuídos a seus atributos. Durante a execução, a geração das classes `EntityMeuExemplo` (Listagem 4.6) e `PrimaryKeyMeuExemplo` (Listagem 4.7) ocorreu automaticamente, com o gerador de classes de tempo de execução.

A Figura 4.5 apresenta o diagrama de objetos desse caso de uso. Inicialmente a aplicação cria um objeto da classe `PersistEntityManager` atráves da classe `EntityManagerFactory`.

Em um *loop* repetido 1.000 vezes, o método `SimpleTransaction.begin` é invocado para

iniciar uma transação. A classe `SimpleTransaction` receberá um objeto da classe `MeuExemplo` através do método `EntityManager.persist`. Ainda nesse *loop*, o método `SimpleTransaction.commit` é invocado para confirmar a transação.

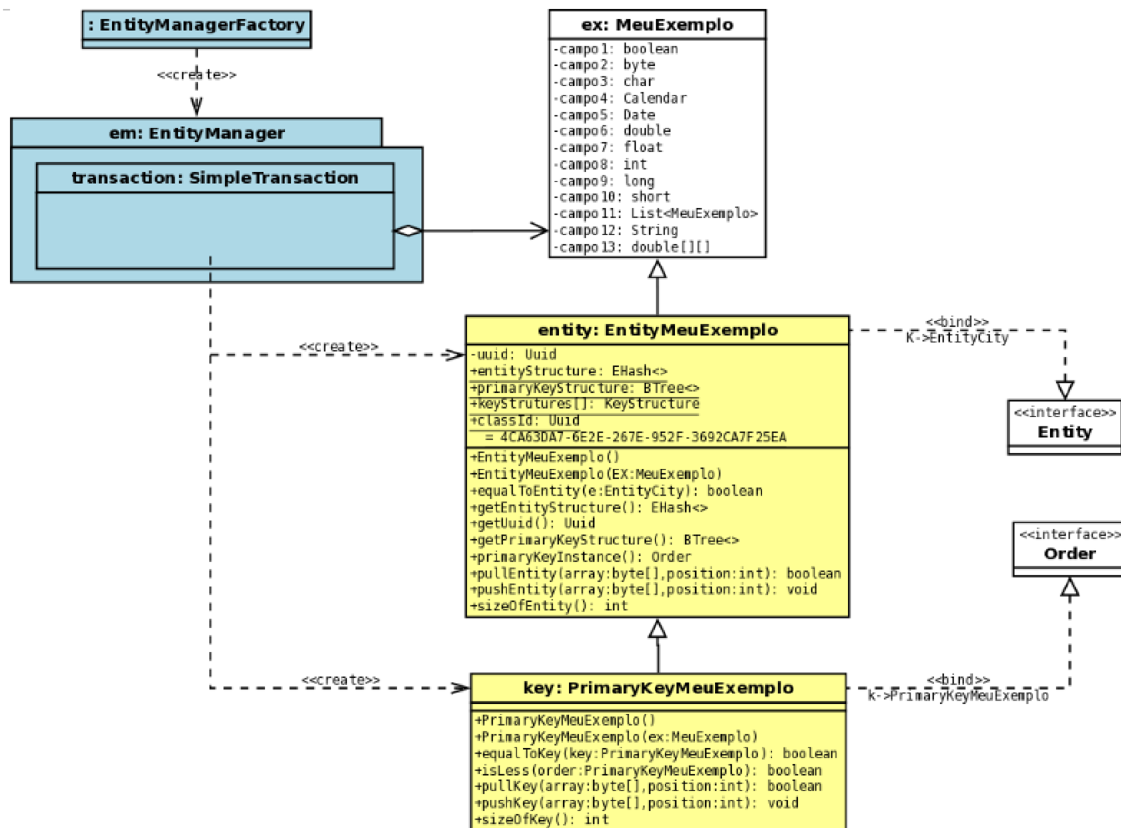

Figura 4.5: Metodologia: Primeiro Caso de Uso

A Listagem 4.6 apresenta a classe `EntityMeuExemplo` que contém 126 linhas de código. Essa classe contém os seguintes 5 atributos: `classId`, `uuid`, `EntityStructure`, `primaryKeyStructure` e `keyStrututures`.

O atributo estático `classId` define um identificador UUID para a classe `MeuExemplo`. O atributo estático `EntityStructure` determina a estrutura de armazenamento para os objetos da classe `MeuExemplo`. O atributo estático `primaryKeyStructure` determina a estrutura de armazenamento para a chave primária dos objetos da classe `MeuExemplo`. O atributo estático `keyStructures` é um vetor que determina todas as chaves secundárias definidas para a classe `MeuExemplo`. Finalmente, o atributo `uuid` define um identificador UUID para o objeto da classe `MeuExemplo`.

A Listagem 4.6 apresenta 2 construtores e 8 métodos para a classe `EntityMeuExemplo`: `equalToEntity`, `getEntityStructure`, `getUuid`, `getPrimaryKeyStructure`,

```
1  package obinject.runtime.test;
2
3  import obinject.device.File;
4  import obinject.metaclass.Entity;
5  import obinject.metaclass.Order;
6  import obinject.metaclass.Uuid;
7  import obinject.storage.BTree;
8  import obinject.storage.KeyStructure;
9  import obinject.storage.Sequential;
10
11 public  class        EntityMeuExemplo
12         extends      MeuExemplo
13         implements   Entity<EntityMeuExemplo> {
14
15     public static final Uuid classId =
16             Uuid.fromString("FDE88657-DA18-23D1-0E5F-2B2020F02BD1");
17
18     public static final EHash<EntityMeuExemplo> entityStructure =
19             new EHash<EntityMeuExemplo>(
20                 new File("build/classes/obinject/runtime/test/MeuExemplo.dbo", 1024)) {};
21
22     public static final BTree<PrimaryKeyMeuExemplo> primaryKeyStructure =
23             new BTree<PrimaryKeyMeuExemplo>(
24                 new File("build/classes/obinject/runtime/test/MeuExemplo.pk", 1024)) {};
25
26     public static final KeyStructure keyStrutures[] = new KeyStructure[0];
27     private Uuid uuid = Uuid.generator();
28
29                 public EntityMeuExemplo() {...}
30                 public EntityMeuExemplo(MeuExemplo obj) {...}
31     @Override   public boolean equalToEntity(EntityMeuExemplo obj) {...}
32     @Override   public EHash getEntityStructure() {...}
33     @Override   public Uuid getUuid() {...}
34     @Override   public BTree getPrimaryKeyStructure() {...}
35     @Override   public Order primaryKeyInstance() {...}
36     @Override   public boolean pullEntity(byte[] array, int position) {...}
37     @Override   public void pushEntity(byte[] array, int position) {...}
38     @Override   public int sizeOfEntity() {...}
39 }
```

Listagem 4.6: Primeiro Caso de Uso: Classe `EntityMeuExemplo`

primaryKeyInstance, pullEntity, pushEntity e sizeOfEntity.

O método equalToEntity é responsável por responder se dois objetos MeuExemplo são iguais. Os métodos getEntityStructure, getUuid e getPrimaryKeyStructure permitem acesso aos atributos da classe a partir da instância. O método pullEntity *puxa* (*pull*) do vetor de *bytes* os valores dos atributos da classe. Por outro lado, o método pushEntity *empurra* (*push*) para o vetor de *bytes* os valores dos atributos da classe. Finalmente, o método sizeOfEntity retorna o tamanho em *bytes* dos valores dos atributos da classe MeuExemplo.

A Listagem 4.7 apresenta a classe PrimaryKeyMeuExemplo, responsável por definir a chave única para classe MeuExemplo através de indexação do atributo campo12. Essa classe apresenta 2 construtores e 5 métodos sendo: equalToKey, isLess, pullKey, pushKey e sizeOfKey.

```
1  package obinject.runtime.test;
2
3  import obinject.metaclass.Order;
4  import obinject.metaclass.PullStream;
5  import obinject.metaclass.PushStream;
6  import obinject.metaclass.Stream;
7
8  public class      PrimaryKeyMeuExemplo
9         extends    EntityMeuExemplo
10        implements Order<PrimaryKeyMeuExemplo> {
11
12                public PrimaryKeyMeuExemplo() {...}
13                public PrimaryKeyMeuExemplo(MeuExemplo obj) {...}
14     @Override  public boolean equalToKey(PrimaryKeyMeuExemplo obj) {...}
15     @Override  public boolean isLess(PrimaryKeyMeuExemplo obj) {...}
16     @Override  public boolean pullKey(byte[] array, int position) {...}
17     @Override  public void pushKey(byte[] array, int position) {...}
18     @Override  public int sizeOfKey() {...}
19 }
```

Listagem 4.7: Primeiro Caso de Uso: Classe PrimaryKeyMeuExemplo

Os métodos equalToKey, pullKey, pushKey e sizeOfKey têm a mesma responsabilidade que os definidos para a classe EntityMeuExemplo, com a diferença que tratam somente o atributo que define chave única da classe MeuExemplo. O método isLess é responsável por responder se um objeto MeuExemplo tem chave menor que outro objeto do mesmo tipo.

4.5 Segundo Caso de Uso

O segundo caso de uso ilustra a modelagem para o filme A Origem[1], constituindo número razoável de classes para um exemplo de *tipos referência*. O cenário e o diagrama de classe em UML para esse segundo caso de uso serão descrito nas próximas seções.

As 14 classes modeladas e codificadas para esse cenário foram: `Sombra`, `Passiv`, `Missao`, `Totem`, `Pessoa`, `Sonhador`, `Sonho`, `Extrator`, `Arquiteto`, `Armador`, `Alvo`, `Falsificador`, `Turista` e `Quimico`.

Foi desenvolvida uma aplicação de teste (Listagem 4.8) para instanciar as classes. Cada uma das 14 classes recebeu as anotações `@Persistent` e `@Id`, resultando na geração de 28 classes (14 implementações da interface `Entity` e 14 implementações da interface `Order`).

As classes `Pessoa`, `Sonho` e suas especializações, implementações das interfaces `Entity` e `Order`, são apresentadas no Apêndice A. Essas classes são representativas do padrão de geração de código adotado, sendo, as demais classes, essencialmente repetições similares.

4.5.1 Cenário

Uma missão é constituída por vários sonhadores e tem um objetivo que pode ou não ser cumprido.

Os sonhadores podem ser: extrator, arquiteta, armador, falsificador, químico, turista e alvo. O extrator, ao final da operação, tem a informação extraída do alvo e pode ou não ter implantado alguma ideia no alvo. A arquiteta usa um estilo arquitetônico para construir o cenário do sonho (descrito como labirinto). O armador enfrenta uma quantidade de problemas durante o sonho para que tudo saia de acordo com os planos. O falsificador assume a forma de uma outra pessoa durante o sonho. O químico usa uma fórmula para compor uma droga para sustentar o nível do sonho. O turista é qualquer pessoa que paga um valor monetário para se juntar a equipe. Em relação ao alvo, ao final da operação, pode ou não ter sido extraído algum tipo de informação ou sido implantada alguma ideia.

Um sonhador é parte de uma pessoa como por exemplo: Dom Cobb (um ladrão), Ariadne (uma estudante), Eames (um trapaceiro), Arthur (um nerd), Yusuf (um biólogo), Saito (um empresário), Robert Fischer (presidente de uma companhia), Peter Browning (executivo de uma companhia). Os sonhadores podem ou não estar vivos e formam um sonho (compartilhado).

[1] A Origem (*Inception*) com direção de *Christopher Nolan* da *Warner Bros* em 2010.

```java
package obinject.sample.inception;

import obinject.EntityManager;
import obinject.EntityManagerFactory;

public class AppGeneratorInception {

    public static void main(String[] args) {

        EntityManager em = EntityManagerFactory.createEntityManager();
        em.getTransaction().begin();

        /* Instanciacao e Persistencia:
           Classe Pessoa. */
        Pessoa pessoa = new Pessoa();
        em.persist(pessoa);

        /* Instanciacao e Persistencia:
           Classe Totem. */
        Totem totem = new Totem();
        em.persist(totem);

        /* Instanciacao e Persistencia:
           Classes Alvo, Armador, Arquiteto,
           Extrator, Falsificador, Missao,
           Pasiv, Quimico, Sombra,
           Sonhador, Sonho e Turista
           (omitidas por brevidade.) */

        em.getTransaction().commit();
    }
}
```

Listagem 4.8: Segundo Caso de Uso: Aplicação de teste

Todo sonho usa máquina PASIV (*Portable Automated Somnacin IntraVenous*) que injeta uma quantidade de droga nos sonhadores. Essa quantidade de droga estabelece o tempo do sonho desde seu início até seu término.

Para entrar na mente de uma pessoa, é preciso que um sonhador construa o espaço do sonho, o qual será povoado pelo inconsciente da vítima por sombras (ou projeções). As sombras são representações humanas masculinas ou femininas e podem ou não ter sido treinadas com algum armamento para buscar e eliminar quem está modificando o sonho, que é geralmente o arquiteto.

Os sonhadores podem fazer ou não mudanças no sonho usando apenas o pensamento. Por esse motivo é muito difícil discernir entre o sonho e a realidade. Assim, toda pessoa tem um totem que é um artefato que realiza uma determinada ação e é usado para distinguir entre a realidade e o sonho.

Para acordar uma pessoa em um sonho é usada a sensação de queda. Essa técnica também chamada de chute (ou pontapé) é combinada em toda missão, junto com o aviso prévio do chute. O aviso prévio é o som de música disparada por quem não está no sonho para sincronizar o acordar de uma equipe do sonho. A música repercute no subconsciente e assim pode-se calcular que o chute virá a qualquer instante.

Em sonhos de mais de um nível, o sonhador que construiu o espaço do sonho fica para trás para sincronizar os chutes sincronizados. Por exemplo, no primeiro nível Yusuf dirige uma van para fora de uma ponte, no segundo nível Arthur explode um elevador com os corpos da equipe em gravidade zero, e no terceiro nível Eames detona explosivos em uma fortaleza na montanha.

Um sonho pode estar dento de outro sonho, formando níveis de sonhos. Por exemplo, estar no segundo nível, significa entrar num sonho dentro de outro sonho, e estar no terceiro nível significa um sonho dentro de outro sonho dentro de outro sonho. A cada nível de sonho aprofunda-se mais no subconsciente do alvo e o tempo fica 12 vezes mais lento. Por exemplo, 5 minutos no mundo real equivalem a 1 hora no primeiro nível, que equivale as 12 horas no segundo nível, que equivale a 144 horas no terceiro nível (6 dias), que equivalem a 1.728 horas no quarto nível (72 dias).

O limbo é o último nível do sonho (um espaço de sonho inacabado) e um minuto de vida normal pode simbolizar dez anos. Para ir ao limbo, uma pessoa deve morrer durante um sonho. Para escapar do limbo você deve realmente acreditar que o mundo em que está é falso.

4.5.2 Diagrama de Classe UML

A Figura 4.6 apresenta o diagrama UML proposto para o cenário do filme A Origem, com 14 classes.

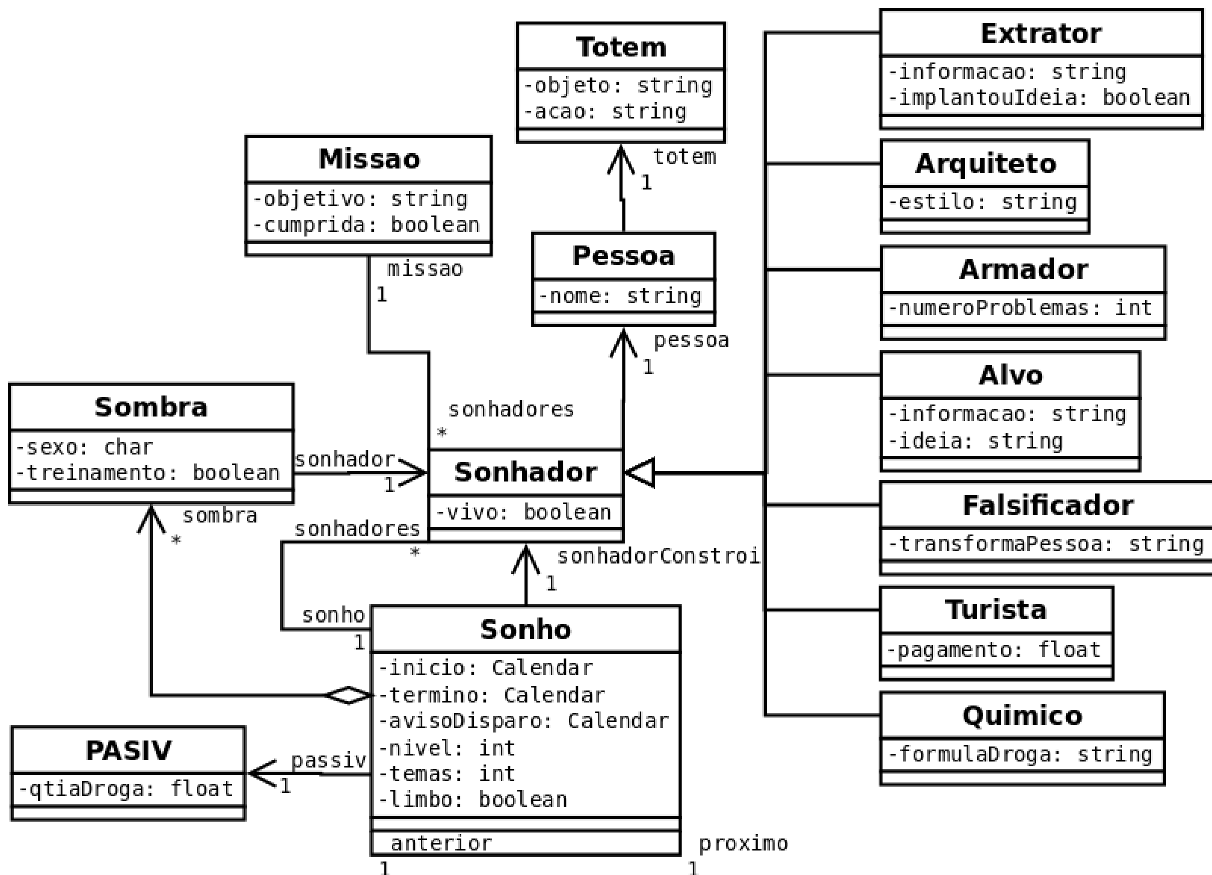

Figura 4.6: Metodologia: Segundo Caso de Uso

4.6 Considerações Finais

Neste capítulo foi apresentada a metodologia desenvolvida para automatizar a implementação de interfaces de persistência e indexação do *framework Object-Injection*, consistindo na combinação de metaclasses do *framework* e metadados em classes de aplicações para alimentação de analisadores e geradores de código, responsáveis por gerar classes auxiliares para aplicações, contendo implementações das interfaces de persistência e indexação requeridas pelo *framework*.

Com a aplicação de mecanismos de metaprogramação para a linguagem de programação Java, como anotações, processadores de anotações e reflexão, foram desenvolvidas anotações de persistência e indexação, utilizadas em classes de aplicações, e validadores e geradores de classes, combinados para gerar classes auxiliares baseados em anotações.

Adicionalmente, as classes de vínculo aplicação-*framework* foram simplificadas, utilizando técnicas desenvolvidas, como índices gerenciados por entidades, geradores de classes e suporte básico à transações.

A validação das técnicas desenvolvidas foi realizada através de 2 casos de uso, abrangendo tipos primitivos e tipos referência.

Capítulo 5 Conclusões

Este trabalho demonstra a possibilidade de automatizar a implementação de interfaces de persistência e indexação do *framework Object-Injection*, efetuando geração de código para aplicações adotando o *framework*, com as seguintes contribuições:

- **Metodologia de automação de implementações**. Utilizando mecanismos de metaprogramação, metaclasses do *framework* e metadados da aplicação são combinados para alimentar analisadores e geradores de código, com mínima intrusão em código de classes de aplicações através de anotações para adição de metadados e classes auxiliares para conter implementações.

- **Anotações de persistência e indexação**. Através de anotações, classes de aplicações podem relacionar e configurar certas funcionalidades do *framework* necessárias à sua operação de forma legível e elegante.

- **Validadores de classes de aplicações**. Analisadores de código realizam verificações sobre anotações, atributos e métodos de classes de aplicações, garantindo que requisitos necessários à operação do *framework* sejam satisfeitos.

- **Geradores de classes auxiliares para aplicações**. Utilizando informações presentes em classes de aplicações através de metadados, geradores de código realizam a construção de classes auxiliares para as aplicações, contendo implementações de interfaces de persistência e indexação requeridas para adoção do *framework*.

- **Validadores e geradores de tempo de compilação e tempo de execução**. Permitindo o favorecimento de desempenho ou flexibilidade durante a execução de aplicações, ambas

ferramentas contam com implementações de tempo de compilação (favorecendo desempenho) e tempo de execução (favorecendo flexibilidade).

- **Demonstração e validação**. Casos de uso com tipos primitivos e tipos referência, acompanhados de especificação e diagramas, foram utilizados para demonstrar e validar a metodologia e as ferramentas desenvolvidas.
- **Índices gerenciados por entidades**. Transferindo para entidades a responsabilidade de gerenciamento de seus índices, as funcionalidades de persistência e indexação obtiveram melhorias de encapsulamento.
- **Classes de vínculo**. Simplificação da interação entre aplicação e *framework*, através de anotações, gerenciadores de entidades, sessões e transações.
- **Incorporação ao *framework Object-Injection***. Com a incorporação do módulo Metaprogramação e classes de vínculo, as contribuições deste trabalho são estendidas à aplicações adotando o *framework*.

5.1 Trabalhos Futuros

Propondo a continuidade do desenvolvimento deste trabalho, são apresentadas as seguintes sugestões para trabalhos futuros:

- **Verificação de classes auxiliares em tempo de definição**. Um transformador sem capacidade de retransformação permite uma eficiente verificação de classe auxiliares (e possível invocação de um gerador de classes), ocorrendo somente uma vez por classe (o mínimo necessário), em tempo de definição de classes.
- **Gerador de classes por instrumentação**. Através de instrumentação de *Java bytecode*, um gerador de classes pode introduzir, diretamente em classes de aplicação, suas implementações, dispensando classes auxiliares e decorrentes conversões de tipo, tornando a utilização do *framework* mais transparente para a aplicação.
- **Suporte à transações**. Um mecanismo transacional, com suporte à operações de inicialização (*begin*), confirmação (*commit*) e cancelamento (*rollback*) de transações, especificadas em classes de vínculo, permitiria garantias de consistência em estruturas de dados do *framework*.

- **Anotações para chaves compostas**. Com o desenvolvimento de anotações relacionando múltiplos atributos, aplicações poderiam empregar chaves compostas de forma simples, através do mecanismo de anotações e classes de vínculo.

- **Compatibilidade com classes sem anotações**. Provendo suporte a classes legadas, sem anotações de persistência e indexação, ainda é possível gerar classes auxiliares, utilizando alternativas à anotações para configuração equivalente e reflexão para garantir independência de código fonte.

Referências Bibliográficas

ATTARDI, G.; CISTERNINO, A. Reflection support by means of template metaprogramming. In: *Proceedings of the Third International Conference on Generative and Component-Based Software Engineering*. London, UK: Springer-Verlag, 2001.

BERLER, M. et al. *The object data standard: ODMG 3.0*. San Francisco, CA, USA: Morgan Kaufmann Publishers, Inc., 2000.

BIESACK, D. J. *Create dynamic applications with javax.tools*. IBM developerWorks, Dezembro 2007. Disponível em: <http://ibm.com/developerworks/java/library/j-jcomp>.

BRACHA, G. et al. Making the future safe for the past: adding genericity to the java programming language. *SIGPLAN Not.*, ACM, New York, NY, USA, v. 33, n. 10, Outubro 1998.

BRACHA, G.; UNGAR, D. Mirrors: design principles for meta-level facilities of object-oriented programming languages. *SIGPLAN Not.*, ACM, New York, NY, USA, v. 39, n. 10, Outubro 2004.

CAMPIONE, M.; WALRATH, K. *About the Java Technology*. Oracle Technology Network Documentation, Outubro 2012. Disponível em: <http://docs.oracle.com/javase/tutorial/getStarted/intro/definition.html>.

CARVALHO, L. O. et al. Obinject: a noodmg indexing and persistence framework. In: *Proceedings of the 2013 ACM symposium on Applied computing (a ser publicado)*. New York, NY, USA: ACM, 2013.

CHIBA, S. Load-time structural reflection in java. In: *Proceedings of the 14th European Conference on Object-Oriented Programming*. London, UK: Springer-Verlag, 2000.

CIACCIA, P.; PATELLA, M.; ZEZULA, P. M-tree: An efficient access method for similarity search in metric spaces. In: *Proceedings of the 23rd International Conference on Very Large Data Bases*. San Francisco, CA, USA: Morgan Kaufmann Publishers, Inc., 1997.

COMER, D. Ubiquitous b-tree. *ACM Comput. Surv.*, ACM, New York, NY, USA, v. 11, n. 2, Junho 1979.

CORRITORE, C. L.; WIEDENBECK, S. An exploratory study of program comprehension strategies of procedural and object-oriented programmers. *Int. J. Hum.-Comput. Stud.*, Academic Press, Inc., Duluth, MN, USA, v. 54, n. 1, Janeiro 2001.

DARWEN, H.; DATE, C. J. The third manifesto. *SIGMOD Rec.*, ACM, New York, NY, USA, v. 24, n. 1, p. 39–49, Março 1995.

DATE, C. *An Introduction to Database Systems*. 8. ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003.

EVANS, I. *Your First Cup: An Introduction to the Java EE Platform*. Abril 2012. Disponível em: <http://docs.oracle.com/javaee/6/firstcup/doc/gkhoy.html>.

FAGIN, R. et al. Extendible hashing – a fast access method for dynamic files. *ACM Trans. Database Syst.*, ACM, New York, NY, USA, v. 4, n. 3, Setembro 1979.

GARCIA-MOLINA, H.; ULLMAN, J. D.; WIDOM, J. *Database Systems: The Complete Book*. 2. ed. Upper Saddle River, NJ, USA: Prentice Hall Press, 2008.

GOSLING, J. et al. *The Java Language Specification (Java SE 7 Edition)*. Julho 2012. Disponível em: <http://docs.oracle.com/javase/specs/jls/se7/html>.

GUTTMAN, A. R-trees: a dynamic index structure for spatial searching. In: *Proceedings of the 1984 ACM SIGMOD international conference on Management of data*. New York, NY, USA: ACM, 1984.

HARRINGTON, J. *Object-Oriented Database Design Clearly Explained*. San Francisco, CA, USA: Morgan Kaufmann Publishers, Inc., 2000.

HOU, D.; RUPAKHETI, C. R.; HOOVER, H. J. Documenting and evaluating scattered concerns for framework usability: A case study. In: *Proceedings of the 2008 15th Asia-Pacific Software Engineering Conference*. Washington, DC, USA: IEEE Computer Society, 2008.

ISO (International Organization for Standardization). *9834-8:2004 Generation and registration of Universally Unique Identifiers (UUIDs) and their use as ASN.1 Object Identifier components*. Geneva, Switzerland: [s.n.], Setembro 2004. ITU-T Recommendation X.667.

JOHNSON, S. C. *Yacc: Yet Another Compiler-Compiler*. Murray Hill, New Jersey, 1979.

JSR-12. *Java Data Objects (JDO) Specification*. Setembro 2003. Disponível em: <http://jcp.org/en/jsr/detail?id=12>.

JSR-199. *Java Compiler API*. Dezembro 2006. Disponível em: <http://jcp.org/en/jsr/detail?id=199>.

JSR-220. *Enterprise JavaBeans 3.0*. Maio 2006. Disponível em: <http://jcp.org/en/jsr/detail?id=220>.

JSR-243. *Java Data Objects 2.0 - An Extension to the JDO specification*. Maio 2006. Disponível em: <http://jcp.org/en/jsr/detail?id=243>.

JSR-316. *Java Platform, Enterprise Edition 6 (Java EE 6) Specification*. Dezembro 2009. Disponível em: <http://jcp.org/en/jsr/detail?id=316>.

JSR-317. *Java Persistence API 2.0*. Dezembro 2009. Disponível em: <http://jcp.org/en/jsr/detail?id=317>.

LEACH, P. J.; MEALLING, M.; SALZ, R. *A Universally Unique IDentifier (UUID) URN Namespace*. Julho 2005. Internet RFC 4122.

LIANG, S. *Java Native Interface: Programmer's Guide and Reference*. 1. ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

LINDHOLM, T. et al. *The Java Virtual Machine Specification (Java SE 7 Edition)*. Julho 2012. Disponível em: <http://docs.oracle.com/javase/specs/jvms/se7/html/>.

OLIVÉ, A. *Conceptual Modeling of Information Systems*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2007.

ORACLE. *Java Platform Standard Edition 7 Documentation*. Outubro 2012. Disponível em: <http://docs.oracle.com/javase/7/docs/>.

ORACLE. *Java Virtual Machine Tool Interface*. Outubro 2012. Disponível em: <http://docs.oracle.com/javase/7/docs/technotes/guides/jvmti/>.

RAMAKRISHNAN, R.; GEHRKE, J. *Database Management Systems*. 3. ed. New York, NY, USA: McGraw-Hill, Inc., 2003.

SADZIAK, M. *Dynamic in-memory compilation*. JavaBlogging, Agosto 2009. Disponível em: <http://www.javablogging.com/dynamic-in-memory-compilation>.

SKALSKI, K.; MOSKAL, M.; OLSZTA, P. *Meta-programming in Nemerle*. 2004. Disponível em: <http://nemerle.org/metaprogramming.pdf>.

STROUSTRUP, B. *The design and evolution of C++*. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1994.

TAHA, W.; SHEARD, T. Metaml and multi-stage programming with explicit annotations. In: *Theoretical Computer Science*. New York, NY, USA: ACM, 1999. p. 203–217.

TAHA, W.; SHEARD, T. Metaml and multi-stage programming with explicit annotations. *Theor. Comput. Sci.*, Elsevier Science Publishers Ltd., Essex, UK, v. 248, n. 1-2, Outubro 2000.

ZAHN, L.; DINEEN, T.; LEACH, P. *Network computing architecture*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1990.

ZEZULA, P. et al. *Similarity Search: The Metric Space Approach*. London, UK: Springer, 2005.

Apêndice A
Segundo Caso de Uso

A.1 Pessoa

A.1.1 Classe Pessoa

```
package obinject.sample.inception;

import obinject.Id;
import obinject.Persistent;

@Persistent
public class Pessoa {

    @Id
    private int codigo;

    private String nome;
    private Totem totem;

    public int getCodigo() {
        return codigo;
    }

    public void setCodigo(int codigo) {
        this.codigo = codigo;
    }

    public String getNome() {
        return nome;
    }

    public void setNome(String nome) {
        this.nome = nome;
    }
```

```
30
31     public Totem getTotem() {
32         return totem;
33     }
34
35     public void setTotem(Totem totem) {
36         this.totem = totem;
37     }
38 }
```

Listagem A.1: Segundo Caso de Uso: Classe `Pessoa`

A.1.2 Classe EntityPessoa

```
1  package obinject.sample.inception;
2
3  import obinject.device.File;
4  import obinject.metaclass.*;
5  import obinject.storage.*;
6
7  public class       EntityPessoa
8         extends     Pessoa
9         implements  Entity<EntityPessoa> {
10
11     public static final Uuid classId =
12         Uuid.fromString("D1A00DFE-F5DA-D61D-25F3-48E7073F7CDB");
13
14     public static final EHash<EntityPessoa> entityStructure =
15         new EHash<EntityPessoa>(
16             new File("build/classes/obinject/sample/inception/Pessoa.dbo", 1024)) {};
17
18     public static final BTree<PrimaryKeyPessoa> primaryKeyStructure =
19         new BTree<PrimaryKeyPessoa>(
20             new File("build/classes/obinject/sample/inception/Pessoa.pk", 1024)) {};
21
22     public static final KeyStructure keyStrutures[] = new KeyStructure[0];
23     private Uuid uuid = Uuid.generator();
24
25     public EntityPessoa() {}
26
27     public EntityPessoa(Pessoa obj) {
28         this.setCodigo(obj.getCodigo());
29         this.setNome(obj.getNome());
30         this.setTotem(obj.getTotem());
31     }
32
33     @Override
34     public boolean equalToEntity(EntityPessoa obj) {
35         return (this.getCodigo() == obj.getCodigo())
36             && (this.getNome().equals(obj.getNome()));
37     }
38
39     @Override
40     public EHash getEntityStructure() {
41         return entityStructure;
42     }
43
44     @Override
45     public Uuid getUuid() {
46         return uuid;
```

```
47      }
48
49      @Override
50      public BTree getPrimaryKeyStructure() {
51          return primaryKeyStructure;
52      }
53
54      public Order primaryKeyInstance() {
55          return new PrimaryKeyPessoa(this);
56      }
57
58      @Override
59      public boolean pullEntity(byte[] array, int position) {
60
61          PullStream pull = new PullStream(array, position);
62          Uuid storedClass = pull.pullUuid();
63
64          if (classId.equals(storedClass) == true) {
65              uuid = pull.pullUuid();
66              this.setCodigo(pull.pullInt());
67              this.setNome(pull.pullString());
68              return true;
69          }
70
71          return false;
72      }
73
74      @Override
75      public void pushEntity(byte[] array, int position) {
76
77          PushStream push = new PushStream(array, position);
78          push.pushUuid(classId);
79          push.pushUuid(uuid);
80          push.pushInt(this.getCodigo());
81
82          if (this.getNome() != null) {
83              push.pushString(this.getNome().getBytes());
84          } else {
85              push.pushInt(0);
86          }
87
88          if (this.getTotem() != null) {
89              if (this.getTotem() instanceof Entity) {
90                  push.pushEntity((Entity) this.getTotem());
91              } else {
92                  push.pushEntity(new EntityTotem(this.getTotem()));
93              }
94          } else {
95              push.pushUuid(Uuid.fromString("00000000-0000-0000-0000-000000000000"));
```

```
96          }
97      }
98
99      @Override
100     public int sizeOfEntity() {
101         return  Stream.sizeOfUuid
102             +   Stream.sizeOfUuid
103             +   Stream.sizeOfInt
104             +   Stream.sizeOfString(this.getNome())
105             +   Stream.sizeOfUuid;
106     }
107 }
```

Listagem A.2: Segundo Caso de Uso: Classe `EntityPessoa`

A.1.3 Classe PrimaryKeyPessoa

```
1  package obinject.sample.inception;
2
3  import obinject.metaclass.Order;
4  import obinject.metaclass.PullStream;
5  import obinject.metaclass.PushStream;
6  import obinject.metaclass.Stream;
7
8  public  class PrimaryKeyPessoa extends EntityPessoa implements Order<PrimaryKeyPessoa> {
9
10     public PrimaryKeyPessoa() {}
11
12     public PrimaryKeyPessoa(Pessoa obj) {
13         this.setCodigo(obj.getCodigo());
14     }
15
16     @Override
17     public boolean equalToKey(PrimaryKeyPessoa obj) {
18         return (this.getCodigo() == obj.getCodigo());
19     }
20
21     @Override
22     public boolean isLess(PrimaryKeyPessoa obj) {
23         return (this.getCodigo() < obj.getCodigo());
24     }
25
26     @Override
27     public boolean pullKey(byte[] array, int position) {
28         PullStream pull = new PullStream(array, position);
29         this.setCodigo(pull.pullInt());
30         return true;
31     }
32
33     @Override
34     public void pushKey(byte[] array, int position) {
35         PushStream push = new PushStream(array, position);
36         push.pushInt(this.getCodigo());
37     }
38
39     @Override
40     public int sizeOfKey() {
41         return Stream.sizeOfUuid
42             +  Stream.sizeOfUuid
43             +  Stream.sizeOfInt;
44     }
45 }
```

Listagem A.3: Segundo Caso de Uso: Classe PrimaryKeyPessoa

A.2 Sonho

A.2.1 Classe Sonho

```
1  package obinject.sample.inception;
2
3  import java.util.ArrayList;
4  import java.util.Calendar;
5  import java.util.GregorianCalendar;
6  import java.util.Iterator;
7  import java.util.List;
8  import obinject.Id;
9  import obinject.Persistent;
10
11 @Persistent
12 public class Sonho {
13
14     @Id
15     private int codigo;
16
17     private Calendar inicio = new GregorianCalendar();
18     private Calendar termino = new GregorianCalendar();
19     private Calendar avisoDisparo = new GregorianCalendar();
20     private int nivel;
21     private boolean limbo;
22     private Sonho sonhoAnterior;
23     private Sonho sonhoProximo;
24     private Pasiv pasiv;
25     private Sonhador sonhadorConstroi;
26     private List<Sonhador> sonhadores = new ArrayList<Sonhador>();
27     private List<Sombra> sombras = new ArrayList<Sombra>();
28
29     public int getCodigo() {
30         return codigo;
31     }
32
33     public void setCodigo(int codigo) {
34         this.codigo = codigo;
35     }
36
37     public Calendar getInicio() {
38         return inicio;
39     }
40
41     public void setInicio(Calendar inicio) {
42         this.inicio = inicio;
43     }
```

```
44
45      public Calendar getTermino() {
46          return termino;
47      }
48
49      public void setTermino(Calendar termino) {
50          this.termino = termino;
51      }
52
53      public Calendar getAvisoDisparo() {
54          return avisoDisparo;
55      }
56
57      public void setAvisoDisparo(Calendar avisoDisparo) {
58          this.avisoDisparo = avisoDisparo;
59      }
60
61      public int getNivel() {
62          return nivel;
63      }
64
65      public void setNivel(int nivel) {
66          this.nivel = nivel;
67      }
68
69      public boolean isLimbo() {
70          return limbo;
71      }
72
73      public void setLimbo(boolean limbo) {
74          this.limbo = limbo;
75      }
76
77      public Sonho getSonhoAnterior() {
78          return sonhoAnterior;
79      }
80
81      public void setSonhoAnterior(Sonho sonhoAnterior) {
82          this.sonhoAnterior = sonhoAnterior;
83      }
84
85      public Sonho getSonhoProximo() {
86          return sonhoProximo;
87      }
88
89      public void setSonhoProximo(Sonho sonhoProximo) {
90          this.sonhoProximo = sonhoProximo;
91      }
92
```

```
93      public Pasiv getPasiv() {
94          return pasiv;
95      }
96
97      public void setPasiv(Pasiv pasiv) {
98          this.pasiv = pasiv;
99      }
100
101     public Sonhador getSonhadorConstroi() {
102         return sonhadorConstroi;
103     }
104
105     public void setSonhadorConstroi(Sonhador sonhadorConstroi) {
106         this.sonhadorConstroi = sonhadorConstroi;
107     }
108
109     public List<Sonhador> getSonhadores() {
110         return sonhadores;
111     }
112
113     public void setSonhadores(List<Sonhador> sonhadores) {
114         this.sonhadores = sonhadores;
115     }
116
117     public List<Sombra> getSombras() {
118         return sombras;
119     }
120
121     public void setSombras(List<Sombra> sombras) {
122         this.sombras = sombras;
123     }
124
125     public int sizeOfSonhador(){
126         return sonhadores.size();
127     }
128
129     public boolean addSonhador(Sonhador v){
130         return sonhadores.add(v);
131     }
132
133     public Sonhador removeSonhador(int idx){
134         return sonhadores.remove(idx);
135     }
136
137     public Iterator<Sonhador> iteratorSonhador(){
138         return sonhadores.iterator();
139     }
140
141     public int sizeOfSombra(){
```

```
142         return sombras.size();
143     }
144
145     public boolean addSombra(Sombra v){
146         return sombras.add(v);
147     }
148
149     public Sombra removeSombra(int idx){
150         return sombras.remove(idx);
151     }
152
153     public Iterator<Sombra> iteratorSombra(){
154         return sombras.iterator();
155     }
156 }
```

Listagem A.4: Segundo Caso de Uso: Classe `Sonho`

A.2.2 Classe EntitySonho

```
1 package obinject.sample.inception;
2
3 import obinject.device.File;
4 import obinject.metaclass.*;
5 import obinject.storage.*;
6
7 public class EntitySonho
8        extends Sonho
9        implements Entity<EntitySonho> {
10
11    public static final Uuid classId =
12        Uuid.fromString("E6904CE1-17D0-0057-A1BC-A0AA259FE50E");
13
14    public static final EHash<EntitySonho> entityStructure =
15        new EHash<EntitySonho>(
16            new File("build/classes/obinject/sample/inception/Sonho.dbo", 1024)) {};
17
18    public static final BTree<PrimaryKeySonho> primaryKeyStructure =
19        new BTree<PrimaryKeySonho>(
20            new File("build/classes/obinject/sample/inception/Sonho.pk", 1024)) {};
21
22    public static final KeyStructure keyStrutures[] = new KeyStructure[0];
23    private Uuid uuid = Uuid.generator();
24
25    public EntitySonho() {}
26
27    public EntitySonho(Sonho obj) {
28        this.setCodigo(obj.getCodigo());
29        this.setInicio(obj.getInicio());
30        this.setTermino(obj.getTermino());
31        this.setAvisoDisparo(obj.getAvisoDisparo());
32        this.setNivel(obj.getNivel());
33        this.setLimbo(obj.isLimbo());
34        this.setSonhoAnterior(obj.getSonhoAnterior());
35        this.setSonhoProximo(obj.getSonhoProximo());
36        this.setPasiv(obj.getPasiv());
37        this.setSonhadorConstroi(obj.getSonhadorConstroi());
38        this.setSonhadores(obj.getSonhadores());
39        this.setSombras(obj.getSombras());
40    }
41
42    @Override
43    public boolean equalToEntity(EntitySonho obj) {
44        return (this.getCodigo() == obj.getCodigo())
45            && (this.getNivel() == obj.getNivel())
46            && (this.isLimbo() == obj.isLimbo());
```

```
47      }
48
49      @Override
50      public EHash getEntityStructure() {
51          return entityStructure;
52      }
53
54      @Override
55      public Uuid getUuid() {
56          return uuid;
57      }
58
59      @Override
60      public BTree getPrimaryKeyStructure() {
61          return primaryKeyStructure;
62      }
63
64      public Order primaryKeyInstance() {
65          return new PrimaryKeySonho(this);
66      }
67
68      @Override
69      public boolean pullEntity(byte[] array, int position) {
70
71          PullStream pull = new PullStream(array, position);
72          Uuid storedClass = pull.pullUuid();
73
74          if (classId.equals(storedClass) == true) {
75              uuid = pull.pullUuid();
76              this.setCodigo(pull.pullInt());
77              this.setInicio(pull.pullCalendar());
78              this.setTermino(pull.pullCalendar());
79              this.setAvisoDisparo(pull.pullCalendar());
80              this.setNivel(pull.pullInt());
81              this.setLimbo(pull.pullBoolean());
82              return true;
83          }
84
85          return false;
86      }
87
88      @Override
89      public void pushEntity(byte[] array, int position) {
90
91          PushStream push = new PushStream(array, position);
92          push.pushUuid(classId);
93          push.pushUuid(uuid);
94          push.pushInt(this.getCodigo());
95          push.pushCalendar(this.getInicio());
```

```
96          push.pushCalendar(this.getTermino());
97          push.pushCalendar(this.getAvisoDisparo());
98          push.pushInt(this.getNivel());
99          push.pushBoolean(this.isLimbo());
100
101         if (this.getSonhoAnterior() != null) {
102             if (this.getSonhoAnterior() instanceof Entity) {
103                 push.pushEntity((Entity) this.getSonhoAnterior());
104             } else {
105                 push.pushEntity(new EntitySonho(this.getSonhoAnterior()));
106             }
107         } else {
108             push.pushUuid(Uuid.fromString("00000000-0000-0000-0000-000000000000"));
109         }
110
111         if (this.getSonhoProximo() != null) {
112             if (this.getSonhoProximo() instanceof Entity) {
113                 push.pushEntity((Entity) this.getSonhoProximo());
114             } else {
115                 push.pushEntity(new EntitySonho(this.getSonhoProximo()));
116             }
117         } else {
118             push.pushUuid(Uuid.fromString("00000000-0000-0000-0000-000000000000"));
119         }
120
121         if (this.getPasiv() != null) {
122             if (this.getPasiv() instanceof Entity) {
123                 push.pushEntity((Entity) this.getPasiv());
124             } else {
125                 push.pushEntity(new EntityPasiv(this.getPasiv()));
126             }
127         } else {
128             push.pushUuid(Uuid.fromString("00000000-0000-0000-0000-000000000000"));
129         }
130
131         if (this.getSonhadorConstroi() != null) {
132             if (this.getSonhadorConstroi() instanceof Entity) {
133                 push.pushEntity((Entity) this.getSonhadorConstroi());
134             } else {
135                 push.pushEntity(new EntitySonhador(this.getSonhadorConstroi()));
136             }
137         } else {
138             push.pushUuid(Uuid.fromString("00000000-0000-0000-0000-000000000000"));
139         }
140
141         for (Sonhador obj : this.getSonhadores()) {
142             if (obj instanceof Entity) {
143                 push.pushEntity((Entity) obj);
144             } else {
```

```
145                 push.pushEntity(new EntitySonhador(obj));
146             }
147         }
148 
149         for (Sombra obj : this.getSombras()) {
150             if (obj instanceof Entity) {
151                 push.pushEntity((Entity) obj);
152             } else {
153                 push.pushEntity(new EntitySombra(obj));
154             }
155         }
156     }
157 
158     @Override
159     public int sizeOfEntity() {
160         return  Stream.sizeOfUuid
161             +   Stream.sizeOfUuid
162             +   Stream.sizeOfInt
163             +   Stream.sizeOfCalendar
164             +   Stream.sizeOfCalendar
165             +   Stream.sizeOfCalendar
166             +   Stream.sizeOfInt
167             +   Stream.sizeOfBoolean
168             +   Stream.sizeOfUuid
169             +   Stream.sizeOfUuid
170             +   Stream.sizeOfUuid
171             +   Stream.sizeOfUuid
172             +   Stream.sizeOfCollection(this.getSonhadores())
173             +   Stream.sizeOfCollection(this.getSombras());
174     }
175 }
```

Listagem A.5: Segundo Caso de Uso: Classe `EntitySonho`

A.2.3 Classe PrimaryKeySonho

```java
package obinject.sample.inception;

import obinject.metaclass.Order;
import obinject.metaclass.PullStream;
import obinject.metaclass.PushStream;
import obinject.metaclass.Stream;

public class PrimaryKeySonho extends EntitySonho implements Order<PrimaryKeySonho> {

    public PrimaryKeySonho() {}

    public PrimaryKeySonho(Sonho obj) {
        this.setCodigo(obj.getCodigo());
    }

    @Override
    public boolean equalToKey(PrimaryKeySonho obj) {
        return (this.getCodigo() == obj.getCodigo());
    }

    @Override
    public boolean isLess(PrimaryKeySonho obj) {
        return (this.getCodigo() < obj.getCodigo());
    }

    @Override
    public boolean pullKey(byte[] array, int position) {
        PullStream pull = new PullStream(array, position);
        this.setCodigo(pull.pullInt());
        return true;
    }

    @Override
    public void pushKey(byte[] array, int position) {
        PushStream push = new PushStream(array, position);
        push.pushInt(this.getCodigo());
    }

    @Override
    public int sizeOfKey() {
        return Stream.sizeOfUuid
            + Stream.sizeOfUuid
            + Stream.sizeOfInt;
    }
}
```

Listagem A.6: Segundo Caso de Uso: Classe PrimaryKeySonho