UNIVERSIDADE FEDERAL DE ITAJUBÁ

PROGRAMA DE PÓS-GRADUAÇÃO EM
CIÊNCIA E TECNOLOGIA DA COMPUTAÇÃO

ENCADEAMENTO DE ANOTAÇÕES E INDEXAÇÃO DE IMAGENS

Helaine Aparecida Correia

UNIFEI
Itajubá
Agosto, 2017

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

Helaine Aparecida Correia

ENCADEAMENTO DE ANOTAÇÕES E INDEXAÇÃO DE IMAGENS

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

Área de Concentração: Sistemas de Computação

Orientador: Prof. Dr. Enzo Seraphim

UNIFEI
Itajubá
Agosto, 2017

Dados Internacionais de Catalogação na Publicação (CIP)

C823e

Correia, Helaine Aparecida.
Encadeamento de Anotações e Indexação de Imagens / Helaine Aparecida Correia. – Itajubá: UNIFEI, 2017.
xiv + 93 f. : il. ; 29.7

Orientador: Enzo Seraphim.
Dissertação (Mestrado em Ciência e Tecnologia da Computação) – Universidade Federal de Itajubá, Itajubá, 2017.

1. Anotação. 2. Indexação de Imagem. 3. Extração de Características. I. Seraphim, Enzo. II. Universidade Federal de Itajubá. III. Título I. Seraphim, Enzo, orient. II. Universidade Federal de Itajubá. III. Título.

CDU 004.65

Exemplar correspondente à versão final do texto, após a realização da defesa, incluindo, portanto, as devidas considerações dos examinadores.

Folha de Aprovação

Dissertação **aprovada** pela Banca Examinadora em 07 de agosto 2017, conferindo ao autor o título de **Mestre em Ciência e Tecnologia da Computação**.

Dr. Enzo Seraphim _____

Orientador

Universidade Federal de Itajubá

Dr. Edmilson Marmo Moreira _____

Membro da Banca

Universidade Federal de Itajubá

Dr^a. Maria Camila Nardini Barioni _____

Convidado

Universidade Federal de Uberlândia

Por vezes sentimos que aquilo que fazemos
não é senão uma gota de água no mar.
Mas o mar seria menor se lhe faltasse uma gota.

Madre Teresa de Calcutá

A minha família

Agradecimentos

A Deus, por sempre me dar forças para enfrentar os desafios que encontro no caminho e por colocar pessoas especiais na minha vida.

Aos meus pais, *Antônio* e *Maria Helena*, pelo amor incondicional.

Ao meu irmão, *Reginaldo*, e sobrinhos, *Pedro*, *João* e *Maria*, pelo carinho.

Ao meu noivo e amigo, *Diego*, pelo carinho e apoio nesses 10 anos. Agradeço também pela compreensão para que eu concluísse essa etapa.

Ao professor e amigo, *Prof. Dr. Enzo Seraphim*, pelos ensinamentos passados, pela orientação, paciência e disponibilidade.

A professora, amiga e "irmã", *Profª. Drª. Thatyana de Faria Piola Seraphim*, pela oportunidade do estágio em docência, pelos ensinamentos e conselhos dados acompanhados com longas conversas e um bom café ou chá. Agradeço também aos pequenos, *Miguel* e *Raphael*, pelas alegrias transmitidas.

Ao *Prof. Dr. Edmilson Marmo Moreira* e ao *Prof. Dr. Otávio Augusto Salgado Carpinteiro*, por me incentivarem a ingressar no ambiente acadêmico.

Aos professores do Programa de Pós-Graduação em Ciência e Tecnologia da Informação, pelos ensinamentos passados dentro e fora da sala de aula.

Ao meu colega, amigo e "irmão", *João Francisco*, pelo apoio e paciência que tem comigo, principalmente no final desta etapa.

Aos colegas do mestrado, do trabalho e amigos, *Luiz Olmes*, *Danillo Goulart*, *Ivan Pereira* e *Armando Neto*, pelo apoio dado.

A Universidade Federal de Itajubá, pela oportunidade de realizar o mestrado.

A banca examinadora, por dispor de seu tempo e contribuir para o enriquecimento deste trabalho.

Enfim, a todos que, mesmo não sendo citados, contribuíram para que eu chegasse até aqui. Obrigada!

Resumo

Na programação moderna tornou-se muito comum utilizar anotações antes dos atributos ou métodos de uma classe para os mais variados objetivos. Por exemplo, no *framework Java Server Faces*, as anotações são usadas para gerenciar as classes que compartilham informações de interfaces visuais. Nos *frameworks* que implementam a injeção de dependência e de contexto, elas são usadas na produção e definição de escopo do objeto. No entanto, a mais conhecida utilização está no mapeamento objeto-relacional, onde objetos são armazenados em tuplas de tabelas dos sistemas gerenciadores de banco de dados relacionais (SGBDR). As anotações do *framework Obinject*, ao serem aplicadas sobre a classe do usuário, utilizando os mecanismos de metaprogramação, permitem a geração automática das classes empacotadoras responsáveis pela persistência e indexação dos objetos. Porém, as anotações de indexação, neste *framework*, apresentam restrições quanto ao posicionamento dos atributos na chave do índice, ou seja, ele segue a ordem como os atributos são declarados; e isto nem sempre é possível manter em indexações complexas com múltiplos índices do mesmo domínio. Dessa maneira, este trabalho propõe a definição de um esquema de encadeamento de anotações, no *framework Obinject*, que garante múltiplos índices usando uma única anotação por domínio e que suporte o posicionamento dos atributos na composição da chave de indexação. Além disso, propõe um novo domínio de indexação para imagens através da extração das características utilizando a abordagem estatística para produzir caracterizações das texturas das imagens. Espera-se, deste trabalho, uma redução do número de anotações no *framework Obinject*; o aumento da flexibilidade, com a definição da posição do atributo que formará a chave de indexação; e o aumento da funcionalidade, tornando as anotações repetíveis e criando a anotação para extração e indexação das imagens . Os experimentos compararam os níveis de usabilidade da versão anterior e atual do *framework Obinject*. Também foram feitos testes para avaliar o desempenho do *framework* ao utilizar sua anotação para extração e indexação de características das imagens.

Palavras-chave: Anotação, Indexação de Imagem, Extração de Características.

Abstract

Annotation Chaining and Image Indexing

In modern programming it has become very common to use annotations before the attributes or methods of a class for the most varied objectives. For example, in the Java Server Faces framework, annotations are used to manage classes that share information of visual interfaces. In frameworks that implement both the dependency injection and of context, they are used in the production and scope definition of the object. However, the best known application is in object-relational mapping, where objects are stored in table tuples of Relational Database Management Systems (RDBMS). The annotations of the Obinject framework, when applied on the user class, using the metaprogramming mechanisms, allow the automatic generation of the packager classes, responsible for the persistence and indexing of the objects. However, the indexing annotations, in this framework, have restrictions regarding the positioning of attributes in the index keys, in other words, it follows the order as the attributes are declared; and this is not always possible to maintain in complex indexing with multiple indexes of the same domain. In this way, the present work proposes the definition of an annotation chaining scheme, in the Obinject framework, which guarantees multiple indexes using a single annotation per domain and that also supports the positioning of the attributes in the composition of the indexing key. In addition, it proposes a new indexing domain for images through the extraction of its features, using the statistical approach to produce characterizations of the textures of the images. This study is expected to reduce the number of annotations in the Obinject framework; the increase in the flexibility, with the definition of the position of the attribute that will form the indexing key; and increased functionality, by making annotations repeatable and creating annotation for the image extraction and indexing. The experiments compared the usability levels of the previous and current version of the Obinject framework. Tests were also made to evaluate the performance of the framework when using its annotation for the extraction and indexing of images features.

Keywords: *Annotation, Image Indexing, Features Extraction.*

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Códigos

Abreviaturas e Siglas

API	–	*Application Programming Interfaces*
CBIR	–	*Content-Based Image Retrieval*
FQN	–	*Fully Qualified Name*
GLCM	–	*Gray Level Coocurrence Matrix*
IDE	–	*Integrated Development Environment*
I/O	–	*input/output*
J2SE	–	*Java Platform, 2º Standard Edition*
Java SE	–	*Java Platform, Standard Edition*
JDK	–	*Java Platform, Standard Edition Development Kit*
JPA	–	*Java Persistence API*
LFW	–	*Labeled Faces in the Wild*
pixel	–	*picture element*
SUS	–	*System Usability Scale*
UUID	–	*Universally Unique Identifier*
UML	–	*Unified Modeling Language*
XML	–	*Extensible Markup Language*

Capítulo 1

Introdução

Uma das primeiras aplicações das imagens digitais ocorreu na indústria dos jornais, na década de 20, quando um cabo submarino (cabo Bartlane) entre Londres e Nova York foi utilizado para a transmissão de imagens. Assim, a fotografia passou a ser transportada em menos de três horas, sendo que antes ela levava mais de uma semana (GONZALEZ; WOODS, 2010).

Na década de 60, devido a disponibilidade de computadores poderosos para realizar as tarefas de processamento de imagens e o programa espacial permitiram o advento do processamento digital de imagens. Nessa época, foram realizadas técnicas computacionais de melhoramento da imagem da Lua para corrigir vários tipos de distorções. Além disso, no final da década de 60 e começo dos anos 70, as técnicas de processamento digital de imagens começaram a ser desenvolvidas para serem aplicadas nas imagens médicas, nas observações remotas da Terra e na astronomia (GONZALEZ; WOODS, 2010).

Depois de 50 anos, a área de processamento de imagem amadureceu rapidamente devido ao menor custo e maior poder do *hardware*, e também ao avanço em termos de algoritmos (SEN; VADIVEL, 2015). As técnicas de processamento de imagens passaram a ser utilizadas cada vez mais na resolução de problemas na área de medicina, biologia, automação industrial, sensoriamento remoto, astronomia, microscopia, artes, área militar, arqueologia, segurança e vigilância (PEDRINI; SCHWARTZ, 2008).

O crescimento da internet e da tecnologia da multimídia levou a uma maior demanda em mercado de processamento de conteúdo e gerenciamento de dados de imagens. O grande número de imagens em aplicações, como *ecommerce*, bibliotecas digitais, imagens médicas, sistemas de informação geográfico, etc; requer um eficiente mecanismo para pesquisa e recuperação no gerenciamento de dados das imagens (ANNAMALAI et al., 2000). Nos bancos de dados de imagens, as características das imagens são extraídas e indexadas para que, ao ser realizada uma pesquisa por informação ou padrão, a busca seja mais eficiente e confiável (ROMDHANE et al., 2010).

Conforme Simpson et al. (2014), existem duas formas para a recuperação de imagens: baseada em texto e baseada em conteúdo. Utilizando métodos para recuperação de imagens baseada em texto, as imagens são representadas por um conjunto de descritores de textos (rótulos). Já nos métodos para recuperação de imagens por conteúdo (CBIR - *Content-Based Image Retrieval* ou Recuperação de Imagem Baseada por Conteúdo), as imagens são representadas por vetores de características numéricos que descrevem sua aparência.

Nos sistemas de recuperação de imagens, os atributos mais utilizados para representar uma imagem são: textura, forma e cor. A textura permite obter a conectividade, densidade e homogeneidade da imagem. Já a análise sobre a forma, embora seja mais difícil, é de grande importância para aplicações médicas; e a maioria dos trabalhos relacionados à extração de características baseadas na distribuição de cores são histogramas de cores (CASTANÓN, 2003; PEDRINI; SCHWARTZ, 2008).

Como o *framework Obinject* não oferece suporte a um tipo de dados cada vez mais utilizado, a imagem digital, isto motivou a realização deste trabalho. Então, para que a imagem possa ser representada por um vetor de características numérico, obtido através dos métodos da abordagem estatística que representam as propriedades da textura da imagem; e este vetor seja indexado em uma estrutura métrica, foi proposta uma nova anotação para o *framework*.

Incorporadas na linguagem de programação Java em 2004, as anotações têm sido um sucesso e são amplamente usadas em muitos projetos dentro da cena de desenvolvimento de *software*, como nos *frameworks* Seam (HAT, 2009), Spring (SOFTWARE, 2002) e Hibernate (BAUER; KING, 2005), que é uma implementação da API (*Application Programming Interfaces*) de Persistência Java ou JPA (*Java Persistence* API) (ORACLE, 2006). Elas podem ser verificadas em tempo de compilação pelos processadores de anotação ou em tempo de execução através da API de Reflexão Java (*Java Reflection* API) (CÓRDOBA-SÁNCHEZ; LARA, 2016) e devem ser apenas uma maneira de adicionar meta-informações (metadados) ao código. Sendo assim, muitos aspectos de uma aplicação são especificados no código da aplicação, eliminando a necessidade de arquivos externos de configuração, por exemplo XML (*Extensible Markup Language* ou Linguagem de Marcação Extensível); ou da inserção de código extra na aplicação (MANCINI et al., 2010).

As anotações do *framework Obinject* (CARVALHO, 2013), definidas em trabalhos anteriores, como de Oliveira (2012) e Ferro (2012), são utilizadas nas classes de aplicação para que as classes empacotadoras sejam geradas através dos mecanismos de metaprogramação. As classes empacotadoras fazem o vínculo da classe de aplicação com o *framework*, sendo responsáveis pela persistência e indexação dos objetos.

No *framework Obinject* existem 6 domínios de indexação: *PrimaryKey*, *GeoPoint*, *Order*, *Edition*, *Point* e *Rectangle*. O nome das anotações dos domínios de indexação, com exceção da anotação `@PrimaryKey`, é formado pelo nome do domínio seguido do ordinal que distingue o

índice. Por exemplo, para que o usuário possa indexar dois atributos em dois índices distintos no domínio *GeoPoint*, é necessário aplicar a anotação `@GeoPointFirst` para o atributo que formará a chave do índice de número um e `@GeoPointSecond` para o atributo que formará a chave do índice de número dois.

Embora o *framework Obinject* suporte a definição de múltiplos índices para diversos domínios de indexação, existe uma restrição na composição da ordem com que os atributos formarão a chave. Esta limitação implica que a composição da chave de indexação esteja na mesma ordem da declaração dos atributos. No entanto, isto nem sempre é possível ser mantido em indexações complexas com múltiplos índices do mesmo domínio. Assim, a definição de um esquema de encadeamento de anotações eficiente para o posicionamento de atributos na composição de chaves foi também uma das motivações para a realização deste trabalho.

1.1 Objetivos

Os objetivos deste trabalho são:

1. definição de um esquema de encadeamento de anotações no *framework Obinject* que garanta múltiplos índices usando uma única anotação por domínio e que suporte o posicionamento dos atributos na composição da chave de indexação;
2. criação de um novo domínio de indexação para imagens, no *framework Obinject*, através da extração de características utilizando abordagem estatística para produzir caracterizações das texturas das imagens. A textura é uma das características usadas pelo sistema de visão humano no processo de análise e interpretação da imagem.

1.2 Organização do Trabalho

Este trabalho é dividido em capítulos como apresentado a seguir:

- Capítulo 2: introduz conceitos do *framework Obinject*, de metaprogramação e dos recursos de metaprogramação da linguagem de programação Java (anotações, tipo genérico e reflexão); e também alguns conceitos de processamento e análise de imagens digitais. Este referencial teórico é utilizado na implementação deste trabalho.
- Capítulo 3: apresenta o encadeamento de anotações e a indexação de imagens.
- Capítulo 4: explica a metodologia utilizada nos dois experimentos realizados e apresenta os resultados obtidos.
- Capítulo 5: apresenta a conclusão deste trabalho e sugere possíveis trabalhos futuros.

Capítulo 2

Fundamentação Teórica

Este Capítulo apresenta os conceitos utilizados na definição de um esquema de encadeamento de anotações e na criação de uma anotação para extração e indexação de características de imagens no *framework Obinject*. A Seção 2.1 menciona conceitos relacionados à metaprogramação e suas taxonomias. As Seções 2.2, 2.3 e 2.4 estão relacionadas aos recursos de metaprogramação disponíveis na linguagem Java: anotações, tipo genérico e reflexão. Já a Seção 2.5 realça os principais conceitos do *framework Obinject*. Finalmente, a Seção 2.6 introduz alguns conceitos relacionados ao processamento e análise de imagens digitais.

2.1 Metaprogramação

A metaprogramação significa escrever programas (metaprogramas) que escrevem ou manipulam outros programas utilizando uma metalinguagem (SHEARD, 2001; DEVRIESE; PIESSENS, 2013).

Além disso, o termo programa objeto, na metaprogramação, é usado tanto para se referir aos programas que estão sendo analisados quanto aqueles que estão sendo gerados, ou seja, esse termo ressalta que os programas são objetos que os metaprogramas realizarão a análise, geração ou transformação. Já a linguagem de programação utilizada para descrever os programas objeto é conhecida como linguagem objeto (OLIVEIRA, 2004).

Os metaprogramas são amplamente utilizados na área de linguagens de programação, por exemplo, para realizar a tradução de códigos escritos em uma linguagem fonte para códigos de outra linguagem (compiladores) (OLIVEIRA, 2004); para criar programas de propósito geral, que a partir de uma especificação, geram programas que apresentam melhor desempenho e atendem soluções específicas, facilitando assim a manutenção e a construção (geradores) (SHEARD, 2001). Além disso, os geradores oferecem uma redução do volume de código necessário que precisa ser escrito manualmente (OLIVEIRA, 2012).

Algumas linguagens de programação, como C, C++, Haskell, Java, ML, OCaml, Racket, Scala e Scheme; oferecem recurso de metaprogramação (BOWMAN et al., 2015).

São apresentadas duas taxonomias que abordam os conceitos de metaprogramação: a taxonomia de Sheard (SHEARD, 2001) e a taxonomia dos conceitos fundamentais de metaprogramação (DAMAŠEVIČIUS; ŠTUIKYS, 2008).

A taxonomia de Sheard classifica a metaprogramação quanto ao tipo de metaprograma, a utilização do tempo, a divisão de código estático e dinâmico do metaprograma e a divisão das linguagens.

A classificação quanto ao tipo correlaciona um metaprograma em gerador ou analisador. Os geradores de programas criam outros programas (programas objeto) usados para solucionar problemas específicos o que os torna mais eficientes do que suas versões não geradas. O analisador de programa extrai os metadados mais relevantes de um programa objeto e os analisa, produzindo resultados como dados, gráficos ou outros programas objeto com propriedades baseadas no programa objeto de origem. Os transformadores de programas, otimizadores e sistemas de avaliação parcial são exemplos de metaprogramas analisadores (SHEARD, 2001; MAGNO, 2015).

Conforme Sheard (2001), um gerador de programa, de acordo com o critério de utilização do tempo, pode ser estático ou em tempo de execução. O gerador de programa estático cria código que é gravado em disco e processado por compiladores normais. Um gerador de programa em tempo de execução escreve ou constrói um programa e imediatamente executa o programa gerado.

Além disso, o código do gerador de programa é dividido em fragmentos de códigos estático, que compreende a utilização da metalinguagem de programação; e dinâmico, que compreende o programa objeto sendo produzido. As anotações são usadas para separar as partes dos programas e podem ser inseridas manualmente por usuários ou automaticamente por um processo (SHEARD, 2001).

Finalmente, um metaprograma pode ser classificado como homogêneo ou heterogêneo segundo a divisão de linguagens. Um metaprograma é homogêneo se a metalinguagem e a linguagem do programa objeto são as mesmas. E um metaprograma é heterogêneo se a metalinguagem e a linguagem do programa objeto são diferentes (SHEARD, 2001).

Já a taxonomia dos conceitos fundamentais de metaprogramação apresentada por Damaševičius e Štuikys (2008) aborda os conceitos de estrutura e de processo que podem ser vistos na tabela 2.1.

Os conceitos de estrutura descrevem abstrações (metaprograma, metadados) e princípios de construção (separação de interesses, níveis de abstração) usados durante o desenvolvimento dos artefatos de metaprogramação. E os conceitos de processo descrevem as operações e

Tabela 2.1: Taxonomia dos conceitos de metaprogramação de Štuikys e Damaševičius

Classes de Conceito	Conceitos	Termos equivalentes usados na literatura
Estrutura	Metaprograma	Meta-componente, meta-especificação, meta-função, template, componente genérico, componente parametrizado
	Níveis de abstração	Níveis de abstração
	Separação de conceitos	Separação de aspectos
	Metadados	Anotações
Processo	Transformação	Manipulação, modificação, adaptação
	Geração	Geração de programa/código
	Reflexão	Introspecção, intercessão
	Generalização	Parametrização

processos realizados por um *designer* dos artefatos de metaprogramação. A transformação, a geração e a reflexão são usadas durante o tempo de compilação ou execução de um metaprograma, e a generalização é utilizada durante a criação dos artefatos de metaprogramação (DAMAŠEVIČIUS; ŠTUIKYS, 2008).

A seguir são apresentados os conceitos da taxonomia apresentada por Damaševičius e Štuikys (2008) sobre metaprograma, níveis de abstração, separação de interesse, metadados, transformação, geração, reflexão e generalização.

Metaprograma

Metaprograma é o programa que usa metalinguagem para manipular o código do programa objeto (DEVRIESE; PIESSENS, 2013; SHEARD, 2001).

As metalinguagens ou linguagens de metaprogramação oferecem recursos que fazem com que a metaprogramação seja uma tarefa mais confortável, mas elas não possuem maior poder computacional do que as linguagens de programação tradicionais (OLIVEIRA, 2004).

Segundo Oliveira (2004), existem duas maneiras de construir metalinguagens. A primeira forma é a extensão de uma linguagem de uso geral com bibliotecas para metaprogramação, permitindo a utilização de todos os recursos disponíveis da linguagem original nos metaprogramas, como APIs e recursos de I/O (*input/output* ou entrada/saída). Já a outra maneira é a construção de uma linguagem completamente nova que ofereça recursos integrados (*built in*) para atender aos requisitos da metaprogramação, possibilitando expressar uma operação sobre o código fonte em alto nível, tornando o metaprograma mais simples.

Os interpretadores, compiladores, programas geradores, verificadores de tipo e programas analisadores são exemplos de metaprogramas (SHEARD, 2001).

Níveis de abstração

A abstração permite que as informações importantes para o desenvolvedor ou usuário sejam enfatizadas e os detalhes não relevantes sejam escondidos (DAMAŠEVIČIUS; ŠTUIKYS, 2008).

O papel dos níveis de abstração, na metaprogramação, é a construção do metaprograma (DAMAŠEVIČIUS; ŠTUIKYS, 2008).

Separação de conceitos

A separação de conceitos, um dos princípios chaves da Engenharia de *Software*, é o processo de quebra de um problema de projeto em tarefas distintas que são implementadas separadamente para lidar com a complexidade e alcançar os fatores de qualidade de engenharia, como flexibilidade, manutenção e confiabilidade (DAMAŠEVIČIUS; ŠTUIKYS, 2008).

Na metaprogramação, este princípio é utilizado para separar variáveis do programa de domínio das partes fixas (DAMAŠEVIČIUS; ŠTUIKYS, 2008).

Metadados

Os metadados representam as informações estruturadas que descrevem, explicam, localizam e tornam mais fácil recuperar, usar e gerenciar um recurso de informação. É chamado de dados sobre dados ou informação sobre informação (PRESS, 2004).

A função dos metadados, em metaprogramação, é descrever e representar informação adicional sobre meta-nível de abstração em metaprogramas (DAMAŠEVIČIUS; ŠTUIKYS, 2008).

Segundo Press (2004), os metadados podem ser descritivos, estruturais ou administrativos. Os metadados descritivos apresentam informação para fins de pesquisa e identificação, tais como os elementos que descrevem título, resumo, autor e palavras-chave. Os metadados estruturais indicam como os objetos compostos são colocados juntos, por exemplo, elementos que descrevem páginas ordenadas para formar capítulos. Os metadados administrativos fornecem informações para ajudar a gerenciar um recurso, como os elementos que descrevem quando foi criado, qual tipo de arquivo e outras informações técnicas.

Transformação

A transformação do programa é definida como uma manipulação da representação do programa resultando na mudança de sua forma (sintaxe) (DAMAŠEVIČIUS; ŠTUIKYS, 2008).

O papel da transformação, na metaprogramação, é que o algoritmo de transformação descreve

a geração de uma instância particular dependendo dos valores dos parâmetros genéricos (DAMAŠEVIČIUS; ŠTUIKYS, 2008).

Geração

A geração de código é o processo em que o gerador de código converte um programa de alto nível em um conjunto de instruções de baixo nível. Ela pode ser feita em tempo de compilação ou em tempo de execução e é usada para produzir código de forma automática, reduzindo a necessidade de codificação manual feita pelo desenvolvedor (DAMAŠEVIČIUS; ŠTUIKYS, 2008).

Na metaprogramação, a função da geração de códigos é o desenvolvimento de programas geradores que geram outros programas adaptados para aplicações específicas (DAMAŠEVIČIUS; ŠTUIKYS, 2008).

Reflexão

Reflexão é a capacidade de um programa em execução examinar a si mesmo, acessando suas propriedades e estrutura; e mudar seu estado, alterando as propriedades e comportamento (FORMAN; FORMAN, 2004; OLIVEIRA, 2012).

Dois níveis de complexidade de reflexão são definidos: introspecção e intercessão. A introspecção refere-se ao acesso do estado de execução do metaprograma e a intercessão refere-se a manipulação/alteração desse estado (OLIVEIRA, 2012).

Segundo Damaševičius e Štuikys (2008), a metaprogramação é uma atividade reflexiva porque permite o desenvolvimento de programas que podem criar outros programas.

Generalização

A generalização fornece uma reapresentação do conhecimento. Sua aplicação significa uma transição para o nível mais alto de abstração, onde o conhecimento do domínio pode ser representado e explicado de forma mais compreensível e eficaz (DAMAŠEVIČIUS; ŠTUIKYS, 2008).

Conforme Damaševičius e Štuikys (2008), a função da generalização, na metaprogramação, é o desenvolvimento de um metaprograma usando um programa de domínio específico (componente) como uma base, capturando os pontos comuns do domínio e expressando, em um nível mais alto de abstração, as variações do domínio.

2.2 Anotações

Anotações são metadados aplicados no código-fonte do programa e fornecem valores aos elementos anotados (MAGNO, 2015). Elas podem ser usadas, por exemplo, para fornecer informações para o compilador detectar erros ou esconder avisos (*warnings*); para ferramentas processarem as informações das anotações e gerarem código-fonte e arquivos XML em tempo de compilação (ORACLE, 2016b). As anotações são abstrações poderosas, mas quando mal aplicadas obscurecem totalmente o próprio código (ARNOLD; GOSLING, 2007).

Na linguagem de programação Java a forma da anotação é definida pelo tipo de anotação. O tipo de anotação é um tipo especial de interface (`@interface`) que tem um nome e pode ter elementos. Esse tipo de anotação estende implicitamente a interface `Annotation` (`java.lang.annotation`) e não pode estender explicitamente qualquer outra interface. Além disso, os tipos de anotação podem ser declarados em qualquer lugar onde uma interface pode ser declarada, isto é, como um tipo de anotação de auto nível ou aninhada dentro de outro tipo; ter os mesmos modificadores aplicados para interfaces e não podem ser declarados como tipo genérico (ARNOLD; GOSLING, 2007).

Os elementos do tipo de anotação são responsáveis por guardarem as informações pertencentes à anotação (MAGNO, 2015). Conforme Arnold e Gosling (2007), um elemento do tipo de anotação pode ter como retorno os tipos: primitivo, enumerado, anotação de outro tipo (não pode ser seu próprio tipo), `Class` (ou uma invocação genérica específica de `Class`) ou *array* de um dos tipos precedentes.

Qualquer elemento do programa, como pacotes, classes, interfaces, enumerações, anotações, campos, métodos, construtores, variáveis locais e de parâmetros, podem ser anotados. Para tanto, é recomendado estar em uma linha separada e fornecer nome da anotação precedido pelo caractere @. Já os valores para os elementos da anotação são fornecidos na forma nome=valor, separados por vírgula e entre parênteses (ARNOLD; GOSLING, 2007).

Para exemplificar, o cenário utilizado é o fornecimento de informações sobre o usuário e a verificação de pendência aplicados ao elemento do programa (classe `Log`). O código 2.1 exemplifica a declaração de anotações (linhas 7 a 19) e suas aplicações (linhas 21 e 22).

No código 2.1, o tipo de anotação `Usuario` possui 4 elementos: `apelido`, `numero`, `vinculo` e `ativo`. O tipo de anotação `Cpf` possui apenas um elemento e por ser denominado `value`, o elemento pode ser omitido ao aplicar a anotação (linha 21). Observe que as anotações `@Usuario` (linha 21), `@Pendente` (linha 22) foram aplicadas sobre a classe `Log` e especificadas em uma linha separada antes da declaração da classe. Na anotação `@Usuario`, o valor do elemento `apelido` está entre aspas. Já o valor do elemento `numero` é precedido pela anotação `@Cpf` e depois o valor. No valor do elemento `vinculo`, como possui apenas um elemento, é dispensado o uso de chaves ao redor dele. O elemento `ativo` e o seu valor foram omitidos,

```
1  enum Funcao {
2      Discente,
3      Docente,
4      ServidorTecnico
5  }
6
7  @interface Cpf{
8      long value();
9  }
10
11 @interface Usuario {
12     String apelido();
13     Cpf numero();
14     Funcao[] vinculo();
15     boolean ativo() default true;
16 }
17
18 @interface Pendente{
19 }
20
21 @Usuario(apelido = "Gabriel", numero = @Cpf(12312312345L), vinculo = Funcao.
       Discente)
22 @Pendente
23 public class Log {
24     //codigo
25 }
```

Código 2.1: Exemplos de declarações e aplicações de anotações

porque o elemento `ativo` inicializa com valor padrão `true` (`default`). Já o tipo de anotação `Pendente` não possui nenhum elemento, então ao anotar a classe (linha 22) o parênteses é omitido.

Segundo Gosling et al. (2015), existem 3 espécies de anotação: anotação marcadora (*Marker Annotation*), anotação de elemento único (*Single Element Annotation*) e anotação normal (*Normal Annotation*).

A anotação marcadora é usada sem elemento ou com elementos em que todos apresentam valores padrões (`default`). A anotação `@Pendente` (linhas 18 e 19) do código 2.1 é um exemplo de anotação marcadora.

A anotação de elemento único é usada com um único elemento chamado `value` ou com múltiplos elementos, sendo um deles chamado `value` e os demais apresentam valores padrões (`default`). Um exemplo de anotação de elemento único é a anotação `@Cpf` (linhas 7 a 9) do código 2.1.

Finalmente, a anotação normal, na sua utilização, deve possuir elementos sem valores padrões (`default`). A anotação `@Usuario` (linha 11 a 16) do código 2.1 é um exemplo de anotação normal.

Na linguagem de programação Java existe algumas anotações definidas no pacote

`java.lang.annotation`: `@Override`, que mostra ao compilador que o elemento declarado em uma superclasse foi sobrescrito (ORACLE, 2016b); `@SuppressWarnings`, que controla as advertências feita pelo compilador Java (GOSLING et al., 2015); e `@Deprecated`, que identifica um elemento do programa que não deve ser usado (ARNOLD; GOSLING, 2007).

Uma definição importante a ser abordada é sobre a meta-anotação que é uma anotação aplicada sobre um tipo de anotação. Como exemplo temos a meta-anotação `@Target` e `@Retention` (definidas no pacote `java.lang.annotation`).

A meta-anotação `@Target` define onde uma anotação é aplicável no elemento do programa (pacotes, classes, interfaces, enumerações, anotações, campos, métodos, construtores, variáveis locais e de parâmetros). Se um tipo de anotação não possui a meta-anotação `@Target`, a anotação é aplicável em qualquer elemento do programa. Mas se possui, o compilador verifica se a anotação foi corretamente aplicada a um elemento do programa. Para definir o elemento do programa, a meta-anotação `@Target` pode conter um ou mais valores do tipo enumerado `ElementType` (`ANOTATION_TYPE`, `CONSTRUCTOR`, `METHOD`, `FIELD`, `LOCAL_VARIABLE`, `PARAMETER`, `PACKAGE` e `TYPE`) (ARNOLD; GOSLING, 2007).

A meta-anotação `@Retention` determina quando uma anotação pode ser acessada através da política de retenção. Para definir as políticas de retenção, a meta-anotação `@Retention` pode conter um ou mais valores do tipo enumerado `RetentionPolicy` (`SOURCE`, `CLASS` e `RUNTIME`). Na política de retenção `SOURCE`, as anotações existem somente no arquivo fonte e são descartadas pelo compilador ao produzir uma representação binária. Na política de retenção `CLASS`, as anotações são preservadas na representação binária da classe, mas não necessitam estar disponíveis durante a execução. Finalmente na política de retenção `RUNTIME`, as anotações são preservadas na representação binária da classe e devem estar disponíveis durante a execução para serem acessadas pelo mecanismo de reflexão (ARNOLD; GOSLING, 2007), usando os métodos de introspecção e intercessão. É importante observar que as anotações são acessadas em tempo de compilação através dos processadores de anotações [1](OLIVEIRA, 2012).

O código 2.2 redefine a anotação `@Usuario` do código 2.1 modificando onde ela é aplicável (`@Target`) e sua política de retenção (`@Retention`).

Na declaração do tipo de anotação `Usuario` (linha 1 do código 2.2), o valor da meta-anotação `@Target` foi definido como `ElementType.TYPE`, então a anotação `@Usuario` somente pode ser aplicada à declarações de tipos de classes, de interfaces, de enumerações ou de tipos de anotação. Além disso, a anotação `@Usuario` pode ser acessada durante a execução do programa, pois a política de retenção foi definida como `RUNTIME` na declaração do tipo de anotação `Usuario` (linha 2).

[1]**Processadores de anotação** são invocados pelo compilador e além de verificarem as anotações aplicadas a qualquer elemento do programa, eles podem realizar outras funções como gerar código (CÓRDOBA-SÁNCHEZ; LARA, 2016)

```
1 @Target (ElementType.TYPE)
2 @Retention(RetentionPolicy.RUNTIME)
3 @interface Usuario {
4     String apelido();
5     Cpf numero();
6     Funcao[] vinculo();
7     boolean ativo() default true;
8 }
```

Código 2.2: Exemplo da aplicação das meta-anotações *Target* e *Retention*.

Algumas meta-anotações, também definidas no pacote `java.lang.annotation`, são: `@Documented`, que documenta anotações, e `@Inherited`, que indica que um tipo de anotação é herdado automaticamente por subclasses da classe anotada (MAGNO, 2015). E, a partir do Java SE (*Java Platform, Standard Edition*) versão 8, foi definida a meta-anotação `@Repeatable` que possibilita uma anotação ser aplicada mais de uma vez em um mesmo elemento do programa.

Para permitir que uma anotação seja repetível, é necessário usar a meta-anotação `@Repeatable` na sua declaração, caso contrário resultará em um erro no tempo de compilação. O valor da meta-anotação `@Repeatable` (que fica entre parênteses) é o tipo de anotação contêiner (*container annotation type*). Além disso, é necessário definir o tipo de anotação contêiner com um elemento *array* chamado `value` do tipo da anotação repetível (ORACLE, 2016b).

O código 2.3 redefine a anotação `@Usuario` do código 2.1 como repetível, define o tipo de anotação contêiner (`ListaUsuarios`), redefine a classe `Log` e declara duas classes (`LogHardware`, `LogSoftware`).

Ao aplicar a anotação `@Usuario`, declarada com uma anotação repetível (`@Repeatable`), sobre a classe `LogHardware` (linhas 23 e 24), o compilador cria automaticamente a anotação contêiner (`@ListaUsuarios`) que armazena todos os valores da anotação repetida (`@Usuario`) no seu elemento `value`. Já a anotação contêiner `@ListaUsuarios` (linhas 29 a 32) agrupa as anotações aplicadas repetidamente sobre a classe `LogSoftware` com seus respectivos valores. É importante ressaltar que esta forma de aplicar nitidamente a anotação contêiner (linhas 29 a 32) era uma solução alternativa utilizada pelos desenvolvedores, porque antes do Java SE versão 8 não era possível anotar mais de uma vez um mesmo elemento do programa com a mesma anotação (PIASECKI, 2014).

Um outro recurso adicionado no Java SE versão 8 permite que as anotações, além de poderem ser aplicadas à declarações, sejam aplicadas também para uso de tipos, como instanciar uma classe, realizar conversão de tipo (*type casting*), implementar cláusula e declarar *thrown exception* (exceção). Esta forma de anotação é chamada de anotação de tipo (*type annotation*) que foi criada para suportar uma análise melhorada dos programas Java de modo a garantir uma verificação de tipo mais forte (ORACLE, 2016b), reduzindo o número de erros e *bugs*

```
1  @Target(ElementType.TYPE)
2  @Retention(RetentionPolicy.RUNTIME)
3  @Repeatable(ListaUsuarios.class)
4  public @interface Usuario{
5      String apelido();
6      Cpf numero();
7      Funcao[] vinculo();
8      boolean ativo() default true;
9  }
10
11 @Target(ElementType.TYPE)
12 @Retention(RetentionPolicy.RUNTIME)
13 public @interface ListaUsuarios{
14     Usuario[] value ();
15 }
16
17 @Usuario(apelido = "Gabriel", numero = @Cpf(123456789L), vinculo = Funcao.
       Discente)
18 @Pendente
19 public class Log {
20     // codigo
21 }
22
23 @Usuario(apelido = "Miguel", numero = @Cpf(00000000000L), vinculo = Funcao.
       Discente)
24 @Usuario(apelido = "Joao", numero=@Cpf(11111111111L), vinculo = Funcao.Docente
       , ativo = false)
25 public class LogHardware extends Log{
26     //codigo
27 }
28
29 @ListaUsuarios({
30 @Usuario(apelido = "Pedro", numero = @Cpf(22222222222L), vinculo = Funcao.
       ServidorTecnico),
31 @Usuario(apelido = "Maria", numero = @Cpf(33333333333L), vinculo = Funcao.
       Docente)
32 })
33 public class LogSoftware extends Log{
34     //codigo
35 }
```

Código 2.3: Exemplo da aplicação da meta-anotação *Repeatable*.

dentro do código. Semelhante a declaração das anotações, anotações de tipo podem conter elementos e valores padrão. Além disso, ao aplicar a meta-anotação `@Target` sobre a anotação de tipo, o tipo enumerado `ElementType` deve ter a constante `TYPE_USE` e pode ter a constante `TYPE_PARAMETER` (JUNEAU, 2014).

O código 2.4 redefine a classe `Log` do código 2.1 e cria duas anotações de tipo: uma aplicada em qualquer tipo e outra aplicada em parâmetro de tipo ou qualquer tipo.

As linhas 1 a 9 do código 2.4 apresentam as declarações das anotações de tipo `@Regra` e `@Utf8`. Como pode-se observar, a anotação `@Regra` foi usada para: implementar a interface `Serializable` na linha 11; instanciar a classe `GregorianCalendar` na linha 13; relançar

```
1  @Target(ElementType.TYPE_USE)
2  @Retention(RetentionPolicy.RUNTIME)
3  public @interface Regra {
4  }
5
6  @Target({ElementType.TYPE_USE, ElementType.TYPE_PARAMETER})
7  @Retention(RetentionPolicy.RUNTIME)
8  public @interface Utf8 {
9  }
10
11 public class LogMemoria extends LogHardware implements @Regra Serializable{
12
13     private Calendar criacao = new @Regra GregorianCalendar();
14     private List<@Utf8 String> itens = new ArrayList<>();
15     public LogMemoria() throws @Regra ExceptionInInitializerError{
16         GregorianCalendar g = (@Regra GregorianCalendar) criacao;
17     }
18     public int totalItens(){
19         return itens.size();
20     }
21
22 }
```

Código 2.4: Exemplo de anotação de tipo.

exceção (*thrown exception*) na linha 15; e converter o tipo para `GregorianCalendar` na linha 16. Finalmente, a anotação `@Utf8`, na linha 14, foi aplicada no parâmetro genérico `String`.

A linguagem Java permite escrever anotações de tipo customizados e processadores para a verificação de tipo. Há também *frameworks* de verificação de tipo, como por exemplo o *Checker Framework* criado pela Universidade de *Washington*. O *Checker Framework* é uma biblioteca que contém anotações de tipo prontas para serem utilizadas juntamente com processadores de anotações que podem ser especificados ao compilar o código (JUNEAU, 2014); e que possibilita a escrita dos próprios verificadores de anotações de tipo. O *framework* permite que os desenvolvedores de *softwares* detectem e evitem erros em seus programas Java; realçando assim, o sistema de tipos do Java (ENGINEERING, 2016).

2.3 Tipo Genérico

Na linguagem de programação Java, o tipo genérico foi incorporado no J2SE (*Java Platform, 2º Standard Edition*) 5.0 tendo como principal finalidade permitir a criação de conjuntos com compatibilidade de tipos, porque até antes as implementações de conjuntos eram declaradas para conter o tipo `Object` (SIERRA et al., 2007).

Vale lembrar que as classes que estão relacionadas a conjuntos, em Java, formam a API *Collection*. Nela estão as classes que implementam a interface `Collection`, como `ArrayList`,

`LinkedList`, `TreeSet`, `HashSet`; e as classes que implementam a interface `Map`, como `TreeMap`, `HashMap`, `HashTable` (SIERRA et al., 2007).

As vantagens da utilização do tipo genérico são: proporcionar reaproveitamento do código para diferentes entradas, segurança de tipos em tempo de compilação e eliminar o uso de *type casting* para fazer a conversão dos objetos (ORACLE, 2016c).

Os tipos genéricos possibilitam que ao definir classes, interfaces e métodos sejam vinculados parâmetros de tipo ou variáveis de tipo de qualquer tipo não primitivo (qualquer classe, interface, *array* ou outra variável de tipo) (ORACLE, 2016c).

Além disso, nas declarações das classes, interfaces e métodos podem conter vários parâmetros de tipos que devem ser separados por vírgula. E por convenção, variáveis de tipo são nomeadas `E` para um tipo de elemento, `K` para um tipo chave, `V` para um tipo de valor, `T` para um tipo geral (ARNOLD; GOSLING, 2007).

A ligação de um tipo genérico, conhecida geralmente como tipo parametrizado, é o fornecimento de um argumento de tipo específico, substituindo cada parâmetro de tipo declarado genérico por um tipo concreto como, por exemplo, `Integer` (ARNOLD; GOSLING, 2007; ORACLE, 2016c).

Os parâmetros de tipos limitados (*bounded type parameters*) são para restringir os tipos que podem ser usados como argumentos de tipo em um tipo parametrizado. Na declaração do parâmetro de tipo limitado é utilizado o nome do parâmetro de tipo, a palavra-chave `extends` e o tipo (classe ou interface) que limita a ligação a esse tipo ou de uma de suas subclasses (limite superior). Neste caso, a palavra-chave `extends` é usada em um sentido geral e significa `extends` para classe ou `implements` para interface. É importante ressaltar que se a variável de tipo apresenta mais de um limite, eles devem ser separados utilizando o símbolo &; e se um dos limites é uma classe, ele deverá ser especificado primeiramente (ORACLE, 2016c).

O código 2.5 apresenta as implementações das classes `Carteira`, `CarteiraEstudantil`, `Pessoa`, `Aluno`, que podem ser vistas no Apêndice A; e da classe principal `App`.

A classe genérica `Carteira` (linha 1) do código 2.5, ao ser declarada, define que a variável de tipo `T` é limitada, ou seja, que ao passar um argumento de tipo é preciso que ele seja do tipo `Number` ou das subclasses de `Number`. Além disso, pode-se observar que este parâmetro de tipo é utilizado na declaração do atributo `numero` (linha 2) e do método genérico `getNumero` (linha 4).

Quando a classe `CarteiraEstudantil` é declarada, o tipo `Integer` (subclasse de `Number`) é passado como argumento de tipo para a classe genérica `Carteira` (linha 9). Assim, ao sobrescrever o método `getNumero` (linha 16), o seu tipo de retorno que antes era do tipo `T` agora passa a ser `Integer`.

```java
public abstract class Carteira<T extends Number> {
    private T numero;

    public T getNumero() {
        return numero;
    }
}

public class CarteiraEstudantil extends Carteira<Integer> {
    private String instituicao;

    public String getInstituicao() {
        return instituicao;
    }
    @Override
    public Integer getNumero() {
        return super.getNumero();
    }
}

public class Pessoa<T>{
    private String nome;
    private T registro;

    public String getNome() {
        return nome;
    }
    public T getRegistro() {
        return registro;
    }
    public <E> void registrar(E documento) {
        //codigo
    }
}

public class Aluno extends Pessoa<CarteiraEstudantil> {

    @Override
    public CarteiraEstudantil getRegistro() {
        return super.getRegistro();
    }
    @Override
    public <String> void registrar(String documento) {
        //codigo
    }
}

public class App {
    public static void main(String[] args) {
        Aluno pedro = new Aluno();
        Funcionario joao = new Funcionario();
        Pessoa<String> raphael = new Pessoa<>();
        String s1= "MG1123456";
        pedro.registrar(s1);
        Long l1 = 12345678910L;
        joao.registrar(l1);
        Double d1 = 123.45;
        raphael.registrar(d1);
    }
}
```

Código 2.5: Usando parâmetros de tipos nas implementações das classes.

Já na declaração da classe genérica `Pessoa` (linha 21), `Pessoa<T>` é lido como "Pessoa de T" e `T` determina o tipo que as instâncias de `Pessoa` podem conter. O parâmetro de tipo `T`, definido na declaração da classe `Pessoa`, é usado também na declaração do atributo `registro` (linha 23) e do método genérico `getRegistro` (linha 28). E o parâmetro de tipo `E` é definido na declaração do método genérico `registrar` (linha 31).

Na declaração da subclasse `Aluno` (linha 36), a classe genérica `Pessoa` recebe como argumento de tipo `CarteiraEstudantil` que substitui o parâmetro de tipo T. Dessa maneira, quando o método genérico `getRegistro` (linha 39) é sobrescrito, a variável de tipo `T` é sobreposta pelo tipo `CarteiraEstudantil`. Já quando o método `registrar` (linha 43) é sobrescrito, o parâmetro de tipo `E` é substituído pelo tipo `String`.

E pode-se observar ainda no código 2.5, que ao instanciar a classe genérica `Pessoa` (linha 52), o construtor genérico da classe é chamado fornecendo um conjunto vazio de argumentos de tipo <> (denominado informalmente de diamante); e que nenhum argumento de tipo é passado ao chamar o método `registrar` (linha 58). Isso se deve ao fato de que o compilador Java irá inferir o argumento de tipo mais específico que fará com que o tipo de chamada seja correto (ORACLE, 2016c; BRACHA, 2004),

Além dos parâmetros de tipos, existem também os curingas que nos códigos genéricos são representados pelo ponto de interrogação (`?`) e que representam um tipo desconhecido (ORACLE, 2016c). Os curingas podem ser usados nas declarações de campos, variáveis locais, tipos de parâmetros e tipos de retorno (ARNOLD; GOSLING, 2007); mas não podem ser utilizados como argumento de tipo para uma invocação de método genérico, uma criação de instância de classe genérica ou um supertipo (ORACLE, 2016c).

Os curingas podem ser ilimitados e limitados. Os curingas ilimitados são especificados utilizando o caractere `?` e representam qualquer espécie (implicitamente o limite superior é `Object`). Já os curingas limitados, ao contrário do parâmetros de tipos limitados, podem ter somente um limite superior ou um limite inferior (ARNOLD; GOSLING, 2007). Na declaração onde o tipo representa o limite superior do tipo do curinga é utilizado o caractere `?`, a palavra-chave `extends` e o seu limite superior; e na declaração onde o tipo representa o limite inferior do tipo do curinga é usado o caractere `?`, a palavra-chave `super` e o seu limite inferior (ORACLE, 2016c).

O tipo curinga com limite superior é utilizado para relaxar as restrições em uma variável de tipo, porque nele são aceitos o tipo que representa o limite superior e as suas subclasses. O tipo curinga com limite inferior maximiza a flexibilidade, porque nele aceita-se objetos que sejam do tipo inferior ou do tipo de suas superclasses (ORACLE, 2016c).

Para exemplificar o uso do curinga com limite superior, inferior e ilimitado; o código 2.6 apresenta a classe `App` do código 2.5 modificada. Nele, os métodos `imprimirNome`, `adicionarAluno` e `imprimirElemento` têm como parâmetros `ArrayList` que utilizam o tipo

curinga como parâmetros de tipos.

```
1  public class App {
2
3      static void imprimirNome(ArrayList<? extends Pessoa> lista) {
4          for (Pessoa pessoa : lista) {
5                  System.out.println(pessoa.getNome());
6          }
7      }
8
9      static void adicionarAluno(ArrayList<? super Aluno> lista, Aluno a) {
10         lista.add(a);
11     }
12
13     static void imprimirElemento(ArrayList<?> lista){
14         for (Object object : lista) {
15                 System.out.println(object);
16         }
17     }
18
19     public static void main(String[] args) {
20         Aluno pedro = new Aluno();
21         Funcionario joao = new Funcionario();
22         Pessoa<String> raphael = new Pessoa<>();
23
24         ArrayList<Pessoa> listaPessoa = new ArrayList<>();
25         ArrayList<Aluno> listaAluno = new ArrayList<>();
26         ArrayList<String> listaString = new ArrayList<>();
27
28         imprimirNome(listaPessoa);
29         imprimirNome(listaAluno);
30         adicionarAluno(listaPessoa, pedro);
31         adicionarAluno(listaAluno, pedro);
32         imprimirElemento(listaPessoa);
33         imprimirElemento(listaAluno);
34         imprimirElemento(listaString);
35     }
36 }
```

Código 2.6: Usando curingas na implementação da classe *App*.

No método `imprimirNome` (linha 3), o parâmetro `ArrayList` precisa ser do tipo da classe `Pessoa` ou de qualquer subclasse de `Pessoa` (`Aluno`, `Funcionario`). Sendo assim, `Pessoa` é o limite superior do tipo curinga. Então, ao chamar o método `imprimirNome`, é possível passar como parâmetro um `ArrayList` do tipo `Pessoa` e um `ArrayList` do tipo `Aluno` (linhas 28 e 29).

O método `adicionarAluno` (linha 9) utiliza como parâmetro um `ArrayList` que precisa ser do tipo da classe `Aluno` ou de qualquer superclasse de `Aluno`. Dessa forma, `Aluno` é o limite inferior do tipo curinga. Assim, ao chamar o método para armazenar as instâncias da classe `Aluno`, é possível passar como parâmetro `ArrayList` do tipo `Aluno` (linha 31) ou da sua superclasse `Pessoa` (linha 30).

Finalmente, o método `imprimirElemento` (linha 13) aceita `ArrayList` de qualquer tipo

(implicitamente o limite superior é `Object`). Então, ao chamar o método são passados como parâmetro um `ArrayList` do tipo `Pessoa` (linha 32), um do tipo `Aluno` (linha 33) e um do tipo `String` (linha 34).

2.4 Reflexão

A reflexão possibilita que programas examinem ou modifiquem o comportamento em tempo de execução. Na linguagem de programação Java, a reflexão é um recurso poderoso utilizado, por exemplo, em depuradores e ferramentas de testes (ORACLE, 2016d). Porém, a reflexão deve ser usada como último recurso, onde os mecanismos de programação orientado a objetos já não atendem as necessidades de encapsulamento (ARNOLD; GOSLING, 2007). O mal uso da reflexão pode causar sobrecarga no desempenho, exigência de permissão em tempo de execução e quebra de abstrações orientadas a objetos (CÓRDOBA-SÁNCHEZ; LARA, 2016).

Na figura 2.1 serão mostradas algumas classes e interfaces que suportam a reflexão. As classes `Class` e `Package` estão no pacote `java.lang` e as demais estão no pacote `java.lang.reflect`.

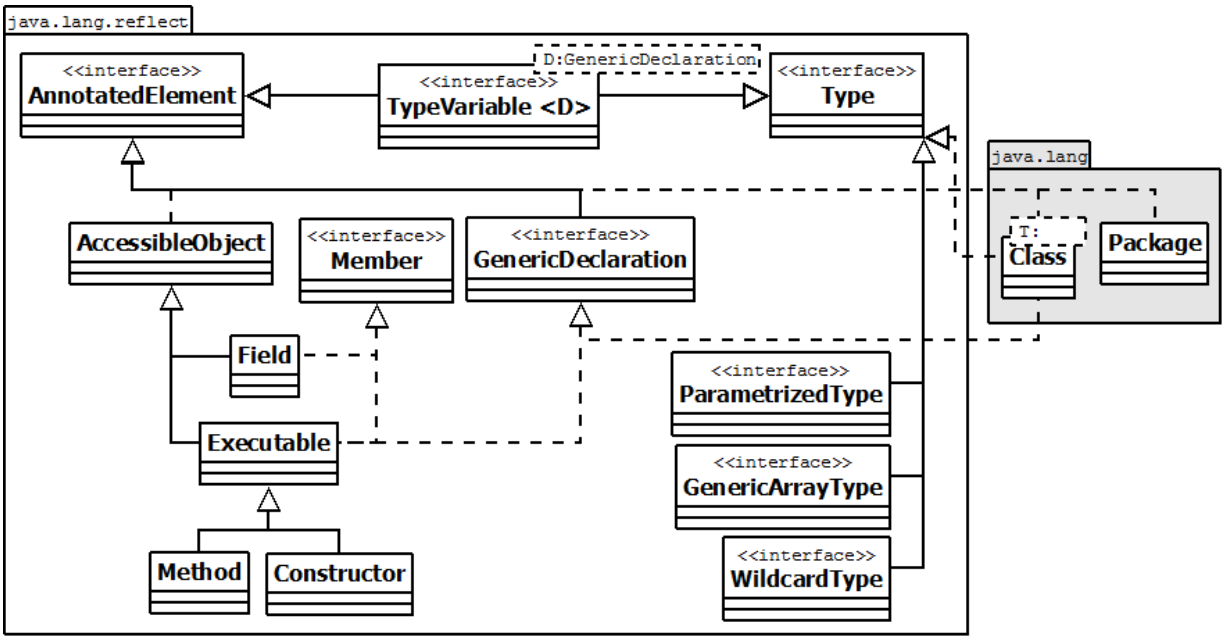


Figura 2.1: A hierarquia de tipos em Java 8 que oferecem suporte a reflexão.

Pode-se observar na figura 2.1 que o mecanismo de reflexão em Java é baseado nas especializações de: `Type`, `Member`, `AccessibleObject`, `GenericDeclaration` e `AnnotatedElement`.

A interface `Type`, uma interface de marcação e sem métodos, representa todos os tipos que existem em um programa Java. Na figura 2.1 observa-se que ela é implementada pela classe `Class`, que representa todos os tipos concretos, e estendida pelas interfaces que representam os tipos genéricos: tipos parametrizados (`ParametrizedType`), variáveis de tipo

(`TypeVariable<D>`), curingas (`WildcardType`) e *arrays* genéricos (`GenericArrayType`) (ARNOLD; GOSLING, 2007).

A classe `Class` é uma classe genérica (`Class<T>`), onde cada objeto é um tipo parametrizado da classe que ele representa. Toda classe, enumerado, interface, anotação e *array* tem uma instância da classe `Class` que é o ponto inicial para a reflexão desse objeto. A partir dessa instância é possível, por exemplo, obter todos os membros de uma classe, informações sobre os tipos da classe, instanciar uma classe e chamar um método. Existem 4 maneiras para obter o objeto `Class`, descritas a seguir (ARNOLD; GOSLING, 2007):

- Utilizando o método `getClass()` através do objeto.
- Usando um literal de classe (nome da classe) seguido de `.class`.
- Empregando o método estático `forName()`, que permite que uma classe procure pelo seu nome plenamente qualificado.
- Aplicando os métodos de reflexão, tais como `Class.getClass`, que retornam objetos `Class` para as classes e interfaces aninhadas.

A classe `Class` possui 3 métodos que fornecem o nome da classe: `getSimpleName()`, `getCanonicalName()` e `getName()`. O método `getSimpleName()` fornece o nome simples do tipo, que é o nome real (literal) de uma classe ou interface. Somente as classes anônimas não possuem o nome simples. Já o método `getCanonicalName()` provê o nome do seu pacote seguido pelo seu nome simples, chamado de nome canônico ou de nome plenamente qualificado ou FQN - *Fully Qualified Name*. E finalmente o método `getName()` providencia o nome binário de uma classe ou interface, que é seu nome canônico, usado para comunicação com a máquina virtual. Para tipos aninhados, o nome binário é o nome binário do seu tipo envolvente seguido por $ e o nome simples do tipo aninhado; para a classe anônima, o nome binário é o nome binário do seu tipo envolvente seguido por $ e um número; para tipos primitivos e `void`, o nome simples, o nome canônico e o nome binário são os mesmos (ARNOLD; GOSLING, 2007).

Alguns métodos da classe `Class` verificam se sua representação é, por exemplo, um tipo enumerado (`isEnum()`), uma interface (`isInterface()`), uma anotação(`isAnnotation()`), um *array* (`isArray()`), ou um tipo primitivo (`isPrimitive()`).

A tabela 2.2 apresenta alguns métodos que examinam os membros de uma classe: atributos (objetos `Field`), métodos (objetos `Method`) e construtores (objetos `Constructor`).

Tabela 2.2: Métodos que examinam os componentes da classe (MAGNO, 2015).

Tipo de Membro: Descrição	Elemento	Método de Instrospecção
Público: Elementos públicos da própria classe e elementos públicos herdados de suas superclasses.	`Field`	`getField(String nome)` `getFields()`
	`Method`	`getMethod(String nome,Class parametros)` `getMethods()`
	`Constructor`	`getConstructor(Class parametros)` `getConstructors()`
Declarado: Elementos da própria classe com qualquer grau de acesso, mas sem herança de outros elementos.	`Field`	`getDeclaredField(String nome)` `getDeclaredFields()`
	`Method`	`getDeclaredMethod(String nome,Class parametros)` `getDeclaredMethods()`
	`Constructor`	`getDeclaredConstructor(Class parametros)` `getDeclaredConstructors()`

A interface `Member` representa os membros de uma classe, como observado na figura 2.1, e suas implementações definem construtores (`Constructor`), métodos (`Method`) e atributos (`Field`).

A classe `Executable` (figura 2.1) é uma classe compartilhada com as funcionalidades comuns das classes `Method` e `Constructor` (ORACLE, 2016a). Embora uma classe ou interface possa ser membro de uma outra classe, a classe `Class`, por razões históricas, não implementa a interface `Member`, mas suporta os métodos com a mesma assinatura (`getDeclaringClass()`, `getName()`, `getModifiers()` e `isSynthetic()`) (ARNOLD; GOSLING, 2007).

A classe `Method` fornece métodos para invocar métodos sobre um dado objeto e para acessar informações sobre modificadores, tipos de retorno, parâmetros, anotações e exceções lançadas (*thrown exceptions*) (ORACLE, 2016d). A seguir são apresentados como invocar um método e como obter informação sobre o tipo de retorno do método (ORACLE, 2016a):

- `invoke(Object obj, Object... args)`: chama o método da classe que o objeto `Method` representa passando como argumento a instância da classe e os parâmetros especificados no método da classe.

- `getReturnType()`: retorna o tipo do retorno no método representado pelo objeto `Method`.

A classe `Constructor` fornece métodos para criar novas instâncias de classe, encontrar construtores e obter informações sobre modificadores, parâmetros, anotações e exceções lançadas (*thrown exceptions*) (ORACLE, 2016d). Por exemplo, o método `newInstance()` cria uma nova instância da classe que objeto `Constructor` representa (ARNOLD; GOSLING, 2007).

A classe `Field` possui métodos para acessar informações do tipo do atributo, configurar e obter valores de um atributo em um determinado objeto e analisar os modificadores do atributo (ORACLE, 2016d). A seguir, alguns desses métodos são apresentados (ORACLE, 2016a):

- `getType()`: identifica o tipo declarado para o atributo representado pelo objeto `Field`.
- `setInt(Object obj, int i)`: define o valor de um atributo do tipo `int` no objeto `obj`.
- `getInt(Object obj)`: obtém o valor do tipo `int` de um atributo estático ou da instância no objeto `obj`.
- `isEnumConstant()`: verifica se o atributo é um elemento do tipo enumerado.

A classe `AccessibleObject`, da figura 2.1, é a classe base para os objetos das classes `Constructor`, `Method` e `Field` (ORACLE, 2016a). Ela permite habilitar ou desabilitar a verificação dos modificadores de acesso no nível da linguagem, como `public` e `private`. Sendo assim, ao desabilitar esta verificação, é possível acessar um membro de uma classe por reflexão indiferentemente do controle de acesso no nível da linguagem. Por exemplo, o método `isAccessible()` verifica o valor do sinalizador atual de acessibilidade e o método `setAccessible (boolean flag)` muda o sinalizador para acessível ou não (ARNOLD; GOSLING, 2007).

A interface `GenericDeclaration`, como mostra a figura 2.1, é implementada pelas classes `Constructor`, `Method` e `Class` porque somente as classes, construtores e métodos podem ser declaradas de modo genérico. A interface `GenericDeclaration` define um único método `getTypeParameters`, retornando um *array* de objetos `TypeVariable` (ARNOLD; GOSLING, 2007) que representa as variáveis de tipo na declaração genérica (ORACLE, 2016a).

E finalmente a interface `AnnotatedElement` é implementada pelas classes `Class`, `Package`, `Field`, `Method` e `Constructor`. Ela permite que as anotações sejam lidas reflexivamente (ORACLE, 2016a) e somente as anotações com uma política de retenção `RetentionPolicy.RUNTIME` (anotações disponíveis durante a execução) podem ser consultadas (ARNOLD; GOSLING, 2007). A seguir são apresentados os métodos da interface `AnnotatedElement` (ORACLE, 2016a):

- `isAnnotationPresent(Class<T> annotationClass)`: verifica se uma anotação está presente em um determinado elemento. O valor retornado pelo método é `true` se a anotação estiver aplicada na classe ou se o tipo de anotação é herdável e a anotação estiver aplicada na classe mãe. Além disso, ele não detecta se a anotação é aplicada mais de uma vez sobre um elemento da classe.

- `getAnnotation(Class<T> annotationClass)`: fornece uma anotação específica de um determinado elemento para um tipo especificado se tal anotação está presente. Se o tipo de anotação é herdável e a anotação não está aplicada na classe filha, o método retorna a anotação da classe mãe. Porém, se a classe filha for anotada pela mesma anotação da classe mãe, a anotação da classe filha será retornada. Ele não detecta se a anotação é aplicada mais de uma vez.

- `getAnnotations()`: fornece anotações que estão presentes em um determinado elemento. Se o tipo de anotação é herdável e as anotações não estão aplicadas na classe filha, o método retorna as anotações da classe mãe. Porém, se a classe filha for anotada pelas mesmas anotações da classe mãe, as anotações das duas classes serão retornadas.

- `getAnnotationsByType(Class<T> annotationClass)`: fornece anotações de um determinado elemento. Se o tipo de anotação é herdável e as anotações não estão aplicadas na classe filha, o método retorna as anotações da classe mãe. Porém, se a classe filha for anotada pelas mesmas anotações da classe mãe, as anotações da classe filha serão retornadas. Ele detecta se o argumento é um tipo de anotação repetível.

- `getDeclaredAnnotation(Class<T> annotationClass)`: fornece uma anotação específica de um determinado elemento para um tipo especificado se tal anotação está diretamente presente e ignora anotações herdadas.

- `getDeclaredAnnotations()`: fornece anotações que estão diretamente presentes em um determinando elemento e ignora anotações herdadas.

- `getDeclaredAnnotationsByType(Class<T> annotationClass)`: fornece anotações de um determinado elemento para um tipo especificado. Este método ignora as anotações herdadas e detecta se o argumento é um tipo de anotação repetível.

O código 2.7 apresenta a declaração da classe `LogHardware` (linhas de 3 a 13) e o uso dos métodos de reflexão sobre a classe `LogHardware` (linhas 15 a 36). O método `getClass()` (linha 19) fornece o objeto `class`; o método `getCanonicalName()` (linha 21) provê o nome canônico da classe; o método `getMethods()` (linha 23) fornece todos o métodos públicos e herdados (`setNome()` e `getNome()`); o método `invoke()` (linha 26) invoca o método `setNome()` obtido pelo método `getMethod()` (linha 25); o método `newInstance()` (linha 28) cria uma nova instância da classe (`lh2`); o método `set()` (linha 32) altera o valor do atributo `nome`

```
1 @Usuario(apelido = "Gabriel", numero = @Cpf(123456789L), vinculo = Funcao.
    Discente)
2 @Usuario(apelido = "Jo", numero=@Cpf(98765432101L), vinculo = Funcao.Docente,
    ativo = false)
3 public class LogHardware extends Log{
4     private String nome;
5 
6         public String getNome() {
7             return nome;
8         }
9 
10        public void setNome(String nome) {
11            this.nome = nome;
12        }
13 }
14 
15 public class App{
16     public static void main(String[] args) {
17 
18         LogHardware lh1 = new LogHardware();
19         Class c = lh1.getClass();
20 
21         String nomeCanonico = c.getCanonicalName();
22 
23         Method[] metodos = c.getMethods();
24 
25         Method m = c.getMethod("setNome", String.class);
26         Object o = m.invoke(lh1, "Log1");
27 
28         LogHardware lh2 = (LogHardware) c.newInstance();
29 
30         Field f = c.getDeclaredField("nome");
31         f.setAccessible(true);
32         f.set(lh1, "Log2");
33 
34         Annotation[] annotations = c.getDeclaredAnnotations();
35     }
36 }
```

Código 2.7: Exemplos de uso dos métodos de reflexão em Java.

(obtido pelo método `getDeclaredField()` (linha 30)) porque o sinalizador de acessibilidade foi configurado para `true` através do método `setAccessible()` (linha 31); e o método `getDeclaredAnnotations()` (linha 34) provê todas as anotações diretamente presentes na classe `LogHardware`.

2.5 *Framework Obinject*

Segundo Carvalho (2013), o principal objetivo do *framework Obinject* é a indexação e persistência orientada a objetos. Além disso, o *framework* está dividido em 4 módulos (Meta, Armazenamento, Dispositivos e Blocos), como é ilustrado na figura 2.2.

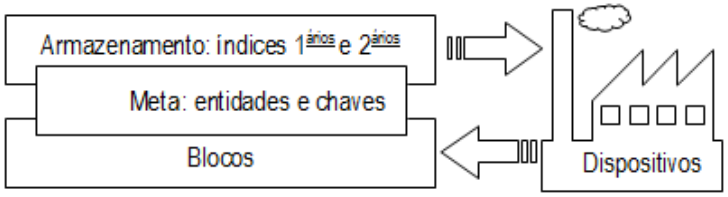


Figura 2.2: Organização dos módulos do *framework Obinject* (CARVALHO, 2013).

O módulo Meta define o vínculo entre as classes de aplicação do usuário e o *framework*, expresso através de entidades persistentes e chaves indexadas associadas aos domínios de indexação.

O módulo Armazenamento especifica a maneira como são implementadas as estruturas de índices primários e secundários.

Já o módulo Dispositivos define os recursos computacionais responsáveis pelo armazenamento físico das estruturas de índices.

E o módulo Blocos define a maneira pela qual as entidades persistentes e chaves indexadas são armazenadas em blocos ou páginas.

A arquitetura do *framework Obinject* é baseada nos conceitos de índices primários e secundários, apresentados a seguir.

Os índices primários armazenam os valores dos atributos de um objeto (entidades persistentes) e indexam através de um UUID (*Universally Unique Identifier* ou Identificador Único Universal). Ou seja, os UUIDs são as identificações utilizadas pelo índices primários para armazenamento e recuperação dos objetos no meio persistente. O índice primário de uma classe é definido pela implementação da interface *Entity* (CARVALHO, 2013; FERRO, 2012).

Os índices secundários replicam os atributos dos objetos que definem uma chave de indexação. Nas estruturas de indexação, baseadas em árvore, os nós internos contêm as chaves de roteamento e os nós folha têm os valores das chaves com os seus respectivos UUIDs dos objetos. Neste caso, a partir de uma chave armazenada no índice secundário, é possível obter o seu UUID e, por sua vez, possibilita recuperar o objeto em um índice primário. O índice secundário de uma classe de aplicação é definido pela implementação das sub-interfaces de *Key* (CARVALHO, 2013). Essas sub-interfaces estão relacionadas aos domínios de indexação como *PrimaryKey*, *Order*, *Point*, *Rectangle*, *Edition* e *GeoPoint* (FERRO, 2012).

A estrutura de dados utilizada no *framework Obinject* para a persistência de entidades é a *BTree+* e para indexar as chaves são usadas as estruturas de dados: *BTree+*, *RTree* e *MTree* (FERRO, 2012).

O código 2.8 mostra a implementação da classe de aplicação `City` utilizando as anotações de persistência e indexação do *framework Obinject*.

```
1  @Persistent
2  public class City {
3
4      @PrimaryKey
5      @EditionFirst
6      @OrderFirst
7      private String name;
8
9      @PointFirst
10     @GeoPointFirst
11     @RectangleFirst
12     private double latitude;
13
14     @PointFirst
15     @GeoPointFirst
16     @RectangleFirst
17     private double longitude;
18
19     //gets e sets
20 }
```

Código 2.8: Exemplos de uso das anotações do *framework Obinject*

Pode-se observar, no código 2.8, a anotação `@Persistent` (linha 1) indicando que a classe `City` será persistida.

Já o atributo `name` (linha 7), ao ser anotado por `@PrimaryKey` (linha 4), será a chave de identificação da classe `City` através de uma estrutura *BTree+*.

Além disso, o atributo `name`, ao ser anotado por `@EditionFirst` (linha 5), será indexado em um índice que agiliza consultas que necessitam de semelhanças entre palavras usando a estrutura *MTree*; e ao ser anotado por `@OrderFirst` (linha 6), será indexado em um índice que agiliza consultas que necessitam de ordenação usando a estrutura *BTree+*.

Finalmente, os atributos `latitude` (linha 12) e `longitude` (linha 17), quando anotados por `@PointFirst`, indica-se que eles serão indexados em uma estrutura que agiliza consultas que necessitam de semelhanças no espaço euclidiano usando a estrutura métrica *MTree*. Quando esses atributos são anotados por `@GeoPointFirst`, indica-se que eles serão indexados em um índice que agiliza consultas que necessitam de semelhanças no espaço esférico usando a estrutura métrica *MTree*. E quando anotados por `@RectangleFirst`, indica-se que eles serão indexados em um índice que agiliza consultas que necessitam de semelhanças no espaço euclidiano usando a estrutura espacial *RTree*.

O atributo `latitude` estará na primeira posição das chaves de indexação dos domínios *Point*, *GeoPoint* e *Rectangle* e o atributo `longitude` estará na segunda posição destas chaves. Isto acontece porque a ordem dos atributos na chave de indexação segue a ordem como eles são declarados na classe.

Vale observar que o nome das anotações dos domínios de indexação, com exceção da anotação

`PrimaryKey`, é formado pelo nome do domínio seguido do ordinal que representa o índice. Por exemplo, para que o usuário possa indexar dois atributos em dois índices distintos no domínio *Order*, é necessário aplicar a anotação `@OrderFirst` para o atributo que formará a chave do índice de número um e `@OrderSecond` para o atributo que formará a chave do índice de número dois.

A figura 2.3 exemplifica a geração das classes empacotadoras, responsáveis pelo vínculo entre a classe de aplicação `City` e o *framework Obinject*.

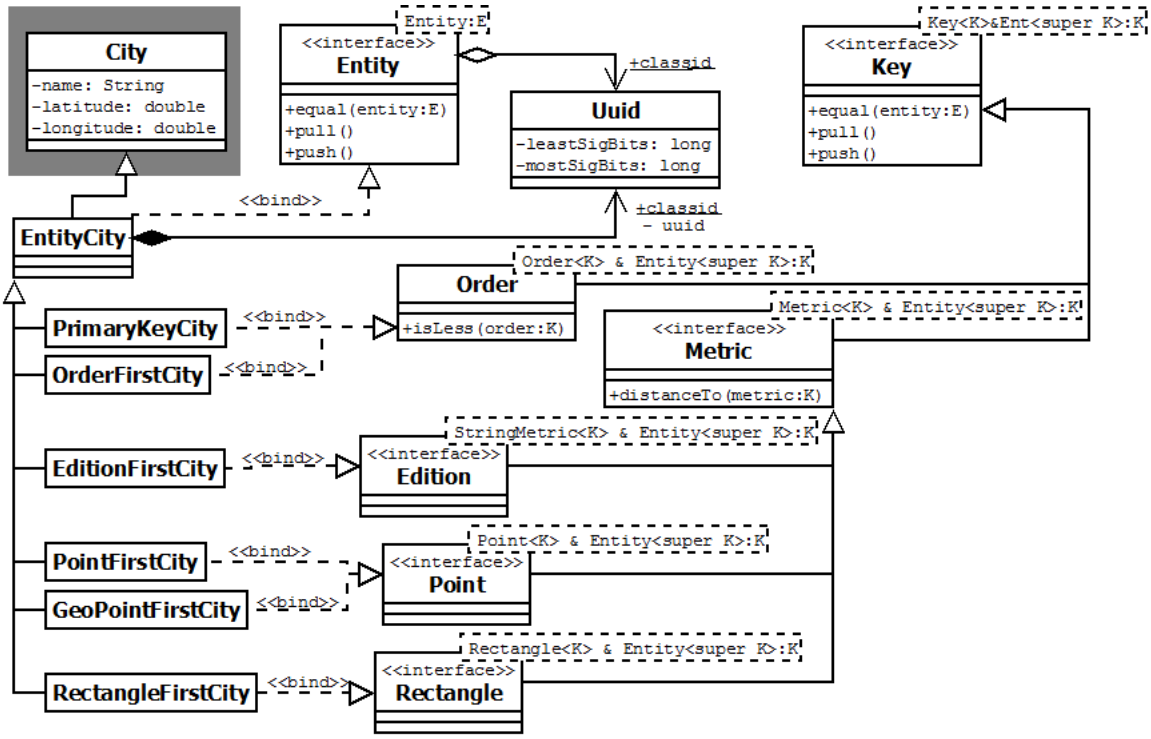


Figura 2.3: Classes empacotadoras geradas pelo *framework Obinject* a partir das anotações aplicadas na classe de aplicação City.

A classe `City`, na figura 2.3, representa a classe que o usuário deseja tornar persistente. Através do uso de metaprogramação, combinando metaclasses do *framework* e metadados (anotações) da classe de aplicação `City` (código 2.8), o *framework* cria a classe empacotadora de persistência (`EntityCity`) e as classes empacotadoras de indexação (`PrimaryKeyCity`, `OrderFirstCity`, `EditionFirstCity`, `PointFirstCity`, `GeoPointCity` e `RectangleFirstCity`). Estas classes implementam as interfaces do *framework* (`Entity` e as sub-interfaces de `Key`) e são especializadas na classe de aplicação `City`.

Vale ressaltar que as anotações do *framework Obinject*, como `@Persistent`, `@PrimaryKey`, `@OrderFirst`; aplicadas à classe de aplicação, permitem que a geração de classes de empacotamento seja de forma automatizada por meio de técnicas de instrumentação e compilação de novas classes em tempo de execução (FERRO, 2012).

2.6 Processamento e Análise de ImagensDigitais

2.6.1 Introdução

A visão computacional tem como objetivo auxiliar, na área de imagens, a interpretação humana e a percepção por máquina (GONZALEZ; WOODS, 2010).

O processamento e a análise de imagens são dois níveis de abstração estabelecidos nas tarefas que utilizam visão computacional. O processamento de imagens (baixo nível) é formado por um conjunto de técnicas para capturar, representar e transformar imagens com o auxílio do computador. Ele inclui métodos como realçar o contraste de imagens, reduzir o ruído de imagens, extrair bordas e comprimir imagens. Já a análise de imagens (alto nível) abrange métodos como segmentação da imagem em regiões ou objetos de interesse, descrição, reconhecimento ou classificação desses objetos (PEDRINI; SCHWARTZ, 2008).

As etapas de um sistema de processamento de imagens, mostradas na figura 2.4, são (PEDRINI; SCHWARTZ, 2008):

- **Aquisição:** é a obtenção de imagens. Os dispositivos que são utilizados para capturar imagens são, por exemplo, câmeras digitais, *scanners*, satélites, raios x, etc.
- **Pré-processamento:** é um melhoramento da qualidade da imagem através do uso de técnicas.
- **Segmentação:** é o uso de técnicas para extração e identificação de objetos de interesses nas imagens.
- **Representação e descrição:** é a utilização de técnicas para armazenar, manipular e extrair características ou propriedades do objeto extraído da imagem na etapa de segmentação.
- **Reconhecimento e interpretação:** é a atribuição de um rótulo aos objetos da imagem, de acordo com as características fornecidas pelos seus descritores.
- **Base de conhecimento:** é a codificação do conhecimento sobre o domínio do problema. Conforme Gonzalez e Woods (2010), a base de conhecimento orienta a operação de cada módulo e controla a interação entre os módulos. Por isso é utilizado setas bidirecionais entre os módulos e a base de conhecimento.

2.6.2 Imagem - Definição, Representação e Atributos

Segundo Gonzalez e Woods (2010), uma imagem pode ser definida como uma função bidimensional, *f(x,y)*, em que x e y são coordenadas espaciais (plano), e a amplitude de f em

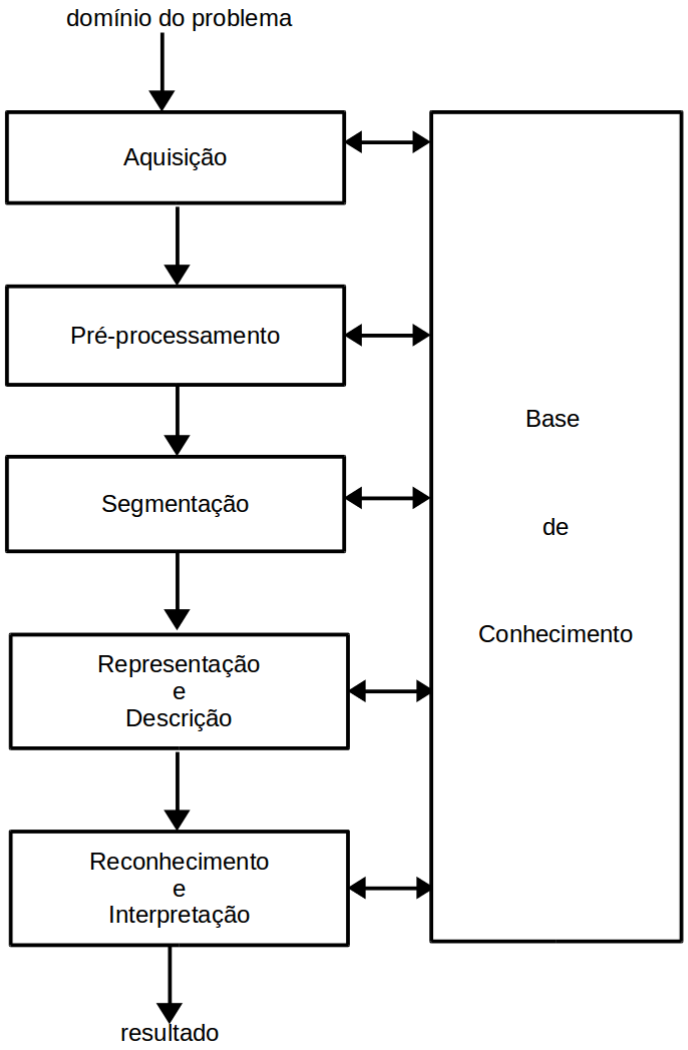


Figura 2.4: Etapas de um sistema de processamento de imagens (PEDRINI; SCHWARTZ, 2008).

qualquer par de coordenadas *(x,y)* é chamada de intensidade ou nível de cinza da imagem nesse ponto. Quando x, y e os valores de intensidade de f são quantidades finitas e discretas é chamado de imagem digital.

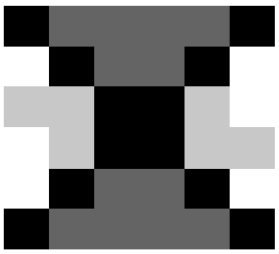
Por convenção, a origem da imagem digital está localizada no canto superior esquerdo da imagem (PEDRINI; SCHWARTZ, 2008). Sendo assim, a largura da imagem corresponde ao eixo x e a altura da imagem corresponde ao eixo y.

A imagem digital pode ser representada por uma matriz bidimensional. Cada elemento da matriz, que corresponde ao *pixel* (*picture element*) da imagem, está associado a um valor naquele ponto, que em uma imagem monocromática representa o nível de cinza. Além disso, a dimensão dessa matriz corresponde ao tamanho da imagem (PEDRINI; SCHWARTZ, 2008).

Vale ressaltar que a escala de cinza apresenta 256 níveis de intensidade e por convenção, o valor 0 é atribuído ao nível de cinza mais escuro (cor preta) e o valor 255 é atribuído ao nível

de cinza mais claro (cor branca) (PEDRINI; SCHWARTZ, 2008).

A figura 2.5(a) mostra uma imagem exemplo e (b) a representação matricial da imagem exemplo.



(a)

	0	1	2	3	4	5
0	0	100	100	100	100	0
1	255	0	100	100	0	255
2	200	200	0	0	200	255
3	255	200	0	0	200	200
4	255	0	100	100	0	255
5	0	100	100	100	100	0

(b)

Figura 2.5: (a) imagem exemplo e (b) representação matricial da imagem exemplo.

Pode-se observar que o valor 0 na linha 0 e coluna 0 da matriz (figura 2.5(b)), corresponde ao nível de cinza mais escuro (preto) no primeiro *pixel* da figura 2.5(a) (canto superior esquerdo). Além disso, como a imagem exemplo mostrada na figura 2.5(a) tem dimensão de 6x6 *pixels*, a representação matricial mostrada na figura 2.5(b) apresenta 6 linhas e 6 colunas.

2.6.3 Textura

A textura é uma das características utilizadas pelo sistema visual humano no processo de análise e interpretação da imagem. Ela permite obter informações da imagem como a distribuição espacial e o arranjo estrutural das superfícies (SCHWARTZ et al., 2012).

Os descritores texturais são obtidos da imagem de entrada e podem ser classificados, conforme apresentado por Schwartz et al. (2012), da seguinte forma:

- **Abordagem estatística:** os métodos, nessa abordagem, representam as propriedades da textura indiretamente e de maneira probabilística.
- **Abordagem baseada no processamento de sinal:** seus métodos extraem descritores de uma representação de imagem resultante de aplicações de transformações da imagem de entrada, tal como *Fourier* ou transformadas *wavelet*.
- **Abordagem geométrica:** nessa abordagem, após identificação de primitivas que compõe a textura, também conhecidas como *textels*, duas classes de métodos podem ser considerados para extração de características: uso de descritores extraídos de primitivas para descrever a textura e a extração de regras para descrever a disposição espacial dessas primitivas.

- **Abordagem baseada em modelos paramétricos:** a análise e síntese de textura é, nessa abordagem, feita utilizando parâmetros.

2.6.3.1 Abordagem Estatística

Essa abordagem apresenta as estatísticas de primeira ordem e segunda ordem. Na primeira ordem, as propriedades estatísticas da imagem são extraídas utilizando histogramas. Já na segunda ordem, elas são obtidas usando a matriz de coocorrência dos níveis de cinza (GLCM - *Gray-Level Coocurrence Matrix*)(SCHWARTZ et al., 2012).

O histograma de uma imagem corresponde à distribuição dos níveis de cinza da imagem, o qual pode ser representado por um gráfico indicando o número de *pixels* na imagem para cada nível de cinza. Além disso, um mesmo histograma pode pertencer a imagens diferentes, pois ele apresenta somente valores de intensidade e não a distribuição espacial (PEDRINI; SCHWARTZ, 2008).

Conforme Gonzalez e Woods (2010), o histograma é normalizado usando a equação 2.1 :

$$P(i) = \frac{h(i)}{n} \tag{2.1}$$

onde *h(i)* representa o número de ocorrências de *pixels* apresentando intensidade i e n indica o número total de *pixels* da imagem. Assim, *P(i)* representa uma estimativa da probabilidade de ocorrência do nível de cinza i em uma imagem. É importante observar que a soma de todos os componentes, após a normalização do histograma, é igual a 1.

A figura 2.6(a) mostra o histograma gerado a partir da figura 2.5(a) sem normalização. O eixo horizontal do histograma corresponde aos valores de intensidade de cinza e o eixo vertical do histograma representa as ocorrências de níveis de cinza (número de *pixels*). A figura 2.6(b) mostra o histograma gerado a partir da figura 2.5(a), mas normalizado. O eixo horizontal do histograma corresponde aos valores de intensidade de cinza e o eixo vertical do histograma indica uma estimativa da probabilidade de ocorrência do nível de cinza.

Pode-se notar na figura 2.6(a) que os níveis de cinza com valores 0 e 100 apresentam a ocorrência igual a 12, ou seja, 12 *pixels* na figura 2.5(a) apresentam valor 0 e 12 *pixels* apresentam valor 100. Os níveis de cinza 200 e 255 possuem a frequência de 6 *pixels*. Já os demais níveis de cinza apresentam ocorrência igual a 0, pois eles não estão presentes na figura 2.5(a).

Na figura 2.6(b), os níveis de cinza 0 e 100 da figura 2.5(a) apresentam os valores 0,333 e os níveis de cinza 200 e 255 apresentam os valores 0,166. Os valores 0,333 e 0,166 foram obtidos pela equação 2.1.

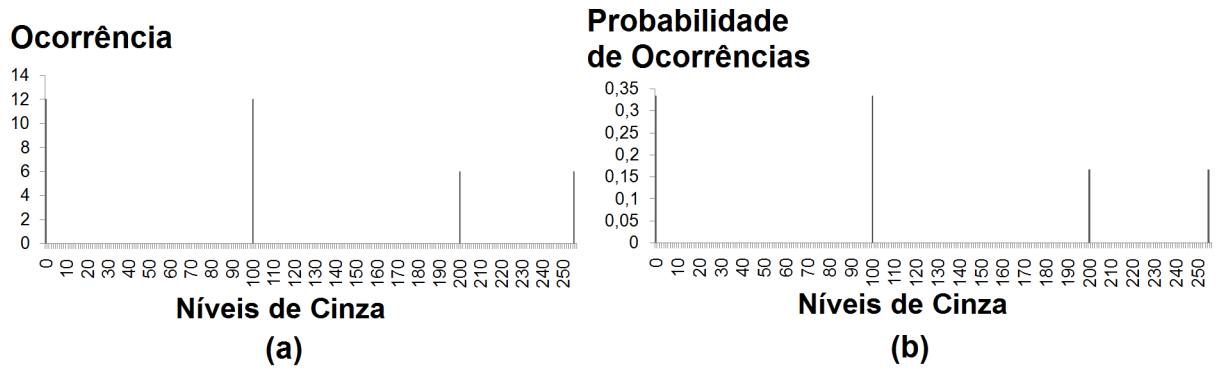


Figura 2.6: Histograma da imagem exemplo (a) sem normalização e (b) com normalização.

Vale ressaltar que os componentes dos histogramas, em imagens escuras, estão concentrados nas intensidades mais escuras na escala de cinza, enquanto que, em imagens claras, estes estão concentrados nas intensidades mais claras na escala de cinza. Nas imagens com baixos contrastes, os componentes dos histogramas concentram-se no meio da escala e nas imagens com altos contrastes, os componentes estão distribuídos por toda escala de cinza (GONZALEZ; WOODS, 2010).

Após a obtenção do histograma de uma imagem, é possível obter os dados estatísticos como intensidade média, variância e assimetria. As equações destas medidas são apresentadas nas equações 2.2, 2.3 e 2.4 respectivamente (GONZALEZ; WOODS, 2010):

$$\mu = \sum_{i=0}^{H_g} g_i h(i) \tag{2.2}$$

$$\sigma^2 = \sum_{i=0}^{H_g} (g_i - \mu)^2 h(i) \tag{2.3}$$

$$s = \sum_{i=0}^{H_g} (g_i - \mu)^3 h(i) \tag{2.4}$$

onde g_i representa o tom de cinza para o i-ésimo *pixel*, *h(i)* denota o número de ocorrências de *pixels* apresentando intensidade i e H_g indica a quantidade de níveis de cinza.

As equações 2.5 e 2.6 calculam a energia e entropia a partir dos histogramas (PEDRINI; SCHWARTZ, 2008).

$$E = \sum_{i=0}^{H_g} [P(i)]^2 \tag{2.5}$$

$$H = -\sum_{i=0}^{H_g} P(i)log[P(i)] \tag{2.6}$$

onde H_g denota a quantidade de níveis de cinza e *P(i)* representa uma estimativa da probabilidade de ocorrência do nível de cinza i em uma imagem (equação 2.1).

Para obter as informações de textura na abordagem estatística de 2ª ordem, são utilizadas 4 matrizes de dependências espaciais de tons de cinza, chamadas de matrizes de coocorrência dos níveis de cinza (GLCM - *Gray Level Coocurrence Matrix*). Elas são calculadas por vários relacionamentos angulares e distância entre os *pixels* vizinhos. Os ângulos utilizados para obter as matrizes são 0°, 45°, 90° e 135° (HARALICK et al., 1973), como mostra a figura 2.7; e o valor da distância geralmente é 1 (ELEYAN; DEMIREL, 2011).

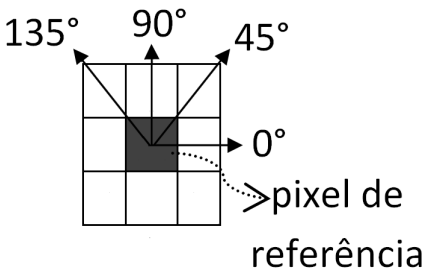


Figura 2.7: Direções das matrizes de coocorrência.

É importante observar que o número de linhas e colunas das matrizes de coocorrência são iguais a quantidade de níveis de cinza contidos na textura (PEDRINI; SCHWARTZ, 2008). Assim, em uma imagem com 4 níveis de cinza, a dimensão da matriz de coocorrência será 4x4.

Como o número de níveis de cinza definem as dimensões das matrizes de coocorrência, um método é utilizado para quantificar as intensidades em algumas faixas permitindo manter sob controle o tamanho da matriz de coocorrência e reduzir o custo computacional. Assim, em uma imagem com 256 níveis de cinza, para que suas matrizes de coocorrência não tenham dimensões 256x256, os primeiros 32 níveis de cinza são iguais a 0, os próximos 32 são iguais 1 e sucessivamente. Dessa forma, a matriz resultante terá dimensão 8x8 (GONZALEZ; WOODS, 2010).

Na literatura são apresentadas as matrizes de coocorrência simétricas e assimétricas. Nas matrizes de coocorrência simétricas não é levado em conta a posição dos tons de cinza dos *pixels* vizinhos. Assim, os *pixels* vizinhos com tons de cinza i, j e j, i terão a mesma frequência. Ao contrário das matrizes de coocorrência simétricas, as matrizes de coocorrência assimétricas consideram qual tom de cinza pertence ao *pixel* de referência e qual tom de cinza pertence ao *pixel* vizinho. Dessa forma, os pares de *pixels* vizinhos i, j e j, i podem apresentar frequências iguais ou diferentes.

A figura 2.8(a) ilustra a representação matricial de uma imagem. As figuras 2.8(b-i) exibem as matrizes de coocorrência da figura 2.8(a). As figura 2.8(b-e) mostram as matrizes de coocorrência simétricas e as figuras 2.8(f-i) ilustram as matrizes de coocorrência assimétricas. É importante observar que cada elemento das matrizes de coocorrência representa a frequência que 2 tons de cinza são vizinhos à distância d e ângulo σ.

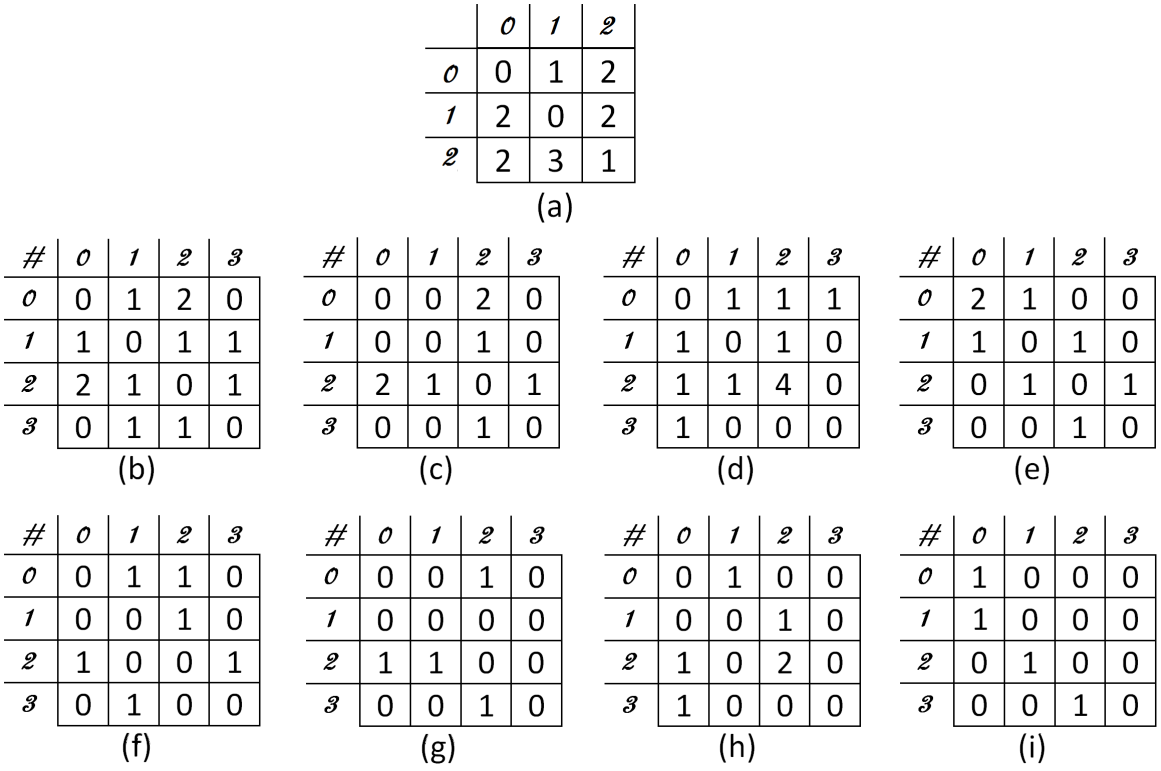


Figura 2.8: Representação matricial de uma imagem e as matrizes de coocorrência.

A figura 2.8(a) é uma representação matricial de uma imagem que possui dimensão 3x3 e 4 níveis de cinza. Já as figuras 2.8(b-e) são as matrizes de coocorrência simétricas de dimensão 4x4, criadas a partir da figura 2.8(a) e descrevem os *pixels* que são adjacentes um para o outro horizontalmente (matriz de coocorrência 0° - figura 2.8(b)), na diagonal direita (matriz de coocorrência 45° - figura 2.8(c)), verticalmente (matriz de coocorrência 90° - figura 2.8(d)) e na diagonal esquerda (matriz de coocorrência 135° - figura 2.8(e)) com distância igual a 1.

Observando a figura 2.8(a), a frequência que os *pixels* com os tons de cinza 0 e 2 são vizinhos a distância 1 e ângulo 0° é igual a 2. Então nas posições (0,2) e (2,0) da matriz de coocorrência simétrica com ângulo igual a 0° terão valores iguais a 2, como mostra a figura 2.8(b). Já a frequência que os *pixels* com os tons de cinza 2 e 2 são vizinhos a distância 1 e ângulo 90° é igual a 2. Mas como são contados em dobro, a posição (2,2) da matriz de coocorrência simétrica com ângulo igual a 90° terá valor 4, como mostra a figura 2.8(d).

E as figuras 2.8(f-i) são as matrizes de coocorrência assimétricas de dimensão 4x4, criadas a partir da figura 2.8(a) e descrevem os *pixels* que são adjacentes um para o outro horizontalmente (matriz de coocorrência 0° - figura 2.8(f)), na diagonal direita (matriz de coocorrência

45° - figura 2.8(g)), verticalmente (matriz de coocorrência 90° - figura 2.8(h)) e na diagonal esquerda (matriz de coocorrência 135° - figura 2.8(i)) com distância igual a 1.

Levando em conta qual tom de cinza pertence ao pixel de referência, qual tom de cinza pertence ao pixel vizinho e analisando a figura 2.8(a), a frequência que os *pixels* com os tons de cinza 3 e 1 são vizinhos a distância 1 e ângulo 0° é igual a 1; e a frequência que os *pixels* com os tons de cinza 1 e 3 são vizinhos a distância 1 e ângulo 0° é 0. Então, na matriz de coocorrência assimétrica com ângulo igual a 0° a posição (3,1) terá valor igual a 1 e a posição (1,3) terá valor igual a 0, como mostra a figura 2.8(f).

Existem duas formas para a normalização das matrizes de coocorrência:

- Calcular o coeficiente de normalização (R) para cada matriz de coocorrência (0°, 45°, 90° e 135°) onde são dados uma imagem com N *pixels* na horizontal N_x, N *pixels* na vertical N_y e uma distância d. Em seguida, dividir cada elemento das matrizes de coocorrência pelo seu coeficiente de normalização (HARALICK et al., 1973). As equações para calcular os coeficientes de normalização (R) são:

  - Coeficiente de normalização da matriz de coocorrência para ângulo igual a 0°:

$$R = 2N_y(N_x - d) \tag{2.7}$$

  - Coeficiente de normalização da matriz de coocorrência para ângulo igual a 45°:

$$R = 2(N_y - d)(N_x - d) \tag{2.8}$$

  - Coeficiente de normalização da matriz de coocorrência para ângulo igual a 90°:

$$R = 2N_x(N_y - d) \tag{2.9}$$

  - Coeficiente de normalização da matriz de coocorrência para ângulo igual a 135°:

$$R = 2(N_x - d)(N_y - d) \tag{2.10}$$

- Dividir cada elemento da matriz de coocorrência pela soma de todos os seus componentes da matriz original (PEDRINI; SCHWARTZ, 2008).

$$p_{m,n} = \frac{P(m,n)}{\sum_{i=0}^{N_g}\sum_{j=0}^{N_g} P(i,j)} \tag{2.11}$$

  onde *P(m,n)* é a (m,n)-ésima entrada na matriz de coocorrência, N_g representa a quantidade de níveis de cinza e *P(i,j)* é a (i,j)-ésima entrada na matriz de coocorrência.

Segundo Baraldi e Parmiggiani (apud PEDRINI; SCHWARTZ, 2008), 6 das 14 características estatísticas apresentadas por Haralick em (HARALICK et al., 1973), calculadas a partir de cada matriz de coocorrência, são mais relevantes para descrever as propriedades contidas nas texturas. Observando que $P_{i,j}$ representa a (i,j)-ésima entrada na matriz de coocorrência normalizada e N_g indica a dimensão da matriz de coocorrência (número de níveis de cinza), as 6 características ou descritores de Haralick são descritas a seguir:

- Homogeneidade: mede a proximidade espacial da distribuição de elementos da matriz de coocorrência.

$$f_{hom} = \sum_{i=0}^{N_g} \sum_{j=0}^{N_g} \frac{1}{1+(i-j)^2} P_{i,j} \tag{2.12}$$

- Contraste: caracteriza-se pela diferença entre os tons de cinza.

$$f_{con} = \sum_{i=0}^{N_g} \sum_{j=0}^{N_g} (i-j)^2 P_{i,j} \tag{2.13}$$

- Segundo Momento Angular (ASM): é conhecido também como energia. Mede a uniformidade da textura.

$$f_{sma} = \sum_{i=0}^{N_g} \sum_{j=0}^{N_g} {P_{i,j}}^2 \tag{2.14}$$

- Entropia: expressa a desordem contida na textura.

$$f_{ent} = -\sum_{i=0}^{N_g} \sum_{j=0}^{N_g} P_{i,j} log(P_{i,j}) \tag{2.15}$$

- Variância: mede a heterogeneidade da textura.

$$f_{var_i} = \sum_{i=0}^{N_g} \sum_{j=0}^{N_g} (i-\mu_i)^2 P_{i,j} \tag{2.16}$$

$$f_{var_j} = \sum_{i=0}^{N_g} \sum_{j=0}^{H_g} (j-\mu_j)^2 P_{i,j} \tag{2.17}$$

  μ_i e μ_j denotam o valor médio das distribuições marginais. Os cálculos são obtidos pelas equações 2.18 e 2.19.

$$\mu_i = \sum_{i=0}^{N_g} \sum_{j=0}^{N_g} i P_{i,j} \tag{2.18}$$

$$\mu_j = \sum_{i=0}^{N_g} \sum_{j=0}^{N_g} j P_{i,j} \tag{2.19}$$

- Correlação: mede a dependência linear entre os tons de cinza da imagem.

$$f_{corr} = \frac{1}{\sigma_x \sigma_y} \sum_{i=0}^{N_g} \sum_{j=0}^{N_g} (i - \mu_i)(j - \mu_j) P_{i,j} \tag{2.20}$$

  μ_i e μ_j, representadas pelas equações 2.18 e 2.19, indicam o valor médio das distribuições marginais; σ_i e σ_j indicam o desvio padrão, que é calculado pela raiz quadrada das variâncias (GONZALEZ; WOODS, 2010).

2.7 Considerações Finais

Neste Capítulo foram apresentados conceitos da metaprogramação e dos recursos de metaprogramação da linguagem de programação Java, como anotações, tipo genérico e reflexão. O *framework Obinject*, através da metaprogramação, processa suas anotações aplicadas nas classes de aplicação para que a aplicação do cliente possa ser persistida e indexada. Além disso, alguns conceitos da área de processamento e análise de imagens foram apresentados, pois serão alvos de implementações no *framework Obinject*.

De maneira geral, a fundamentação teórica deste Capítulo foi apresentada com o intuito de auxiliar na compreensão do encadeamento de anotações e da extração/indexação das imagens no *framework Obinject*.

Capítulo 3

Encadeamento de Anotações e Indexação de Imagens

O objetivo deste Capítulo é apresentar e explicar as realizações deste trabalho. Na seção 3.1 são descritas as modificações estruturais nas anotações do *framework Obinject*. E na Seção 3.2 é proposta uma anotação responsável pela extração e indexação das características das imagens destinada aos atributos do tipo `BufferedImage`; e descrita a implementação dos métodos e do código da classe empacotadora gerada a partir da anotação.

3.1 Cadeias de Anotações

O *framework Obinject* possui anotações que, aplicadas às classe de aplicação, permitem a geração automática das classes empacotadoras. Essa combinação define uma metodologia que promove o vínculo da classe de aplicação com o *framework Obinject* e que, segundo Oliveira (2012), conduz à uma abordagem conveniente e pouco intrusiva para a aplicação do cliente.

Nos trabalhos de Oliveira (2012) e Ferro (2012) foram definidas anotações que possibilitam a persistência e a indexação dos atributos dos objetos no *framework Obinject*, pois permitem a geração das classes empacotadoras. Para gerar uma classe empacotadora responsável pela persistência do objeto utiliza-se a anotação `@Persistent`, aplicável à tipos (classes).

As modificações realizadas no *framework Obinject* ocorreram no esquema das anotações dos domínios de indexação (*Unique*, *Sort*, *Origin/Extension*, *Point*, *Coordinate*, *Edition*, *Protein* e *Feature*) e possibilitam: múltiplos índices usando uma única anotação por domínio e a definição do posicionamento dos atributos na composição da chave de indexação.

A primeira modificação estrutural consiste na definição do índice que o atributo será indexado

em um dos domínios. Para realizar isso, é preciso que o usuário atribua valores ao elemento `number` da anotação. Por exemplo, para que dois atributos da classe de aplicação sejam indexados em índices distintos do domínio *Point*, o usuário precisa aplicar a anotação `@Point` sobre os atributos e atribuir ao elemento `number` da anotação o valor `Number.One` no atributo que será indexado no índice um e o valor `Number.Two` no atributo que será indexado no índice dois. Anteriormente, para indexar em vários índices era preciso utilizar uma das anotações do domínio que representavam o índice escolhido.

A segunda modificação estrutural é a definição do posicionamento dos atributos na composição da chave de indexação. Para isso, o usuário precisa atribuir um valor ao elemento `order` da anotação. Por exemplo, para criar um índice no domínio *Point* com dois atributos, o usuário deve: anotar os dois atributos com a anotação `@Point`, atribuir o mesmo valor para o elemento `number`; e atribuir ao elemento `order` dessa anotação o valor `Order.First` para o atributo da primeira posição e `Order.Second` para o atributo da segunda posição. Anteriormente, a posição dos atributos na chave de indexação seguia a ordem de como eram declarados na classe.

Além disso, cada um dos 8 domínios de indexação são definidos através das anotações de indexação que podem ser aplicadas mais de uma vez sobre o mesmo atributo da classe (repetíveis), graças ao recurso suportado a partir do Java SE versão 8 (GOSLING et al., 2015).

É importante observar que antes o *framework Obinject* possuía 101 anotações, sendo 1 anotação de persistência e as demais de indexação. A partir deste trabalho, o número de anotações do *framework* passou para 9, sendo 1 de persistência e 8 anotações de indexação. Utilizando apenas uma anotação de um domínio é possível definir os índices de um domínio através da atribuição de valores para o elemento `number` da anotação e a posição do atributo na chave de indexação através da atribuição de valores para o elemento `order` da anotação.

No Apêndice B mostra-se o novo diagrama UML (*Unified Modeling Language*) do pacote `generator` (responsável por encapsular todos o metadados necessários para geração das classes empacotadoras (FERRO, 2012)) e `annotation` (contém todos os tipos de anotações) do *framework Obinject*.

3.1.1 Exemplo de uso das anotações definidas neste trabalho

O código 3.1 exemplifica o uso das anotações definidas neste trabalho para o *framework Obinject* em uma classe que encapsula os atributos de exames médicos.

A anotação `@Persistent` (linha 1) do código 3.1 indica que a classe `Exame` será persistida.

Os atributos `numero` (linha 5), `paciente` (linha 8), `data` (linha 11) e `medico` (linha 15), quando anotados por `@Unique`, identificam unicamente os atributos a classe `Exame` e serão indexados na estrutura *BTree+*. O atributo `numero` será indexado no índice um do domínio

```
1  @Persistent
2  public class Exame {
3
4      @Unique(number = Number.One, order = Order.First)
5      private String numero;
6
7      @Unique(number = Number.Two, order = Order.Second)
8      private String paciente;
9
10     @Unique(number = Number.Two, order = Order.First)
11     private Date data = new Date();
12
13     @Unique(number = Number.Two, order = Order.Third)
14     @Edition(number = Number.One, order = Order.First)
15     private String medico;
16
17     @Sort(number = Number.One, order = Order.First)
18     private String clinica;
19
20     @Feature(number = Number.One, order = Order.First,
21         method = ExtractionMethod.HistogramStatistical)
22     @Feature(number = Number.Two, order = Order.First,
23         method = ExtractionMethod.HaralickSymmetric)
24     @Feature(number = Number.Two, order = Order.Second,
25         method = ExtractionMethod.HistogramStatistical)
26     @Feature(number = Number.Two, order = Order.Third,
27         method = ExtractionMethod.Histogram)
28     private BufferedImage imagem;
29
30     @Point(number = Number.One, order = Order.First)
31     @Coordinate(number = Number.One, order = Order.First)
32     @Origin(number = Number.One, order = Order.First)
33     private double latitudeImagem;
34
35     @Point(number = Number.One, order = Order.Second)
36     @Coordinate(number = Number.One, order = Order.Second)
37     @Origin(number = Number.One, order = Order.Second)
38     private double longitudeImagem;
39
40     //gets e sets
41 }
```

Código 3.1: Exemplos de uso das anotações atuais do *framework Obinject*

de indexação `Unique` por causa do valor `Number.One` atribuído ao elemento `number` da anotação; e os atributos `data`, `paciente` e `medico` serão indexados no índice dois do domínio de indexação `Unique` em virtude do valor `Number.Two` atribuído ao elemento `number`; na primeira, segunda e terceira posição das chaves de indexação, respectivamente, em razão dos valores atribuídos ao elemento `order` da anotação.

O atributo `medico` (linha 15), anotado também por `@Edition`, será indexado no índice um do domínio de indexação `Edition` e na primeira posição da chave de indexação.

O atributo `clinica` (linha 18), anotado por `@Sort`, será indexado no índice um do domínio de indexação `Sort` e na primeira posição da chave de indexação.

O atributo `imagem` (linha 28) é anotado por `@Feature`, anotação definida neste trabalho. Os dados estatísticos do histograma da imagem serão extraídos em razão do valor atribuído ao elemento `method` da anotação e indexados no índice de número um do domínio de indexação `Feature` na primeira posição da chave de indexação. E os 6 descritores mais relevantes de Haralick, obtidos através das matrizes de coocorrência simétricas da imagem do atributo `imagem`; os dados estatísticos do histograma da imagem e o histograma da imagem serão extraídos em virtude dos valores atribuídos ao elemento `method` da anotação e indexados no índice de número dois do domínio de indexação `Feature` na primeira, segunda e terceira posição, respectivamente, da chave de indexação.

Por último, estão os atributos `latitudeImagem` (linha 33) e `longitudeImagem` (linha 38) anotados por `@Point`, `@Coordinate` e `@Origin`. Eles serão indexados nos índices de número um dos domínios de indexação `Point`, `Coordinate` e `Origin/Extension`. O atributo `latitudeImagem` estará na primeira posição e o atributo `longitudeImagem` estará na segunda posição das chaves de indexação.

Vale ressaltar que para uma classe de aplicação possa ser manipulável pelo *framework*, é opcional conter índices; no entanto, ela deve ser persistente (`@Persistent`) e possuir uma chave de identificação (`@Unique`).

A figura 3.1 apresenta o diagrama UML das classes geradas pelo *framework Obinject* a partir das anotações aplicadas à classe `Exame` (código 3.1).

O diagrama UML, da figura 3.1, ilustra que a partir do atributo anotado o gerador de classes gera classes empacotadoras. Se a anotação é de persistência, a classe empacotadora `$Exame` implementa a interface `Entity` e especializa a classe de aplicação `Exame`. Por outro lado, se a anotação é de indexação, as classes empacotadoras implementam as subinterfaces de `Key` e especializam a classe empacotadora responsável pela indexação `$Exame`.

Pode-se observar na figura 3.1 que ao aplicar a anotação `Persistent` na declaração da classe `Exame` (código 3.1), uma classe empacotadora responsável pela persistência (`$Exame`) será gerada.

Além disso, o atributo `numero` da classe `Exame`, quando anotado por `@Unique` com `number=Number.One`, indicará a criação da classe empacotadora `UniqueOneExame`. Já os atributos `paciente`, `data` e `medico`, quando anotados por `@Unique` com `number=Number.Two`, indicará a criação da classe empacotadora `UniqueTwoExame`. Já o atributo `clinica`, quando anotado por `@Sort`, indicará a geração da classe empacotadora `SortOneExame`. Estas classes empacotadoras citadas implementam a interface `Sort` (filha da interface `Key`) e especializam a classe `$Exame`. Estes atributos serão indexados utilizando a *BTree+*.

Os atributos da classe `Exame`, quando anotados por `@Edition`, `@Point`, `@Coordinate` e `@Origin`, indicarão a geração das classes empacotadoras (`EditionOneExame`, `PointOneExame`, `CoordinateOneExame` e `RectangleOneExame`, respectivamente) que implementam a interface

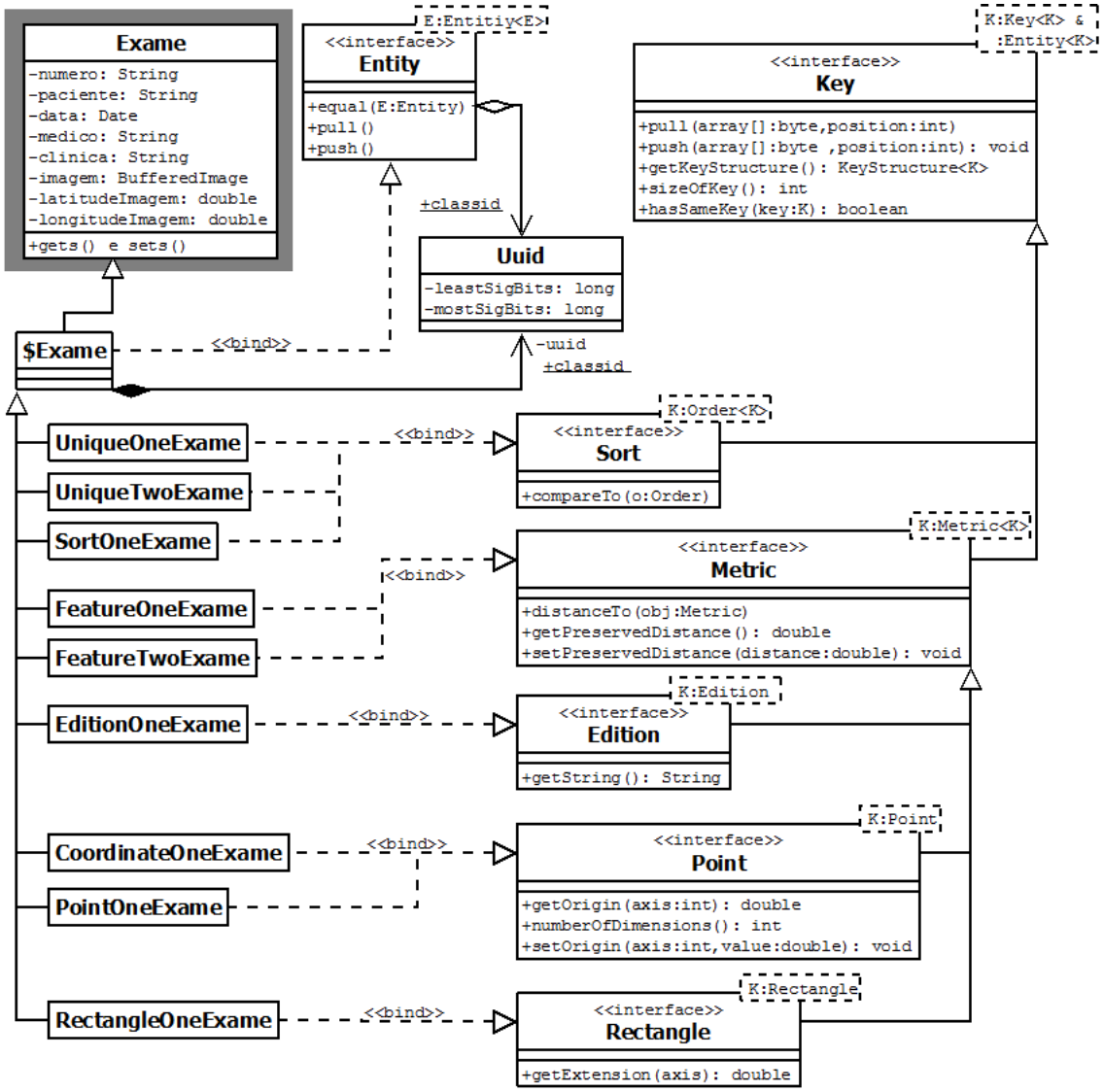


Figura 3.1: Classes empacotadoras geradas pelo *framework Obinject* a partir da classe de aplicação Exame.

`Metric` (filha da interface `Key`) e especializam a classe `$Exame`. Os atributos `latitudeImagem` e `longitudeImagem`, quando anotados por `@Origin`, serão indexados em uma *RTree*. Estes atributos, quando anotados por `@Point` e `@Coordinate`; e o atributo `medico`, quando anotado por `@Edition`, serão indexados em uma *MTree*.

E a partir das anotações `@Feature` aplicadas sobre o atributo `imagem`, duas classes empacotadoras `FeatureOneExame` e `FeatureTwoExame`, que implementam a interface `Metric` e especializam a classe `$Exame`, serão geradas devido ao valor atribuído ao elemento `number`. Já as características das imagens serão extraídas e indexadas em uma *MTree*.

3.2 Anotação para extração e indexação de imagens

A nova anotação `@Feature`, definida neste trabalho, possibilita que as características das imagens da classe de aplicação sejam extraídas e indexadas. Mas para isso, o atributo da classe de aplicação deve ser do tipo `BufferedImage`, como pode-se observar no código 3.1. A classe `BufferedImage` é uma subclasse de `Image` que descreve uma imagem usando um bloco de memória e está na API Java.

Para definir o método de extração de características da imagem, é preciso atribuir ao elemento `method` da anotação `@Feature` os valores presentes no enumerado `ExtractionMethod` (pacote `org.obinject.annotation`) que são:

- `Histogram`: cria um vetor de características da distribuição dos níveis de cinza de uma imagem.
- `HistogramStatistical`: cria um vetor de características com os dados estatísticos (média, variância, assimetria, energia e entropia) do histograma de uma imagem.
- `HaralickSymmetric`: cria um vetor de características com os 6 descritores mais relevantes de Haralick (homogeneidade, contraste, segundo momento angular, entropia, variância e correlação) usando as matrizes de coocorrência simétricas da imagem.
- `HaralickAssymmetric`: cria um vetor de características com os 6 descritores mais relevantes de Haralick (homogeneidade, contraste, segundo momento angular, entropia, variância e correlação) usando as matrizes de coocorrência assimétricas da imagem.

3.2.1 Métodos para extração de características das imagens

Responsáveis pela extração das características das imagens, os métodos histograma, dados estatísticos sobre histograma e 6 descritores mais relevantes de Haralick usando as matrizes de coocorrência simétricas e assimétricas estão na classe `ExtractionUtil` (pacote `org.obinject.meta.generator`).

É importante ressaltar que, antes de serem extraídas as características, as imagens são convertidas em tons de cinza através do método `TYPE_BYTE_GRAY` da classe `BufferedImage`.

O método que calcula o histograma da imagem utiliza um vetor para armazenar a quantidade de tonalidades de cinza presentes na imagem. Para não ter um vetor com tamanho de 256 posições (níveis de cinza) e assim reduzir o custo computacional, as tonalidades próximas foram unidas no mesmo índice do vetor. Assim, as 16 primeiras tonalidades de cinza foram adicionadas no primeiro índice do vetor, as outras 16 no segundo índice do vetor e assim sucessivamente.

Já o histograma, que é obtido para realizar o cálculo dos dados estatísticos, não é reduzido, ou seja, ele possui um tamanho de 256 posições. Além disso, foi utilizado um mesmo *loop* para calcular a variância, assimetria, energia e entropia.

Como na literatura encontram-se as matrizes de coocorrência simétricas e assimétricas, foram implementados dois métodos para obter os 6 descritores mais relevantes de Haralick. Com o objetivo de reduzir o tempo de execução para o cálculo das características de textura de Haralick, foram realizadas algumas otimizações.

Na implementação das matrizes de coocorrência simétricas, é verificado se a matriz de coocorrência para um dado ângulo (0°, 45°, 90° e 135°) está vazia e, se estiver, é feito o cálculo da frequência de relacionamento entre os *pixels* vizinhos *n* com *n+1* e *n+1* com *n*. Dessa forma, não é preciso percorrer toda a matriz de coocorrência para armazenar o resultado da frequência, uma vez que é simétrica.

E na implementação das matrizes de coocorrência assimétricas, durante o cálculo da frequência de relacionamento dos *pixels* vizinhos *n* e *n+1*, é feita a soma das frequências para que sejam utilizadas na normalização das matrizes de coocorrência assimétricas.

Além disso, nesses 2 métodos, alguns cálculos dos 6 descritores mais relevantes de Haralick foram combinados dentro de um *loop* para que reduzisse o número de acessos à memória. E os valores dos descritores, armazenados no vetor de características, foram obtidos pela média dos valores dos descritores das matrizes de coocorrência para cada ângulo (0°, 45°, 90° e 135°).

3.2.2 Indexação de Imagens

O código da classe empacotadora, gerada a partir da anotação `@Feature`, inclui a declaração do(s) atributo(s) vetor(es) que armazena(m) as características que é/são definida(s) pelo elemento `method` da anotação.

Para exemplificar, é utilizado o código da classe empacotadora `FeatureTwoExame` (figura 3.1) gerada a partir da anotação `@Feature` aplicada à classe de aplicação `Exame` (código 3.1).

O código 3.2 apresenta os vetores de características declarados na classe empacotadora `FeatureTwoExame`.

O uso da anotação `@Feature` sobre o atributo `imagem` na classe de aplicação `Exame` (código 3.1) indicará a criação de uma classe empacotadora, chamada `FeatureTwoExame` (figura 3.1), por exemplo, que implementará a interface `Metric` (filha da interface `Key`) e especializará a classe empacotadora responsável pela persistência `$Exame`. Neste exemplo, como serão indexadas três características no mesmo índice do domínio de indexação *Feature*, três vetores do tipo `double` serão declarados: `featureVectorFirst`, `featureVectorSecond` e

```
1  public class FeatureTwoExame extends $Exame implements Metric<FeatureTwoExame>
       {
2      private double featureVectorFirst[] =
3              new double[ExtractionUtil.LENGTH_HARALICK_SYMMETRIC];
4      private double featureVectorSecond[] =
5              new double[ExtractionUtil.LENGTH_HISTOGRAM_STATISTICAL];
6      private double featureVectorThird[] =
7              new double[ExtractionUtil.LENGTH_HISTOGRAM];
8
9      //código
10 }
```

Código 3.2: *FeatureTwoExame*: os vetores de características

`featureVectorThird`. Cada vetor armazenará a característica extraída da imagem correspondente ao valor atribuído no elemento `method` da anotação `@Feature`.

A classe empacotadora `FeatureTwoExame` (figura 3.1) também sobrescreve os métodos da classe `$Exame`, filha da classe `Exame`, e das interfaces `Key` e `Metric`. A codificação de alguns desses métodos, da classe `FeatureTwoExame` (figura 3.1), será analisada a seguir.

O código 3.3 apresenta a implementação do método `setImagem` que muda a imagem.

```
1 @Override
2 public void setImagem(BufferedImage image) {
3     super.setImagem(image);
4     featureVectorFirst = ExtractionUtil.haralickSymmetric(image);
5     featureVectorSecond = ExtractionUtil.histogramStatistical(image);
6     featureVectorThird = ExtractionUtil.histogram(image);
7 }
```

Código 3.3: *FeatureTwoExame*: métodos *setImagem*

O método do código 3.3 sobrescreve o método `setImagem` da classe `Exame`. No método `setImagem`, os 6 descritores mais relevantes de Haralick obtidos a partir das matrizes de coocorrência simétricas da imagem do atributo `imagem` serão extraídos e armazenados no vetor `featureVectorFirst`; os dados estatísticos usando o histograma da imagem do atributo `imagem` será calculado e armazenado no vetor `featureVectorSecond` e o histograma da imagem do atributo `imagem` será obtido e armazenado no vetor `featureVectorThird`.

O código 3.4 apresenta a implementação do método `distanceTo` usado para estabelecer a distância entre dois objetos na indexação da *MTree*.

O método do código 3.4 sobrescreve o método `distanceTo` da interface `Metric`. A distância euclidiana entre dois objetos com vetor de características `featureVectorFirst`, a distância euclidiana entre dois objetos com vetor de características `featureVectorSecond` e a distância euclidiana entre dois objetos com vetor de características `featureVectorThird` são calculadas e, em seguida, o método retorna a soma de todas as distâncias.

O código 3.5 apresenta a implementação do método `pushKey` usado na serialização da chave

```
1 @Override
2 public double distanceTo(FeatureTwoExame obj) {
3     return DistanceUtil.euclidean(this.featureVectorFirst, obj.
          featureVectorFirst)
4         + DistanceUtil.euclidean(this.featureVectorSecond, obj.
              featureVectorSecond)
5         + DistanceUtil.euclidean(this.featureVectorThird, obj.
              featureVectorThird);
6 }
```

Código 3.4: *FeatureTwoExame*: o método *distanceTo*

de indexação da *MTree*.

```
1 @Override
2 public void pushKey(byte[] array, int position) {
3     PushPage push = new PushPage(array, position);
4     push.pushMatrix(featureVectorFirst);
5     push.pushMatrix(featureVectorSecond);
6     push.pushMatrix(featureVectorThird);
7 }
```

Código 3.5: *FeatureTwoExame*: o método *pushKey*

O método do código 3.5 sobrescreve o método `pushKey` da interface `Key`. Ele é responsável por armazenar os vetores de características `featureVectorFirst`, `featureVectorSecond` e `featureVectorThird` em um vetor de *bytes*.

E o código 3.6 apresenta a implementação do método `pullKey` usado para recuperar a chave da serialização na indexação da *MTree*.

```
1 @Override
2 public boolean pullKey(byte[] array, int position) {
3     PullPage pull = new PullPage(array, position);
4     featureVectorFirst = (double[]) pull.pullMatrix();
5     featureVectorSecond = (double[]) pull.pullMatrix();
6     featureVectorThird = (double[]) pull.pullMatrix();
7 }
```

Código 3.6: *FeatureTwoExame*: o método *pullKey*

O método do código 3.6 sobrescreve o método `pullKey` da interface `Key`. Ele é responsável por recuperar os vetores de características `featureVectorFirst`, `featureVectorSecond` e `featureVectorThird` do vetor de *bytes*.

3.3 Considerações Finais

O *framework Obinject* possui 8 domínios de indexação que são definidos através anotações: `@Unique`, `@Sort`, `@Coordinate`, `@Point`, `@Origin/@Extension`, `@Edition`, `@Protein` e

`@Feature`). Estas anotações permitem a definição de múltiplos índices através do elemento `number` e do posicionamento dos atributos na composição da chave de indexação através do elemento `order`; e podem ser aplicadas mais de uma vez sobre o mesmo atributo (repetíveis).

É importante observar que a anotação `@Feature`, no *framework*, possibilita extrair e indexar características das imagens, como histograma, dados estatísticos a partir de histogramas e 6 descritores mais relevantes de Haralick usando as matrizes de coocorrência simétricas e assimétricas; através do elemento `method` da anotação.

A realização deste trabalho permitiu uma redução do número de anotações de indexação (de 100 para 8 anotações) e possibilitou um acréscimo do número de funcionalidades que essas anotações permitem. Além disso, para que o *framework Obinject* suporte mais valores de índices e posições, as únicas modificações que precisam ser feitas são as adições de valores nas declarações dos enumerados `KeyNumber` e `FieldOrder` do pacote `org.obinject.annotation`.

Capítulo 4

Experimentos e Resultados

Neste Capítulo são apresentados dois experimentos realizados com o *framework Obinject*.

O primeiro deles tem o objetivo de analisar a usabilidade do *framework* antes e após as modificações estruturais nas anotações do *framework* realizadas por este trabalho (descritas no Capítulo 3). A Seção 4.1 descreve as configurações, processos utilizados neste experimento e apresenta os gráficos produzidos a partir dos resultados das observações.

O segundo experimento visa avaliar o desempenho do *framework Obinject* ao utilizar sua anotação `@Feature` (também descrita no Capítulo 3) para extração e indexação de características das imagens. A Seção 4.2 apresenta os recursos usados neste experimento, sua implementação, a tabela produzida a partir dos resultados, juntamente com suas observações.

4.1 Experimento 1: Utilização das anotações do *framework Obinject* para persistir e indexar os atributos

O experimento 1 foi realizado em 2 dias e contou com a participação de 25 alunos da Engenharia da Computação da Universidade Federal de Itajubá, sendo que 13 alunos utilizaram o *framework Obinject* antes das modificações realizadas por este trabalho e 12 alunos utilizaram o *framework Obinject* com as modificações realizadas por este trabalho. Esses alunos possuem conhecimento na linguagem de programação Java e nenhum conhecimento no *framework Obinject*.

Para garantir a realização dos testes de comparação entre o *framework Obinject* antes e após as contribuições deste trabalho, é preciso que eles sejam feitos em condições semelhantes;

por isso, eles foram realizados no Laboratório da Engenharia da Computação I (LEC I) que possui 28 máquinas com o mesmo *hardware* e *software*.

As configurações de *hardware* dos computadores do LEC I são: processador Intel *core* i5-4590 3.30 GHz com 4 núcleos, 4 *threads* e 6 MB de *cache*; memória DDR 3-1600 com 4 GB e HD com 500 GB 7200 RPM SATA III.

As configurações de *software* dos computadores do LEC I, utilizados no experimento, são: *Windows* 7, Netbeans 8.1, JDK (*Java Platform, Standard Edition Development Kit*) 1.8.0.

O experimento 1 é dividido em 2 partes, sendo que uma parte utilizou o *framework Obinject* antes das alterações realizadas por este trabalho e a outra parte utilizou o *framework Obinject* com as alterações realizadas por este trabalho. Cada parte possui 3 fases: treinamento, execução e avaliação.

Na fase de treinamento foram apresentadas as anotações e suas definições, o código de uma classe exemplificando o uso das anotações, uma breve explicação das funções das anotações no código exemplo e as classes, já codificadas, responsáveis pela geração das classes empacotadoras e da persistência do objeto da classe exemplo.

Na fase de avaliação foi dado um diagrama UML para que os participantes codificassem as classes, aplicassem nelas as anotações definidas nos enunciados e executassem as classes para gerar as classes empacotadoras e realizar a persistência.

E, finalmente, na fase de avaliação, os participantes responderam um questionário para medir o nível de usabilidade do *framework Obinject*.

Para facilitar a compreensão, o *framework Obinject* antes das alterações realizadas por este trabalho foi definido como *framework Obinject* versão 1 e, com as alterações, *framework Obinject* versão 2.

Neste experimento foram criados dois roteiros para a realização das tarefas: o primeiro roteiro está relacionado com o *framework Obinject* versão 1 e encontra-se no Apêndice C; e o segundo roteiro está relacionado com o *framework Obinject* versão 2 e está no Apêndice D. Cada participante, antes de iniciar o experimento, recebeu um roteiro com as tarefas de cada fase.

4.1.1 Fase de Treinamento do Experimento 1

O objetivo da fase de treinamento é apresentar aos participantes as definições e o uso das anotações do *framework Obinject*, já que eles não possuem conhecimento sobre o *framework*.

Nesta fase, as anotações do *framework Obinject* são definidas, o código da classe `Exame` com as anotações é mostrado e explicado no roteiro. A figura 4.1 mostra o diagrama UML da classe `Exame`.

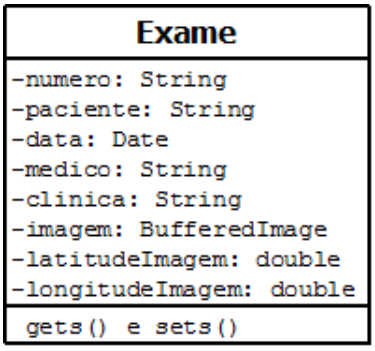


Figura 4.1: Diagrama UML da classe Exame.

Nesta fase é solicitado, através do roteiro, que o participante abra o pacote modelo usando a IDE Netbeans. No pacote já estão codificadas as classes `Exame` (classe exemplo), `AppGerarClasses` (responsável por gerar as classes empacotadoras) e `AppPersistir` (responsável por persistir o objeto). Em seguida, é pedido para que o participante execute estas duas últimas classes para que as classes empacotadoras, baseadas na anotações utilizadas na classe `Exame`, sejam criadas e o objeto da classe `Exame` seja persistido.

4.1.2 Fase de Execução do Experimento 1

O objetivo desta fase é a utilização das anotações do *framework Obinject* apresentadas na fase de treinamento.

Na fase de execução é dado um diagrama UML (figura 4.2) para que o participante codifique e em seguida aplique as anotações.

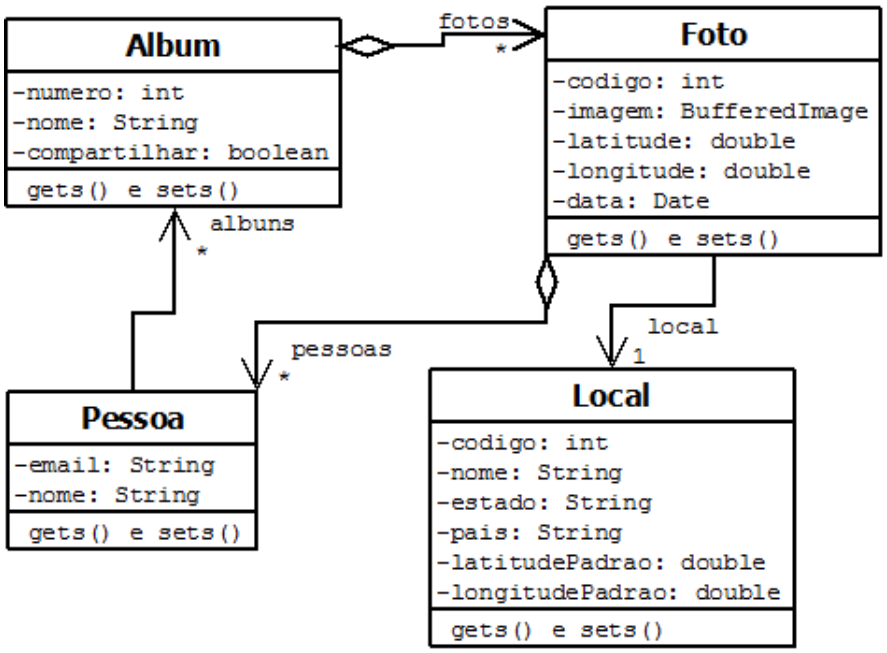


Figura 4.2: Diagrama UML das classes Album, Foto, Local e Pessoa

Após a codificação do diagrama UML da figura 4.2, é solicitado ao participante que execute as classes `AppGerarClasses` para gerar as classes empacotadoras e `AppPersistir` para realizar a persistência dos objetos das classes `Album`, `Foto`, `Local` e `Pessoa`.

4.1.3 Fase de Avaliação do Experimento 1

A usabilidade, conforme (ROCHA; BARANAUSKAS, 2003), é um conceito chave em Interface Homem Computador e refere-se a quão bem os usuários podem usar a funcionalidade definida.

Para medir o nível de usabilidade na versão 1 e 2 do *framework Obinject*, foi utilizado o questionário SUS (*System Usability Scale*) de Brooke et al. (1996), que tem sido usado a mais de 25 anos para mensurar percepções de usabilidade (SAURO, 2011).

Embora existam várias alternativas para avaliar a usabilidade, o SUS apresenta as seguintes características, segundo Bangor et al. (2008):

- é independente da tecnologia;
- é rápido e fácil de usar tanto por participantes do estudo como administradores;
- fornece uma pontuação única que é facilmente compreendida por várias pessoas;
- ele não é proprietário, tornando assim uma ferramenta que pode ser utilizada sem maiores custos.

O questionário SUS é composto de 10 questões, que alternam entre o positivo e o negativo, onde o participante seleciona uma das opções na escala de 1 a 5; sendo 1 “Discordo Totalmente” e 5 “Concordo Totalmente”. Para calcular o nível de usabilidade, a partir do SUS, são utilizados os critérios a seguir:

- O valor final das questões ímpares será a resposta do participante menos 1.
- O valor final das questões pares será 5 menos a resposta do participante.
- Todas as questões estarão em uma escala de 0 a 4.
- Some todas as questões e multiplique por 2,5.

O resultado final varia de 0 a 100 e indicará a pontuação do SUS que representa uma medida da usabilidade global do sistema que está sendo estudado. Pontuações mais altas indicam uma melhor usabilidade (BANGOR et al., 2008).

Além do questionário SUS, foram adicionadas 4 perguntas na fase de avaliação: uma questão para identificar o tempo de conhecimento do participante na linguagem de programação

Java, duas questões para verificar se o participante realizou as tarefas de maneira correta e uma última questão aberta permite que o participante dê sugestões.

4.1.4 Resultados do Experimento 1

Nesta seção são apresentados os resultados obtidos com o experimento 1, responsável por avaliar a usabilidade do *framework Obinject* na versão 1 e 2.

Os dados obtidos com o experimento 1 podem ser consultados no Apêndice E.

Em cada fase (treinamento, execução e avaliação) é calculado o tempo mínimo, máximo e a média. Além disso, é obtida a média do tempo dos participantes na realização de todas as fases.

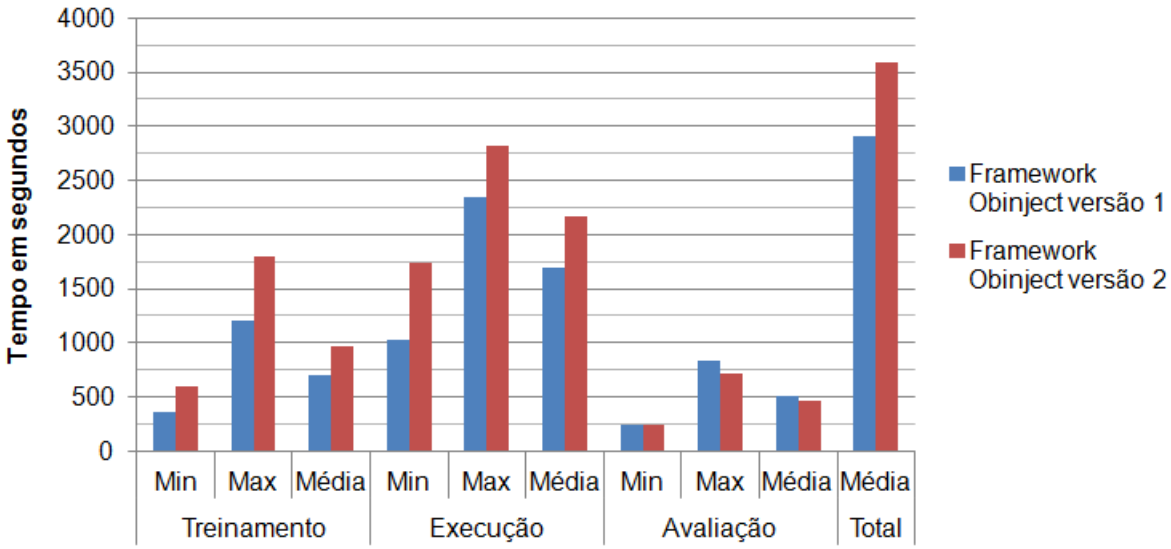


Figura 4.3: Mínimo, máximo e média do tempo de cada fase e a média do tempo total usando a versão 1 e 2 do *framework Obinject*.

Pelo gráfico da figura 4.3, pode-se observar que o tempo nas fases de treinamento e execução utilizando a versão 1 do *framework Obinject* é menor se comparado com a versão 2 do *framework*. Na média da fase de treinamento, a versão 2 do *framework Obinject* apresentou um aumento de 38,176% no tempo em comparação com a versão 1. Já na média da fase de execução, a versão 2 do *framework* apresentou um aumento de 27,208% no tempo em comparação com a versão 1. Na fase de avaliação, observando a média, os participantes que utilizaram a versão 2 do *framework Obinject* gastaram menos tempo do que os participantes que utilizaram a versão 1 do *framework*, ou seja, uma redução de 8,548% em comparação com o tempo da fase de avaliação da versão 1. Finalmente, na média do tempo total, os participantes que utilizaram a versão 2 do *framework Obinject* gastaram 23,665% mais tempo para realizar as tarefas de todas as fases do que os participantes que utilizaram a versão 1.

Esse aumento na média do tempo para que os participantes realizassem as tarefas de todas as fases é justificável uma vez que a versão 2 do *framework* apresenta mais funcionalidades e flexibilidade do que a versão 1, como extrair as características das imagens e indexá-las, tornar as anotações repetíveis; e definir o índice e o posicionamento dos atributos na chave de indexação utilizando apenas uma anotação por domínio. Embora seja mais trabalhoso anotar os atributos porque as anotações possuem elementos; é importante ressaltar que houve uma redução do número de anotações da versão 1 para a versão 2 do *framework Obinject*: de 101 para 9 anotações.

As figuras 4.4(a) e (b) mostram a porcentagem de acertos e erros das anotações aplicadas na fase de execução do experimento 1.

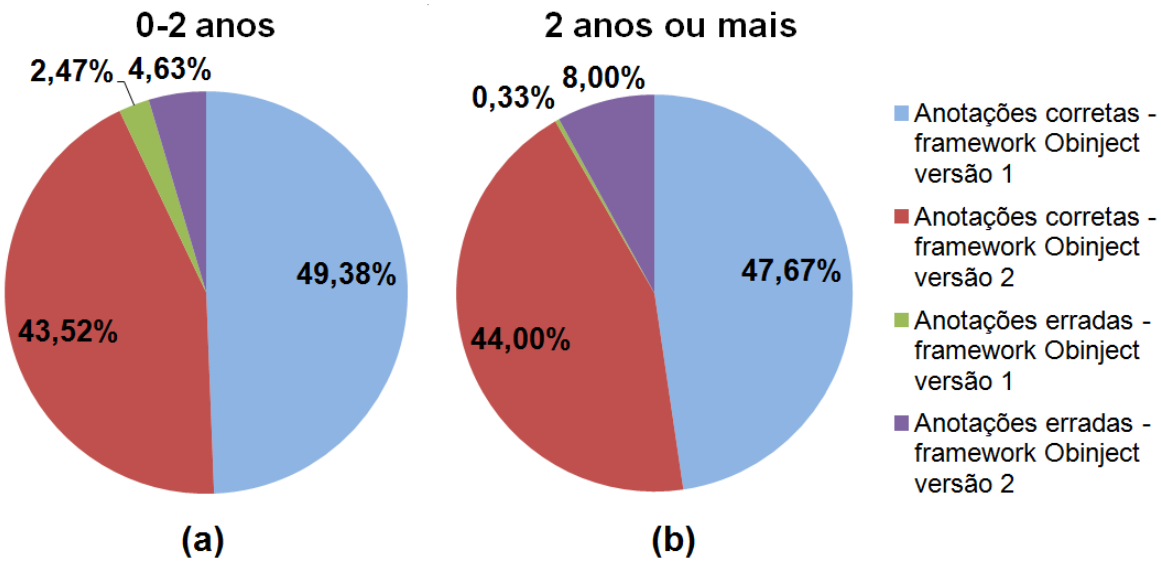


Figura 4.4: Avaliação das anotações aplicadas na fase de execução utilizando a versão 1 e 2 do *framework Obinject*, sendo que o gráfico (a) representa as anotações do grupo com 0 a 2 anos de conhecimento em Java e (b) as anotações do grupo com 2 anos ou mais de conhecimento em Java.

A figura 4.4(a) mostra a porcentagem de acertos e erros das anotações para o grupo de participantes com 0 a 2 anos de conhecimento na linguagem de programação Java. Os participantes deste grupo deveriam aplicar 168 anotações na versão 1 e 156 anotações na versão 2. No total foram aplicadas 324 anotações, sendo que na utilização da versão 1 do *framework Obinject*, os participantes acertaram 160 anotações (49,38%) e erraram 8 (2,47%). Os participantes que utilizaram a versão 2 do *framework Obinject* acertaram 141 anotações (43,52%) e erraram 15 (4,63%).

Já a figura 4.4(b) mostra a porcentagem de acertos e erros das anotações para o grupo de participantes com 2 anos ou mais de conhecimento na linguagem de programação Java. Os participantes deste grupo deveriam aplicar 144 anotações na versão 1 e 156 anotações na versão 2. No total foram aplicadas 300 anotações, sendo que na utilização da versão 1 do *framework Obinject*, os participantes acertaram 143 anotações (47,67%) e erraram 1 (0,33%).

Os participantes que utilizaram a versão 2 do *framework Obinject* acertaram 132 anotações (44%) e erraram 24 (8%).

As figuras 4.5(a) e (b) mostram a porcentagem de acertos e erros das classes empacotadoras geradas a partir das anotações aplicadas na fase de execução do experimento 1.

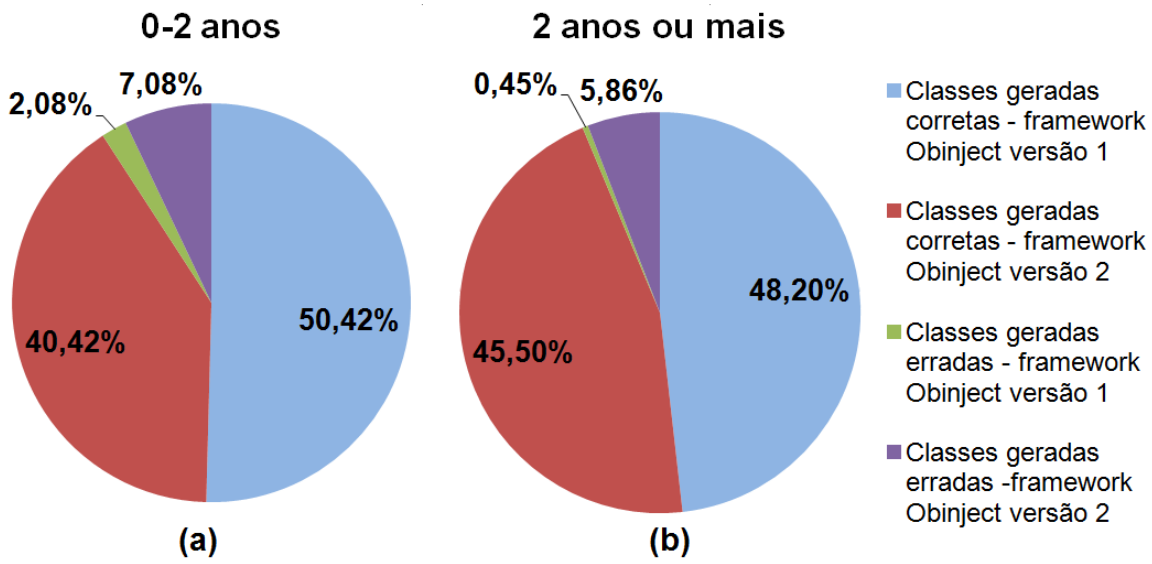


Figura 4.5: Avaliação das classes empacotadoras geradas na fase de execução utilizando a versão 1 e 2 do *framework Obinject*, sendo que o gráfico (a) representa as classes geradas pelo grupo com 0 a 2 anos de conhecimento em Java e (b) as classes geradas pelo grupo com 2 anos ou mais de conhecimento em Java.

A figura 4.5(a) mostra a porcentagem de acertos e erros das classes geradas para o grupo de participantes com 0 a 2 anos de conhecimento da linguagem de programação Java. Os participantes deste grupo deveriam gerar 126 classes empacotadoras na versão 1 e 114 classes empacotadoras na versão 2. No total foram geradas 240 classes, sendo que na utilização da versão 1 do *framework Obinject*, os participantes geraram 121 classes corretas (50,42%) e erraram 5 (2,08%). Os participantes que utilizaram a versão 2 do *framework Obinject* geraram 97 classes corretas (40,42%) e erraram 17 (7,08%).

Já a figura 4.5(b) mostra a porcentagem de acertos e erros das classes geradas para o grupo de participantes com 2 anos ou mais de conhecimento na linguagem de programação Java. Os participantes deste grupo deveriam gerar 108 classes empacotadoras na versão 1 e 114 classes empacotadoras na versão 2. No total foram geradas 222 classes, sendo que na utilização da versão 1 do *framework Obinject*, os participantes geraram 107 classes corretas (48,20%) e erraram 1 (0,45%). Os participantes que utilizaram a versão 2 do *framework Obinject* geraram 101 classes corretas (45,50%) e erraram 13 (5,86%).

Pode-se analisar, pelos gráficos das figuras 4.4 e 4.5, que os participantes que utilizaram a versão 1 do *framework Obinject* acertaram mais que os participantes que utilizaram a versão 2. Isto porque na versão 1, para o usuário indexar os atributos em mais de um índice

de um domínio, era preciso aplicar sobre os atributos uma das anotações do domínio que representavam o índice escolhido. Já na versão 2 do *framework*, embora exista uma única anotação para cada um dos domínios, elas possuem elementos que definem o número do índice de um domínio que o atributo será indexado e a sua posição na chave de indexação; tornando assim o processo de anotar os atributos da classe mais suscetível a erros.

A figura 4.6 mostra o nível de usabilidade das versões 1 e 2 do *framework Obinject* obtido pelo questionário SUS aplicado aos participantes na fase de avaliação.

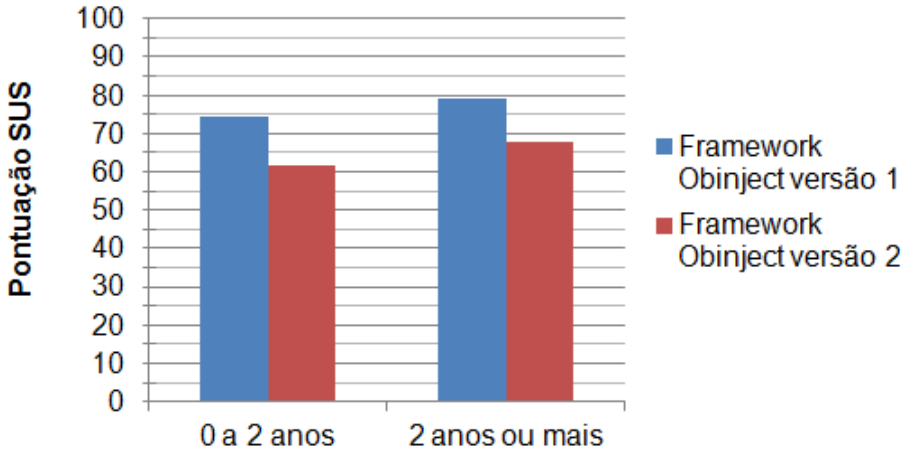


Figura 4.6: Teste do nível de usabilidade da versão 1 e 2 do *framework Obinject*.

No gráfico da figura 4.6, observa-se que houve uma diminuição no valor da pontuação SUS da versão 1 do *framework Obinject* para a versão 2. Ou seja, uma redução de 17,384% para o grupo de participantes com 0 a 2 anos de conhecimento em Java e 14,210% para o grupo com 2 anos ou mais de conhecimento em Java. Esta diminuição na pontuação SUS se deve ao fato das anotações na versão 2 do *framework Obinject* possuírem elementos que definem funcionalidades de indexação, como o número do índice de um domínio que o atributo será indexado e a sua posição na chave de indexação; tornando o processo de anotação mais trabalhoso para o usuário do que na versão 1 do *framework*.

Um estudo feito por Sauro (2011) onde 5.000 usuários avaliaram 500 programas diversos com questionário SUS, a pontuação média foi de 68 pontos. Dessa maneira, utilizando esta pontuação como base, as pontuações SUS da versão 2 do *framework Obinject* para os grupos com conhecimento em Java de 0 a 2 anos e de 2 anos ou mais são aproximadamente 9,313% e 0,122% menores. Apesar dos valores estarem abaixo da média, o valor da pontuação SUS para os participantes com maior tempo de conhecimento na linguagem de programação Java chegou próximo à pontuação média obtida no estudo realizado por Sauro (2011).

4.2 Experimento 2: Utilização das anotações @Persistent, @Unique e @Feature do *framework Obinject*

No experimento 2 são realizadas as medições das alturas das estruturas *BTree+* e *MTree*, tempo médio para inserções, número de acessos ao disco e verificações (comparação e distância) para persistir (`@Persistent`), identificar o objeto da classe (`@Unique`), extrair as características das imagens e indexá-las (`@Feature`). Essas características extraídas das imagens são dados estatísticos usando histograma (`HistogramStatistical`), os 6 descritores mais relevantes de Haralick usando as matrizes de coocorrência simétricas (`HaralickSymmetric`) e assimétricas (`HaralickAssymmetric`). Além disso, elas são definidas no elemento `method` da anotação `@Feature` quando aplicada sobre um atributo da classe.

Para realizar este experimento foi utilizado um computador com processador Intel *core* i5-459 3.30 GHz com 4 núcleos, 4 *threads* e 6 MB de *cache*; memória DDR 3-1600 com 4 GB, HD com 500 GB 7200 RPM SATA III, sistema operacional Linux Ubuntu 14.04, IDE Netbeans 8.1, JDK 1.8.0. O banco de imagens de faces usado foi LFW (*Labeled Faces in the Wild*) (HUANG et al., 2007). Este banco de imagens possui 13.233 imagens de faces coletadas da *web* de 5.749 pessoas, sendo que 1.680 pessoas têm duas ou mais imagens. Além disso, cada imagem de face foi rotulada com o nome da pessoa representada.

Foi implementada a classe `Face`, apresentada na figura 4.7, e aplicada as anotações `@Persistent` na declaração da classe, `@Unique` sobre o atributo `label` e `@Feature` sobre o atributo `face` (do tipo `BufferedImage`). A anotação `@Feature` é aplicada mais de um vez sobre o mesmo atributo com diferentes índices e métodos para extração de características das imagens. Além disso, é criada uma classe principal, que, ao ser executada, percorre todos os diretórios do banco de faces e atribui uma imagem para o atributo da classe `Face`. Em cada imagem atribuída, as características da imagem são extraídas de acordo com o valor atribuído ao elemento `method` da anotação `@Feature`, indexadas e o objeto da classe `Face` é persistido.

A seguir é apresentado, na figura 4.7, o diagrama UML da classe Face:

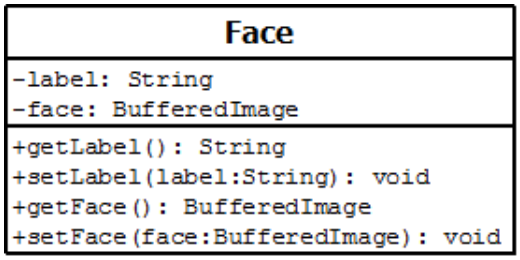


Figura 4.7: Diagrama UML da classe Face

Esse experimento gastou aproximadamente 79.192 segundos.

4.2.1 Código-fonte da classe AppInsere

O código 4.1 apresenta o método `main` da classe `AppInsere` (classe principal).

```
1  public static void main(String[] args) {
2      Face face = new Face();
3      List<File> files = new LinkedList<>();
4      PersistentManager pm = new PersistentManager();
5      addFileInDirectories(new File("/dataset/lfw/"), files);
6      for (File f : files) {
7          try {
8              face.setFace(ImageIO.read(f));
9              face.setLabel(f.toString());
10             pm.getTransaction().begin();
11             pm.persist(face);
12             pm.getTransaction().commit();
13         } catch (IOException ex) {
14             Logger.getLogger(AppInsere.class.getName()).log(Level.SEVERE, null
                   , ex);
15         }
16     }
17 }
```

Código 4.1: Método main da classe AppInsere

Na linha 2, do código 4.1, um objeto da classe `Face` é criado (`face`). Na linha 4, o objeto `pm` (responsável por gerenciar a persistência) é criado. O método `addFileInDirectories` (linha 5) percorre todos os diretórios que estão no banco de faces LFW e armazena, na lista do tipo `Files` (`files`), os caminhos onde estão as imagens. Para cada item da lista `files`, a imagem é lida e atribuída ao campo `face` do objeto (linha 8); e o caminho da imagem é atribuído ao campo `label` do objeto (linha 9). Em seguida, nas linhas de 10 a 12, é realizada a persistência do objeto `face`, a indexação do atributo identificador `label` no domínio *Unique* e a extração/indexação dos dados estatísticos usando o histograma da imagem, dos 6 descritores mais relevantes de Haralick usando as matrizes de coocorrência simétricas e assimétricas, cada um em um índice do domínio *Feature*.

4.2.2 Resultados do Experimento 2

Nesta Seção são apresentados os resultados obtidos com o experimento 2 encarregado de avaliar o desempenho do *framework Obinject* ao utilizar sua anotação `@Feature` para extração e indexação de características das imagens.

Os resultados obtidos no experimento 2 são as alturas das estruturas *BTree+* e *MTree*, tempo médio para inserções, número de acessos ao disco e verificações (comparação e distância); e estão na tabela 4.1.

Tabela 4.1: Experimento 2: dados obtidos

	Altura da árvore	Tempo	Acesso ao disco	Verificação
Persistent	3	0,227 ms	3,778	30,133
Unique	3	0,034 ms	3,998	32,513
HistogramStatistical	3	0,155 ms	4,031	44,859
HaralickSymmetric	3	0,202 ms	4,059	43,674
HaralickAssymmetric	3	0,199 ms	4,055	43,383

A tabela 4.1 faz uma avaliação das anotações `@Persistent` e `@Unique` que utilizam a *BTree+* e dos métodos da anotação `@Feature` que extraem as características da imagens e indexam na *MTree*. Cada anotação indexa em uma árvore.

Pode-se observar na tabela 4.1 que todas as árvores (2 *BTree+* e 3 *MTree*) têm altura 3.

Além disso, o tempo médio para realizar a indexação na *BTree+* usando a anotação `@Unique` é menor que o tempo médio para extrair e indexar na *MTree* os dados estatísticos usando os histogramas, os 6 descritores mais relevantes de Haralick usando as matrizes de coocorrência assimétricas e simétricas. Além disso, ele também é menor que o tempo médio para persistir o objeto na *BTree+* usando anotação `@Persistent`. A extração e indexação dos 6 descritores mais relevantes de Haralick usando as matrizes de coocorrência simétricas levou mais tempo que a extração e indexação dos dados estatísticos usando os histogramas das imagens e dos 6 descritores mais relevantes de Haralick usando as matrizes de coocorrência assimétricas, ou seja, uma diferença aproximadamente de 30,5906% em relação ao tempo do dados estatísticos usando histogramas e de 1,5630% em relação ao tempo dos 6 descritores mais relevantes de Haralick usando as matrizes de coocorrência assimétricas.

Nota-se também que as anotações `@Persistent` e `@Unique` acessaram menos o disco do que as demais estruturas. Já a extração e indexação dos 6 descritores mais relevantes de Haralick usando as matrizes de coocorrência simétricas acessou mais vezes o disco, uma diferença de aproximadamente 0,6784% em relação a extração e indexação dos dados estatísticos usando histogramas e 0,0857% em relação aos 6 descritores mais relevantes de Haralick usando as matrizes de coocorrência assimétricas.

Percebe-se que as anotações `@Persistent` e `@Unique` realizaram menos verificações de comparação para utilizarem as *BTree+* do que os outros 3 métodos da anotação `@Feature` que realizaram verificações de distância para indexarem nas *MTree*. O número de verificações de distância para indexar os dados estatísticos usando histogramas é aproximadamente 2,7132% maior do que as verificações dos 6 descritores mais relevantes de Haralick usando matrizes de coocorrência simétricas e 3,4036% maior do que as verificações dos 6 descritores mais relevantes de Haralick usando matrizes de coocorrência assimétricas.

4.3 Considerações finais

A realização do experimento 1, com o propósito de comparar o nível de usabilidade do *framework Obinject* na versão anterior e após as contribuições deste trabalho, demonstrou que o uso de elementos nas anotações para definirem as funcionalidades de indexação tornou o processo de aplicação das anotações de indexação sobre os atributos mais trabalhoso e suscetível a erros. Isto é atestado pela queda da pontuação da usabilidade e no aumento de anotações erradas aplicadas nas classes, que por consequência geraram classes empacotadoras erradas.

A execução do experimento 2, com o objetivo de avaliar o desempenho do *framework Obinject* ao utilizar sua anotação `@Feature` para extração e indexação de características das imagens, mostrou que a utilização do método para extrair os 6 descritores mais relevantes de Haralick usando as matrizes de coocorrência simétricas levou mais tempo e acessou mais o disco do que os outros métodos para a extração de características das imagens; e que o método para obter os dados estatísticos usando o histograma da imagem realizou mais verificação para o cálculo de distância comparado aos demais métodos que extraem as características e indexam também na *MTree*.

Capítulo 5

Conclusão

O *framework Obinject* tem se mostrado uma solução completa e flexível para persistência e indexação de objetos.

Neste sentido, este trabalho apresentou como contribuições principais, no *framework Obinject*, a definição de um esquema de encadeamento de anotações que garante múltiplos índices usando uma única anotação por domínio e que suporte o posicionamento dos atributos na composição da chave de indexação. Além disso, a criação de um novo domínio de indexação para imagens através da extração de características utilizando abordagem estatística para produzir caracterizações das texturas das imagens.

A realização deste trabalho proporcionou uma redução do número de anotações no *framework Obinject*, de 101 para 9 anotações; um aumento da flexibilidade, permitindo que o usuário defina a posição que o atributo formará a chave de indexação; e da funcionalidade, tornando as anotações repetíveis e criando uma anotação para extração e indexação de imagens.

É importante destacar que para realizar este trabalho as técnicas de metaprogramação da linguagem de programação Java, como anotações, tipos genéricos e reflexão, além de alguns conceitos de processamento e análise de imagens foram estudados para que novas funcionalidades fossem adicionadas ao *framework Obinject*.

Durante o estudo para a realização deste trabalho houve uma preocupação na preparação de um material teórico que facilitasse ao leitor durante a aprendizagem de conceitos importantes dos recursos de metaprogramação em Java e de alguns conceitos do processamento e análise de imagens.

O experimento 1 mostrou que com a realização deste trabalho o processo de aplicação das anotações de indexação sobre os atributos tornou-se mais trabalhoso e suscetível a erros, atestado pela queda da pontuação da usabilidade e no aumento de anotações erradas aplicadas nas classes, que por consequência geraram classes empacotadoras erradas.

O experimento 2 demonstrou que a utilização do método para extrair os 6 descritores mais relevantes de Haralick usando as matrizes de coocorrência simétricas levou mais tempo e acessou mais o disco do que os outros métodos para a extração de características das imagens; e que o método para obter os dados estatísticos usando o histograma da imagem realizou mais verificação para o cálculo de distância comparado aos demais métodos que extraem as características e indexam também em uma *MTree*.

5.1 Trabalhos Futuros

As propostas futuras para possíveis trabalhos são:

1. Implementações de métodos de representação e descrição de imagens que utilizam outras abordagens, como abordagem baseada no processamento de sinal, abordagem geométrica, abordagem baseada em modelos paramétricos; para que auxilie nas consultas realizadas por similaridade ou dissimilaridade. Essas implementações restringem a expansão dos valores para o elemento `method` da anotação `@Feature`.

2. Implementação de esquema de anotações que permita pré-processamento, segmentação e representação das imagens através de outras definições de encadeamento de anotações.

3. Análise de busca por semelhança para investigar melhores resultados na indexação de imagens de uma área específica, por exemplo, imagens médicas.

4. Aprimorar a gramática da linguagem de consulta do *framework Obinject* para realizar busca por semelhança de imagens usando as definições de indexação deste trabalho.

Referências Bibliográficas

ANNAMALAI, M.; CHOPRA, R.; DEFAZIO, S.; MAVRIS, S. Indexing images in oracle8i. In: **Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data**. New York, NY, USA: ACM, 2000. (SIGMOD '00), p. 539–547. ISBN 1-58113-217-4. http://dl.acm.org/citation.cfm?id=335463&CFID=799695344&CFTOKEN=62642343 (visitado em 06/06/2017).

ARNOLD, K.; GOSLING, J. **A linguagem de programação Java**. 4. ed. São Paulo: Bookman Editora, 2007. ISBN 0-321-34980-6.

BANGOR, A.; KORTUM, P. T.; MILLER, J. T. An empirical evaluation of the system usability scale. **International. Journal of Human–Computer Interaction**, v. 24, n. 6, p. 574–594, 2008. http://www.tandfonline.com/doi/full/10.1080/10447310802205776?scroll=top&needAccess=true (visitado em 13/06/2017).

BARALDI, A.; PARMIGGIANI, F. An investigation of the textural characteristics associated with gray level cooccurrence matrix statistical parameters. v. 33, n. 2, p. 293–304, April 1995.

BAUER, C.; KING, G. 2005. http://hibernate.org/orm/ (visitado em 06/06/2017).

BOWMAN, W. J.; MILLER, S.; ST-AMOUR, V.; DYBVIG, R. K. Profile-guided meta-programming. **SIGPLAN Not.**, ACM, New York, NY, USA, v. 50, n. 6, p. 403–412, jun. 2015. ISSN 0362-1340. http://dl.acm.org/citation.cfm?id=2737990&CFID=799695344&CFTOKEN=62642343 (visitado em 23/05/2016).

BRACHA, G. **Generics in the Java programming language**. 2004. http://www.cs.unibo.it/martini/PP/generics-tutorial.pdf (visitado em 24/05/2016).

BROOKE, J. et al. Sus-a quick and dirty usability scale. **Usability evaluation in industry**, London, United Kingdom, v. 189, n. 194, p. 4–7, 1996.

CARVALHO, L. O. **Object-Injection: Um Framework de Indexação e Persistência**. http://saturno.unifei.edu.br/bim/0039959.pdf (visitado em 18/05/2016). Dissertação (Mestrado) — Universidade Federal de Itajubá, Março 2013.

CASTANÓN, C. A. B. **Recuperação de imagens por conteúdo através de análise multiresolução por wavelets**. http://www.teses.usp.br/teses/disponiveis/55/55134/tde-29072004-194807/pt-br.php (visitado em 19/04/2016). Dissertação (Mestrado) — Instituto de Ciências Matemáticas e de Computação - Universidade de São Paulo, Fevereiro 2003.

CÓRDOBA-SÁNCHEZ, I.; LARA, J. de. Ann: A domain-specific language for the effective design and validation of java annotations. **Computer Languages, Systems & Structures**, v. 45, p. 164–190, 2016. ISSN 1477-8424. http://www.sciencedirect.com/science/article/pii/S1477842416300318 (visitado em 06/06/2017).

DAMAŠEVIČIUS, R.; ŠTUIKYS, V. Taxonomy of the fundamental concepts of metaprogramming. **Information Technology and Control**, v. 37, n. 2, 2008. http://itc.ktu.lt/index.php/ITC/article/view/11931 (visitado em 23/05/2016).

DEVRIESE, D.; PIESSENS, F. Typed syntactic meta-programming. **SIGPLAN Not.**, ACM, New York, NY, USA, v. 48, n. 9, p. 73–86, set. 2013. ISSN 0362-1340. http://dl.acm.org/citation.cfm?id=2500575&CFID=799695344&CFTOKEN=62642343 (visitado em 23/05/2016).

ELEYAN, A.; DEMIREL, H. Co-occurrence matrix and its statistical features as a new approach for face recognition. **Turkish Journal of Electrical Engineering & Computer Sciences**, The Scientific and Technological Research Council of Turkey, v. 19, n. 1, p. 97–107, 2011. https://journals.tubitak.gov.tr/elektrik/abstract.htm?id=11424 (visitado em 16/10/2015).

ENGINEERING, U. of W. C. S. . **The Checker Framework**. 2016. http://types.cs.washington.edu/checker-framework/ (visitado em 10/11/2016).

FERRO, C. **ObInject Query Language**. https://repositorio.unifei.edu.br/xmlui/bitstream/handle/123456789/381/dissertacao_ferro_2012.pdf?sequence=1&isAllowed=y (visitado em 18/05/2016). Dissertação (Mestrado) — Universidade Federal de Itajubá, Novembro 2012.

FORMAN, I. R.; FORMAN, N. **Java reflection in action**. Greenwich, CT, USA: Manning Publications Co, 2004. ISBN 1-932394-18-4.

GONZALEZ, R.; WOODS, R. **Digital Image Processing**. 3. ed. São Paulo: Pearson Prentice Hall, 2010. ISBN 978-85-7605-401-6.

GOSLING, J.; JOY, B.; STEELE, G.; BRACHA, G.; BUCKLEY, A. **The Java® Language Specification Java SE 8 Edition**. 2015. http://docs.oracle.com/javase/specs/jls/se8/jls8.pdf (visitado em 23/05/2016).

HARALICK, R. M.; SHANMUGAM, K.; DINSTEIN, I. H. Textural features for image classification. **IEEE Transactions on systems, man and cybernetics**, IEEE, n. 6, p. 610–621, 1973. http://haralick.org/journals/TexturalFeatures.pdf (visitado em 30/04/2015).

HAT, I. R. 2009. http://www.seamframework.org/ (visitado em 06/06/2017).

HUANG, G. B.; RAMESH, M.; BERG, T.; LEARNED-MILLER, E. **Labeled Faces in the Wild: A Database for Studying Face Recognition in Unconstrained Environments**. [S.l.], October 2007. `http://vis-www.cs.umass.edu/lfw/#download` (visitado em 25/05/2017).

JUNEAU, J. Jsr 308 explained: Java type annotations. **Java Magazine**, Oracle Inc., p. 56–60, 2014.

MAGNO, D. G. **Aplicação da Técnica de Scaffolding para a Criação de Sistemas CRUD.** `https://repositorio.unifei.edu.br/xmlui/bitstream/handle/123456789/197/dissertacao_magno_2015.pdf?sequence=1&isAllowed=y` (visitado em 18/05/2016). Dissertação (Mestrado) — Universidade Federal de Itajubá, Setembro 2015.

MANCINI, F.; HOVLAND, D.; MUGHAL, K. A. Investigating the limitations of java annotations for input validation. In: **2010 International Conference on Availability, Reliability and Security**. [S.l.]: IEEE, 2010. p. 513–518. `http://ieeexplore.ieee.org/abstract/document/5438045/?reload=true` (visitado em 06/06/2017).

OLIVEIRA, A. de A. **MetaJ: Um Ambiente para Meta-Programação em Java**. `http://www.facom.ufu.br/~marcmaia/arquivos/dissertacaoAdemir.pdf` (visitado em 03/01/2017). Dissertação (Mestrado) — Universidade Federal de Minas Gerais, Novembro 2004.

OLIVEIRA, M. F. de. **Metaprogramação e Metadados de Persistência e Indexação para o Framework Object-Injection**. `http://saturno.unifei.edu.br/bim/0040022.pdf` (visitado em 23/05/2016). Dissertação (Mestrado) — Universidade Federal de Itajubá, Dezembro 2012.

ORACLE. 2006. `http://www.oracle.com/technetwork/java/javaee/tech/persistence-jsp-140049.html` (visitado em 06/06/2017).

ORACLE, C. a. i. a. **Java Platform, Standard Edition 8 API Specification**. 2016. `https://docs.oracle.com/javase/8/docs/api/` (visitado em 24/11/2016).

_____. **The Java Tutorials – Lesson: Annotations**. 2016. `https://docs.oracle.com/javase/tutorial/java/annotations/index.html` (visitado em 26/05/2016).

_____. **The Java Tutorials – Lesson: Generics (updated)**. 2016. `http://docs.oracle.com/javase/tutorial/java/generics/index.html` (visitado em 25/05/2016).

_____. **The Java Tutorials – Trail: The Reflection API**. 2016. `https://docs.oracle.com/javase/tutorial/reflect/` (visitado em 25/05/2016).

PEDRINI, H.; SCHWARTZ, W. R. **Análise de imagens digitais: princípios, algoritmos e aplicações**. São Paulo: Thomson Learning, 2008. ISBN 978-85-221-0595-3.

PIASECKI, R. **softwarecave: Repeating annotations in Java 8**. 2014. https://softwarecave.org/2014/05/20/repeating-annotations-in-java-8/ (visitado em 19/01/2017).

PRESS, N. Understanding metadata. **National Information Standards**, v. 20, 2004. http://www.lter.uaf.edu/metadata_files/UnderstandingMetadata.pdf (visitado em 23/05/2016).

ROCHA, H. V. D.; BARANAUSKAS, M. C. C. **Design e avaliação de interfaces humano-computador**. [S.l.]: Universidade Estadual de Campinas – Unicamp, 2003. http://www.nied.unicamp.br/?q=content/design-e-avalia%C3%A7%C3%A3o-de-interfaces-humano-computador (visitado em 12/06/2017).

ROMDHANE, L. B.; BANNOUR, H.; AYEB, B. Imiol: A system for indexing images by their semantic content based on possibilistic fuzzy clustering and adaptive resonance theory neural networks learning. **Applied Artificial Intelligence**, v. 24, n. 9, p. 821–846, 2010. http://www.tandfonline.com/doi/full/10.1080/08839514.2010.514194?scroll=top&needAccess=true (visitado em 06/06/2017).

SAURO, J. **Measuring Usability with the System Usability Scale (SUS)**. 2011. https://measuringu.com/sus/ (visitado em 30/05/2017).

SCHWARTZ, W. R.; SIQUEIRA, F. R. de; PEDRINI, H. Evaluation of feature descriptors for texture classification. **Journal of Electronic Imaging**, v. 21, p. 17, 2012. https://www.spiedigitallibrary.org/journals/Journal-of-Electronic-Imaging/volume-21/issue-2/023016/Evaluation-of-feature-descriptors-for-texture-classification/10.1117/1.JEI.21.2.023016.full?SSO=1 (visitado em 11/12/2015).

Introduction to fast indexing method for images in database, v. 9631. 9631 - 9631 - 5 p. https://www.spiedigitallibrary.org/conference-proceedings-of-spie/9631/1/Introduction-to-fast-indexing-method-for-images-in-database/10.1117/12.2197176.full (visitado em 06/06/2017).

SHEARD, T. Accomplishments and research challenges in meta-programming. In: **In 2nd Int. Workshop on Semantics, Applications, and Implementation of Program Generation, LNCS 2196**. Berlin, Heidelberg: Springer-Verlag, 2001. p. 2–44. ISBN 3-540-42558-6. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.15.7818 (visitado em 23/05/2016).

SIERRA, K.; BATES, B.; COELHO, A. J. **Use a cabeça!: Java**. 2. ed. Rio de Janeiro: Alta Books, 2007. ISBN 978-85-7608-173-9.

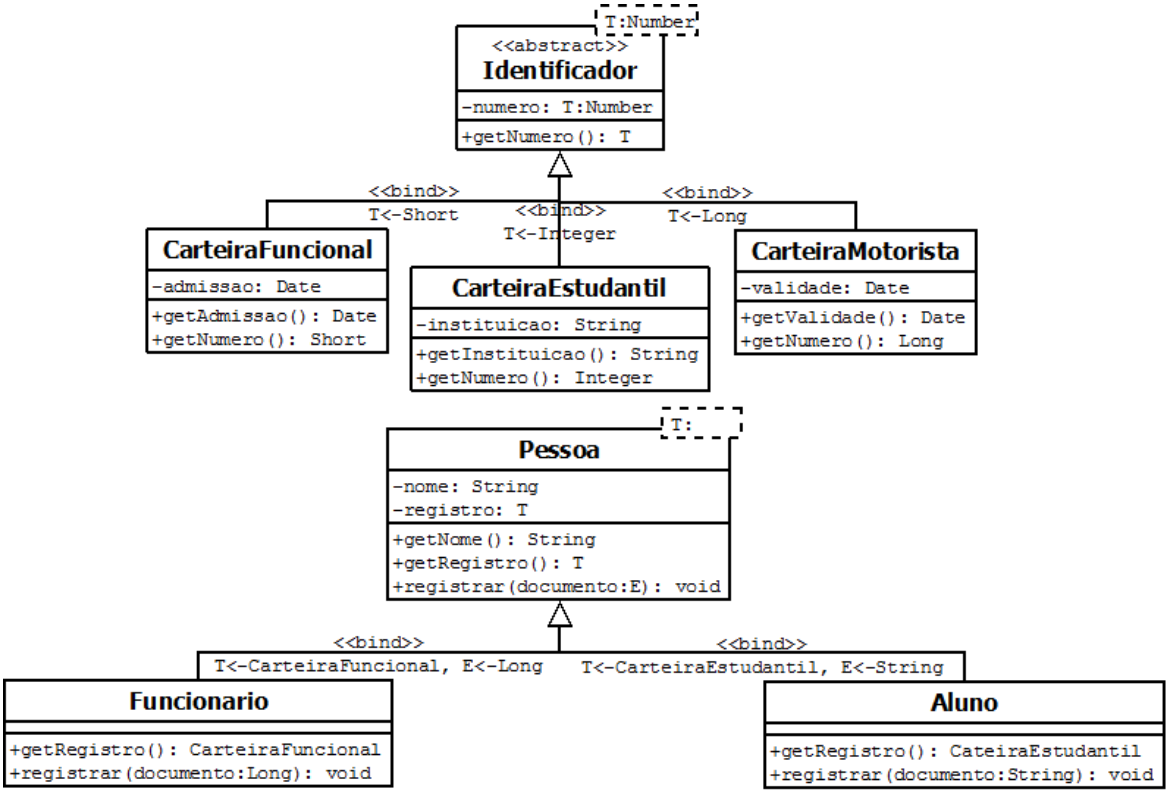
SIMPSON, M. S.; DEMNER-FUSHMAN, D.; ANTANI, S. K.; THOMA, G. R. Multimodal biomedical image indexing and retrieval using descriptive text and global feature mapping. **Information retrieval**, Springer, v. 17, n. 3, p. 229–264, Jun 2014. `https://link.springer.com/article/10.1007/s10791-013-9235-2` (visitado em 06/06/2017).

SOFTWARE, P. 2002. `http://spring.io/` (visitado em 06/06/2017).

Apêndice A

Diagrama UML das classes genéricas Carteira e Pessoa com suas subclasses

A figura abaixo é um diagrama UML (*Unified Modeling Language*) da classe genérica `Carteira`, com suas subclasses `CarteiraEstudantil`, `CarteiraFuncional` e `CarteiraMotorista`; e da classe genérica `Pessoa`, com suas subclasses `Aluno` e `Funcionario`.

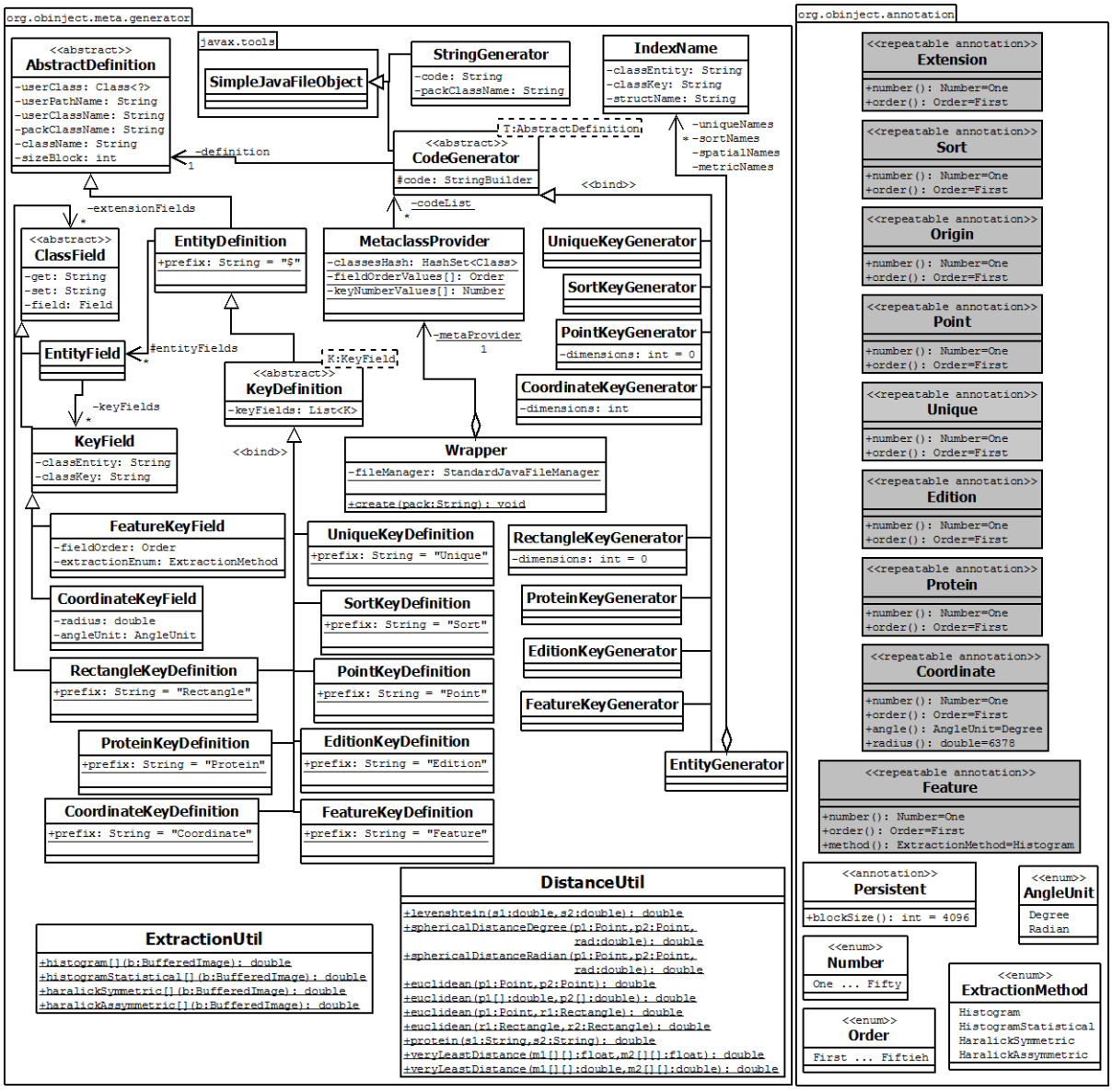


O diagrama UML mostra que as classes `CarteiraEstudantil`, `CarteiraFuncional` e `CarteiraMotorista` passam os tipos `Integer`, `Short` e `Long`, respectivamente, substituindo a variável de tipo `T` da superclasse `Carteira`. Já as classes `Aluno` e `Funcionario` passam os tipos `CarteiraEstudantil` e `CarteiraFuncional`, respectivamente, sobrepondo o parâ-

metro de tipo `T` da superclasse `Pessoa`; e passam os tipos `String` e `Long`, respectivamente, substituindo a variável de tipo `E` definida na declaração do método `registrar` da superclasse `Pessoa`.

Apêndice B

Pacotes *Meta* e *Annotation* do *Framework Obinject*



Apêndice C

Roteiro do experimento 1: *Framework Obinject* versão 1

Fase 1: Treinamento		
Curso: **Período/Ano de início:**		
Data:	**Hora de Início:**	**Hora de Término:**

O *Framework Obinject* é um *framework* de persistência e indexação de objetos escrito na linguagem de programação Java. A persistência possibilita que os objetos sejam armazenados e a indexação possibilita que chaves agilizem as consultas por objetos.

Para persistir uma classe é preciso aplicar a anotação **@Persistent** antes da declaração da classe.

Cada anotação de indexação forma um domínio. Para indexar uma classe, um ou mais atributos, que formarão a chave, devem receber as anotações que estão na tabela.

Anotações	Função	Aplicadas ao(s) tipo(s)
@PrimaryKey	Define que os atributos não podem conter valores repetidos.	Primitivos e String
@Order[First... Twentieth]	Define que os atributos terão um índice que agiliza consultas que necessitam de ordenação.	Primitivos e String
@Edition[First... Twentieth]	Define que os atributos terão um índice que agiliza consultas que necessitam de semelhanças entre palavras e proteínas.	String
@Point[First... Twentieth]	Define que os atributos terão um índice que agiliza consultas que necessitam de semelhanças no espaço euclidiano usando a estrutura métrica *MTree*.	Primitivo numérico
@GeoPoint[First... Twentieth]	Define que os atributos terão um índice que agiliza consultas que necessitam de semelhanças no espaço esférico (Terra).	Primitivo numérico
@Rectangle[First... Twentieth]	Define que os atributos terão um índice que agiliza consultas que necessitam de semelhanças no espaço euclidiano usando a estrutura espacial *RTree*.	Primitivo numérico

Vale ressaltar que a mesma anotação não pode ser aplicada mais de uma vez sobre o mesmo atributo.

A anotação **@PrimaryKey** não pode ser utilizada para criar mais de um índice, mas ela pode ser aplicada em mais de um atributo.

Para indexar dois ou mais atributos no domínio **Point** utilizando a mesma estrutura, por exemplo, basta anotá-los com **@PointFirst**.

E para indexar dois ou mais atributos no domínio **Point** utilizando estruturas diferentes, por exemplo, é preciso anotar um atributo com **@PointFirst**, o outro atributo com **@PointSecond** e assim por diante até **@PointTwentieth**. Ou seja, para especificar mais de um índice do mesmo domínio, é necessário utilizar uma das anotações do domínio (nome do domínio seguido do ordinal).

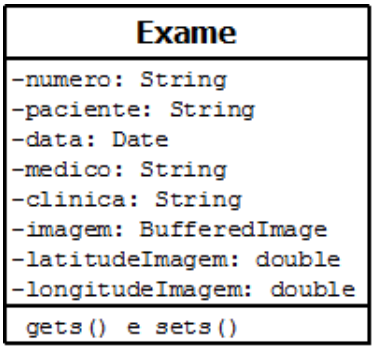
É importante destacar que para persistir objetos de uma classe deve-se aplicar a anotação **@Persistent** e **@PrimaryKey**.

A seguir é apresentado um exemplo que mostra o uso das anotações do *framework Obinject* sobre a classe Exame. Para ver este exemplo, abra a IDE Netbeans e acesse **Meus Documentos/Experimento/ projeto encadeamento-modelo1**.

Código-fonte da classe Exame:

```
1  @Persistent
2  public class Exame {
3
4      @PrimaryKey
5      private String numero;
6
7      @OrderFirst
8      @EditionSecond
9      private String paciente;
10
11     private Date data = new Date();
12
13     @EditionFirst
14     @EditionSecond
15     private String medico;
16
17     @OrderFirst
18     @EditionFirst
19     private String clinica;
20
21     private BufferedImage imagem;
22
23     @PointFirst
24     @GeoPointFirst
25     @RectangleFirst
26     private double latitudeImagem;
27
28     @PointFirst
29     @GeoPointFirst
30     @RectangleFirst
31     private double longitudeImagem;
32
33     //gets e sets
34
35 }
```

UML da classe Exame:



A classe Exame, anotada por **@Persistent**, é persistida.

O atributo **numero**, anotado por **@PrimaryKey**, é utilizado como chave de identificação do objeto da classe Exame.

Os atributos **paciente** e **clinica**, anotados por **@OrderFirst**, indica que eles são indexados em um mesmo índice que agiliza consultas que necessitam de ordenação.

Os atributos **medico** e **clinica**, anotados por **@EditionFirst**, são indexados em um índice que agiliza consultas que necessitam de semelhanças entre palavras.

Os atributos **paciente** e **medico**, anotados por **@EditionSecond**, indica que eles são indexados em um outro índice que agiliza consultas que necessitam de semelhanças entre palavras.

E, finalmente, os atributos **latitudeImagem** e **longitudeImagem**, quando anotados por **@PointFirst**, indica que eles serão indexados em um mesmo índice que agiliza consultas que necessitam de semelhanças no espaço euclidiano usando a estrutura métrica *MTree*. Quando esses atributos são anotados por **@GeoPointFirst**, indica que eles serão indexados em um mesmo índice que agiliza consultas que necessitam de semelhanças no espaço esférico. E quando anotados por **@RectangleFirst**, indica que eles serão indexados em um mesmo índice que agiliza consultas que necessitam de semelhanças no espaço euclidiano usando a estrutura espacial *RTree*.

Para realizar a persistência e a indexação da classe de aplicação Exame, além das anotações que precisam ser aplicadas, como **@Persistent**, **@PrimaryKey**; é preciso executar a classe **AppGerarClasses** (responsável por gerar as classes que fazem o vínculo entre a classe de aplicação Exame e o *Framework Obinject*) e a classe **AppPersistir** (responsável por persistir os objetos da classe Exame). Execute primeiramente a classe AppGerarClasses e, em seguida, a classe AppPersistir.

A seguir é apresentado o código da classe AppGerarClasses e da classe AppPersistir.

Código-fonte da classe AppGerarClasses:

```
1 public class AppGerarClasses {
2
3     public static void main(String[] args) {
4
5         Wrapper.create("org.obinject.sample.exame");
6
7     }
8 }
```

Código-fonte da classe AppPersistir:

```
1  public class AppPersistir {
2      public static void main(String[] args) {
3          PersistentManager pm = PersistentManagerFactory.
               createPersistentManager();
4          pm.getTransaction().begin();
5          Exame p = new Exame();
6          p.setNumero("123abc");
7          p.setPaciente("ze");
8          p.setMedico("Dr.Tião");
9          p.setClinica("Clinica Modelo");
10         p.setLatitudeImagem(-22.43);
11         p.setLongitudeImagem(-45.45);
12         pm.persist(p);
13         pm.getTransaction().commit();
14     }
15 }
```

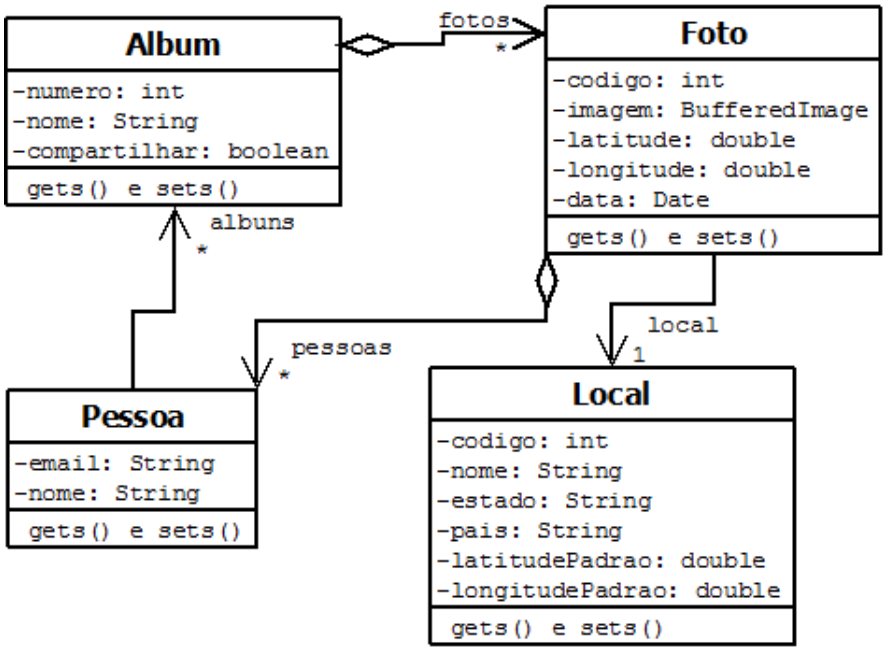
Fase 2: Execução

Curso:
Período/Ano de início:

Data: **Hora de Início:** **Hora de Término:**

A figura abaixo é um diagrama UML de classes que modela um sistema de armazenamento de álbuns com identificação das pessoas e da localidade nas fotos. Uma pessoa pode ter vários álbuns que contêm várias fotos. Cada foto têm a identificação das pessoas e da localidade.

1. **Abra na IDE Netbeans o projeto chamado encadeamento-experimento1 que está no diretório Meus Documentos/ Experimento 1**. Codifique as 4 classes do diagrama UML abaixo.



2. Após a codificação das classes, é preciso torná-las persistentes.

   Além disso, para cada classe é necessário:

3. Na classe Album:

   - tornar o atributo **numero** identificador da classe **Album**.
   - indexar o atributo **nome** no índice que agiliza consultas que necessitam de semelhanças entre palavras.

4. Na classe Foto:

   - tornar o atributo **codigo** identificador da classe Foto.

- indexar os atributos **latitude** e **longitude** no índice que agiliza consultas que necessitam de semelhanças no espaço euclidiano usando a estrutura métrica *MTree*.
- indexar os atributos **latitude** e **longitude** no índice que agiliza consultas que necessitam de semelhanças no espaço esférico.
- indexar os atributos **latitude** e **longitude** no índice que agiliza consultas que necessitam de semelhanças no espaço euclidiano usando a estrutura espacial *RTree*.

5.Na classe Local:

- tornar o atributo **codigo** identificador da classe **Local**.
- indexar o atributo **nome** no índice que agiliza consultas que necessitam de semelhanças entre palavras.
- indexar os atributos **latitudePadrao** e **longitudePadrao** em um índice que agiliza consultas que necessitam de semelhanças no espaço euclidiano usando a estrutura métrica *MTree*.
- indexar os atributos **latitudePadrao** e **longitudePadrao** em um índice que agiliza consultas que necessitam de semelhanças no espaço esférico.
- indexar os atributos **latitudePadrao** e **longitudePadrao** em um índice que agiliza consultas que necessitam de semelhanças no espaço euclidiano usando a estrutura espacial *RTree*.

6.Na classe Pessoa:

- tornar o atributo **email** identificador da classe **Pessoa**.
- indexar o atributo **email** em um índice que agiliza consultas que necessitam de ordenação.
- indexar o atributo **nome em um outro índice** que agiliza consultas que necessitam de semelhanças entre palavras.

7.Executar a classe **AppGerarClasses** e em seguida a classe **AppPersistir**.

Fase 3: Avaliação da versão 1 do framework Obinject

Curso:
Período/Ano de início:

Data: **Hora de Início:** **Hora de Término:**

Questionário de Avaliação de Usabilidade - Questões de 1 a 10 Traduzido do original *"SUS - A quick and dirty usability scale"*, John Brooke, 1996

Nas questões de 1 a 10, selecione uma das opções na escala de 1 a 5, sendo 1 discordo totalmente e 5 concordo totalmente.

*** Preenchimento obrigatório**

1.Acha que gostaria de usar esse sistema com frequência. *

Discordo totalmente 1 2 3 4 5 Concordo totalmente

2.Achei o sistema desnecessariamente complexo. *

Discordo totalmente 1 2 3 4 5 Concordo totalmente

3.Achei o sistema fácil de usar. *

Discordo totalmente 1 2 3 4 5 Concordo totalmente

4.Eu acho que precisaria do apoio de um técnico para conseguir usar esse sistema. *

Discordo totalmente 1 2 3 4 5 Concordo totalmente

5.Achei que as várias funções desse sistema estavam bem integradas. *

Discordo totalmente 1 2 3 4 5 Concordo totalmente

6.Achei que tinha muitas inconsistências nesse sistema. *

Discordo totalmente 1 2 3 4 5 Concordo totalmente

7.Acho que a maioria das pessoas aprenderia a usar esse sistema facilmente. *

Discordo totalmente 1 2 3 4 5 Concordo totalmente

8.Achei o sistema muito complicado de se usar. *

Discordo totalmente 1 2 3 4 5 Concordo totalmente

9.Eu me senti muito confiante usando o sistema. *

Discordo totalmente 1 2 3 4 5 Concordo totalmente

10.Eu precisei aprender várias coisas antes de continuar usando o sistema. *

Discordo totalmente 1 2 3 4 5 Concordo totalmente

11.Conhece a linguagem de programação Java a quanto tempo? *

( ) 0 meses - 1 ano
( ) 1 ano - 2 anos
( ) 2 anos - 3 anos
( ) 3 anos - 5 anos
( ) Acima de 5 anos

12.Com base na realização da fase de execução, preencha a tabela abaixo com o total de anotações (persistência e indexação) aplicadas para cada classe: *

Obs: As anotações com o mesmo nome podem ser contabilizadas.

Classe	**Total de Anotações**
Album	
Foto	
Local	
Pessoa	

13.Após a execução do classe AppGerarClasses, quantas classes foram geradas? *

Obs: Não contabilizar as classes: AppGerarClasses, AppPersistir, Album, Foto, Local e Pessoa.

Resposta:

14.Quais foram as dificuldades encontradas?

Resposta:

Apêndice D

Roteiro do experimento 1: *Framework Obinject* versão 2

Fase 1: Treinamento		
Curso: **Período/Ano de início:**		
Data:	**Hora de Início:**	**Hora de Término:**

O *Framework Obinject* é um *framework* de persistência e indexação de objetos escrito na linguagem de programação Java. A persistência possibilita que os objetos sejam armazenados e a indexação possibilita que chaves agilizem as consultas por objetos.

Anotações	Função	Aplicadas ao(s) tipo(s)	Elementos da anotação
@Unique	Define que os atributos não podem conter valores repetidos.	Primitivos, String e Date	number, order
@Sort	Define que os atributos terão um índice que agiliza consultas que necessitam de ordenação.	Primitivos e String	number, order
@Edition	Define que os atributos terão um índice que agiliza consultas que necessitam de semelhanças entre palavras.	String	number, order
@Feature	Define que os atributos terão um índice que agiliza consultas que necessitam de semelhanças de características das imagens.	BufferedImage	number, order, method
@Protein	Define que os atributos terão um índice que agiliza consultas que necessitam de semelhanças entre proteínas.	String	number, order
@Point	Define que os atributos terão um índice que agiliza consultas que necessitam de semelhanças no espaço euclidiano usando a estrutura métrica *MTree*.	Primitivo numérico	number, order
@Coordinate	Define que os atributos terão um índice que agiliza consultas que necessitam de semelhanças no espaço esférico (Terra).	Primitivo numérico	number, order
@Origin	Define que os atributos terão um índice que agiliza consultas que necessitam de semelhanças no espaço euclidiano usando a estrutura espacial *RTree*.	Primitivo numérico	number, order

Para persistir uma classe é preciso aplicar a anotação **@Persistent** antes da declaração da classe. E para indexar uma classe, um ou mais atributos, que formarão a chave, devem receber as anotações da tabela.

Cada uma das 8 anotações de indexação forma um domínio. Além disso, elas podem ser aplicadas tanto no mesmo atributo quanto em atributos diferentes.

Para indexar dois ou mais atributos no domínio **Point** utilizando a mesma estrutura, por exemplo, é preciso anotá-los com **@Point** e atribuir ao elemento **number** da anotação o valor **Number.One**.

Para indexar dois ou mais atributos no domínio **Point** utilizando índices diferentes, por exemplo, é preciso anotá-los com **@Point** e atribuir para o elemento **number** da anotação o valor **Number.One** no atributo que será indexado no índice um, o valor **Number.Two** no atributo que será indexado no índice dois e assim sucessivamente.

E para especificar a posição que os atributos, por exemplo, serão indexados na chave de

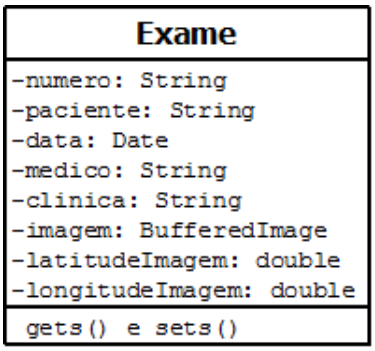
indexação do domínio **Point**, é preciso anotá-los com **@Point** e atribuir para o elemento **order** da anotação o valor **Order.First** no atributo que ficará na primeira posição da chave de indexação, o valor **Order.Second** no atributo será indexado na segunda posição da chave de indexação e assim por diante.

Vale ressaltar que a anotação **@Feature** possui também o elemento **method** que define que característica da imagem será extraída. Se o valor atribuído ao elemento method for **ExtractionMethod.Histogram**, o histograma da imagem será obtido; se for **ExtractionMethod.HistogramStatistical**, os dados estatísticos usando histogramas serão obtidos; se o valor for **ExtractionMethod.HaralickSymmetric**, os descritores de Haralick usando matrizes de coocorrência simétricas serão obtidos; e se o valor for **ExtractionMethod.HaralickAssymmetric** os descritores de Haralick usando matrizes de coocorrência assimétricas serão obtidos.

É importante destacar que para persistir objetos de uma classe deve-se aplicar a anotação **@Persistent** e **@Unique**.

A seguir é apresentado um exemplo que mostra o uso das anotações do *framework Obinject* sobre a classe Exame. **Para ver este exemplo, abra a IDE Netbeans e acesse Meus Documentos/Experimento 2/ projeto encadeamento-modelo2**.

UML da classe Exame:



Código-fonte da classe Exame:

```
1  @Persistent
2  public class Exame {
3
4      @Unique(number = Number.One, order = Order.First)
5      private String numero;
6
7      @Sort(number = Number.One, order = Order.Second)
8      @Edition(number = Number.Two, order = Order.Second)
9      private String paciente;
10
11     private Date data = new Date();
12
13     @Edition(number = Number.One, order = Order.First)
14     @Edition(number = Number.Two, order = Order.First)
15     private String medico;
16
17     @Sort(number = Number.One, order = Order.First)
18     @Edition(number = Number.One, order = Order.Second)
19     private String clinica;
20
21     @Feature(number = Number.One, order = Order.First,
22                         method = ExtractionMethod.HistogramStatistical)
23     @Feature(number = Number.Two, order = Order.First,
24                         method = ExtractionMethod.HaralickSymmetric)
25     private BufferedImage imagem;
26
27     @Point(number = Number.One, order = Order.First)
28     @Coordinate(number = Number.One, order = Order.First)
29     @Origin(number = Number.One, order = Order.First)
30     private double latitudeImagem;
31
32     @Point(number = Number.One, order = Order.Second)
33     @Coordinate(number = Number.One, order = Order.Second)
34     @Origin(number = Number.One, order = Order.Second)
35     private double longitudeImagem;
36
37     //gets e sets
38
39 }
```

A classe Exame, anotada por **@Persistent**, é persistida.

O atributo **numero**, anotado por **@Unique**, é utilizado como chave de identificação do objeto da classe Exame. Ele está indexado no índice UM **(number = Number.One)** do domínio **Unique** e na PRIMEIRA posição da chave de indexação **(order = Order.First)**.

Os atributos **paciente** e **clinica**, anotados por **@Sort** com elemento number igual a **Number.One**, indica que eles são indexados em um mesmo índice que agiliza consultas que necessitam de ordenação. Além disso, o atributo **clinica** estará na primeira posição devido ao valor **Order.First** atribuído ao elemento **order**; e o atributo **paciente** na segunda posição da chave de indexação, devido ao valor **Order.Second** atribuído ao elemento **order** da

anotação **@Sort**.

Os atributos **medico** e **clinica**, anotados por **@Edition** com elemento **number** igual a **Number.One**, serão indexados em um índice que agiliza consultas que necessitam de semelhanças entre palavras, sendo que o atributo **medico** está na primeira posição e o atributo **clinica** na segunda posição da chave de indexação devido ao valor atribuído ao elemento **order** da anotação **@Edition**.

Os atributos **paciente** e **medico**, anotados por **@Edition** com elemento number igual a **Number.Two**, são indexados **em outro índice** que agiliza consultas que necessitam de semelhanças entre palavras, sendo que o atributo **medico** está na primeira posição e o atributo **paciente** na segunda posição da chave de indexação devido ao valor atribuído ao elemento **order** da anotação **@Edition**.

A imagem contida no atributo **imagem**, anotada por **@Feature**, terá suas características extraídas e indexadas em um índice que agiliza consultas que necessitam de semelhanças de características das imagens. A primeira anotação **@Feature** com elemento **method** igual a **ExtractionMethod.HistogramStatistical** indica que os dados estatísticos a partir dos histogramas serão obtidos. Além disso, eles serão indexados no índice UM do domínio **Feature** devido ao valor do elemento **number** e na primeira posição devido ao valor do elemento **order**. Será indexado **em outro índice** que agiliza consultas que necessitam de semelhanças de características das imagens os descritores de Haralick usando matrizes de coocorrência simétricas da imagem devido ao valor atribuído ao elemento **method** da anotação **@Feature**.

E, finalmente, os atributos **latitudeImagem** e **longitudeImagem**, quando anotados por **@Point** com elemento **number** igual a **Number.One**, indica que eles serão indexados em um mesmo índice que agiliza consultas que necessitam de semelhanças no espaço euclidiano usando a estrutura métrica *MTree*. Quando esses atributos são anotados por **@Coordinate** com elemento **number** igual a **Number.One**, indica que eles serão indexados em um mesmo índice que agiliza consultas que necessitam de semelhanças no espaço esférico. E quando anotados por **@Origin** com elemento **number** igual a **Number.One**, indica que eles serão indexados em um mesmo índice que agiliza consultas que necessitam de semelhanças no espaço euclidiano usando a estrutura espacial *RTree*. O atributo **latitudeImagem** será indexado na primeira posição das chaves de indexação do domínios **Point**, **Coordinate** e **Origin** e o atributo **longitudeImagem** será indexado na segunda posição das chaves de indexação dos domínios **Point**, **Coordinate** e **Origin**.

Para realizar a persistência e a indexação da classe de aplicação Exame, além das anotações que precisam ser aplicadas, como **@Persistent**, **@Unique**, é preciso executar a classe **AppGerarClasses** (responsável por gerar as classes que fazem o vínculo entre a classe de aplicação Exame e o *framework Obinject*) e a classe **AppPersistir** (responsável por persistir

os objetos da classe Exame). Execute primeiramente a classe AppGerarClasses e, em seguida, a classe AppPersistir.

A seguir é apresentado o código da classe AppGerarClasses e da classe AppPersistir.

Código-fonte da classe AppGerarClasses:

```
1 public class AppGerarClasses {
2
3     public static void main(String[] args) {
4
5         Wrapper.create("org.obinject.sample.exame");
6
7     }
8 }
```

Código-fonte da classe AppPersistir:

```
1 public class AppPersistir {
2
3     public static void main(String[] args) {
4
5         PersistentManager pm = PersistentManagerFactory.
              createPersistentManager();
6         pm.getTransaction().begin();
7
8         Exame p = new Exame();
9         try {
10            p.setNumero("123abc");
11            p.setPaciente("ze");
12            p.setMedico("Dr.Tião");
13            p.setClinica("Clinica Modelo");
14            p.setImagem(ImageIO.read(new File("tomografia.png")));
15            p.setLatitudeImagem(-22.43);
16            p.setLongitudeImagem(-45.45);
17        } catch (IOException ex) {
18                Logger.getLogger(AppPersistir.class.getName()).log(Level.
                      SEVERE, null, ex);
19        }
20
21        pm.persist(p);
22        pm.getTransaction().commit();
23    }
24 }
```

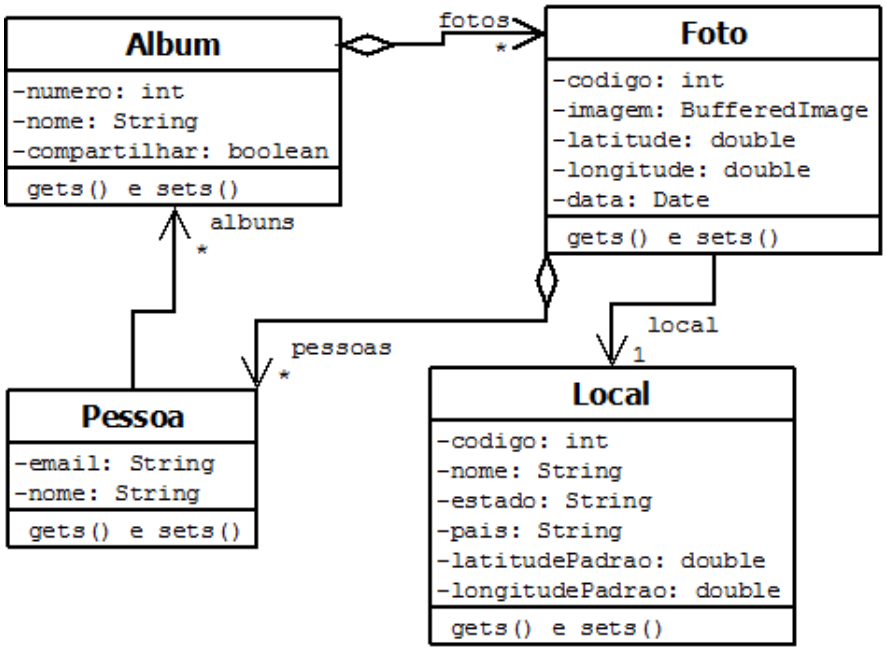
Fase 2: Execução

Curso:
Período/Ano de início:

Data: **Hora de Início:** **Hora de Término:**

A figura abaixo é um diagrama UML de classes que modela um sistema de armazenamento de álbuns com identificação das pessoas e da localidade nas fotos. Uma pessoa pode ter vários álbuns que contêm várias fotos. Cada foto têm a identificação das pessoas e da localidade.

1. **Abra na IDE Netbeans o projeto chamado encadeamento-experimento2 que está no diretório Meus Documentos/ Experimento 2**. Codifique as 4 classes do diagrama UML abaixo.



2. Após a codificação das classes, é preciso torná-las persistentes.

   Além disso, para cada classe é necessário:

3. Na classe Album:

   - tornar o atributo **numero** identificador da classe **Album**. O atributo deve ser indexado em um índice e na primeira posição da chave de indexação.
   - indexar o atributo **nome** no índice que agiliza consultas que necessitam de semelhanças entre palavras. Ele deve ser indexado em um índice e na primeira posição da chave de indexação.

4. Na classe Foto:

- tornar o atributo **codigo** identificador da classe **Foto**. O atributo deve ser indexado em um índice e na primeira posição da chave de indexação.
- indexar o atributo **imagem** no índice que agiliza consultas que necessitam de semelhanças de características de imagens. Deve-se obter o histograma da imagem e indexá-la em um índice do domínio e na primeira posição da chave de indexação. Deve indexar também **no mesmo índice** os descritores de Haralick usando as matrizes de coocorrência assimétricas na segunda posição da chave da chave de indexação.
- indexar os atributos **latitude** e **longitude** no índice que agiliza consultas que necessitam de semelhanças no espaço euclidiano usando a estrutura métrica *MTree*. O atributo latitude deve estar na primeira posição e o atributo longitude deve estar na segunda posição da chave de indexação.
- indexar os atributos **latitude** e **longitude** no índice que agiliza consultas que necessitam de semelhanças no espaço esférico. O atributo latitude deve estar na primeira posição e o atributo longitude na segunda posição da chave de indexação.
- indexar os atributos **latitude** e **longitude** no índice que agiliza consultas que necessitam de semelhanças no espaço euclidiano usando a estrutura espacial *RTree*. O atributo latitude deve estar na primeira posição o atributo longitude na segunda posição da chave de indexação.

5.Na classe Local:

- tornar o atributo **codigo** identificador da classe **Local**. O atributo deve ser indexado em um índice e na primeira posição da chave de indexação.
- indexar o atributo **nome** no índice que agiliza consultas que necessitam de semelhanças entre palavras. Ele deve ser indexado em uma estrutura e na primeira posição da chave de indexação.
- indexar os atributos **latitudePadrao** e **longitudePadrao** em um índice que agiliza consultas que necessitam de semelhanças no espaço euclidiano usando a estrutura métrica *MTree*. O atributo latitude deve estar na primeira posição e o atributo longitude deve estar na segunda posição da chave de indexação.
- indexar os atributos **latitudePadrao** e **longitudePadrao** em um índice que agiliza consultas que necessitam de semelhanças no espaço esférico. O atributo latitude deve estar na primeira posição e o atributo longitude na segunda posição da chave de indexação.
- indexar os atributos **latitudePadrao** e **longitudePadrao** em um índice que agiliza consultas que necessitam de semelhanças no espaço euclidiano usando a estrutura espacial *RTree*. O atributo latitude deve estar na primeira posição o atributo longitude na segunda posição da chave de indexação.

6.Na classe Pessoa:

- tornar o atributo **email** identificador da classe **Pessoa**. O atributo deve ser indexado em um índice e na primeira posição da chave de indexação.
- indexar o atributo **email** em um índice que agiliza consultas que necessitam de ordenação. Ele deve ser indexado em um índice e na primeira posição da chave de indexação.
- indexar o atributo nome no índice que agiliza consultas que necessitam de semelhanças entre palavras. Ele deve ser indexado **em um outro índice** e na primeira posição da chave de indexação.

7.Executar a classe **AppGerarClasses** e em seguida a classe **AppPersistir**.

Fase 3: Avaliação da versão 2 do framework Obinject

Curso:
Período/Ano de início:

Data: **Hora de Início:** **Hora de Término:**

Questionário de Avaliação de Usabilidade - Questões de 1 a 10 Traduzido do original *"SUS - A quick and dirty usability scale"*, John Brooke, 1996

Nas questões de 1 a 10, selecione uma das opções na escala de 1 a 5, sendo 1 discordo totalmente e 5 concordo totalmente.

*** Preenchimento obrigatório**

1.Acha que gostaria de usar esse sistema com frequência. *

Discordo totalmente 1 2 3 4 5 Concordo totalmente

2.Achei o sistema desnecessariamente complexo. *

Discordo totalmente 1 2 3 4 5 Concordo totalmente

3.Achei o sistema fácil de usar. *

Discordo totalmente 1 2 3 4 5 Concordo totalmente

4.Eu acho que precisaria do apoio de um técnico para conseguir usar esse sistema. *

Discordo totalmente 1 2 3 4 5 Concordo totalmente

5.Achei que as várias funções desse sistema estavam bem integradas. *

Discordo totalmente 1 2 3 4 5 Concordo totalmente

6.Achei que tinha muitas inconsistências nesse sistema. *

Discordo totalmente 1 2 3 4 5 Concordo totalmente

7.Acho que a maioria das pessoas aprenderia a usar esse sistema facilmente. *

Discordo totalmente 1 2 3 4 5 Concordo totalmente

8.Achei o sistema muito complicado de se usar. *

Discordo totalmente 1 2 3 4 5 Concordo totalmente

9.Eu me senti muito confiante usando o sistema. *

Discordo totalmente 1 2 3 4 5 Concordo totalmente

10.Eu precisei aprender várias coisas antes de continuar usando o sistema. *

Discordo totalmente 1 2 3 4 5 Concordo totalmente

11.Conhece a linguagem de programação Java a quanto tempo? *

( ) 0 meses - 1 ano
( ) 1 ano - 2 anos
( ) 2 anos - 3 anos
( ) 3 anos - 5 anos
( ) Acima de 5 anos

12.Com base na realização da fase de execução, preencha a tabela abaixo com o total de anotações (persistência e indexação) aplicadas para cada classe: *

Obs: As anotações com o mesmo nome podem ser contabilizadas.

Classe	**Total de Anotações**
Album	
Foto	
Local	
Pessoa	

13.Após a execução do classe AppGerarClasses, quantas classes foram geradas? *

Obs: Não contabilizar as classes: AppGerarClasses, AppPersistir, Album, Foto, Local e Pessoa.

Resposta:

14.Quais foram as dificuldades encontradas?

Resposta:

Apêndice E

Dados obtidos no experimento 1

Neste Apêndice são apresentados os dados obtidos no experimento 1.

A tabela E.1 mostra o tempo gasto por cada participante em cada fase e o tempo total na realização do experimento 1 utilizando a versão 1 do *framework Obinject*.

Tabela E.1: Experimento 1: Tempo de cada participante utilizando a versão 1 do *framework Obinject*.

	Participante													**Média**
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	
Treinamento	480	1200	960	900	780	840	660	540	540	360	420	540	900	702
Execução	1860	1620	2340	1740	2340	1440	1920	1020	1680	1020	1680	1800	1620	1698
Avaliação	600	420	240	420	240	720	420	840	660	540	360	240	840	503
Total	2940	3240	3540	3060	3360	3000	3000	2400	2880	1920	2460	2580	3360	2903

A tabela E.2 mostra o tempo gasto por cada participante em cada fase e o tempo total na realização do experimento 1 utilizando a versão 2 do *framework Obinject*.

Tabela E.2: Experimento 1: Tempo de cada participante utilizando a versão 2 do *framework Obinject*.

	Participante												**Média**
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	
Treinamento	720	660	720	720	840	720	600	1320	1020	1800	1620	900	970
Execução	1800	1860	2040	2220	2220	2820	2640	1740	1980	2220	2400	1980	2160
Avaliação	540	720	540	600	360	240	360	420	420	240	360	720	460
Total	3060	3240	3300	3540	3420	3780	3600	3480	3420	4260	4380	3600	3590

As tabelas E.3 e E.4 exibem as respostas dos Questionário de Avaliação de Usabilidade e as pontuações do SUS relativas as versões 1 e 2 do *framework Obinject*. Para cada versão do *framework*, os participantes foram divididos em 2 grupos de acordo com o tempo de conhecimento da linguagem de programação Java.

Tabela E.3: Experimento 1: Respostas do Questionário de Avaliação de Usabilidade e as pontuações do SUS utilizando a versão 1 do *framework Obinject*.

			Questionário SUS										Pontuação SUS	Média
			1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**		
Participante	**Grupo de 0 a 2 anos**	**1**	4	1	5	1	5	1	5	1	5	3	92,5	**74,642**
		2	3	4	3	3	4	1	4	3	2	4	52,5	
		3	3	2	3	2	4	2	4	2	2	1	67,5	
		4	4	1	5	2	4	1	5	1	4	1	90	
		5	5	2	4	4	5	1	5	1	5	4	80	
		6	4	2	3	3	3	2	4	2	2	1	65	
		7	4	4	4	2	4	2	4	1	4	1	75	
	Grupo de 2 anos ou mais	**8**	3	1	5	1	4	4	4	2	3	1	75	**79,166**
		9	1	1	3	2	2	1	3	1	4	2	65	
		10	5	1	4	2	5	1	5	1	4	1	92,5	
		11	4	2	3	4	5	2	4	4	3	3	60	
		12	5	1	5	4	5	2	5	1	4	1	87,5	
		13	4	1	5	1	5	1	4	1	5	1	95	

Tabela E.4: Experimento 1: Respostas do Questionário de Avaliação de Usabilidade e as pontuações do SUS utilizando a versão 2 do *framework Obinject*.

			Questionário SUS										Pontuação SUS	Média
			1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**		
Participante	**Grupo de 0 a 2 anos**	**1**	5	3	4	3	5	2	2	1	4	1	75	**61,666**
		2	1	5	2	2	2	2	1	4	1	2	30	
		4	4	3	2	4	5	2	1	4	4	4	47,5	
		8	1	1	5	1	5	1	5	1	3	1	85	
		9	3	1	5	3	1	2	4	1	4	1	72,5	
		12	4	3	3	2	2	1	3	3	3	2	60	
	Grupo de 2 anos ou mais	**2**	4	1	4	3	4	2	4	2	4	2	75	**67,916**
		5	2	3	4	2	5	1	2	1	2	5	57,5	
		6	4	1	4	3	4	1	4	1	4	1	82,5	
		7	4	3	4	1	4	1	4	1	4	3	77,5	
		10	2	5	3	2	5	3	1	5	2	1	42,5	
		11	4	3	4	1	3	3	5	2	3	1	72,5	

As tabelas E.5 e E.6 apresentam os resultados das tarefas da fase de execução: as anotações aplicadas nas classes codificadas pelos participantes e as classes empacotadoras geradas, a partir das anotações, utilizando as versões 1 e 2 do *framework Obinject*.

As classes Album, Foto, Local e Pessoa serão representadas nas tabelas E.5 e E.6 pelas letras A, F, L e P respectivamente.

Tabela E.5: Experimento 1: Resultados das anotações aplicadas e das classes geradas pelos participantes utilizando a versão 1 do *framework Obinject* de acordo com os roteiro.

			Anotações corretas nas classes				Anotações erradas nas classes				Classes geradas corretas	Classes geradas erradas
			A	F	L	P	A	F	L	P		
Participante	Grupo de 0 a 2 anos	**1**	3	8	9	4	-	-	-	-	18	-
		2	3	8	9	4	-	-	-	-	18	-
		3	3	6	9	4	-	2	-	-	17	1
		4	3	8	9	4	-	-	-	-	18	-
		5	3	8	9	3	-	-	-	1	17	1
		6	3	8	9	4	-	-	-	-	18	-
		7	3	6	7	3	-	2	2	1	15	3
	Grupo de 2 anos ou mais	**8**	3	8	9	4	-	-	-	-	18	
		9	3	8	9	4	-	-	-	-	18	-
		10	3	8	9	4	-	-	-	-	18	-
		11	3	8	9	4	-	-	-	-	18	-
		12	3	8	9	4	-	-	-	-	18	-
		13	3	8	9	3	1	-	1	1	17	3

Tabela E.6: Experimento 1: Resultados das anotações aplicadas e das classes geradas pelos participantes utilizando a versão 2 do *framework Obinject* de acordo com o roteiro.

			Anotações corretas nas classes				Anotações erradas nas classes				Classes geradas corretas	Classes geradas erradas
			A	F	L	P	A	F	L	P		
Participante	Grupo de 0 a 2 anos	**1**	3	10	9	4	-	-	-	-	19	-
		3	3	7	8	4	-	3	1	-	16	3
		4	3	8	9	4	-	2	-	-	18	1
		8	3	8	9	4	-	2	-	-	17	2
		9	3	8	8	4	-	2	1	-	17	2
		12	3	8	9	2	-	2	-	2	10	9
	Grupo de 2 anos ou mais	**2**	3	8	9	4	-	2	-	-	18	1
		5	3	8	9	4	-	2	-	-	18	1
		6	3	8	9	4	-	2	-	-	18	1
		7	3	2	3	3	-	8	6	1	11	8
		10	3	8	9	3	-	2	-	1	17	2
		11	3	10	9	4	-	-	-	-	19	-

Anotações