UNIVERSIDADE FEDERAL DE ITAJUBÁ

PROGRAMA DE PÓS GRADUAÇÃO EM
CIÊNCIA E TECNOLOGIA DA COMPUTAÇÃO

Estudo quantitativo de reusabilidade de software a partir dos conceitos de classe, herança, tipos genéricos e CRTP

Lais Reis Vilela

Itajubá, Setembro de 2013

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

Lais Reis Vilela

Estudo quantitativo de reusabilidade de software a partir dos conceitos de classe, herança, tipos genéricos e CRTP

Dissertação 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

Setembro de 2013
Itajubá - MG

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

V699e

Vilela, Lais Reis
Estudo quantitativo de reusabilidade de software a partir dos conceitos de classe, herança, tipos genéricos e CRTP / Lais Reis Vilela. -- Itajubá, (MG) : [s.n.], 2013.
68 p. : il.

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

1. Estudo qunatitativo. 2. Metadados de códigos fontes. 3. Reusabilidade de software. I. Seraphim, Enzo, orient. II. Univer_ sidade Federal de Itajubá. III. Título.

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

Lais Reis Vilela

Estudo quantitativo de reusabilidade de software a partir dos conceitos de classe, herança, tipos genéricos e CRTP

Dissertação aprovada por banca examinadora em 09 de setembro de 2013, conferindo ao autor o título de Mestre em Ciência e Tecnologia da Computação.

Banca Examinadora:
Dr. Enzo Seraphim (Orientador)
Dr. Rodrigo Duarte Seabra
Dr. Marcelo de Almeida Maia

Setembro de 2013
Itajubá - MG

"O que não se conhece não se pode controlar.
O que não se controla não se pode medir.
O que não se mede não se pode gerir e
o que não se gere não se pode melhorar."

James Harrington

A meu pai, Lourenço
e a minha mãe, Ana.

Agradecimentos

Agradeço a Deus, que me guiou nessa caminhada, os talentos recebidos e vitórias conquistadas. Aos meus pais, irmãos, avós e familiares, pelo incentivo, dedicação, amor, confiança, paciência e principalmente por sempre estarem presentes em minha vida, mesmo nos momentos que estive distante.

A meu orientador, Prof. Dr. Enzo Seraphim por todo o aprendizado, esforço, dedicação e apoio empreendido neste trabalho. Principalmente por não me deixar desistir, por acreditar em mim e por toda paciência nos momentos que estive ausente.

Ao Afrânio, pela compreensão, amor, paciência e apoio. Aos meus amigos, que sempre me deram coragem, pelo simples gesto de amizade. Especialmente ao Luiz Olmes, amigo de graduação e pós-graduação, pelas conversas infinitas, pelo apoio, conselhos e incentivos durante a realização deste trabalho.

Agradeço também aos colegas de trabalho da B2ML e do Peixe Urbano pelo apoio e incentivo. Obrigada por toda a oportunidade de trabalho e pela compreensão da minha ausência no ambiente de trabalho para que eu pudesse finalizar este projeto.

À CAPES, pelo auxílio financeiro, à universidade e aos colegas do programa de Mestrado em Ciência e Tecnologia da Computação. Aos demais docentes do curso, por me mostrarem novos caminhos e repartirem conhecimentos. E a Prof. Thatyana Seraphim por toda ajuda nos momentos finais deste trabalho.

Emfim, meu agradecimento a todos cujo apoio, torcida e incentivo foram fundamentais para vencer esta etapa da vida.

Sumário

Lista de Figuras

Lista de Tabelas

Abreviaturas e Siglas

CRTP *Curiously Recurring Template Pattern*

CRGP *Curiously Recurring Generic Pattern*

DAO *Data Access Object*

DIT *Depth of Inheritance Tree*

GUI *Graphical User Interface*

MVC *Model View Control*

NOC *Number of Children*

OOD *Object-Oriented Design*

POO Programação Orientada a Objetos

UML *Unified Modeling Language*

RESUMO

A reutilização está presente em várias técnicas da Programação Orientada a Objeto, como herança, tipos genéricos e padrões. No entanto, poucos estudos quantitativos foram realizados para verificar a utilização destas técnicas. Para avaliar essas técnicas orientadas a objetos, são necessárias medidas e métricas que quantificam suas estruturas. Este trabalho apresenta um estudo quantitativo de tipos, herança, programação genérica e *Curiously Recurring Template Pattern* (CRTP) em projetos de *software Java* contendo 236.676 arquivos de código fonte aberto. Para realizar este estudo foi desenvolvida uma ferramenta que extrai metadados de arquivos de projeto Java e armazena-os em uma base de dados relacional. Esta base de dados contém informações sobre código fonte orientado a objetos, tais como tipos, heranças, parâmetros genéricos, restrições de parâmetros genéricos e invocações de tipos genéricos. A partir desta base foi possível realizar medições e aplicar as métricas *Depth of Inheritance Tree* (DIT) e *Number of Children* (NOC). Os resultados mostraram que a herança é muita utilizada. Já tipos genéricos e CRTP mostraram serem poucos utilizados.

Abstract

Reuse is present in various techniques of Object-Oriented Programming, as an inheritance, generic types and patterns. However, few quantitative studies have been conducted to verify these techniques. To evaluate these object-oriented techniques are necessary measures and metrics that quantify their structures. This paper presents a study quantitative of types, inheritance, generic programming and CRTP in projects Java software containing 236676 files open source. To conduct this study was developing a tool that extracts metadata from Java project files and stores them in a relational database. This database contains information on object-oriented source code such as types, inheritance, generic parameters, bounds and binds. On this basis it was possible to perform measurements and apply metrics *Depth of Inheritance Tree* (DIT) and *Number of Children* (NOC). The results showed that the inheritance is much used. Already generic types and CRTP showed are few used.

Capítulo 1
Introdução

Introduzida no início da década de 1960 por Kristen Nygaard e Ole-Johan Dahl por meio da linguagem Simula 67 (DAHL, 1968; NYGAARD; DAHL, 1978), a orientação a objetos é um paradigma de programação bastante disseminado dentre as técnicas de programação. Porém, a primeira linguagem orientada a objetos que se destacou foi o Smalltalk (GOLDBERG; ROBSON, 1983) desenvolvida pela Xerox PARC nos anos 70. Essa linguagem introduziu muitos dos conceitos do paradigma orientado a objetos, sendo seus desenvolvedores os primeiros a utilizarem o termo Programação Orientada a Objetos (POO).

Booch et al. (2007) define a POO como um método de implementação em que os programas são organizados como coleção de objetos, onde cada um deles representa uma instância de uma classe. A herança é um dos conceitos fundamentais da POO. É um processo de derivar uma classe nova a partir de classes já existentes. Segundo Booch et al. (2007), a programação sem herança não pode ser considerada uma programação orientada a objetos. Outra técnica da POO são os tipos genéricos. Esta técnica permite o desenvolvimento de modelos para classes que podem ser modificados por meio de parâmetros em tempo de compilação.

Através do uso da herança é possível obter reutilização de *software*. A reutilização é um princípio da engenharia de *software* onde um *software* deve incorporar componentes pré-construídos. A reutilização é capaz de produzir um grande impacto na produtividade e na qualidade do *software* produzido. Para Booch et al. (2007), qualquer artefato de desenvolvimento de *software* pode ser reutilizado. A reutilização permite uma economia de recursos que seriam necessários para reinventar algum problema anteriormente resolvido.

A reutilização de *software* também é possível por meio do uso das técnicas de tipos genéricos e padrões de projeto. Padrões de projeto descrevem soluções em uma forma de escrita para problemas

recorrentes em um determinado contexto. Desde a introdução do primeiro catálogo de padrões em Gamma et al. (1995), padrões foram rapidamente aceitos pela engenharia de *software*.

Um outro padrão chamado CRTP, descrito por Coplien (1995), utiliza os recursos de herança e programação genérica para criar uma classe derivada que herde de uma instanciação de uma classe base usando ela própria como argumento genérico.

A medição pode quantificar práticas orientadas a objetos. Medição tornou-se uma área de pesquisa popular. O primeiro passo no processo de medição é derivar as métricas de *software*. Métricas são ferramentas poderosas de apoio no desenvolvimento de *software* e manutenção. Elas são usadas para avaliar a qualidade do *software*, estimar a complexidade, custo e esforço, controlar e melhorar os processos (PRESSMAN, 2006).

1.1 Motivação

Nas últimas décadas, a necessidade de um *software* confiável fez com que a engenharia de *software* tornasse uma importante área. No entanto, é muito difícil provar que as várias abordagens da engenharia de *software* geram benefícios, devido à falta de dados empíricos.

O *software* desenvolvido utilizando orientação a objetos é apresentado como mais confiável, fácil de manter e reutilizar. Muitas destas afirmações se baseiam em estudos que utilizaram métricas em orientação a objetos como os trabalhos de Chidamber e Kemerer (1994), Basili, Briand e Melo (1995) e Briand et al. (1998).

Como, atualmente, a POO é uma técnica amplamente aceita, a utilização de seus recursos precisam ser verificadas. Poucos estudos quantitativos foram realizados para verificar o modo como as características de *softwares* orientados a objetos estão sendo utilizadas. Para avaliar os atributos das estruturas orientadas a objetos são necessárias medidas que quantificam essas estruturas.

1.2 Objetivo

O objetivo deste trabalho é apresentar uma análise quantitativa envolvendo os aspectos de reusabilidade: classes, herança, tipos genéricos e CRTP. Os objetivos específicos são:

- Desenvolver uma ferramenta que extrai metadados de arquivos de projeto Java armazenando-os em banco de dados relacional.

- Criar um modelo de entidade para metadados de códigos fontes orientados a objetos que encapsulam tipos, heranças, parâmetros genéricos, restrições de parâmetros genéricos e invocações de tipos genéricos.
- Conceber um algoritmo capaz de resolver ambiguidades sintáticas dos nomes das classes.
- Desenvolver medições para análise de técnicas orientadas a objeto.

Com esta análise espera-se conhecer se estes aspectos são muitos utilizados e de que modo estão sendo utilizados.

1.3 Organização do Trabalho

Este documento está organizado da seguinte maneira:

- *Capítulo 2 - Revisão Bibliográfica*: discute sobre metadados, reusabilidade, programação orientada a objetos e métricas de *software*.
- *Capítulo 3 - Metadados de Programação Orientada a Objetos (MetaPOO)*: descreve o *software* desenvolvido, mostrando seu funcionamento e detalhes de implementação.
- *Capítulo 4 - Análise Quantitativa*: apresenta a análise realizada com os dados obtidos com a ferramenta desenvolvida, discutindo a metodologia e resultados.
- *Capítulo 5 - Conclusão*: apresenta as conclusões do trabalho e as propostas para trabalhos futuros.
- *Apêndice A - Consultas*: contém as consultas realizadas no banco de dados utilizadas para gerar os gráficos obtidos no Capítulo 4.

CAPÍTULO
2
Revisão Bibliográfica

Este capítulo apresenta uma revisão dos principais temas envolvidos neste trabalho. A Seção 2.1 discute sobre as definições de metadados e os aspectos relacionados. A Seção 2.2 apresenta as principais características para reusabilidade de *software*. A Seção 2.3 introduz os fundamentos de programação orientada a objetos utilizados neste trabalho. Serão apresentados exemplos na linguagem de programação Java. A Seção 2.4 introduz as propriedades de medições e métricas em orientação a objetos, apresentando duas métricas que serão utilizadas. Finalmente, a Seção 2.5 apresenta trabalhos relacionados.

2.1 Metadados

Metadados são dados que descrevem as características de entidades portadoras de informação e ajudam na identificação, descoberta, avaliação e gestão das entidades descritas (DAMASEVICIUS; STUIKYS, 2008).

Uma outra definição comumente usada é que metadados são dados que descrevem outros dados. De maneira geral, são utilizados para facilitar o entendimento, o uso e o gerenciamento de dados em um contexto de uso. Por exemplo, no contexto de uma biblioteca, onde os dados podem ser representados pelo conteúdo dos livros, os metadados incluem uma descrição do conteúdo, o autor, a data de publicação e sua localização física. No contexto de uma câmera, onde os dados podem ser representados pelo conteúdo de imagem fotográfica, os metadados normalmente incluem a data em que a foto foi tirada e detalhes da configuração da câmera. Finalmente, no contexto de projeto de sistema, onde os dados são as classes que encapsulam o negócio, os metadados destas classes incluem informações sobre qual é o pacote em que a classe está localizada, qual é sua superclasse, entre outras

informações.

Bretherton e Singley (1994) distinguem metadados em dois tipos: estrutura e guia. Por exemplo, metadados de estrutura são usados para descrever a estrutura e organização de um item. Metadados de guia são usados na localização de itens específicos e são geralmente descritos com o uso de um conjunto de palavras-chave em uma linguagem natural.

Por outro lado, Press (2004) distingue três tipos de metadados: descritivos, estruturais e administrativos. Os metadados descritivos são as informações usadas para pesquisar e localizar um objeto. Os metadados estruturais descrevem como os componentes dos dados são organizados. Finalmente, os metadados administrativos irão identificar informações que servirão para preservação e controle, permitindo gerenciar desde o acesso a um determinado recurso, até o controle de autoridade e de validade deste recurso.

A tecnologia de metadados surgiu devido às organizações necessitarem conhecer melhor os dados que elas mantêm. Os metadados têm um papel importante na gestão de dados, pois a partir deles, as informações são processadas, atualizadas e consultadas (HUNER; OTTO, 2009).

2.1.1 Metaprogramação

Metaprogramação é a interpretação de um código de programa como dados a serem analisados e transformados por metaprogramas (LOWE; NOGA, 2002). Um metaprograma é um programa de computador capaz de representar e manipular outros programas por meio de uma metalinguagem de programação (SHEARD, 2001).

Exemplos de metaprogramas incluem analisadores, interpretadores, compiladores e geradores de programas, sendo, em comum, manipuladores de dados relacionados às sentenças de linguagens de programação. Um analisador de programa efetua a observação de propriedades, estrutura e ambiente de execução do programa-objeto, de forma a produzir ações ou resultados (SHEARD, 2001).

Dependendo do tempo da aplicação da metaprogramação é possível distinguir entre estática e dinâmica. A metaprogramação estática realiza análises e transformações em tempo de compilação. Por outro lado, a metaprogramação dinâmica age em tempo de execução (LOWE; NOGA, 2002).

O conceito de metalinguagem é definido como sendo qualquer linguagem usada para analisar outro sistema de linguagem. Um sistema de metaprogramação homogêneo é definido como sendo um sistema com a mesma metalinguagem e linguagem-objeto (SHEARD, 2001).

No contexto da metaprogramação, os metadados são as descrições das propriedades ou preocupações de uma camada específica da abstração em um sistema (DAMASEVICIUS; STUIKYS, 2008).

Uma ferramenta proposta para descrever metaprogramas na linguagem Java é o METAJ (OLIVEIRA et al., 2004). Esta ferramenta oferece uma visão orientada a objetos da metaprogramação, através de abstrações que oferecem uma estrutura flexível de descrição de metaprogramas e permite maior legibilidade. Uma vantagem dessa ferramenta é não realizar modificações na estrutura sintática da linguagem Java.

2.2 Reusabilidade de software

Os primeiros conceitos sobre reutilização de *software* são do ano de 1968, quando Doug McIlory apresentou em seu trabalho *"Mass Produced Software Components"* (MCILROY, 1968) a sua motivação por *softwares* que pudessem ser fabricados em massa.

Apesar de os primeiros conceitos sobre reutilização de *software* terem sido idealizados no final da década de 1960, somente nos anos 80 é que receberam atenção da indústria e das universidades; quando a complexidade dos sistemas começou a aumentar e as empresas foram forçadas a procurar por métodos mais eficientes de construção de sistemas.

A reutilização é um princípio da engenharia de *software* que diz que um *software* deve incorporar componentes pré-construídos. Qualquer artefato de desenvolvimento de *software* pode ser reutilizado. Classes são os artefatos primários para reutilização, pois fazem parte da herança e de outras técnicas que permitem a reutilização (BOOCH et al., 2007).

Para Meyer (1997), os benefícios de *softwares* reutilizáveis são:

- **Produtividade:** A construção de *software* é mais rápida com a utilização de componentes prontos;
- **Confiabilidade:** Ao contar com componentes de uma fonte renomada, tem-se a garantia de que os autores aplicaram os cuidados necessários de que aquele componente é confiável, tornando o *software* mais confiável;
- **Eficiência:** Os desenvolvedores de componentes reutilizáveis utilizam os melhores algoritmos e estruturas de dados;

- **Consistência:** A reusabilidade propõe um projeto coerente para a qualidade do *software* produzido.

A vantagem da reutilização de *software* está no potencial que ela exibe para se desenvolver *software* mais rápido, melhor e mais barato. Para que esse potencial seja o maior possível, o reuso não deve se limitar apenas ao código das aplicações, mas também aos requisitos, especificações, projetos, testes, e todos os outros elementos produzidos durante as fases de desenvolvimento (SAMETINGER, 1997).

O surgimento das tecnologias orientadas a objetos constituiu um marco essencial na reutilização de *software*. Isso porque elas conseguiram ampliar esse escopo e permitiram o reaproveitamento de classes de análise, projeto e implementação em diferentes projetos de *software*. Com isso, elas tornaram as antigas bibliotecas de funções obsoletas e proporcionaram o desenvolvimento de bibliotecas de classes, as quais modelam com mais coesão as regras de negócio dos sistemas (MCCLURE, 1997).

Mais tarde, novos passos foram dados com a criação de *frameworks* reutilizáveis, que são estruturas de classes montadas e preparadas para um conjunto específico de problemas. Um *framework* captura as decisões de projeto que são comuns ao domínio de aplicação. Assim, *frameworks* enfatizam reutilização de projetos em relação à reutilização de código, embora um *framework* inclua subclasses concretas que podem ser utilizadas diretamente. Os *frameworks* estão se tornando cada vez mais comuns e importantes e são a maneira pela qual os sistemas orientados a objetos conseguem a maior reutilização.

2.2.1 Padrões de Projeto: Elementos de reusabilidade

Os padrões de projeto surgiram do trabalho de um arquiteto, Christopher Alexander, no final da década de 70. Ele escreveu dois livros nos quais exemplificava o uso e descrevia seu raciocínio para documentar os padrões: *"A Pattern Language: Towns, Buildings, Construction" (Oxford University Press, 1977)* e *"The Timeless Way of Building" (Oxford University Press, 1979)* (METSKER; WAKE, 2006).

Em 1995 foi lançado o livro *"Design Patterns: Elements of Reusable Object-Oriented Software"*(GAMMA et al., 1995), um catálogo com 23 padrões e, desde então, este é um dos livros de padrões mais referenciados.

Os padrões de projeto têm sido reconhecidos por sua grande importância para a criação de sistemas complexos, pois tornam mais fáceis reutilizar projetos e arquiteturas bem-sucedidas. De acordo

Tabela 2.1: Classificação dos vinte e três padrões

Fonte: Gamma et al. (1995)

Padrão	**Finalidade**	**Escopo**	**Característica que pode variar**
Abstract Factory	Criação	Objeto	Famílias de objetos-produto
Builder		Objeto	Como um objeto composto é criado
Factory Method		Classe	Subclasse de objeto que é instanciada
Prototype		Objeto	Classe de objeto que é instanciada
Singleton		Objeto	A única instância de uma classe
Adapter	Estrutural	Classe	Interface para um objeto
Bridge		Objeto	Implementação de um objeto
Composite		Objeto	Estrutura e composição de um objeto
Decorator		Objeto	Responsabilidade de um objeto sem usar subclasses
Façade		Objeto	Interface para um subsistema
Flyweight		Objeto	Custos de armazenamento de objetos
Proxy		Objeto	Como um objeto é acessado
Chain of Responsability	Comporta-	Objeto	Objeto que pode atender a uma solicitação
Command	mental	Objeto	Quando e como uma solicitação é atendida
Interpreter		Classe	Gramática e interpretação de uma linguagem
Iterator		Objeto	Como os elementos de um agregado são acessados
Mediator		Objeto	Como e quais objetos interagem uns com os outros
Memento		Objeto	Que informação privada é armazenada fora de um objeto
Observer		Objeto	Como os objetos dependentes se mantêm atualizados
State		Objeto	Estados de um objeto
Strategy		Objeto	Um algoritmo
Template Method		Classe	Passos de um algoritmo
Visitor		Objeto	Operações que podem ser aplicadas sem mudar sua classe

com Gamma et al. (1995), padrões de projeto são soluções reutilizáveis para problemas recorrentes, que podem ser encontrados durante o desenvolvimento do *software*. Padrões de projeto são ideias que mostram como alcançar um objetivo, podendo ser aplicadas a determinados casos. A Tabela 2.1 apresenta um resumo dos 23 padrões propostos por Gamma et al. (1995).

Os padrões de projeto podem ser classificados por dois critérios. O primeiro critério, chamado **finalidade**, reflete o que um padrão faz e pode ter a finalidade de criação, estrutural ou comportamental. Os padrões de criação se preocupam com o processo de criação. Os padrões estruturais lidam com a composição de classes ou de objetos. Os padrões comportamentais caracterizam as maneiras pelas quais classes ou objetos interagem e distribuem responsabilidades (GAMMA et al., 1995).

O segundo critério, **escopo**, especifica se o padrão se aplica a classes ou objetos. Os padrões para

classes lidam primariamente com os relacionamentos entre classes e suas subclasses. Os padrões para objetos lidam com relacionamento entre objetos que pode ser mudado em tempo de execução e são mais dinâmicos (GAMMA et al., 1995).

A combinação do escopo com a finalidade definem comportamentos comuns entre os padrões de projeto. Os padrões de criação voltados para classes deixam alguma parte da criação de objetos para subclasses, enquanto que os padrões de criação voltados para objetos deixam esse processo para outro objeto. Os padrões estruturais voltados para classes utilizam a herança para compor classes, enquanto que os padrões estruturais voltados para objetos, descrevem maneiras de montar objetos. Os padrões comportamentais voltados para classes, usam a herança para descrever algoritmos, enquanto que os voltados para objetos descrevem como um grupo de objetos coopera para executar uma tarefa que um único objeto não pode executar sozinho (GAMMA et al., 1995).

2.3 Fundamentos de Orientação a Objetos

A orientação a objetos é um paradigma de programação bastante disseminado dentre as técnicas de construção de *software*. Seus principais conceitos foram introduzidos por volta de 1960 com Kristen Nygaard e Ole-Johan Dahl através da linguagem Simula 67 (DAHL, 1968; NYGAARD; DAHL, 1978). Porém, a primeira linguagem orientada a objetos que se destacou foi o Smalltalk (GOLDBERG; ROBSON, 1983) desenvolvida pela Xerox PARC nos anos 70. Essa linguagem introduziu muitos dos conceitos do paradigma orientado a objetos, sendo seus desenvolvedores os primeiros a utilizarem o termo POO.

Durante os anos 90, POO se tornou a principal técnica de programação, influenciada principalmente pela popularidade das interfaces gráficas de usuário (em inglês, *Graphical User Interface* (GUI)), onde as técnicas de orientação a objetos foram extensivamente aplicadas (SEBESTA, 2009). Hoje as linguagens com conceitos de orientação a objetos mais representativas na indústria de *software* são C++, Java e C# (TIOBE, 2013).

As seções seguintes descrevem os conceitos fundamentais da linguagem de programação orientada a objetos Java.

2.3.1 Classe

A unidade principal da linguagem de programação Java é a classe. Uma classe define atributos e funcionalidades (métodos) para um conjunto de objetos. Esses atributos e métodos agregados a um só objeto é denominado encapsulamento. A especificação de uma classe é composta pelo nome da classe que é um identificador que permite referenciá-la posteriormente (ARNOLD; GOSLING; HOLMES, 2006).

O diagrama de classe de uma classe concreta é mostrado na Figura 2.1(a). Uma classe é declarada usando a palavra reservada *class* e atribuindo um nome à classe, conforme mostra a linha 1 do código da Figura 2.1(b). O corpo de uma classe é formado pelos atributos e métodos listados entre as chaves após a declaração. A linha 3 apresenta um exemplo de um atributo (*attribute*) e as linhas 5,6 e 7 definem um método.

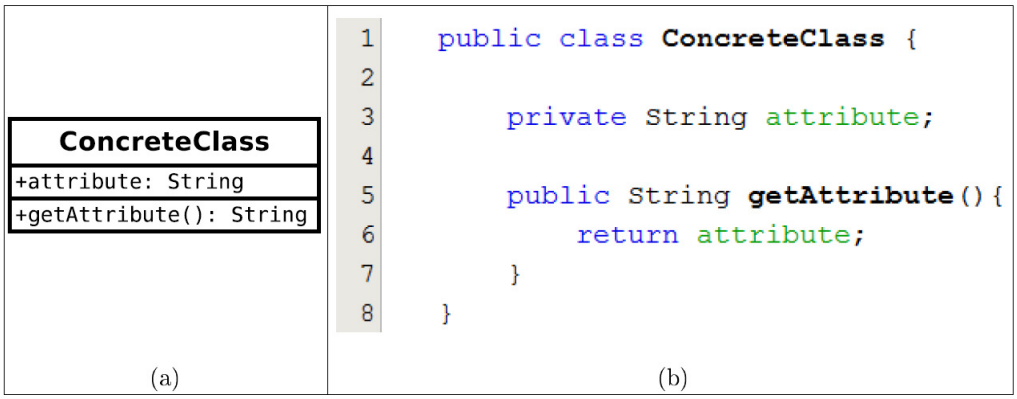

Figura 2.1: Exemplo de uma classe concreta

As classes podem ser classificadas em abstratas e concretas. Segundo Gamma et al. (1995), uma classe abstrata é uma classe cuja finalidade principal é definir uma interface comum para suas subclasses. Uma classe abstrata postergará parte de, ou toda, sua implementação para operações definidas em subclasses e, portanto, uma classe abstrata não pode ser instanciada. Já uma classe concreta possui objetos instanciados a partir dela mesma.

O diagrama de classe de uma classe abstrata é mostrado na Figura 2.2(a). Uma classe abstrata é declarada usando a palavra reservada *abstract* e atribuindo um nome à classe, conforme mostra a linha 1 do código da Figura 2.2(b). O método *abstractMethod* da linha 9 é um método abstrato que deverá ser implementado pelas classes filhas concretas desta classe.

Além da palavra reservada *abstract*, que é utilizada para definir classes abstratas, uma declaração de classe pode possuir modificadores como *public* e *final*. O modificador *public* possibilita que

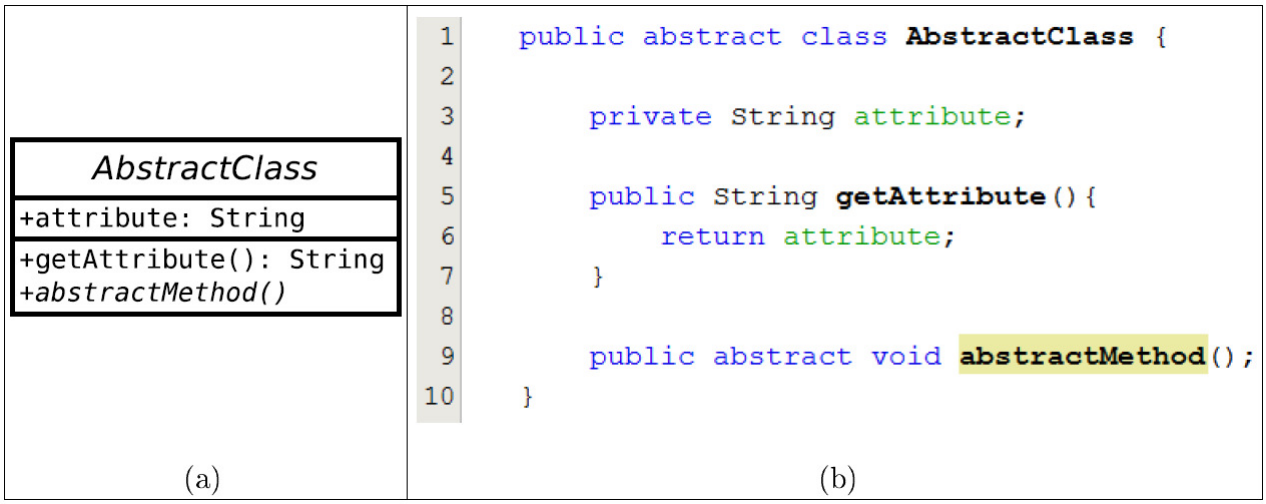

Figura 2.2: Exemplo de uma classe abstrata

a classe seja referenciada de outros pacotes e o modificador *final* não permite que a classe tenha subclasses (GOSLING et al., 2013).

Interface

Uma interface define um conjunto de métodos, constantes ou variáveis que não possuem nenhum valor ou ação preestabelecidos e que podem ser implementados em qualquer classe que seja definida. Interfaces são como as classes abstratas, mas nelas não é possível implementar nenhum método. Uma classe concreta ao herdar uma interface deverá escrever todos os seus métodos (ARNOLD; GOSLING; HOLMES, 2006).

Conforme está apresentado no código da Figura 2.3(b), uma interface é declarada usando a palavra reservada *interface* (linha 1). Essa declaração deve fornecer um nome à interface e relacionar os membros da interface entre chaves ({}). A Figura 2.3(a) apresenta um diagrama de classe de uma interface.

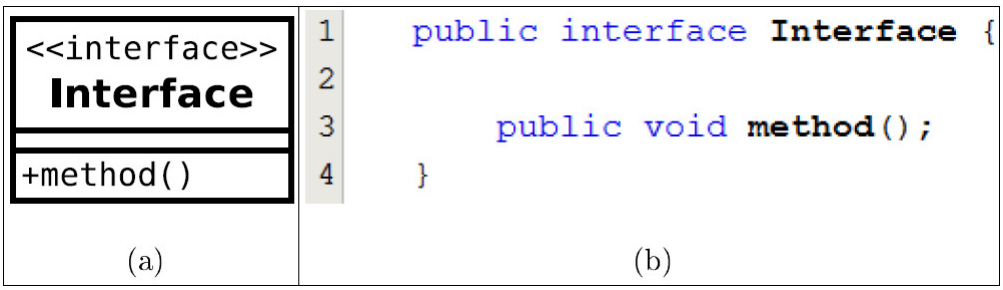

Figura 2.3: Exemplo de uma interface

Enum

Enum é um tipo de classe especial que permite um conjunto de constantes pré-definidas. Todos os tipos *enums* implicitamente estendem a classe *java.lang.Enum* e não podem estender nenhuma outra classe (GOSLING et al., 2013).

O diagrama de classe de um *enum* está ilustrado na Figura 2.4(a). O código da Figura 2.4(b) mostra a implementação de um *enum* em Java. Um *enum* é declarado utilizando a palavra reservada *enum* (linha 1), um identificador que o nomeia e um corpo que nomeia cada um dos valores ou constantes do *enum* (linha 3).

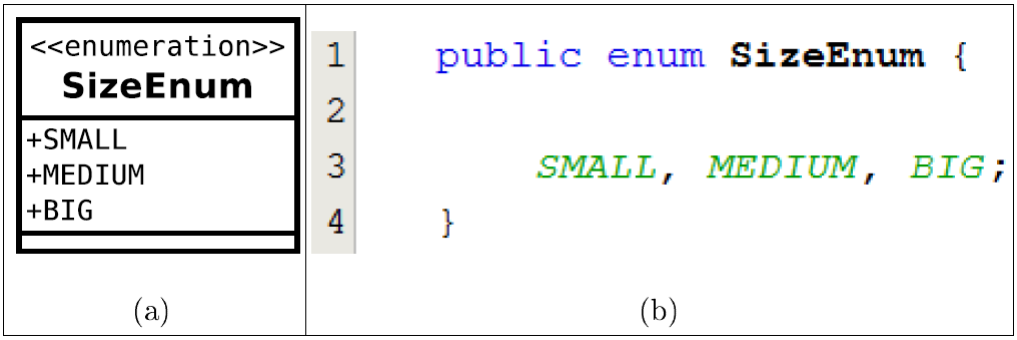

Figura 2.4: Exemplo de um enum

Classes aninhadas

Uma classe aninhada é uma classe cuja declaração ocorre dentro do corpo de outra classe ou interface. As classes aninhadas são classificadas em estáticas e internas. Classes aninhadas estáticas utilizam o modificador *static* e não tem acesso aos membros da instância da classe externa, somente aos membros estáticos (linha 3 do código da Figura 2.5(b)) Já uma classe interna é uma classe aninhada sem a palavra reservada *static* que está associada a uma instância de sua classe externa e tem acesso direto aos métodos e atributos do objeto (linha 6 do código da Figura 2.5) (GOSLING et al., 2013).

A Figura 2.5(a) ilustra o diagrama de classe de classes aninhadas. Um exemplo de uso de classes aninhadas pode ser encontrado na implementação de *Iterators* no pacote de *Collections* do Java. Os *iterators* estão aninhados nas listas e, com isso, o *iterator* acessa o vetor sem que a classe exponha métodos para compartilhar objetos e resultados necessários.

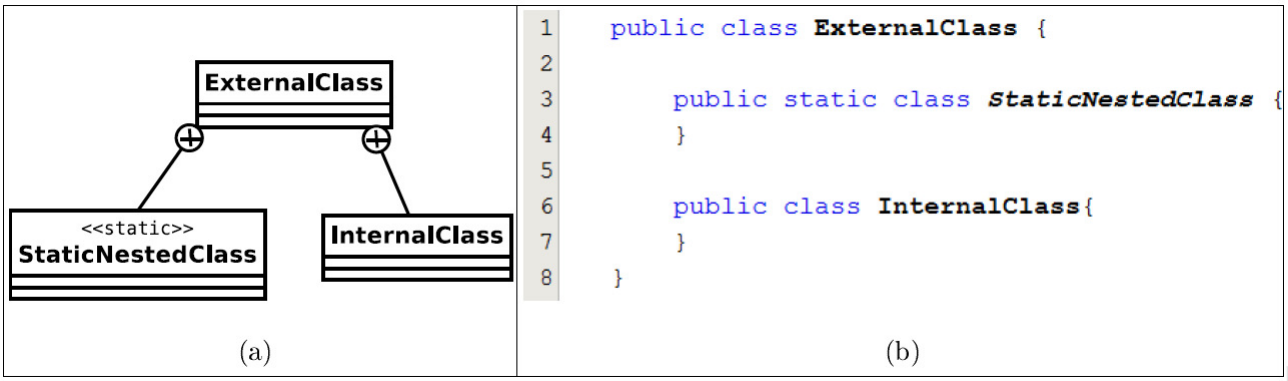

Figura 2.5: Exemplo de classes aninhadas

2.3.2 Herança

Um dos grandes diferenciais da programação orientada a objetos em relação a outros paradigmas de programação está no conceito de herança. A herança permite que novas classes sejam definidas em termos de classes existentes. A classe existente é chamada de superclasse, classe mãe ou classe base e a nova classe é chamada de subclasse, classe filha ou classe derivada. Quando uma classe filha herda de uma classe mãe, ela inclui as definições de todos os dados e operações da superclasse (DEITEL; DEITEL, 2005).

Assim, a herança é um mecanismo que permite que características comuns a diversas classes sejam definidas em uma classe base. A partir da superclasse, outras classes podem ser especificadas. Cada classe derivada apresenta características da classe base e acrescenta a elas o que for definido de particularidade (GAMMA et al., 1995).

Herança de implementação

A herança de implementação é aquela onde todo o código da superclasse é herdado. É um mecanismo para compartilhar código e representação. A forma básica de herança em Java é a extensão simples entre uma superclasse e sua classe filha (ARNOLD; GOSLING; HOLMES, 2006).

O código da Figura 2.6(b) mostra a implementação de uma herança simples entre classes. Para tanto, utiliza-se na definição da classe derivada a palavra reservada *extends* seguida pelo nome da superclasse, conforme mostra a linha 4. O diagrama de classe de herança de implementação com classes está representado na Figura 2.6(a).

Uma classe pode estender exatamente uma superclasse. A linguagem Java não permite utilizar herança múltipla, um mecanismo encontrado em algumas outras linguagens de programação orien-

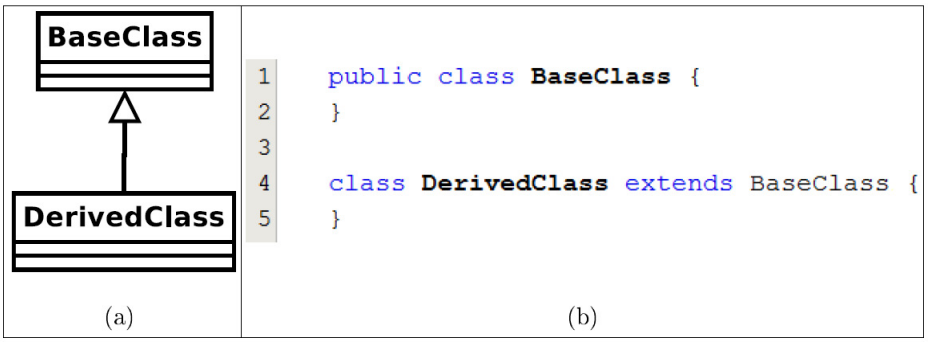

Figura 2.6: Exemplo de herança de implementação com classes

tada a objetos. A herança múltipla ocorre quando a classe é derivada de mais de uma superclasse (GOSLING et al., 2013).

Diferentemente de classes, interfaces podem estender mais de uma interface, como mostra o código da Figura 2.7(b), utilizando a palavra reservada *extends* (linha 8). O diagrama de classe de herança de implementação com interfaces está representado na Figura 2.7(a).

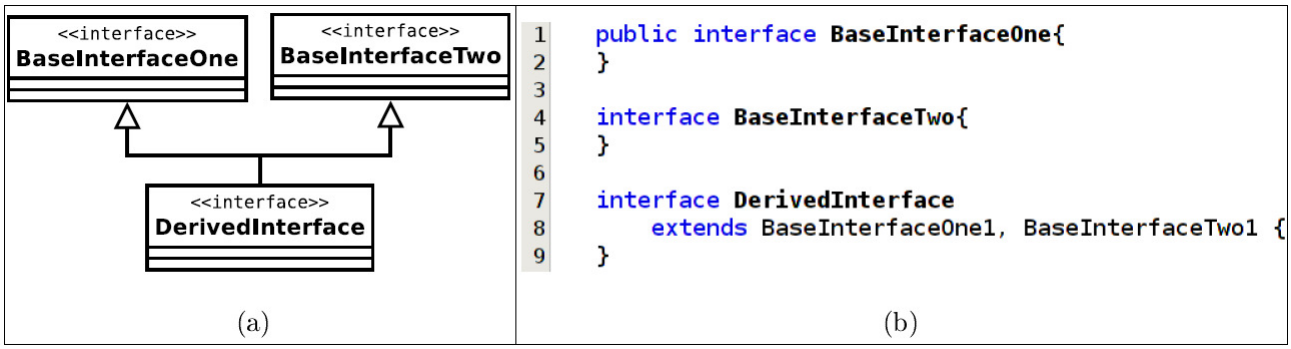

Figura 2.7: Exemplo de herança de implementação com interfaces

Herança de interface

Na herança de interface são herdadas as assinaturas de métodos e propriedades. A palavra reservada *implements* é utilizada para indicar que uma classe implementa uma determinada interface (linha 1 do código da Figura 2.8(b)). Quando uma classe implementa uma interface, garante-se que todas as funcionalidades especificadas pela interface serão oferecidas pela classe. Uma classe pode implementar várias interfaces e não é permitido para uma interface implementar outra interface (ARNOLD; GOSLING; HOLMES, 2006). O diagrama de classe de herança de interface está ilustrado na Figura 2.8(a).

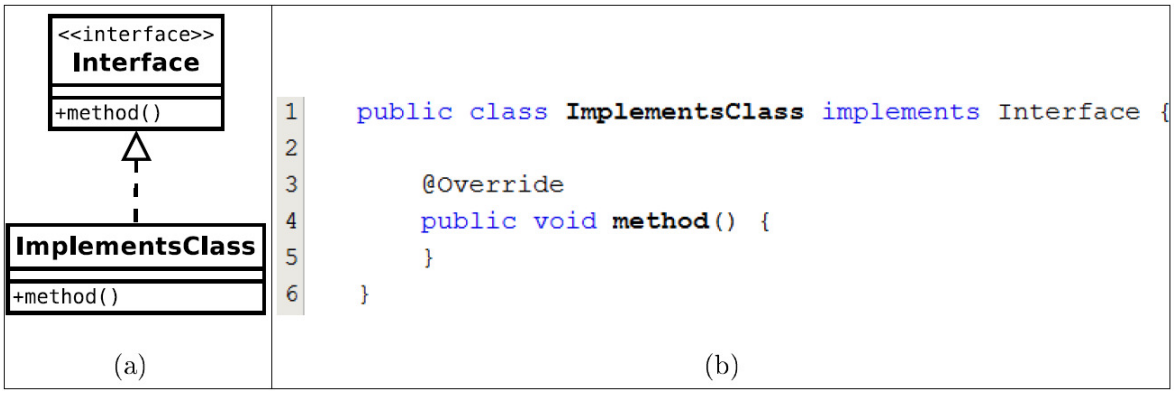

Figura 2.8: Exemlo de herança de interface

2.3.3 Tipos Genéricos

Essa técnica, também conhecida como tipos parametrizados, permite definir um tipo sem especificar todos os outros tipos que ele usa. Os tipos não especificados são fornecidos como parâmetros no ponto de utilização (GAMMA et al., 1995).

Classes e interfaces genéricas permitem a utilização de parâmetros de tipo, capazes de representar diversos tipos, com segurança de tipo em tempo de compilação. Com a programação genérica, é possível escrever programas que resolvam uma classe de problemas de uma só vez, sendo uma técnica importante na reutilização (DEITEL; DEITEL, 2005).

Para declarar uma classe genérica é necessário definir uma variável conhecida como parâmetro genérico (ou parâmetro de tipo) na declaração da classe genérica. A declaração de tipo genérico pode conter vários parâmetros de tipos, separados por vírgula. Estes parâmetros genéricos são substituídos por argumentos de tipo quando o tipo genérico é instanciado (ou declarado) (LANGER, 2013).

Parâmetros genéricos podem ser declarados com restrições (*bounds*). As restrições limitam o conjunto de tipos que podem ser utilizados como argumentos de tipo e podem ser restringidos por tipos ou parâmetros genéricos. Para informar que um parâmetro de tipo seja um subtipo de alguma classe utiliza-se a palavra reservada *extends* na declaração do tipo genérico (LANGER, 2013).

Para criar um objeto da classe genérica é necessário informar ao compilador que tipo específico será utilizado para substituir o parâmetro. Este processo é conhecido como invocação de um tipo genérico (*bind*) (ARNOLD; GOSLING; HOLMES, 2006).

O código da Figura 2.9(b) mostra a implementação de uma classe genérica, *GenericClass*, que possui três parâmetros genéricos (T, U e V). Já a Figura 2.9(a) ilustra o diagrama de classe de uma classe genérica.

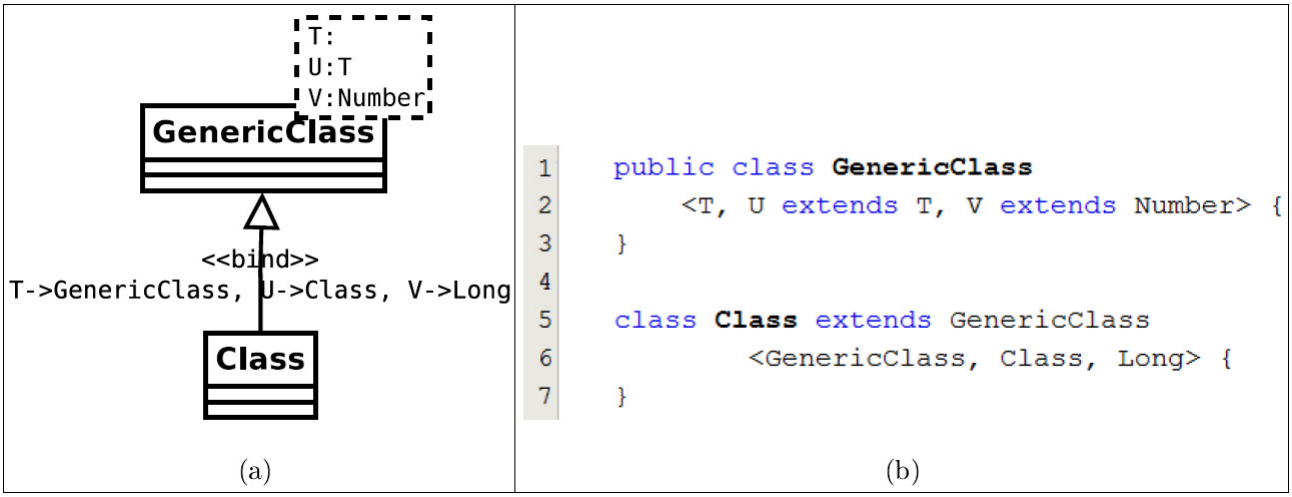

Figura 2.9: Exemplo de uma classe genérica

O parâmetro genérico T definido na linha 2 do código da Figura 2.9(b) é um parâmetro não restringido. Já o parâmetro U é um parâmetro genérico restringido pelo parâmetro T e o parâmetro V é restringido pelo tipo *Number*.

A linha 6 do código da Figura 2.9(b) mostra a invocação do tipo *GenericClass* definindo o parâmetro T como um tipo *GenericClass*, o parâmetro U como um tipo *Class* e o parâmetro V como um tipo *Long*.

2.3.4 Curiously Recurring Template Pattern (CRTP)

O CRTP, descrito por Coplien (1995), utiliza os recursos de herança e programação genérica para criar uma classe derivada que herde de uma instanciação de uma classe base usando ela própria como argumento genérico. Dessa forma, pode-se, em tempo de compilação, especializar a classe base usando a subclasse.

Em Eckel (2005), o autor propõe o termo *Curiously Recurring Generic Pattern* (CRGP) como análogo a CRTP para a linguagem de programação Java devido o termo *generic* ser utilizado na linguagem ao invés do termo *template*, mas os aspectos do padrão permanecem os mesmos.

A Figura 2.10(a) apresenta um exemplo de modelagem *Unified Modeling Language* (UML) do uso de CRTP. *BaseClass* é uma classe genérica cujo parâmetro é T. *BaseClass* declara um método chamado *getThis*, responsável por retornar a instância do objeto T. Por sua vez, *DerivedClass* deriva de *BaseClass* e passa a si própria como argumento genérico através do estereótipo *bind*. Este argumento é utilizado por *BaseClass* na assinatura do método *getThis*.

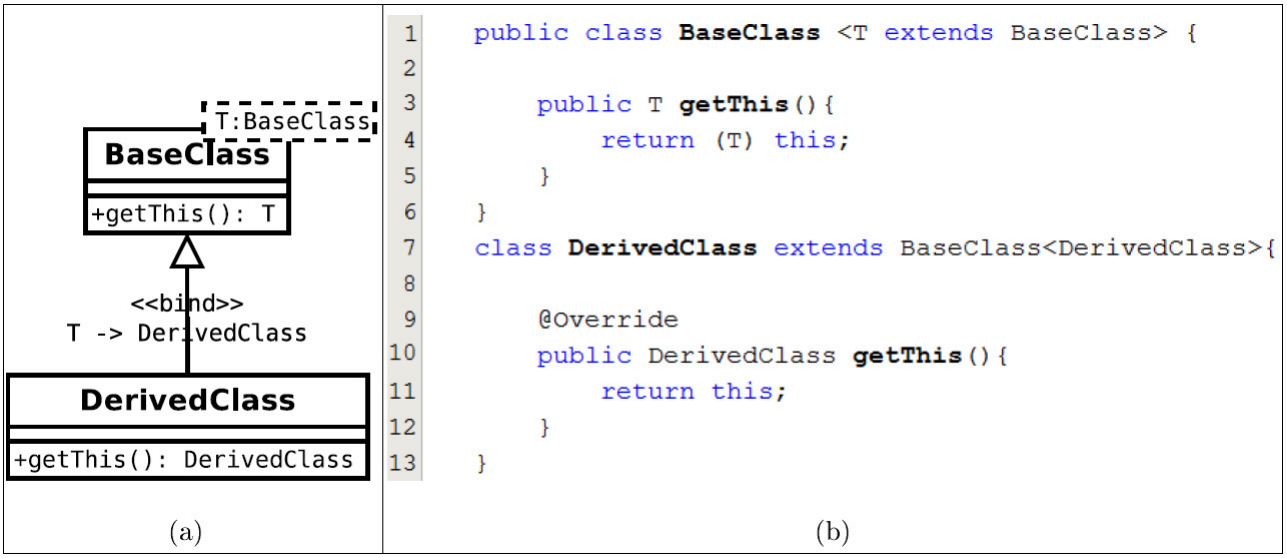

Figura 2.10: Exemplo de CRTP

O código da Figura 2.10(b) mostra a implementação correspondente à Figura 2.10(a) em Java. Na implementação de *DerivedClass*, devido ao CRTP, a assinatura do parâmetro do método *getThis* foi modificada, recebendo o tipo de *DerivedClass* no lugar do argumento T de *BaseClass*.

2.4 Métricas em Orientação a Objetos

As métricas originaram-se da execução prática de avaliação para quantificar indicadores sobre o processo de desenvolvimento de um sistema, sendo adotados a partir de 1970. Uma medida fornece uma indicação quantitativa da extensão, quantidade, dimensão, capacidade ou tamanho de algum atributo de um produto ou processo (PRESSMAN, 2006).

Métricas são ferramentas poderosas de apoio no desenvolvimento de *software* e manutenção. Elas são usadas para avaliar a qualidade do *software*, para estimar a complexidade, custo e esforço, controlar e melhorar os processos.

O primeiro passo no processo de medição é derivar as medidas e métricas de *software* que são adequadas para a representação do *software* que está sendo considerado. A seguir, os dados necessários para derivar as métricas formuladas são coletados. Uma vez calculadas, as métricas adequadas são analisadas com base em diretrizes preestabelecidas. Os resultados da análise são interpretados e levam a modificações dos modelos quando necessário (PRESSMAN, 2006).

A POO requer uma abordagem diferente na implementação de métricas de *software* devido às diferenças deste paradigma. Existem várias propostas para métricas orientadas a objetos que levam

em consideração as características básicas e interações de um sistema orientado a objetos como classe, encapsulamento e herança.

Um dos conjuntos de métricas de *software* orientadas a objetos mais amplamente referenciado foi proposto por Chidamber e Kemerer (1994). Frequentemente conhecido como conjunto de métricas CK, os autores propuseram seis métricas baseadas em classes: (1) Métodos ponderados por classe, (2) Profundidade da árvore de herança, (3) Número de filhos (4) Acoplamento entre classes de objetos, (5) Resposta para uma classe e (6) Falta de coesão em métodos. Serão apresentadas duas métricas utilizadas neste trabalho: Número de filhos (NOC) e Profundidade da árvore de herança (DIT).

A métrica NOC foi introduzida por Chidamber e Kemerer (1994) e define o número de subclasses imediatamente subordinadas a uma classe na hierarquia. Essa métrica mede quantas subclasses vão herdar métodos da classe base. Quanto maior o número de NOC, maior a reutilização. Por outro lado, à medida que aumenta o NOC, o esforço necessário para testar e manter a classe também aumenta, visto que qualquer mudança pode afetar suas subclasses.

A métrica DIT, também introduzida por Chidamber e Kemerer (1994), mede a profundidade de herança de uma classe. Nos casos envolvendo herança múltipla, a DIT será a profundidade máxima do nó à raiz da árvore. Essa métrica é uma medida de quantas classes ancestrais podem afetar essa classe. Quanto maior a profundidade de uma classe na hierarquia, maior o número de métodos que está herdando, tornando-a mais complexa para prever seu comportamento. Por outro lado, valores altos implicam que muitos métodos podem ser reutilizados.

A Figura 2.11 ilustra um exemplo das métricas DIT e NOC. A classe A possui dois filhos, B e C, e, portanto, seu NOC é dois. A classe D possui um DIT de dois porque seus ancestrais são B e A.

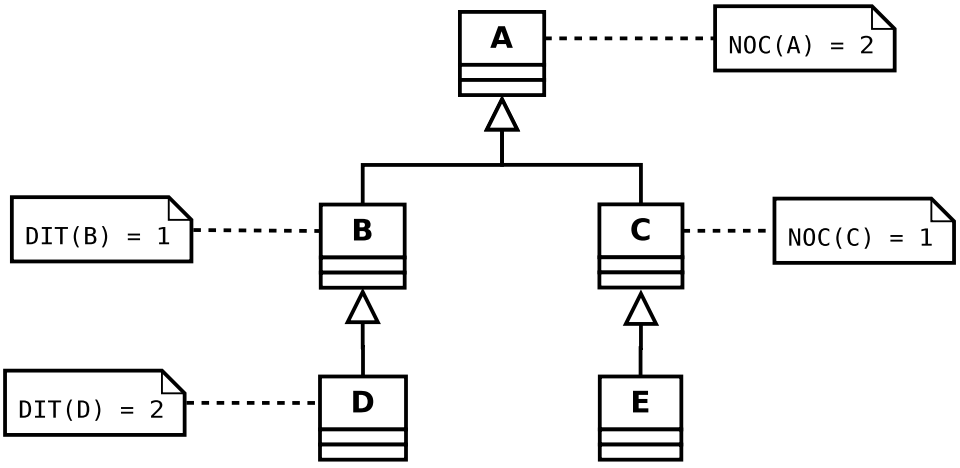

Figura 2.11: Métricas DIT e NOC

A medição permite obter entendimento do processo e projeto, sendo um mecanismo para ava-

liação objetiva e pode ser aplicada com o objetivo de melhorar o processo de *software* de forma contínua. Uma das razões pelas quais é utilizada a medição é para obter entendimento dos recursos e para estabelecer referências para comparação com futuras avaliações (PRESSMAN, 2006).

2.5 Trabalhos similares

Existem vários estudos realizados que abrangem métricas na orientação a objetos, como os trabalhos de Basili, Briand e Melo (1995), Briand et al. (1998), Huston (2001) e Am, Satwinder e S (2009). Estes trabalhos apresentam estudos sobre as métricas propostas por Chidamber e Kemerer (1994). Porém, poucos estudos quantitativos foram encontrados para verificar como as técnicas de reusabilidade de *software* estão sendo utilizadas. Foram encontrados três trabalhos correlatos.

O trabalho de Bieman e Zhao (1995) apresenta um estudo quantitativo em *software* C++. Neste estudo foram utilizados 19 sistemas com 2.744 classes. Somente a técnica de herança foi analisada e o estudo chegou a conclusão de um uso limitado da herança nos *softwares* analisados.

O trabalho de Hahsler (2003) apresenta um estudo quantitativo da aplicação de padrões de projeto em *software* Java. Este trabalho utilizou 1.319 projetos de código fonte aberto. Foram analisados os vinte e três padrões propostos por Gamma et al. (1995) e o estudo apresenta a limitação de não analisar o código fonte por meio de métricas e medições orientadas a objetos. Através deste estudo, os autores concluíram que são poucos os desenvolvedores que utilizam padrões.

O trabalho de Dagenais e Hendren (2008) apresenta uma ferramenta para análise estática de programas com a linguagem Java. Neste trabalho, é apresentada uma função para resolver as ambiguidades sintáticas, porém esta função não implementa todas as etapas que o compilador Java utiliza para resolver o nome completo de um tipo.

Para realizar a análise quantitativa, os trabalhos citados se basearam em poucos projetos e poucas técnicas foram analisadas. O trabalho proposto difere desses estudos por apresentar um estudo em uma base de dados com mais de 300.000 classes onde são analisados quatro aspectos.

2.6 Considerações Finais

Este capítulo apresentou os conceitos que constituem o embasamento teórico deste trabalho. A Seção 2.1 apresentou os conceitos de metadados que será o modelo extraído e armazenado proposto no

Capítulo 3. A Seção 2.2 apresentou a teoria envolvendo reusabilidade de *software*. Estes conceitos serão utilizados para definir as técnicas que serão medidas no estudo quantitativo proposto no Capítulo 4. Já a Seção 2.3 introduziu os conceitos fundamentais da programação orientada a objetos. Estes conceitos serão utilizados na implementação do programa proposto no Capítulo 3 e no estudo quantitativo proposto no Capítulo 4. A Seção 2.4 apresentou a teoria envolvendo medições e métricas de *software* que será utilizada no Capítulo 4 para a análise quantitativa dos dados. Por fim, a Seção 2.5 mostrou trabalhos similares a este.

CAPÍTULO
3
Metadados de Programação Orientada a Objetos (MetaPOO)

Este capítulo apresenta uma das contribuições deste trabalho, que é a ferramenta *MetaPOO*, com os conceitos que guiaram a sua modelagem e desenvolvimento. Esta descrição utiliza os fundamentos da linguagem de programação Java.

O objetivo dessa ferramenta é extrair os metadados de arquivos de projeto Java e armazena-los em banco de dados relacional. Para tanto, é necessário percorrer e analisar todos os subdiretórios de um projeto Java.

A Seção 3.1 apresenta aspectos teóricos considerados durante a implementação. A Seção 3.2 descreve o modelo para os metadados e para a ferramenta *MetaPOO*. Já a Seção 3.3 apresenta as fases da ferramenta desenvolvida, composta por três etapas: Limpeza (3.3.1), Tipos (3.3.2) e Relações (3.3.3).

3.1 Conceitos iniciais

Esta seção descreve os conceitos que inspiraram as fases da ferramenta *MetaPOO* e apresenta as descrições para identificação de classes referentes à linguagem de programação Java que foram empregados no seu desenvolvimento.

O objetivo da ferramenta *MetaPOO* é a decomposição de código fonte orientado a objetos em metadados para o estudo quantitativo. O processo adotado para esta decomposição foi inspirado nas definições de Projeto Orientado a Objetos proposto por Booch et al. (2007).

Em Booch et al. (2007), o autor apresenta um estudo em que a decomposição de código fonte é realizada em quatro etapas. A primeira etapa é a identificação de classes, seguida pela etapa de identificar a semântica das classes. A terceira etapa é a identificação das relações entre as classes. Relações são os relacionamentos das classes com herança e parâmetros genéricos. E, por fim, a quarta etapa é a implementação dos detalhes internos da classe (BOOCH et al., 2007).

Uma das limitações deste estudo consiste em não capturar o comportamento dinâmico dos projetos analisados. Baseado neste estudo a ferramenta *MetaPOO* irá analisar e decompor códigos fontes usando as três primeiras etapas. A quarta etapa, referente ao corpo da classe, não será analisada neste trabalho, pois envolve aspectos que não serão analisados.

3.1.1 Identificação das classes

Em Java, classes são definidas em arquivos e organizadas em pacotes. A organização dos arquivos em pacotes é necessária para evitar conflitos de nomes e permitir a localização do código da classe de forma eficiente (GOSLING et al., 2013).

Um pacote é uma unidade de organização de código que agrega arquivos relacionados. Para indicar que as definições de um arquivo Java fazem parte de um determinado pacote a primeira linha de código deve ser a declaração de pacote utilizando a palavra reservada *package*. Caso a declaração não esteja presente, as classes farão parte de um pacote anônimo (ARNOLD; GOSLING; HOLMES, 2006).

De acordo com Gosling et al. (2013), toda classe possui um nome completo que é o nome do pacote em que a classe está localizada seguido de ponto (’.’) e seguido do nome da classe. Caso a classe esteja localizada em um pacote anônimo, seu nome completo é somente seu nome de classe. O nome completo é importante para utilizar importações.

O código da Figura 3.1 mostra o uso das instruções *package* e *import*. A linha 1 apresenta a definição de um pacote. O nome completo desta classe é *pacote.exemplo.MainClass*.

Importações

Uma instrução de importação permite incluir o código fonte de outras classes em um arquivo fonte no momento da compilação. Para realizar uma importação utiliza-se a palavra reservada *import* seguida pelo caminho do pacote, delimitado por pontos, terminando com um nome de classe ou um asterisco. Estas instruções ocorrem após a instrução *package* opcional e antes da definição de classe (LIGUORI;

```
1   package pacote.exemplo;
2
3   // Importa a classe ArrayList do pacote java.util
4   import java.util.ArrayList;
5   // Importa todas as classes do pacote java.io
6   import java.io.*;
7   //Importa membro estático ITALY
8   import static java.util.Locale.ITALY;
9
10  public class MainClass {
11  }
```

Figura 3.1: Declaração de pacote e importação em Java

FINEGAN, 2009).

O uso da importação terminada com um nome de classe é conhecido como importação explícita (linha 4 do código da Figura 3.1) e irá importar somente a classe especificada. Já a importação terminada com asterisco é conhecida como importação implícita (linha 6 do código da Figura 3.1) e irá importar todas as classes daquele pacote (LIGUORI; FINEGAN, 2009).

Outro tipo de importação são as importações estáticas, que permitem a importação de membros estáticos. Podem ser explícitas ou implícitas (LIGUORI; FINEGAN, 2009). A linha 8 do código da Figura 3.1 mostra uma importação estática explícita.

Segundo Arnold, Gosling e Holmes (2006), a classe de um pacote recebe implicitamente todo o código do seu pacote. Portanto, tudo que é definido em um pacote está disponível para os outros tipos no mesmo pacote. Além disso, o pacote *java.lang* é implicitamente importado em qualquer código porque é considerado essencial para a interpretação de qualquer programa Java.

O comando de importação simplesmente indica ao compilador como ele pode determinar o nome completo para uma classe que é usada no programa se ele não conseguir achar aquele tipo definido localmente. O compilador irá procurar pela classe na seguinte ordem: (1) A classe atual; (2) Uma classe aninhada à classe atual; (3) Classe importada explicitamente; (4) Outras classes declaradas no mesmo pacote; (5) Classes importadas implicitamente (ARNOLD; GOSLING; HOLMES, 2006).

As características de pacotes e importações da linguagem Java são fundamentais para as definições das heurísticas na função de resolver a ambiguidade de uma classe que será descrita na Seção 3.3.3.

3.2 Modelo de Classes para os Metadados

Para a análise quantitativa envolvendo as técnicas de reusabilidade heranças e tipos genéricos, optou-se por armazenar metadados dos arquivos de código fonte Java. Esse estudo é realizado por meio de uma abordagem automática diretamente a partir do código utilizando persistência em um banco de dados relacional. O diagrama de classes das entidades persistentes está representado na Figura 3.2.

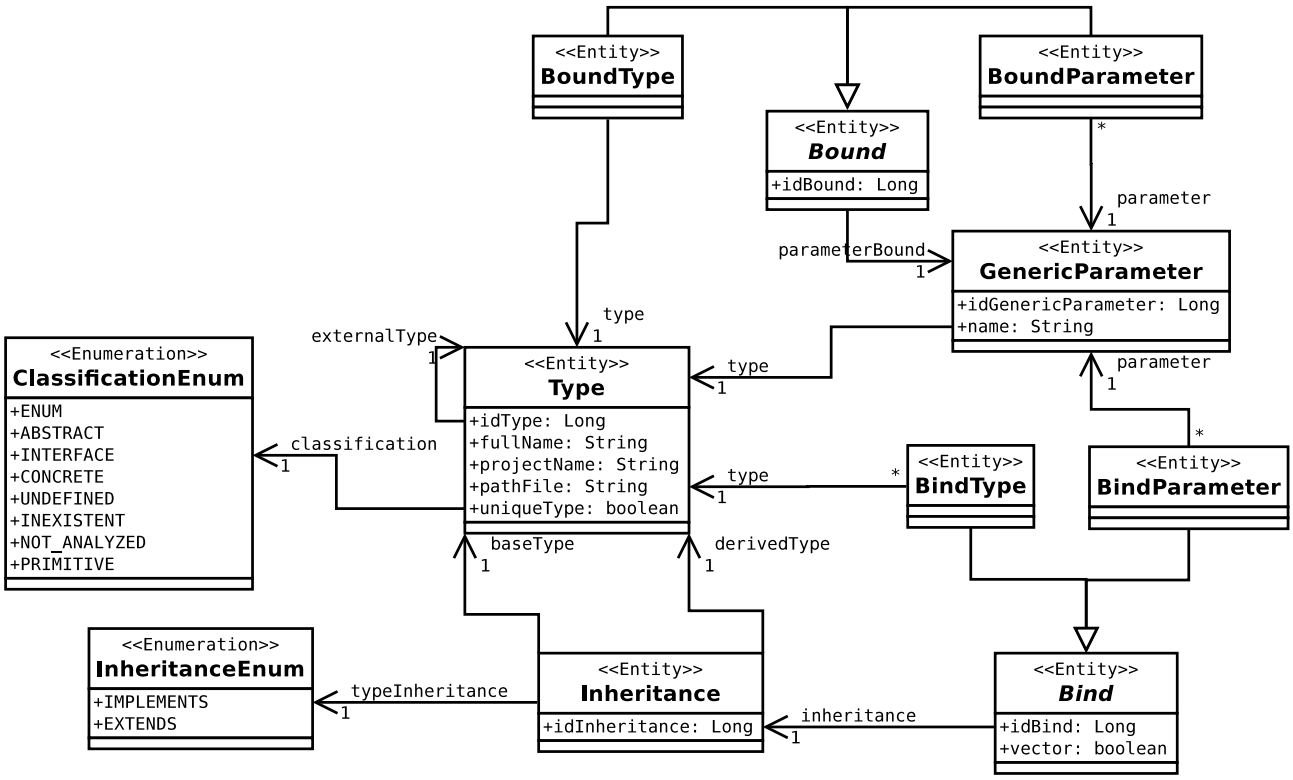

Figura 3.2: Diagrama de classe para metadados do código fonte.

A classe *Type* é responsável pela representação de tipos encontrados nos arquivos Java. Os tipos podem ser uma interface, um *enum* ou uma classe. Esta classe é caracterizada por um identificador, o nome completo do tipo, o nome do projeto que o tipo pertence, o caminho do arquivo e se o tipo é único. Caso seja uma classe interna, é definida também sua classe externa. Todo tipo possui uma classificação que é definida pelo *enum ClassificationEnum*. O detalhamento dessa classificação será descrito nas próximas seções.

A classe *Inheritance* define as características de herança de um tipo (apresentado na seção 2.3.2). Essa classe tem um atributo identificador e dois relacionamentos entre a classe *Type*: um que define o tipo da superclasse (atributo *baseType*) e outro que define o tipo da subclasse (atributo *derivedType*). Uma herança pode ser classificada como de implementação (*extends*) ou de interface (*implements*). Essa classificação é definida pelo valor do *enum InheritanceEnum*.

A classe *GenericParameter* define os parâmetros genéricos e é caracterizada por um identificador, o nome do parâmetro e o tipo que o parâmetro pertence (classe *Type*).

A classe abstrata *Bound* define as restrições dos parâmetros genéricos através do relacionamento com a classe *GenericParameter* (atributo *parameterBound*). Como discutido na Seção 2.3.3, as restrições dos parâmetros genéricos podem ser por tipo (subclasse *BoundType*) ou por parâmetro (subclasse *BoundParameter*). A classe *BoundType* é restrita pelo tipo (atributo *type*). Diferentemente, a classe BoundParameter é restrita por um parâmetro (atributo *parameter*).

A classe abstrata *Bind* define a ligação entre herança e tipo genérico (atributo *inheritance*), conforme foi apresentado na Seção 2.3.3. Além dessa ligação, esta classe abstrata tem um identificador e a informação se a ligação é feita por um vetor.

A Figura 3.3 ilustra o exemplo de um código em Java onde são extraídos os metadados que geram os objetos do modelo apresentado anteriormente (Figura 3.2). Na linha 1, a classe *ConcreteA* é um *Type* e possui três *GenericParameter*: T, U e V. O *GenericParameter* T tem uma restrição de *BoundType* e o *GenericParameter* U tem uma restrição de *BoundParameter*. Na linha 2, a classe *ConcreteB* também é um *Type* e possui um *GenericParameter*: K. Além disso, esta classe possui um *Inheritance*, onde *ConcreteA* é o *baseType* e *ConcreteB* é o *derivedType*. *ConcreteB* possui três *Bind*, sendo *ConcreteA* uma invocação de *BindType*, *ConcreteB* uma invocação de *BindType* e K uma invocação de *BindParameter*. Na linha 3, a classe *ConcreteC* é um *Type* que possui *ConcreteB* como *externalType*.

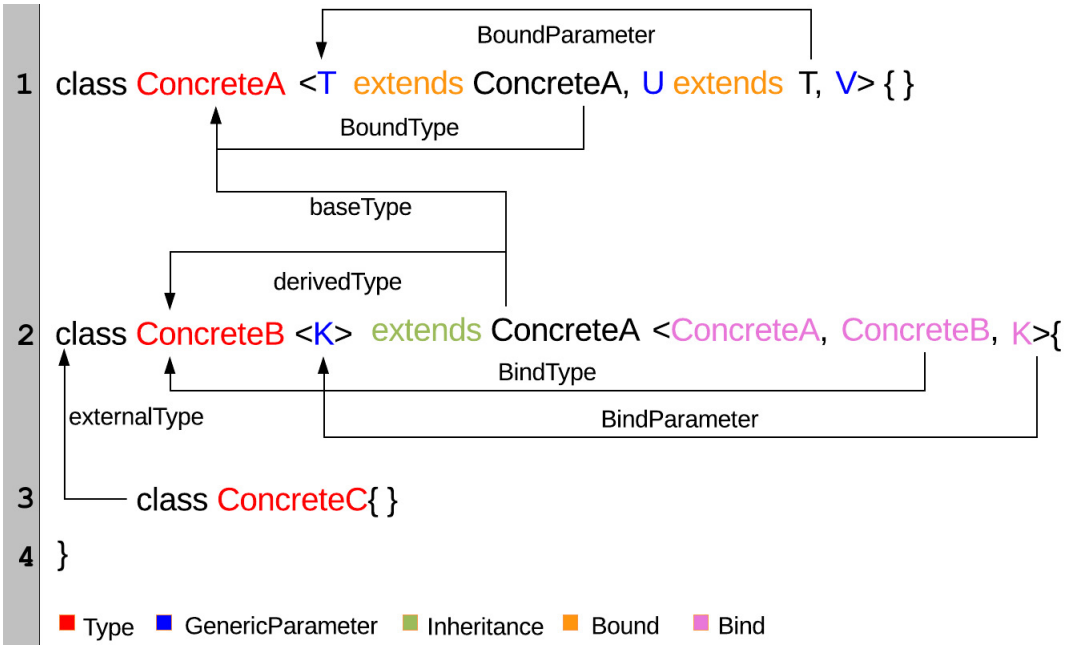

Figura 3.3: Exemplo de decomposição de classes em metadados

A geração e a persistência do modelo para os metadados é realizada nas três fases de execução da ferramenta que serão descritas na Seção 3.3.

3.2.1 Modelo de Classes para Ferramenta MetaPOO

A Figura 3.4 apresenta o diagrama de classes da ferramenta *MetaPOO* utilizando o padrão arquitetural *Model View Control* (MVC). A classe *AppManager* presente na camada *view* é responsável por executar vários *threads* e gerenciar as três fases da ferramenta que serão descritas na Seção 3.3. A camada *control* encapsula as três fases da ferramenta. A camada *persistence* utiliza o padrão *Data Access Object* (DAO) para cada uma das entidades do modelo de metadados (Figura 3.2).

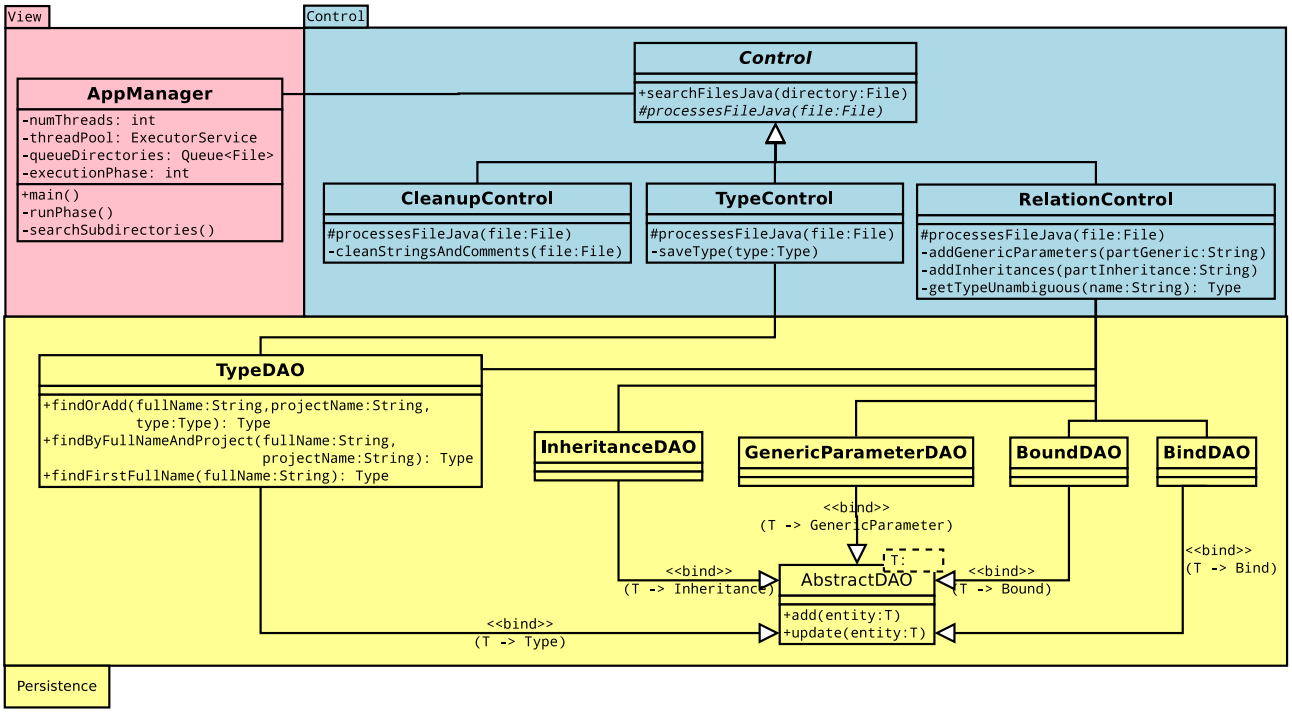

Figura 3.4: Diagrama de classes para a ferramenta *MetaPOO*

3.3 Fases da Ferramenta MetaPOO

A ferramenta *MetaPOO* encontra-se particionada em três fases, organizadas de acordo com a Figura 3.5.

1. **Limpeza**: limpa do código fonte elementos que não serão tratados neste trabalho (Seção 3.3.1)
   .

2. **Tipos**: extrai e persiste metadados referentes aos tipos (Seção 3.3.2).

3. **Relações**: extrai e persiste metadados referentes às heranças e tipos genéricos (Seção 3.3.3).

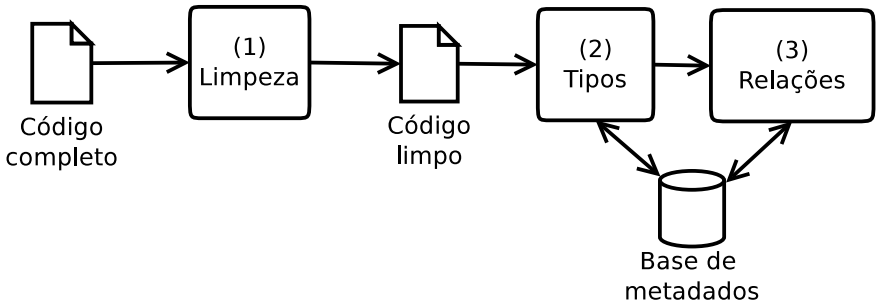

Figura 3.5: Fases da ferramenta

3.3.1 Etapa Limpeza

Esta fase é representada pela classe *CleanupControl* (Figura 3.4) e é responsável por reescrever os arquivos e limpar comentários, *strings*, métodos e atributos. Essa operação é feita invocando o método *cleanStringAndComments*.

Um exemplo de um código completo pode ser visto na Figura 3.6(a) e o resultado após a fase de limpeza pode ser visto na Figura 3.6(b). A classe *CodigoCompleto* tem um atributo do tipo *String* chamado *problema* (linha 10 do código da Figura 3.6(a)) que é inicializado com uma cadeia de caracteres que formam palavras reservadas usadas nas próximas fases da ferramenta. Para evitar estes problemas de interpretações das fases seguintes, *strings* e comentários também são removidos.

```java
1  package pacote.exemplo;
2
3  import java.math.BigInteger;import java.util.Collections;
4
5  /**
6   * Exemplo de um codigo completo
7   */
8  public class CodigoCompleto {
9
10     private static final String problema = "package class";
11
12     private BigInteger bigInt;
13
14     //Metodo get
15     public BigInteger getBigInt(){
16         return bigInt;
17     }
18
19     class ClasseInterna{ }
20 }
```

(a)

```java
1  package pacote.exemplo;
2  import java.math.BigInteger;
3  import java.util.Collections;
4  public class CodigoLimpo {
5      class ClasseInterna{
6      }
7  }
```

(b)

Figura 3.6: Entrada e saída da fase Limpeza

Para atingir um melhor desempenho nas etapas seguintes, esta etapa também remove linhas vazias, adiciona quebra de linha entre chaves duplas e após símbolo de terminação de linha (ponto e vírgula).

3.3.2 Etapa Tipos

Nesta etapa, a classe *TypeControl* (Figura 3.4) processará os arquivos limpos para encontrar as declarações de tipos. Ao identificar uma declaração de classe ou interface são extraídas as características para persistir um objeto *Type*. As características são o nome da classe ou interface, o pacote que está localizada, um relacionamento reflexivo que identifica o seu tipo externo (caso exista), a classificação (*ClassificationEnum* da seção 3.2) e um booleano que informa se o tipo é único.

Os tipos não únicos são aqueles em que a tripla projeto, pacote e nome se repetem durante esta etapa. Isso acontece muitas vezes em projetos que possuem pastas para diversos sistemas operacionais, mas que definem o mesmo tipo com o mesmo nome de pacote. Como o identificador único de um tipo é composto pelo seu nome completo e pelo nome do projeto, esses tipos não são únicos. Os valores possíveis para a classificação nesta etapa são: *Interface, Enum, Concrete e Abstract*.

Como pode ser visto no diagrama de objetos da Figura 3.7, os metadados extraídos do código limpo da Figura 3.3 persistem três objetos: *ConcreteA*, *ConcreteB* e *ConcreteC*.

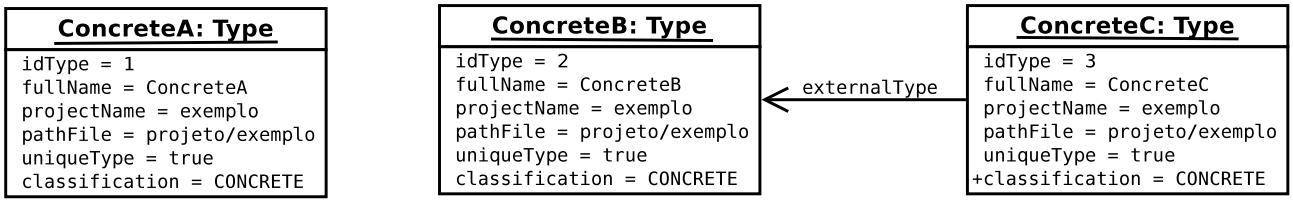

Figura 3.7: Exemplo de um diagrama de objetos da fase 2

3.3.3 Etapa Relações

Nesta etapa, a classe *RelationControl* (Figura 3.4) processará cada arquivo limpo novamente para adicionar, caso existam, as características sobre parâmetros genéricos por meio do método *addGenericParameters* e para adicionar, caso existam, as características sobre heranças por meio do método *addInheritances*.

As características sobre parâmetros genéricos são definidas pelas persistências e associações entre as classes *Type*, *GenericParameter* e *Bound*. Por outro lado, as características sobre heranças são

definidas pelas persistência e associações entre as classes *Type*, *Inheritance* e *Bind*.

A Figura 3.8, mostra o diagrama de objetos que está persistido dos metadados extraídos sobre os parâmetros genéricos e das heranças do código da Figura 3.3. Os objetos *ConcreteA* e *ConcreteB* foram persistidos na etapa anterior. O objeto *ConcreteA* possui três objetos *GenericParameter*, além de estar associado a *BindType*, *BoundType* e *Inheritance*. O objeto *ConcreteB* possui um objeto *GenericParameter*, um objeto *Inheritance*, além de estar associado a *BindType*.

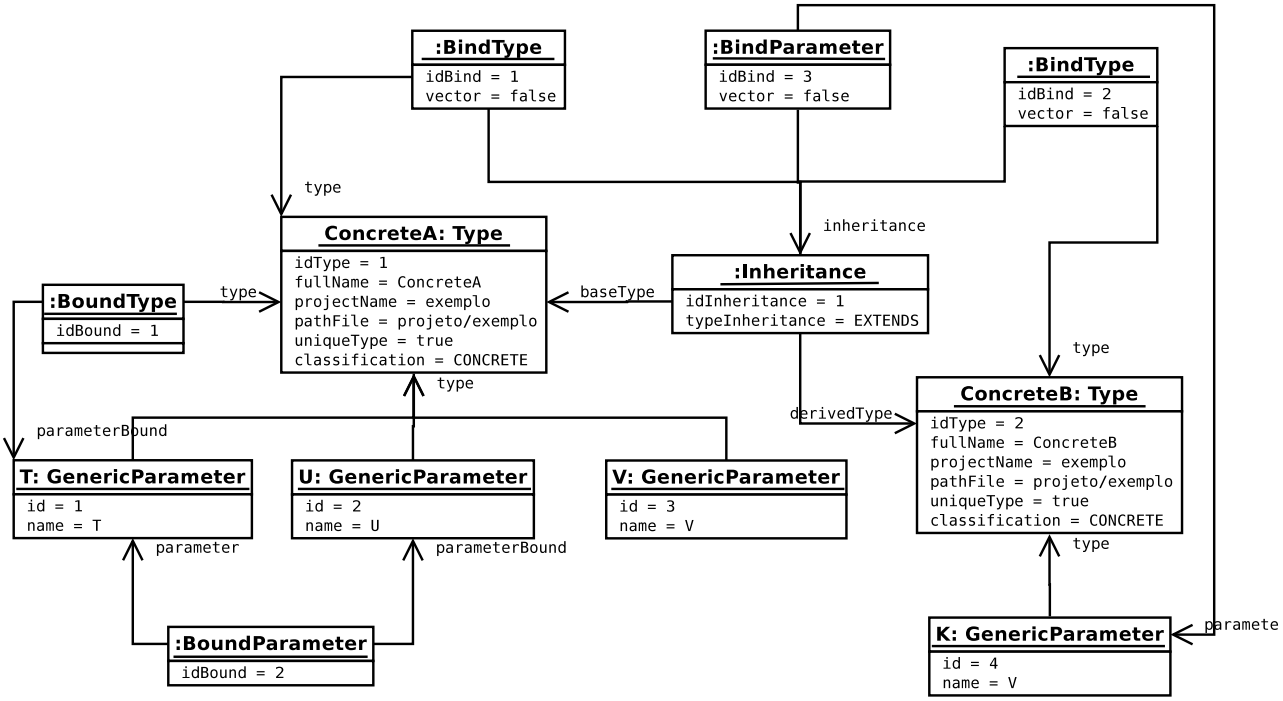

Figura 3.8: Exemplo de um diagrama de objetos da fase 3

Esta fase pode armazenar valores inválidos sobre as características dos parâmetros genéricos e das heranças caso o código fonte contenha erros de compilação. A validação de compilação do código fonte não foi implementada neste trabalho, pois os experimentos usaram projetos estáveis que foram obtidos por meio de sistemas de controle de versão.

Um grande desafio enfrentado nessa etapa foram as ambiguidades sintáticas dos nomes das classes e será descrito a seguir.

Ambiguidade sintática

O método *getTypeUnambiguous* é responsável por resolver ambiguidades sintáticas, já que nesta fase deve ser determinado o pacote e nome (nome completo) de tipos declarados para as heranças e para os tipos genéricos. Ao tentar inferir um nome completo de um tipo, pode acontecer de não ser possível

recuperar toda a informação.

A desambiguidade do nome completo não leva em consideração bibliotecas binárias adicionadas ao projeto. No caso de haver bibliotecas adicionadas ao projeto, deve-se processar, primeiramente, todos os códigos fontes dessas bibliotecas.

Ao executar a função *getTypeUnambiguous* o tipo que será resolvido a ambiguidade pode estar totalmente qualificado, ou seja, com pacote e nome do tipo. Para separar o pacote do nome do tipo foi utilizada a convenção de nomenclatura Java, onde pacotes iniciam com letras minúsculas e nomes de tipos iniciam com letras maiúsculas.

O atributo classificação (*ClassificationEnum* da seção 3.2) pode receber os seguintes valores nesta etapa: *Primitive, Not Analyzed, Inexistent, Undefined*.

Caso a função *getTypeUnambiguous* encontre um tipo primitivo, este será adicionado ao banco de dados com a classificação *Primitive*.

A classificação *Not Analyzed* ocorre quando a função consegue resolver o nome completo do tipo, porém, o tipo não foi encontrado na base de dados por pertencer a alguma biblioteca do projeto que não foi analisada.

A classificação *Inexistent* ocorre quando a função consegue resolver o nome completo do tipo e deveria ser um tipo presente na base de dados, mas por algum motivo desconhecido, o tipo não existe.

Por fim, a classificação *Undefined* ocorre quando a função *getTypeUnambiguous* não consegue resolver o nome completo do tipo.

Para resolver a ambiguidade do nome completo, a função *getTypeUnambiguous* utiliza a heurística a seguir, a partir de um nome que representa a ambiguidade sintática e a definição do tipo no qual foi declarado:

1. Se o nome é um tipo primitivo, retorna este tipo primitivo.
2. Se o nome se refere ao próprio tipo declarado, retorna o tipo declarado.
3. Se o nome é um tipo interno do tipo declarado, retorna o tipo interno.
4. Se o nome já está completo, retorna o tipo com este nome completo.
5. Se existe uma importação explícita para o nome, retorna o tipo da importação explícita.
6. Se não existe uma importação explícita para o nome e também não existe nenhuma importação implícita:

   (a) Se o nome existe no mesmo pacote do tipo principal, retorna o tipo deste pacote.

   (b) Caso contrário, retorna o tipo do pacote *java.lang*.

7. Se não existe uma importação explícita para o nome, mas existe uma importação implícita:

   (a) Se existe o nome no mesmo pacote do tipo principal, retorna o tipo deste pacote.

   (b) Se existe o nome no pacote *java.lang*, retorna o tipo *java.lang*.

   (c) Caso contrário, retorna o tipo da importação implícita.

8. Se não existe uma importação explícita para o nome e existem várias importações implícitas:

   (a) Se existe o nome no mesmo pacote do tipo principal, retorna o tipo deste pacote.

   (b) Se existe o nome no pacote *java.lang*, retorna o tipo *java.lang*.

   (c) Para cada importação implícita: se existe na base o tipo com aquela importação implícita, retorna o tipo da importação implícita.

   (d) Para cada importação implícita: se o nome pertence à importação utilizando a função *Class.forName*, retorna o tipo da importação implícita.

9. Caso contrário, retorna um tipo com classificação *Undefined*.

3.4 Considerações Finais

Este capítulo apresentou a ferramenta *MetaPOO* utilizada para decompor o código fonte orientado a objetos em metadados para o estudo quantitativo.

A Seção 3.1 introduziu os conceitos que inspiraram as fases da ferramenta *MetaPOO* e apresentou as descrições para identificação de classes referentes à linguagem de programação Java. A Seção 3.2 apresentou os diagramas de classes das entidades do modelo de metadados e do modelo da ferramenta. Por fim, a Seção 3.3 mostrou como a ferramenta foi organizada e detalhou cada uma das três fases.

O Capítulo 4 apresentará e analisará o estudo quantitativo sobre os metadados obtidos com a execução da ferramenta *MetaPOO*. Esse estudo é realizado por meio de consulta aos metadados armazenados em banco de dados relacional.

CAPÍTULO
4
Análise Quantitativa

Este capítulo descreve os resultados obtidos com a execução da ferramenta *MetaPOO*. Visando obter dados para uma análise quantitativa, foram utilizados mais de 200.000 arquivos.

A Seção 4.1 descreve os conjuntos de dados utilizados nas análises, os parâmetros utilizados como referência e as configurações adotadas. As Seções 4.2, 4.3, 4.4 e 4.5 apresentam e discutem os comportamentos dos gráficos confeccionados a partir dos resultados obtidos agrupando-os por tipos, heranças, tipos genéricos e CRTP, respectivamente.

O estudo quantitativo foi realizado por meio de consultas SQL das tabelas mapeadas pelo modelo objeto-relacional (Código A.1) do Apêndice A dos metadados gerados pela ferramenta *MetaPOO*.

4.1 Metodologia

Neste trabalho foram analisados projetos de código aberto disponíveis no repositório web GitHub (GITHUB, 2013a) que utilizam a linguagem de programação orientada a objetos Java. Os projetos deste repositório apresentam uma coleção de arquivos individuais que são armazenados em uma pasta raiz denominada *src*.

Para a análise quantitativa, os projetos foram obtidos entre março e abril de 2013 de quatro comunidades. A Tabela 4.1, mostra as comunidades, suas quantidades de projetos e informações sobre os arquivos.

A primeira comunidade gerencia os projetos da distribuição OpenJDK (GITHUB, 2013c). O OpenJDK foi escolhido por conter uma biblioteca padrão implementada em Java e usada em muitos outros projetos (CORPORATION, 2013). Esta distribuição possui 12 projetos com 11.471 arquivos.

Tabela 4.1: Conjuntos de dados

	Projeto	**Arquivo**	**Tipo**	**Tamanho (MB)**
OpenJDK	12	11.471	16.742	307,70
Apache	5.578	201.682	259.622	4.505,60
Spring	152	9.971	13.825	126,40
Jboss	117	13.552	15.988	331,10
Total	**5.859**	**236.676**	**306.177**	**5.270,80**

A segunda comunidade consiste de 5.578 projetos com 201.682 arquivos da distribuição Apache Software Foundation (FOUNDATION, 2010). A Apache é uma organização sem fins lucrativos que fornece suporte organizacional para mais de 140 projetos de *software* de código aberto (FOUNDATION, 2012).

A terceira comunidade foi obtida do Spring Source (GITHUB, 2013d), que possui um dois mais populares *frameworks* para aplicações Java (GOPIVOTAL, 2013). Este conjunto de dados contém 152 projetos com 9.971 arquivos.

A quarta comunidade é formada por 117 projetos com 13.552 arquivos da distribuição JBoss (GITHUB, 2013b). Esta comunidade, também de código aberto, possui um dos *frameworks* de persistência mais populares, o Hibernate (REDHAT, 2013).

A análise foi dividida em quatro seções. A Seção 4.2 analisará os dados obtidos com a entidade *Type*. A Seção 4.3 discutirá os dados obtidos com a entidade *Inheritance*. Já a Seção 4.4 apresentará os dados relacionados a tipos genéricos envolvendo as entidades *GenericParameter*, *Bound* e *Bind*. Por fim, a Seção 4.5 analisará os dados referentes a CRTP.

4.2 Tipos

Esta seção apresenta um estudo das características de classes implementadas em projetos orientados a objetos.

O total de tipos encontrados com projeto é de 306.177. Conforme descrito na Seção 3.3.2, cada tipo tem uma classificação: *Interface*, *Enum*, *Concrete*, *Abstract*, *Primitive*, *Not Analyzed*, *Inexistent* e *Undefined*. A Figura 4.1 mostra a distribuição dessas classificações.

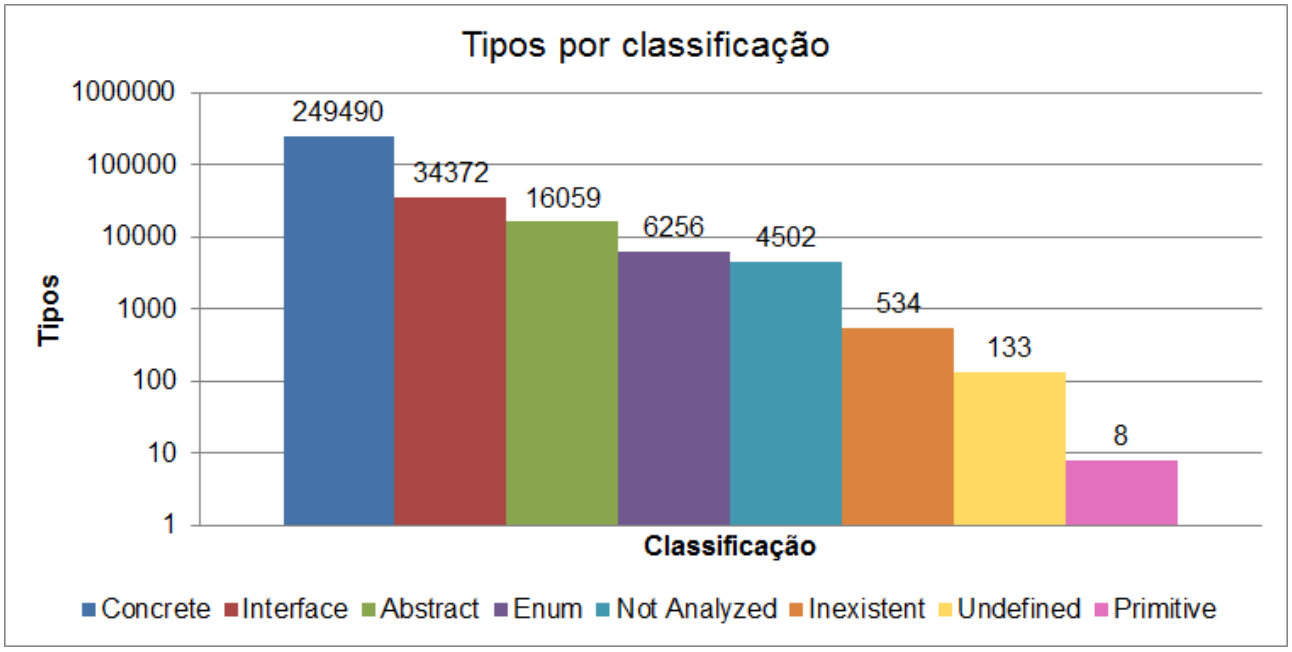

Figura 4.1: Quantidade de tipos por classificação

Pode-se perceber que a quantidade de classes concretas é superior à soma de todas as outras classificações.

A classificação *Primitive* indica que o tipo não é uma classe, mas um tipo primitivo e representa cerca de 0,002% do total de tipos analisados, que são realmente os oitos tipos suportados pela linguagem Java.

Como a ferramenta *MetaPOO* não acessa as bibliotecas binárias presentes nos projetos, a classificação *Not Analyzed* representa os tipos que tiveram a ambiguidade resolvida, porém seus arquivos estão nas bibliotecas binárias. Cerca de 1,47% dos tipos foram classificados como *Not Analyzed*.

A classificação *Inexistent* é dada quando é possível resolver a ambiguidade do tipo, mas o arquivo não existe no projeto. Foram classificados como *Inexistent* 0,17% do total de tipos analisados.

A classificação *Undefined* significa que a função não conseguiu resolver o nome completo do tipo, como foi descrita na Seção 3.3.3. Essa classificação obteve cerca de 0,043% do total de tipos analisados.

As classificações *Primitive*, *Not Analyzed*, *Inexistent* e *Undefined* não possibilitam extrações de metadados e não serão usadas para este estudo quantitativo.

Ainda com relação a tipos, foram encontrados 67.757 tipos internos. A distribuição desses tipos internos de acordo com a classificação *Interface*, *Enum*, *Concrete* e *Abstract* pode ser vista na Figura 4.2. O gráfico mostra que, para cada classificação, existe mais tipos externos do que internos,

exceto pelo tipo *Enum*, que possui cerca de 71,69% *enums* internos.

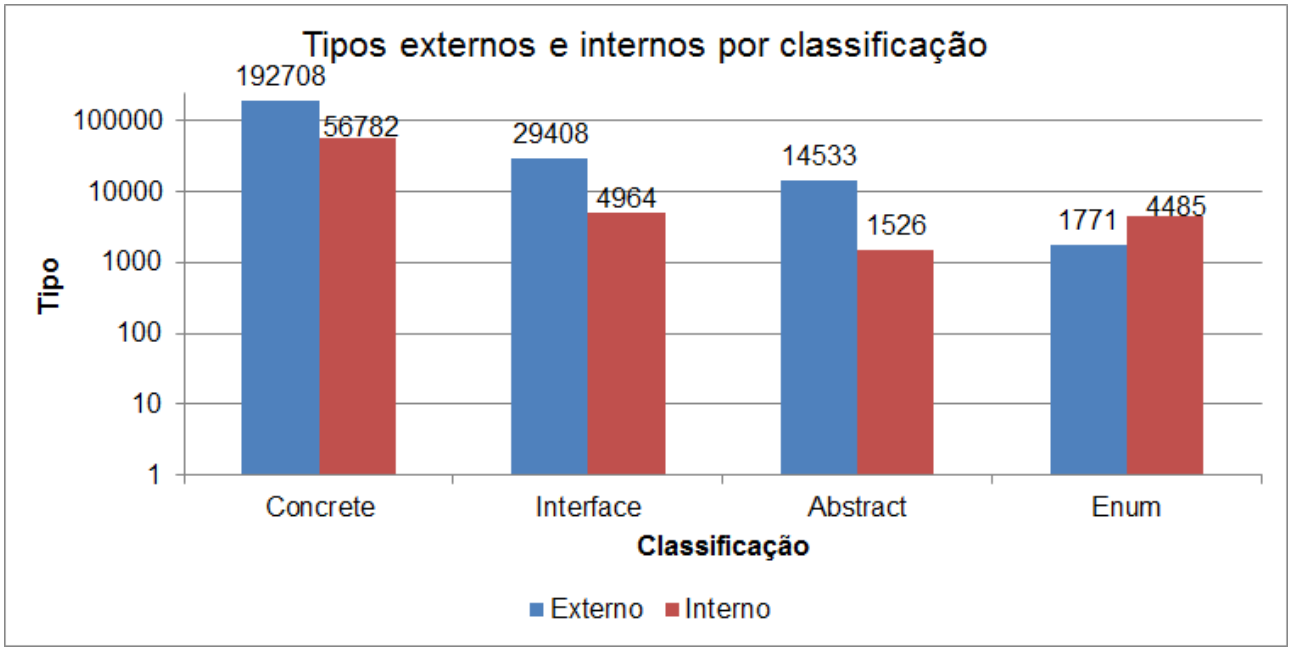

Figura 4.2: Quantidade de tipos externos e internos

4.3 Heranças

A herança é um dos principais recursos de reusabilidade. É uma relação entre duas classes, onde a classe mãe é denominada base e a classe filha é denominada derivada. Além disso, a herança pode ser feita de duas formas: *extends* ou *implements*.

Na linguagem Java pode-se caracterizar uma classe combinando bases e derivadas com implementações e extensões obtendo os seguintes conjuntos:

- Classe base que é implementada (*BaseImp*) é exemplificada pelas interfaces O, P, Q e R da Figura 4.3.
- Classe base que é estendida (*BaseExt*) é exemplificada pela classe concreta F Figura 4.3.
- Classe derivada que estende (*DerivExt*) é exemplificada pela classe concreta K Figura 4.3.
- Classe derivada que implementa (*DerivImp*) é exemplificada pelos *enums* S e T da Figura 4.3.

Ainda é possível caracterizar uma classe Java combinando estes conjuntos e obtendo os subconjuntos:

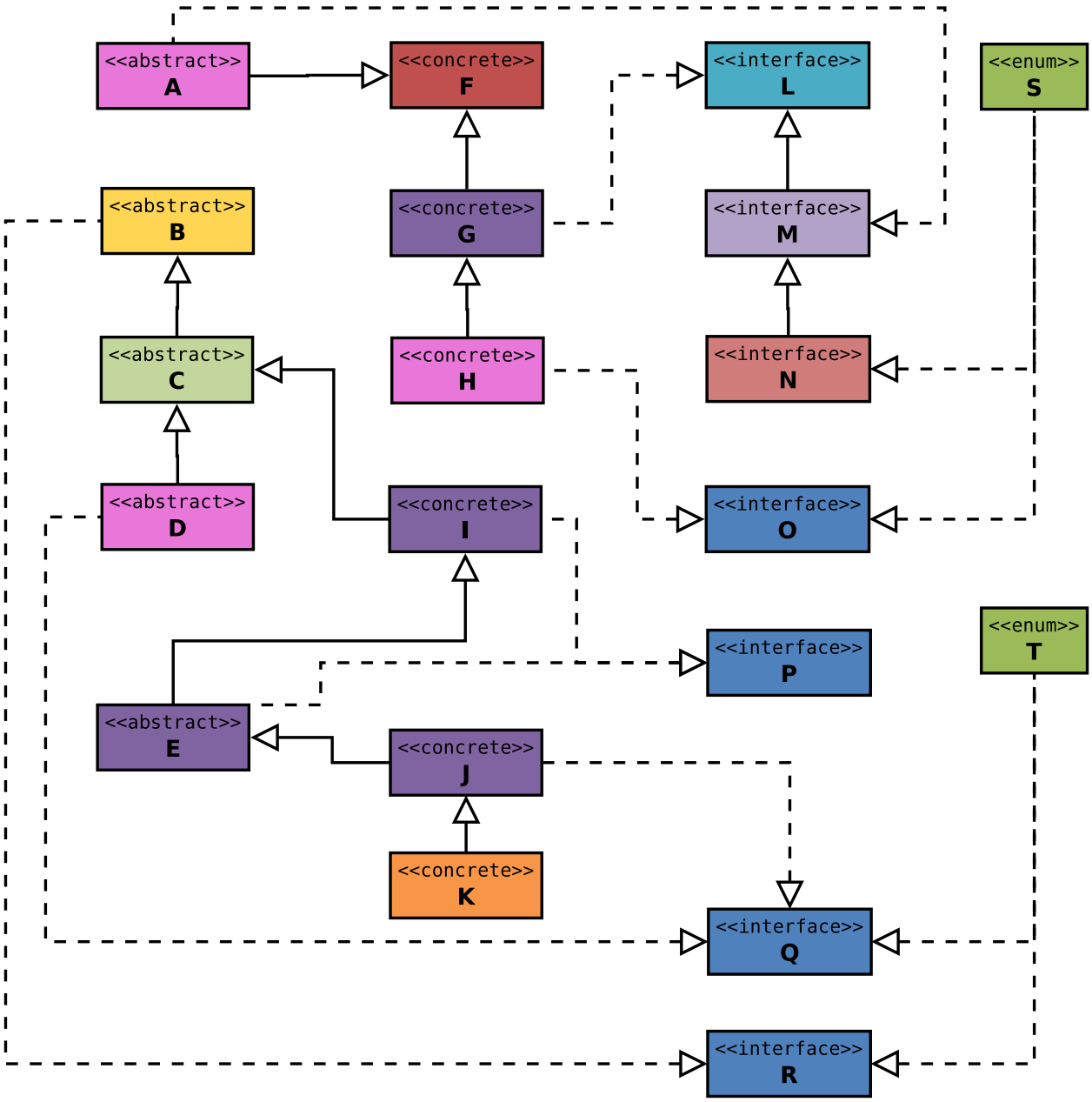

Figura 4.3: Diagrama de classes de herança

- Classe base que é implementada e estendida (*BaseImpExt*) é exemplificada pela interface L da Figura 4.3.
- Classe derivada que implementa e estende (*DerivImpExt*) é exemplificadas pelas classes abstratas A e D e pela classe concreta H da Figura 4.3.
- Classe base que é estendida e ainda é derivada que implementa (*BaseExt e DerivImp*) é exemplificada pela classe abstrata B da Figura 4.3.
- Classe base que é implementada e ainda é derivada que estende (*BaseImp e DerivExt*) é exemplificada pela interface N da Figura 4.3.

- Classe base que é estendida e ainda é derivada que estende (*BaseExt e DerivImp*) é exemplificada pela classe abstrata C da Figura 4.3.
- Classe base que é implementada e estendida e ainda é derivada que estende (*BaseImpExt e DerivExt*) é exemplificada pela interface M da Figura 4.3.
- Classe base que é estendida e ainda é derivada que implementa e estende (*BaseExt e DerivImpExt*) é exemplificadas pelas classes concretas G, I e J e pela classe abstrata E da Figura 4.3.

A Figura 4.4 ilustra as intersecções possíveis na linguagem Java entre os quatro conjuntos que geram mais sete subconjuntos. Os únicos conjuntos que não possuem intersecção são *BaseImp* e *DerivImp*, isso porque uma interface não pode implementar outra interface.

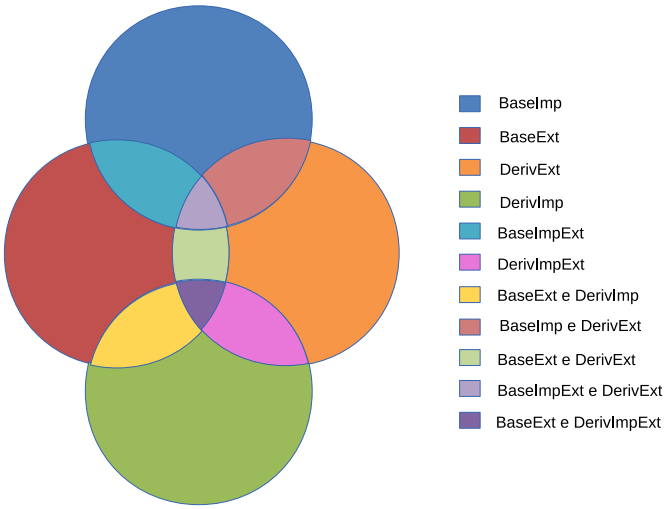

Figura 4.4: Conjunto de herança

Este trabalho aferiu as duas métricas para herança discutidas na Seção 2.4: DIT e NOC.

O resultado da métrica DIT para herança (extensão ou implementação) pode ser visto na Figura 4.5. Optou-se por começar com nível de herança igual a 1 porque em Java todo tipo é uma extensão da classe *Object*. Assim, foi considerado que Java somente tem uma classe de nível 0 que é *Object*.

A profundidade máxima da árvore de herança é 14 e cerca de 31% dos tipos herdam diretamente de *Object*, mas 69% possuem algum nível de herança.

O resultado da métrica NOC está ilustrado na Figura 4.6, onde 89,95% dos tipos possuem zero filhos. Por outro lado, apenas 10,05% dos tipos apresentam um ou mais filhos.

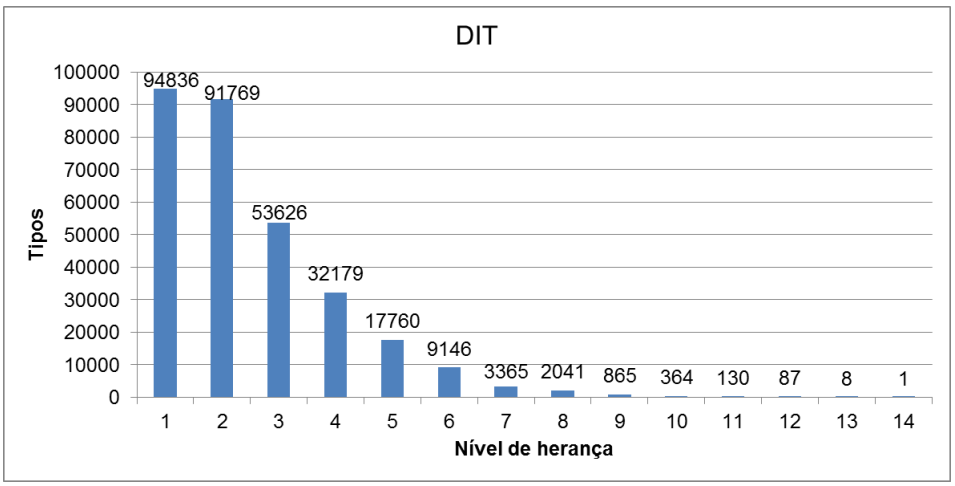

Figura 4.5: DIT

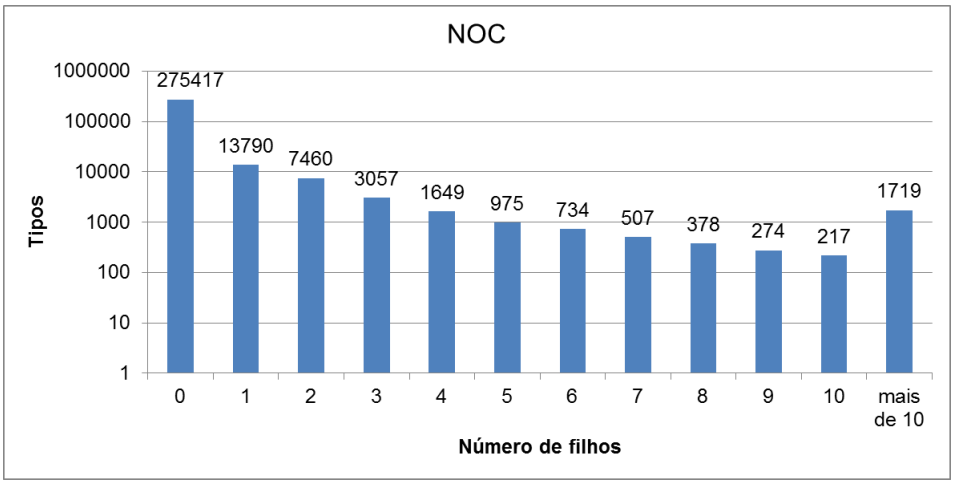

Figura 4.6: NOC

Além das métricas DIT e NOC, a análise de herança utiliza medições envolvendo os quatro valores (Base, Derivada, *Extends*, *Implements*).

A primeira medição sobre a herança pode ser vista na Figura 4.7 e apresenta classes bases e derivadas que utilizam herança de extensão.

A Figura 4.7(a) ilustra a quantidade de bases não genéricas que são estendidas: por somente *Interface*, por somente *Concrete*, por somente *Abstract* e por *Abstract* e *Concrete*. O maior número de classes não genéricas é o de base concreta sendo estendida por somente concreto. Em seguida, é a de base abstrata sendo estendida por somente concreto. Chama a atenção o número de interfaces sendo estendidas por interfaces. O menor número deste gráfico é a base concreta sendo estendida por somente abstrato.

A Figura 4.7(b) ilustra a quantidade de bases genéricas que são estendidas: por somente *Interface*, por somente *Concrete*, por somente *Abstract* e por *Abstract* e *Concrete*. O maior número de classes

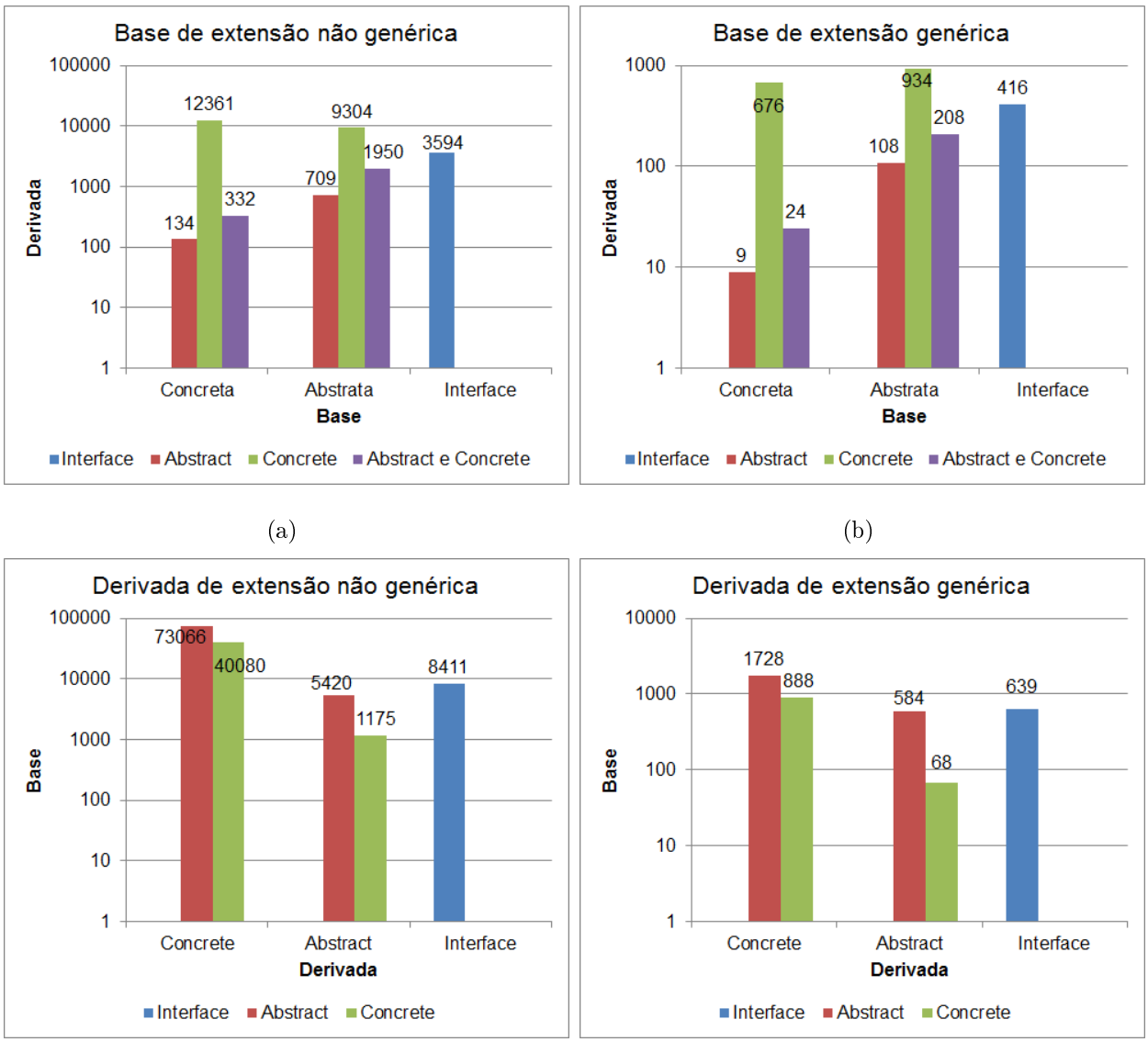

Figura 4.7: Herança de extensão

genéricas é o de base abstrata genérica sendo estendida por somente concreto. O menor número é a base concreta genérica sendo estendida por somente abstrato.

A Figura 4.7(c) mostra a quantidade de derivadas não genéricas que estende: *Interface*, *Abstract* e *Concrete*. As classes não genéricas que são derivadas concretas estendem mais tipos abstratos do que tipos concretos. E as classes não genéricas que são derivadas abstratas também estendem mais tipos abstratos do que tipos concretos.

A Figura 4.7(d) mostra a quantidade de derivadas genéricas que estende : *Interface*, *Abstract* e *Concrete*. As classes genéricas que são derivadas concretas estendem mais tipos abstratos do que tipos concretos. E as classes genéricas que são derivadas abstratas também estendem mais tipos abstratos do que tipos concretos.

A segunda medição sobre a herança pode ser vista na Figura 4.8 e apresenta classes bases e derivadas que utilizam herança de implementação.

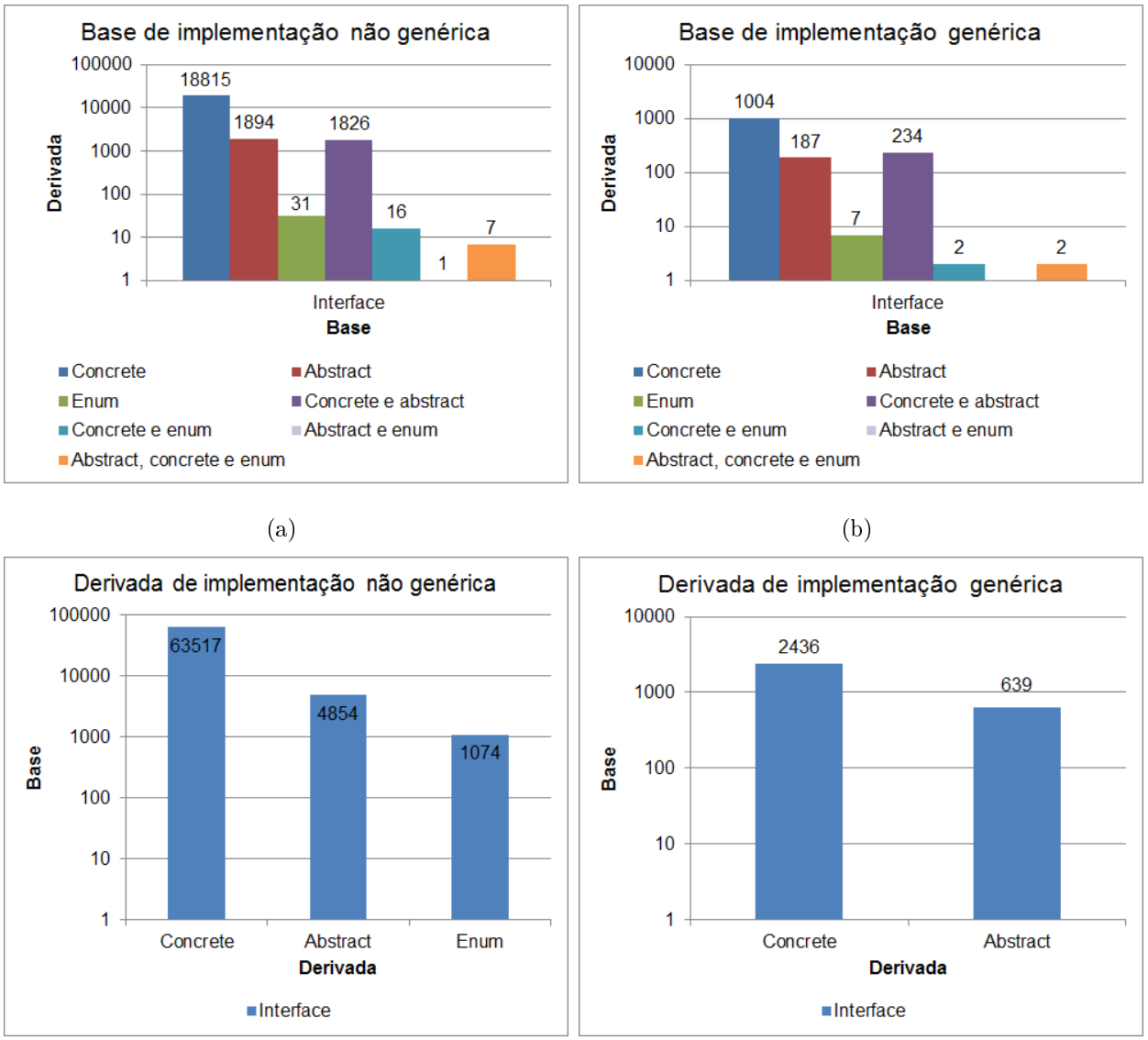

Figura 4.8: Herança de interface

A Figura 4.8(a) mostra a quantidade de interface não genérica que está sendo implementada: por somente *Concrete*, por somente *Abstract*, por somente *Enum*, por *Concrete* e *Abstract*, por *Concrete* e *Enum*, por *Abstract* e *Enum*, por *Abstract*, *Concrete* e *Enum*. A maioria das interfaces são implementadas por somente classes concretas. O número de interfaces sendo implementada por somente classes abstratas e por classes concretas e abstratas é quase o mesmo.

A Figura 4.8(b) mostra a quantidade de interface genérica que está sendo implementada: por somente *Concrete*, por somente *Abstract*, por somente *Enum*, por *Concrete* e *Abstract*, por *Concrete* e *Enum*, por *Abstract* e *Enum*, por *Abstract*, *Concrete* e *Enum*. O gráfico segue a mesma tendência da

Figura 4.8(a).

A Figura 4.8(c) ilustra a quantidade de derivadas não genéricas que implementam interfaces. As classes concretas não genéricas são as que mais implementam interfaces, seguida pelas classes abstratas não genéricas e, por último, os *enums*.

A Figura 4.8(d) apresenta a quantidade de derivadas genéricas que implementam interfaces. Assim como na Figura 4.8(c), as classes concretas são as que mais implementam interfaces.

4.4 Tipos Genéricos

A programação genérica é uma técnica da POO pouco usada pelas projetos analisados, pois apenas 8.691 classes ou interfaces são genéricas.

A Figura 4.9 apresenta as distribuições destes tipos genéricos de acordo com a classificação: *Abstract*, *Concrete* e *Interface*. Dos tipos que são genéricos, 61% são classes concretas, 22% são interfaces e 17% são classes abstratas.

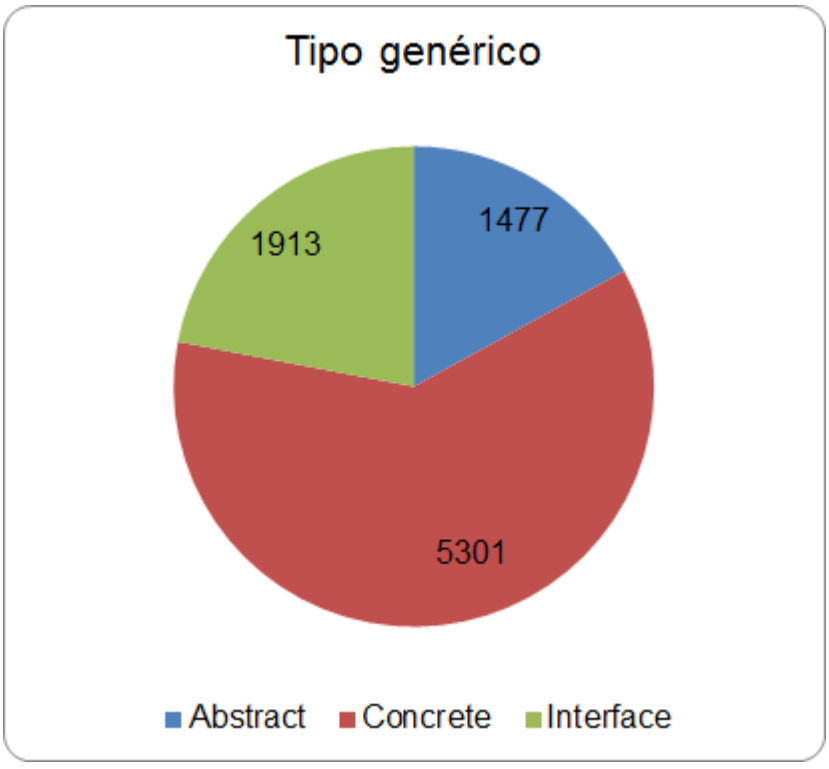

Figura 4.9: Tipos genéricos

Os tipos genéricos podem ter um ou mais parâmetros genéricos. A Figura 4.10 mostra a quantidade de tipos por número de parâmetro. O número de parâmetros varia de 1 a 7 e cerca de 69,59% dos tipos genéricos possuem somente um parâmetro genérico.

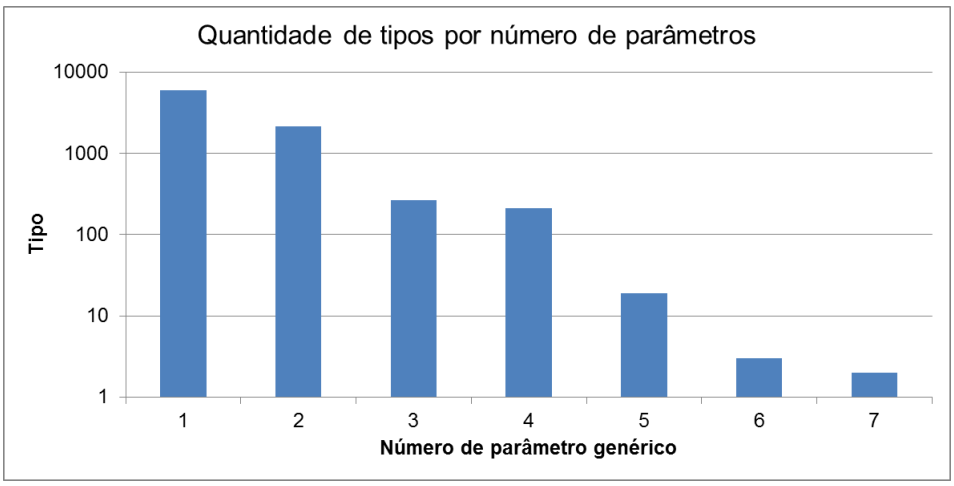

Figura 4.10: Número de parâmetros genéricos

Os tipos genéricos possuem parâmetros genéricos que podem ou não ser restritos (*bound*). Quando os parâmetros genéricos são restritos, podem ser por tipo ou por outros parâmetros genéricos (Seção 2.3.3). A Tabela 4.2 mostra essa característica dos parâmetros genéricos, onde 26% dos parâmetros possuem restrição e a maioria é restrita por tipo.

Tabela 4.2: Quantidade de Bound e Bind

	Bound	**Bind**
Tipo	3043	20445
Parâmetro Genérico	136	6127

A invocação de um tipo genérico (*bind*) é outra característica da programação genérica. Ao invocar um tipo genérico em uma declaração deve-se definir o parâmetro genérico através de um tipo ou de outro parâmetro. A Tabela 4.2 também apresenta os dados referentes a como a invocação de tipos é utilizada, onde a maioria utiliza a invocação por um tipo.

4.5 CRTP

O padrão CRTP foi apresentado na Seção 2.2.1 e pode acontecer em qualquer tipo de herança (implementação ou interface). Para o estudo deste padrão optou-se pela análise de duas formas.

A primeira é denominada CRTP explícito e está representada na Figura 4.11. Sua característica consiste na restrição do parâmetro genérico pelo tipo da superclasse. O código da Figura 4.11(a)

apresenta o CRTP explícito utilizando herança de implementação. Já o código da Figura 4.11(b) apresenta o CRTP explícito utilizando herança de interface.

```
1  public class BaseClass <T extends BaseClass> {
2  }
3
4  class DerivedClass extends BaseClass<DerivedClass>{
5  }
```

(a)

```
1  public interface BaseClass <T extends BaseClass> {
2  }
3
4  class DerivedClass implements BaseClass<DerivedClass>{
5  }
```

(b)

Figura 4.11: CRTP explícito

A segunda forma é denominada CRTP implícito e está ilustrada na Figura 4.12. A principal diferença é a não restrição do parâmetro genérico pelo tipo da superclasse. O código da Figura 4.12(a) apresenta o CRTP implícito utilizando herança de implementação. Já o código da Figura 4.12(b) apresenta o CRTP implícito utilizando herança de interface.

```
1  public class BaseClass <T> {
2  }
3
4  class DerivedClass extends BaseClass<DerivedClass>{
5  }
```

(a)

```
1  public interface BaseClass <T> {
2  }
3
4  class DerivedClass implements BaseClass<DerivedClass>{
5  }
```

(b)

Figura 4.12: CRTP implícito

As duas formas caracterizam o padrão CRTP por transferir o tipo derivado para a superclasse na invocação do tipo genérico.

Do total de classes analisadas, apenas 109 tipos foram encontrados utilizando CRTP. A Figura 4.13 apresenta os dados obtidos para estas formas de implementação do padrão, explícito e implícito.

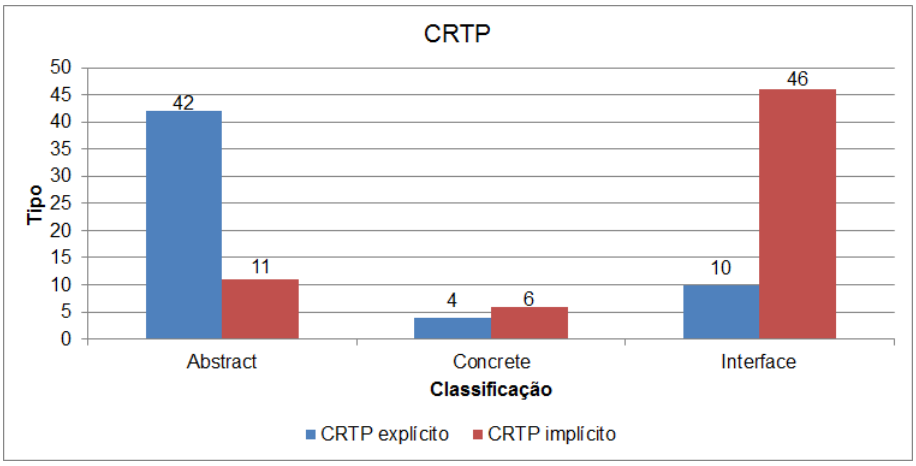

Figura 4.13: Quantidade de CRTP explícito e implícito por tipo

O maior uso do CRTP acontece de forma implícita utilizando interfaces como superclasse. Também é bastante utilizada a forma de CRTP explícita com classes abstratas como superclasse.

4.6 Considerações Finais

Este capítulo apresentou uma análise quantitativa de algumas técnicas envolvendo Programação Orientada a Objetos.A Tabela 4.3 resume os dados que foram apresentados neste capítulo.

Tabela 4.3: Resumo das seções analisadas

	Quantidade	**Superclasse**	**Subclasse**	**Tipo Genérico**	**CRTP**
Abstrato	16.059	13.213	11.927	1.477	46
Concreto	249.490	13.537	188.355	5.301	10
Interface	34.372	25.521	9.914	1.913	53
Enum	6.256		1.145		
Total	**306.177**	**52.271 (17,07%)**	**211.341 (69,02%)**	**8.691 (2,84%)**	**109 (0,035%)**

Na Seção 4.2, foram analisados dados referentes aos tipos. A maioria dos tipos são classes concretas, como mostra a Tabela 4.3. Apenas 0,043% foram os tipos que a ferramenta *MetaPOO* não conseguiu resolver a ambiguidade sintática.

Na Seção 4.3, cerca de 69% dos tipos possuem algum tipo de herança. Já a métrica NOC mostrou que 89,95% dos tipos não possuem filhos. Ainda com relação à herança, os gráficos desta seção mostraram que a maioria das superclasses são interfaces enquanto que a maioria das subclasses são classes concretas.

Na Seção 4.4 foram analisados dados referentes aos tipos genéricos. De acordo com a Tabela 4.3, apenas 2,84% dos tipos são genéricos, sendo que o tipo que mais utiliza a programação genérica é a classe concreta.

Na Seção 4.5 foram analisados dados referentes ao padrão CRTP. Somente 109 tipos utilizam CRTP, como mostra a Tabela 4.3. Este padrão ocorre mais em interfaces do que em classes.

Capítulo 5
Conclusões

O objetivo principal deste trabalho foi consolidado ao apresentar no Capítulo 4 a análise quantitativa envolvendo os aspectos de reusabilidade: classe, herança, tipos genéricos e CRTP.

No estudo quantitativo de classes, 81,48% dos tipos são classes concretas, comprovando sua grande utilização. Com relação as classes abstratas, esperava-se encontrar um número maior, devido a análise de alguns *frameworks*.

Na análise dos conceitos envolvendo herança, a métrica DIT mostrou que 69% dos tipos utilizam a herança de interface ou de implementação. Este número comprova que a herança é um recurso muito utilizado. Já a métrica NOC mostrou que apenas 10,05% dos tipos possuem filhos, ou seja, as superclasses estão concentradas em poucos tipos.

Ao analisar tipos genéricos, esperava-se encontrar um número maior de tipos que utiliza a programação genérica. Porém, menos de 3% dos tipos são genéricos. Em relação ao padrão CRTP, também era esperado um número maior de utilização. Como este padrão depende dos recursos de herança e tipos genéricos, sua utilização foi de apenas 109 tipos.

5.1 Contribuições

As principais contribuições deste trabalho são:

- Apresentação e análise dos dados envolvendo reusabilidade: classes, herança, tipos genéricos e CRTP. Esta análise utilizou-se de várias medições e das métricas DIT e NOC.
- O desenvolvimento de uma ferramenta que extrai metadados de arquivos de projeto Java e

armazena-os em banco de dados relacional. As fases de Tipos e Relações são executadas em múltiplos *threads*.

Como contribuições secundárias pode-se citar:

- Modelo de entidade para metadados de códigos fontes orientados a objetos que encapsulam tipos, heranças, parâmetros genéricos, restrições de parâmetros genéricos e invocações de tipos genéricos.
- Um algoritmo capaz de resolver ambiguidades sintáticas dos nomes das classes. Este algoritmo foi capaz de resolver ambiguidade de 99,95%.
- A ferramenta não apresentou problemas ao analisar 306.177 tipos armazenados em aproximadamente 5GB.
- Desenvolvimento de medições para análise de técnicas orientadas a objeto.

5.2 Trabalhos Futuros

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

- Extração de metadados sobre métodos e atributos de classes. Esses metadados podem ser utilizados para estabelecer padrões de comportamentos.
- Estabelecimento de métricas para as medições realizadas.
- Estender a funcionalidade para quantificar padrões de projeto envolvendo classes, atributos e métodos.
- Adicionar, na ferramenta *MetaPOO*, o suporte para extração e armazenamento de metadados de outras linguagens de programação orientada a objetos.

Referências Bibliográficas

AM, K.; SATWINDER, S.; S, K. K. *Evaluation and Metrication of Object Oriented System*. 2009.

ARNOLD, K.; GOSLING, J.; HOLMES, D. *Java(TM) Programming Language, The (4th Edition)*. [S.l.]: Addison-Wesley Professional, 2006. (Java (Addison-Wesley)). ISBN 9780321349804.

BASILI, V. R.; BRIAND, L.; MELO, W. L. A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering*, v. 22, p. 751–761, 1995.

BIEMAN, J. M.; ZHAO, J. X. Reuse through inheritance: a quantitative study of c++ software. *SIGSOFT Softw. Eng. Notes*, ACM, New York, NY, USA, v. 20, n. SI, p. 47–52, ago. 1995. ISSN 0163-5948.

BOOCH, G. et al. *Object-Oriented Analysis and Design with Applications (3rd Edition)*. Third. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 2007. ISBN 9780201895513.

BRETHERTON, F. P.; SINGLEY, P. T. Metadata: a user's view. In: *Proceedings of the 7th international conference on Scientific and Statistical Database Management*. Washington, DC, USA: IEEE Computer Society, 1994. (SSDBM'1994), p. 166–174. ISBN 0-1234-5678-2.

BRIAND, L. C. et al. *Exploring the Relationships between Design Measures and Software Quality in Object-Oriented Systems*. 1998.

CHIDAMBER, S. R.; KEMERER, C. F. A metrics suite for object oriented design. *IEEE Trans. Softw. Eng.*, IEEE Press, Piscataway, NJ, USA, v. 20, n. 6, p. 476–493, jun. 1994. ISSN 0098-5589.

COPLIEN, J. O. Curiously recurring template patterns. *C++ Rep.*, SIGS Publications, Inc., New York, NY, USA, v. 7, n. 2, p. 24–27, fev. 1995. ISSN 1040-6042.

CORPORATION, O. *OpenJDK*. 2013. Disponível em: <http://openjdk.java.net/>.

DAGENAIS, B.; HENDREN, L. Enabling static analysis for partial java programs. *SIGPLAN Not.*, ACM, New York, NY, USA, v. 43, n. 10, p. 313–328, out. 2008. ISSN 0362-1340.

DAHL, O.-J. *SIMULA 67 common base language*. [S.l.]: Norwegian computing center, 1968. ISBN B0007JZ9J6.

DAMASEVICIUS, R.; STUIKYS, V. Taxonomy of the fundamental concepts of metaprogramming. *124X Information Tecnhology and Control*, v. 37, p. 124–132, 2008.

DEITEL, H. M.; DEITEL, P. J. *Java how to program*. 6th edition. ed. Upper Saddle River, NJ, USA: Pearson Prentice Hall, 2005. ISBN 0131483986.

ECKEL, B. *Curiously Recurring Generic Pattern*. 2005. Disponível em: <http://www.artima.com/weblogs/viewpost.jsp?thread=133275>.

FOUNDATION, T. A. S. *Git at Apache*. 2010. Disponível em: <http://git.apache.org/>.

FOUNDATION, T. A. S. *The Apache Software Foundation*. 2012. Disponível em: <http://apache.org/>.

GAMMA, E. et al. *Design patterns: elements of reusable object-oriented software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN 0-201-63361-2.

GITHUB. *GitHub*. 2013. Disponível em: <https://github.com/>.

GITHUB. *Hibernate*. 2013. Disponível em: <https://github.com/hibernate>.

GITHUB. *Mirror of OpenJDK repositories*. 2013. Disponível em: <https://github.com/openjdk-mirror>.

GITHUB. *Spring Source*. 2013. Disponível em: <https://github.com/SpringSource>.

GOLDBERG, A.; ROBSON, D. *Smalltalk-80: the language and its implementation*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1983. ISBN 0-201-11371-6.

GOPIVOTAL. *Spring Source*. 2013. Disponível em: <http://www.springsource.org/>.

GOSLING, J. et al. *The Java(TM) Language Specification. Java SE 7 Edition*. [S.l.], 2013. Disponível em: <http://docs.oracle.com/javase/specs/jls/se7/html/index.html>.

HAHSLER, M. *A Quantitative Study of the Application of Design Patterns in Java*. 2003.

HUNER, K.; OTTO, B. The effect of using a semantic wiki for metadata management: A controlled experiment. In: *System Sciences, 2009. HICSS '09. 42nd Hawaii International Conference on*. [S.l.: s.n.], 2009. p. 1–9. ISSN 1530-1605.

HUSTON, B. The effects of design pattern application on metric scores. *Journal of Systems and Software*, v. 58, n. 3, p. 261–269, 2001.

LANGER, A. *Java Generics FAQs - Frequently Asked Questions*. 2013. Disponível em: <http://www.angelikalanger.com/GenericsFAQ/JavaGenericsFAQ.html>.

LIGUORI, R.; FINEGAN, E. *SCJA Sun Certified Java Associate Study Guide (Exam CX-310-019)*. [S.l.]: McGraw Hill Professional, 2009. (Certification Press). ISBN 9780071594844.

LOWE, W.; NOGA, M. L. Metaprogramming applied to web component deployment. *Electronic Notes in Theoretical Computer Science*, v. 65, p. 11, 2002.

MCCLURE, C. *Software reuse techniques: adding reuse to the system development process*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1997. ISBN 0-13-661000-5.

MCILROY, M. D. Mass produced software components. *Proc. NATO Conf. on Software Engineering, Garmisch, Germany*, Springer-Verlag, 1968.

METSKER, S. J.; WAKE, W. C. *Design Patterns in Java*. 2nd edition. ed. [S.l.]: Addison-Wesley Professional, 2006. ISBN 0321333020.

MEYER, B. *Object-Oriented Software Construction*. 2st. ed. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1997. ISBN 0-13-629155-4.

NYGAARD, K.; DAHL, O.-J. The development of the simula languages. *SIGPLAN Not.*, ACM, New York, NY, USA, v. 13, n. 8, p. 245–272, ago. 1978. ISSN 0362-1340.

OLIVEIRA, A. A. de et al. Metaj: An extensible environment for metaprogramming in java. *Journal of Universal Computer Science*, v. 10, n. 7, p. 872–891, jul 2004.

PRESS, N. *Understanding Metadata*. [S.l.]: National Information Standards Organization Press, 2004. ISBN 1-880124-62-9.

PRESSMAN, R. *Engenharia de software*. [S.l.]: McGraw-Hill, 2006. ISBN 9788586804571.

REDHAT. *JBoss Community*. 2013. Disponível em: <http://www.jboss.org/overview/>.

SAMETINGER, J. *Software engineering with reusable components*. New York, NY, USA: Springer-Verlag New York, Inc., 1997.

SEBESTA, R. W. *Concepts of Programming Languages*. 9th. ed. USA: Addison-Wesley Publishing Company, 2009. ISBN 0136073476.

SHEARD, T. Accomplishments and research challenges in meta-programming. In: *Proceedings of the 2nd international conference on Semantics, applications, and implementation of program generation*. Berlin, Heidelberg: Springer-Verlag, 2001. (SAIG'01), p. 2–44.

TIOBE. *TIOBE Programming Community Index for May 2013*. TIOBE Software, 2013. Http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html. Disponível em: <http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html>.

Apêndice

A

Consultas

Este capítulo apresenta as consultas realizadas no Capítulo 4 para a análise quantitativa dos dados.

O modelo relacional do banco de dados está representado a seguir:

Listing A.1: Modelo Relacional

Type = {id_type, classification, full_name, level_inheritance, path_file, project_name, unique_type, nogp, id_external_type(Type.id_type)}

Inheritance = {id_inheritance, type_inheritance, id_base_type(Type.id_type), id_derived_type(Type.id_type)}

GenericParameter = {id_generic_parameter, name, id_type(Type.id_type)}

Bound = {id_bound, discriminator, id_parameter_bound(GenericParameter.id_generic_parameter), id_type(Type.id_type), id_parameter(GenericParameter.id_generic_parameter)}

Bind = {id_bind, discriminator, vector, id_inheritance(Inheritance.id_inheritance), id_parameter(GenericParameter.id_generic_parameter), id_type(Type.id_type)}

Tipos

Listing A.2: Total de projetos

```
SELECT count(distinct project_name) FROM type;
```

Listing A.3: Total de Tipos

```
SELECT count(*) FROM type;
```

Listing A.4: Conjunto de dados do OpenJDK

```
--Quantidade de projetos do OpenJDK
SELECT count(distinct project_name) FROM type WHERE project_name like 'OpenJDK%'

--Quantidade de tipos do OpenJDK
SELECT count(distinct id_type) FROM type WHERE project_name like 'OpenJDK%';

--Quantidade de arquivos do OpenJDK
SELECT count(distinct path_file) FROM type WHERE project_name like 'OpenJDK%';
```

Listing A.5: Conjunto de dados do Apache

```
--Quantidade de projetos do Apache
SELECT count(distinct project_name) FROM type WHERE project_name like 'Apache%'

--Quantidade de tipos do Apache
SELECT count(distinct id_type) FROM type WHERE project_name like 'Apache%';

--Quantidade de arquivos do Apache
SELECT count(distinct path_file) FROM type WHERE project_name like 'Apache%';
```

Listing A.6: Conjunto de dados do Spring

```
--Quantidade de projetos do Spring
SELECT count(distinct project_name) FROM type WHERE project_name like 'Spring%'

--Quantidade de tipos do Spring
SELECT count(distinct id_type) FROM type WHERE project_name like 'Spring%';

--Quantidade de arquivos do Spring
SELECT count(distinct path_file) FROM type WHERE project_name like 'Spring%';
```

Listing A.7: Conjunto de dados do JBoss

```
--Quantidade de projetos do JBoss
SELECT count(distinct project_name) FROM type WHERE project_name like 'JBoss%'

--Quantidade de tipos do JBoss
SELECT count(distinct id_type) FROM type WHERE project_name like 'JBoss%';

--Quantidade de arquivos do JBoss
SELECT count(distinct path_file) FROM type WHERE project_name like 'JBoss%';
```

Listing A.8: Quantidade de tipos completos

```
SELECT classification, count(*) FROM type
  WHERE classification in ('INTERFACE', 'ABSTRACT', 'ENUM', 'CONCRETE')
  group by classification;
```

Listing A.9: Quantidade de tipos por classificação

```
SELECT classification, count(*) FROM type group by classification;
```

Listing A.10: Quantidade de tipos externos por classificação

```
SELECT classification, count(distinct id_type) FROM type
  WHERE id_external_type is null
  group by classification;
```

Listing A.11: Quantidade de tipos internos por classificação

```
SELECT classification, count(distinct id_type) FROM type
  WHERE id_external_type is not null
  group by classification;
```

Herança

Listing A.12: Quantidade de superclasses por classificação

```
SELECT t.classification, count(distinct t.id_type)
  FROM type t, inheritance i
  WHERE t.id_type = i.id_base_type
  group by t.classification
```

Listing A.13: Quantidade de subclasses por classificação

```
SELECT t.classification, count(distinct t.id_type)
  FROM type t, inheritance i
  WHERE t.id_type = i.id_derived_type
  group by t.classification
```

Listing A.14: Profundidade de herança por número de tipo

```
SELECT level_inheritance, count(id_type)
  FROM type
  group by level_inheritance
  order by level_inheritance
```

Listing A.15: Número de filhos por número de tipo

```
SELECT noc, count(id_type)
  FROM type
  WHERE noc is not null
  group by noc
  order by noc
```

Base Extensão Não Genérica

Listing A.16: Quantidade de superclasses concretas não genéricas estendidas por somente abstratas

```
SELECT count(distinct h1.id_base_type)
  FROM inheritance h1, type cD1, type cB1
  WHERE h1.type_inheritance = 'EXTENDS'
  AND h1.id_base_type = cB1.id_type
  AND h1.id_derived_type = cD1.id_type
  AND cB1.classification = 'CONCRETE'
  AND cD1.classification = 'ABSTRACT'
  AND h1.id_base_type not in (
    SELECT distinct h2.id_base_type
    FROM inheritance h2, type cD2, type cB2
    WHERE h2.type_inheritance = 'EXTENDS'
    AND h2.id_base_type = cB2.id_type
    AND h2.id_derived_type = cD2.id_type
```

```
    AND cB2.classification = 'CONCRETE'
    AND cD2.classification = 'CONCRETE'
  )
  AND h1.id_base_type not in (
    SELECT distinct id_type FROM generic_parameter
  );
```

Listing A.17: Quantidade de superclasses concretas não genéricas estendidas por somente concretas

```
SELECT count(distinct h1.id_base_type)
  FROM inheritance h1, type cD1, type cB1
  WHERE h1.type_inheritance = 'EXTENDS'
  AND h1.id_base_type = cB1.id_type
  AND h1.id_derived_type = cD1.id_type
  AND cB1.classification = 'CONCRETE'
  AND cD1.classification = 'CONCRETE'
  AND h1.id_base_type not in (
    SELECT distinct h2.id_base_type
    FROM inheritance h2, type cD2, type cB2
    WHERE h2.type_inheritance = 'EXTENDS'
    AND h2.id_base_type = cB2.id_type
    AND h2.id_derived_type = cD2.id_type
    AND cB2.classification = 'CONCRETE'
    AND cD2.classification = 'ABSTRACT'
  )
  AND h1.id_base_type not in (
    SELECT distinct id_type FROM generic_parameter
  );
```

Listing A.18: Quantidade de superclasses concretas não genéricas estendidas por abstratas e concretas

```
SELECT count(distinct h1.id_base_type)
  FROM inheritance h1, type cD1, type cB1
  WHERE h1.type_inheritance = 'EXTENDS'
  AND h1.id_base_type = cB1.id_type
  AND h1.id_derived_type = cD1.id_type
  AND cB1.classification = 'CONCRETE'
  AND cD1.classification = 'ABSTRACT'
  AND h1.id_base_type in (
    SELECT h2.id_base_type
    FROM inheritance h2, type cD2, type cB2
    WHERE h2.type_inheritance = 'EXTENDS'
    AND h2.id_base_type = cB2.id_type
    AND h2.id_derived_type = cD2.id_type
    AND cB2.classification = 'CONCRETE'
    AND cD2.classification = 'CONCRETE'
  )
  AND h1.id_base_type not in (
    SELECT distinct id_type FROM generic_parameter
  );
```

Listing A.19: Quantidade de superclasses abstratas não genéricas estendidas por somente abstratas

```
SELECT count(distinct h1.id_base_type)
  FROM inheritance h1, type cD1, type cB1
  WHERE h1.type_inheritance = 'EXTENDS'
  AND h1.id_base_type = cB1.id_type
```

```
  AND h1.id_derived_type = cD1.id_type
  AND cB1.classification = 'ABSTRACT'
  AND cD1.classification = 'ABSTRACT'
  AND h1.id_base_type not in (
    SELECT h2.id_base_type
    FROM inheritance h2, type cD2, type cB2
    WHERE h2.type_inheritance = 'EXTENDS'
    AND h2.id_base_type = cB2.id_type
    AND h2.id_derived_type = cD2.id_type
    AND cB2.classification = 'ABSTRACT'
    AND cD2.classification = 'CONCRETE'
  )
  AND h1.id_base_type not in (
    SELECT distinct id_type FROM generic_parameter
  );
```

Listing A.20: Quantidade de superclasses abstratas não genéricas estendidas por somente concretas

```
SELECT count(distinct h1.id_base_type)
  FROM inheritance h1, type cD1, type cB1
  WHERE h1.type_inheritance = 'EXTENDS'
  AND h1.id_base_type = cB1.id_type
  AND h1.id_derived_type = cD1.id_type
  AND cB1.classification = 'ABSTRACT'
  AND cD1.classification = 'CONCRETE'
  AND h1.id_base_type not in (
    SELECT h2.id_base_type
    FROM inheritance h2, type cD2, type cB2
    WHERE h2.type_inheritance = 'EXTENDS'
    AND h2.id_base_type = cB2.id_type
    AND h2.id_derived_type = cD2.id_type
    AND cB2.classification = 'ABSTRACT'
    AND cD2.classification = 'ABSTRACT'
  )
  AND h1.id_base_type not in (
    SELECT distinct id_type FROM generic_parameter
  );
```

Listing A.21: Quantidade de superclasses abstratas não genéricas estendidas por abstratas e concretas

```
SELECT count(distinct h1.id_base_type)
  FROM inheritance h1, type cD1, type cB1
  WHERE h1.type_inheritance = 'EXTENDS'
  AND h1.id_base_type = cB1.id_type
  AND h1.id_derived_type = cD1.id_type
  AND cB1.classification = 'ABSTRACT'
  AND cD1.classification = 'ABSTRACT'
  AND h1.id_base_type in (
  SELECT h2.id_base_type
    FROM inheritance h2, type cD2, type cB2
    WHERE h2.type_inheritance = 'EXTENDS'
    AND h2.id_base_type = cB2.id_type
    AND h2.id_derived_type = cD2.id_type
    AND cB2.classification = 'ABSTRACT'
    AND cD2.classification = 'CONCRETE'
  )
  AND h1.id_base_type not in (
    SELECT distinct id_type FROM generic_parameter
```

```
);
```

Listing A.22: Quantidade de interfaces não genéricas estendidas por interfaces

```
SELECT count(distinct h.id_base_type)
  FROM inheritance h, type cD, type cB
  WHERE h.type_inheritance = 'EXTENDS'
  AND h.id_base_type = cB.id_type
  AND h.id_derived_type = cD.id_type
  AND cD.classification = 'INTERFACE'
  AND cB.classification = 'INTERFACE'
  AND h.id_base_type not in (
    SELECT distinct id_type FROM generic_parameter
  );
```

Base Extensão Genérica

Listing A.23: Quantidade de superclasses concretas genéricas estendidas por somente abstratas

```
SELECT count(distinct h1.id_base_type)
  FROM inheritance h1, type cD1, type cB1
  WHERE h1.type_inheritance = 'EXTENDS'
  AND h1.id_base_type = cB1.id_type
  AND h1.id_derived_type = cD1.id_type
  AND cB1.classification = 'CONCRETE'
  AND cD1.classification = 'ABSTRACT'
  AND h1.id_base_type not in (
    SELECT distinct h2.id_base_type
    FROM inheritance h2, type cD2, type cB2
    WHERE h2.type_inheritance = 'EXTENDS'
    AND h2.id_base_type = cB2.id_type
    AND h2.id_derived_type = cD2.id_type
    AND cB2.classification = 'CONCRETE'
    AND cD2.classification = 'CONCRETE'
  )
  AND h1.id_base_type in (
    SELECT distinct id_type FROM generic_parameter
  );
```

Listing A.24: Quantidade de superclasses concretas genéricas estendidas por somente concretas

```
SELECT count(distinct h1.id_base_type)
  FROM inheritance h1, type cD1, type cB1
  WHERE h1.type_inheritance = 'EXTENDS'
  AND h1.id_base_type = cB1.id_type
  AND h1.id_derived_type = cD1.id_type
  AND cB1.classification = 'CONCRETE'
  AND cD1.classification = 'CONCRETE'
  AND h1.id_base_type not in (
    SELECT distinct h2.id_base_type
    FROM inheritance h2, type cD2, type cB2
    WHERE h2.type_inheritance = 'EXTENDS'
    AND h2.id_base_type = cB2.id_type
    AND h2.id_derived_type = cD2.id_type
    AND cB2.classification = 'CONCRETE'
    AND cD2.classification = 'ABSTRACT'
```

```
  )
  AND h1.id_base_type in (
    SELECT distinct id_type FROM generic_parameter
  );
```

Listing A.25: Quantidade de superclasses concretas genéricas estendidas por abstratas e concretas

```
SELECT count(distinct h1.id_base_type)
  FROM inheritance h1, type cD1, type cB1
  WHERE h1.type_inheritance = 'EXTENDS'
  AND h1.id_base_type = cB1.id_type
  AND h1.id_derived_type = cD1.id_type
  AND cB1.classification = 'CONCRETE'
  AND cD1.classification = 'ABSTRACT'
  AND h1.id_base_type in (
    SELECT h2.id_base_type
    FROM inheritance h2, type cD2, type cB2
    WHERE h2.type_inheritance = 'EXTENDS'
    AND h2.id_base_type = cB2.id_type
    AND h2.id_derived_type = cD2.id_type
    AND cB2.classification = 'CONCRETE'
    AND cD2.classification = 'CONCRETE'
  )
  AND h1.id_base_type in (
    SELECT distinct id_type FROM generic_parameter
  );
```

Listing A.26: Quantidade de superclasses abstratas genéricas estendidas por somente abstratas

```
SELECT count(distinct h1.id_base_type)
  FROM inheritance h1, type cD1, type cB1
  WHERE h1.type_inheritance = 'EXTENDS'
  AND h1.id_base_type = cB1.id_type
  AND h1.id_derived_type = cD1.id_type
  AND cB1.classification = 'ABSTRACT'
  AND cD1.classification = 'ABSTRACT'
  AND h1.id_base_type not in (
    SELECT h2.id_base_type
    FROM inheritance h2, type cD2, type cB2
    WHERE h2.type_inheritance = 'EXTENDS'
    AND h2.id_base_type = cB2.id_type
    AND h2.id_derived_type = cD2.id_type
    AND cB2.classification = 'ABSTRACT'
    AND cD2.classification = 'CONCRETE'
  )
  AND h1.id_base_type in (
    SELECT distinct id_type FROM generic_parameter
  );
```

Listing A.27: Quantidade de superclasses abstratas genéricas estendidas por somente concretas

```
SELECT count(distinct h1.id_base_type)
  FROM inheritance h1, type cD1, type cB1
  WHERE h1.type_inheritance = 'EXTENDS'
  AND h1.id_base_type = cB1.id_type
  AND h1.id_derived_type = cD1.id_type
  AND cB1.classification = 'ABSTRACT'
```

```
  AND cD1.classification = 'CONCRETE'
  AND h1.id_base_type not in (
    SELECT h2.id_base_type
    FROM inheritance h2, type cD2, type cB2
    WHERE h2.type_inheritance = 'EXTENDS'
    AND h2.id_base_type = cB2.id_type
    AND h2.id_derived_type = cD2.id_type
    AND cB2.classification = 'ABSTRACT'
    AND cD2.classification = 'ABSTRACT'
  )
  AND h1.id_base_type in (
    SELECT distinct id_type FROM generic_parameter
  );
```

Listing A.28: Quantidade de superclasses abstratas genéricas estendidas por abstratas e concretas

```
SELECT count(distinct h1.id_base_type)
  FROM inheritance h1, type cD1, type cB1
  WHERE h1.type_inheritance = 'EXTENDS'
  AND h1.id_base_type = cB1.id_type
  AND h1.id_derived_type = cD1.id_type
  AND cB1.classification = 'ABSTRACT'
  AND cD1.classification = 'ABSTRACT'
  AND h1.id_base_type in (
  SELECT h2.id_base_type
    FROM inheritance h2, type cD2, type cB2
    WHERE h2.type_inheritance = 'EXTENDS'
    AND h2.id_base_type = cB2.id_type
    AND h2.id_derived_type = cD2.id_type
    AND cB2.classification = 'ABSTRACT'
    AND cD2.classification = 'CONCRETE'
  )
  AND h1.id_base_type in (
    SELECT distinct id_type FROM generic_parameter
  );
```

Listing A.29: Quantidade de interfaces genéricas estendidas por interfaces

```
SELECT count(distinct h.id_base_type)
  FROM inheritance h, type cD, type cB
  WHERE h.type_inheritance = 'EXTENDS'
  AND h.id_base_type = cB.id_type
  AND h.id_derived_type = cD.id_type
  AND cD.classification = 'INTERFACE'
  AND cB.classification = 'INTERFACE'
  AND h.id_base_type in (
    SELECT distinct id_type FROM generic_parameter
  );
```

Derivada Extensão Não Genérica

Listing A.30: Quantidade de subclasses não genéricas por classificação

```
SELECT cD.classification , cB.classification , count(distinct h.id_derived_type)
  FROM inheritance h, type cD, type cB
  WHERE h.type_inheritance = 'EXTENDS'
  AND h.id_base_type = cB.id_type
```

```
  AND h.id_derived_type = cD.id_type
  AND ((cD.classification in ('CONCRETE', 'ABSTRACT')
  AND cB.classification in ('CONCRETE', 'ABSTRACT'))
  OR (cD.classification = 'INTERFACE'
  AND cB.classification = 'INTERFACE'))
  AND h.id_derived_type not in (SELECT distinct id_type from generic_parameter)
  group by cD.classification, cB.classification;
```

Derivada Extensão Genérica

Listing A.31: Quantidade de subclasses genéricas por classificação

```
SELECT cD.classification, cB.classification, count(distinct h.id_derived_type)
  FROM inheritance h, type cD, type cB
  WHERE h.type_inheritance = 'EXTENDS'
  AND h.id_base_type = cB.id_type
  AND h.id_derived_type = cD.id_type
  AND ((cD.classification in ('CONCRETE', 'ABSTRACT')
  AND cB.classification in ('CONCRETE', 'ABSTRACT'))
  OR (cD.classification = 'INTERFACE'
  AND cB.classification = 'INTERFACE'))
  AND h.id_derived_type in (SELECT distinct id_type from generic_parameter)
  group by cD.classification, cB.classification;
```

Base Implementação Não Genérica

Listing A.32: Quantidade de interfaces não genéricas implementadas por somente concretas

```
SELECT count(distinct h1.id_base_type)
  FROM inheritance h1, type cD1, type cB1
  WHERE h1.type_inheritance = 'IMPLEMENTS'
  AND h1.id_base_type = cB1.id_type
  AND h1.id_derived_type = cD1.id_type
  AND cB1.classification = 'INTERFACE'
  AND cD1.classification = 'CONCRETE'
  AND h1.id_base_type not in (
    SELECT distinct h2.id_base_type
    FROM inheritance h2, type cD2, type cB2
    WHERE h2.type_inheritance = 'IMPLEMENTS'
    AND h2.id_base_type = cB2.id_type
    AND h2.id_derived_type = cD2.id_type
    AND cB2.classification = 'INTERFACE'
    AND cD2.classification = 'ABSTRACT'
  )
  AND h1.id_base_type not in (
    SELECT distinct h3.id_base_type
    FROM inheritance h3, type cD3, type cB3
    WHERE h3.type_inheritance = 'IMPLEMENTS'
    AND h3.id_base_type = cB3.id_type
    AND h3.id_derived_type = cD3.id_type
    AND cB3.classification = 'INTERFACE'
    AND cD3.classification = 'ENUM'
  )
  AND h1.id_base_type not in (
    SELECT distinct id_type FROM generic_parameter
  );
```

Listing A.33: Quantidade de interfaces não genéricas implementadas por somente abstratas

```
SELECT count(distinct h1.id_base_type)
  FROM inheritance h1, type cD1, type cB1
  WHERE h1.type_inheritance = 'IMPLEMENTS'
  AND h1.id_base_type = cB1.id_type
  AND h1.id_derived_type = cD1.id_type
  AND cB1.classification = 'INTERFACE'
  AND cD1.classification = 'ABSTRACT'
  AND h1.id_base_type not in (
    SELECT distinct h2.id_base_type
    FROM inheritance h2, type cD2, type cB2
    WHERE h2.type_inheritance = 'IMPLEMENTS'
    AND h2.id_base_type = cB2.id_type
    AND h2.id_derived_type = cD2.id_type
    AND cB2.classification = 'INTERFACE'
    AND cD2.classification = 'CONCRETE'
  )
  AND h1.id_base_type not in (
    SELECT distinct h3.id_base_type
    FROM inheritance h3, type cD3, type cB3
    WHERE h3.type_inheritance = 'IMPLEMENTS'
    AND h3.id_base_type = cB3.id_type
    AND h3.id_derived_type = cD3.id_type
    AND cB3.classification = 'INTERFACE'
    AND cD3.classification = 'ENUM'
  )
  AND h1.id_base_type not in (
    SELECT distinct id_type FROM generic_parameter
  );
```

Listing A.34: Quantidade de interfaces não genéricas implementadas por somente enum

```
SELECT count(distinct h1.id_base_type)
  FROM inheritance h1, type cD1, type cB1
  WHERE h1.type_inheritance = 'IMPLEMENTS'
  AND h1.id_base_type = cB1.id_type
  AND h1.id_derived_type = cD1.id_type
  AND cB1.classification = 'INTERFACE'
  AND cD1.classification = 'ENUM'
  AND h1.id_base_type not in (
    SELECT distinct h2.id_base_type
    FROM inheritance h2, type cD2, type cB2
    WHERE h2.type_inheritance = 'IMPLEMENTS'
    AND h2.id_base_type = cB2.id_type
    AND h2.id_derived_type = cD2.id_type
    AND cB2.classification = 'INTERFACE'
    AND cD2.classification = 'CONCRETE'
  )
  AND h1.id_base_type not in (
    SELECT distinct h3.id_base_type
    FROM inheritance h3, type cD3, type cB3
    WHERE h3.type_inheritance = 'IMPLEMENTS'
    AND h3.id_base_type = cB3.id_type
    AND h3.id_derived_type = cD3.id_type
    AND cB3.classification = 'INTERFACE'
    AND cD3.classification = 'ABSTRACT'
  )
  AND h1.id_base_type not in (
    SELECT distinct id_type FROM generic_parameter
  );
```

Listing A.35: Quantidade de interfaces não genéricas implementadas por somente concreta e abstrata

```
SELECT count(distinct h1.id_base_type)
  FROM inheritance h1, type cD1, type cB1
  WHERE h1.type_inheritance = 'IMPLEMENTS'
  AND h1.id_base_type = cB1.id_type
  AND h1.id_derived_type = cD1.id_type
  AND cB1.classification = 'INTERFACE'
  AND cD1.classification = 'ABSTRACT'
  AND h1.id_base_type in (
    SELECT distinct h2.id_base_type
    FROM inheritance h2, type cD2, type cB2
    WHERE h2.type_inheritance = 'IMPLEMENTS'
    AND h2.id_base_type = cB2.id_type
    AND h2.id_derived_type = cD2.id_type
    AND cB2.classification = 'INTERFACE'
    AND cD2.classification = 'CONCRETE'
  )
  AND h1.id_base_type not in (
    SELECT distinct h3.id_base_type
    FROM inheritance h3, type cD3, type cB3
    WHERE h3.type_inheritance = 'IMPLEMENTS'
    AND h3.id_base_type = cB3.id_type
    AND h3.id_derived_type = cD3.id_type
    AND cB3.classification = 'INTERFACE'
    AND cD3.classification = 'ENUM'
  )
  AND h1.id_base_type not in (
    SELECT distinct id_type FROM generic_parameter
  );
```

Listing A.36: Quantidade de interfaces não genéricas implementadas por concreta e enum

```
SELECT count(distinct h1.id_base_type)
  FROM inheritance h1, type cD1, type cB1
  WHERE h1.type_inheritance = 'IMPLEMENTS'
  AND h1.id_base_type = cB1.id_type
  AND h1.id_derived_type = cD1.id_type
  AND cB1.classification = 'INTERFACE'
  AND cD1.classification = 'ENUM'
  AND h1.id_base_type in (
    SELECT distinct h2.id_base_type
    FROM inheritance h2, type cD2, type cB2
    WHERE h2.type_inheritance = 'IMPLEMENTS'
    AND h2.id_base_type = cB2.id_type
    AND h2.id_derived_type = cD2.id_type
    AND cB2.classification = 'INTERFACE'
    AND cD2.classification = 'CONCRETE'
  )
  AND h1.id_base_type not in (
    SELECT distinct h3.id_base_type
    FROM inheritance h3, type cD3, type cB3
    WHERE h3.type_inheritance = 'IMPLEMENTS'
    AND h3.id_base_type = cB3.id_type
    AND h3.id_derived_type = cD3.id_type
    AND cB3.classification = 'INTERFACE'
    AND cD3.classification = 'ABSTRACT'
  )
  AND h1.id_base_type not in (
    SELECT distinct id_type FROM generic_parameter
```

```
);
```

Listing A.37: Quantidade de interfaces não genéricas implementadas por somente abstrata e enum

```
SELECT count(distinct h1.id_base_type)
  FROM inheritance h1, type cD1, type cB1
  WHERE h1.type_inheritance = 'IMPLEMENTS'
  AND h1.id_base_type = cB1.id_type
  AND h1.id_derived_type = cD1.id_type
  AND cB1.classification = 'INTERFACE'
  AND cD1.classification = 'ENUM'
  AND h1.id_base_type in (
    SELECT distinct h2.id_base_type
    FROM inheritance h2, type cD2, type cB2
    WHERE h2.type_inheritance = 'IMPLEMENTS'
    AND h2.id_base_type = cB2.id_type
    AND h2.id_derived_type = cD2.id_type
    AND cB2.classification = 'INTERFACE'
    AND cD2.classification = 'ABSTRACT'
  )
  AND h1.id_base_type not in (
    SELECT distinct h3.id_base_type
    FROM inheritance h3, type cD3, type cB3
    WHERE h3.type_inheritance = 'IMPLEMENTS'
    AND h3.id_base_type = cB3.id_type
    AND h3.id_derived_type = cD3.id_type
    AND cB3.classification = 'INTERFACE'
    AND cD3.classification = 'CONCRETE'
  )
  AND h1.id_base_type not in (
    SELECT distinct id_type FROM generic_parameter
  );
```

Listing A.38: Quantidade de interfaces não genéricas implementadas por concreta, abstrata e enum

```
SELECT count(distinct h1.id_base_type)
  FROM inheritance h1, type cD1, type cB1
  WHERE h1.type_inheritance = 'IMPLEMENTS'
  AND h1.id_base_type = cB1.id_type
  AND h1.id_derived_type = cD1.id_type
  AND cB1.classification = 'INTERFACE'
  AND cD1.classification = 'ABSTRACT'
  AND h1.id_base_type in (
    SELECT distinct h2.id_base_type
    FROM inheritance h2, type cD2, type cB2
    WHERE h2.type_inheritance = 'IMPLEMENTS'
    AND h2.id_base_type = cB2.id_type
    AND h2.id_derived_type = cD2.id_type
    AND cB2.classification = 'INTERFACE'
    AND cD2.classification = 'CONCRETE'
  )
  AND h1.id_base_type in (
    SELECT distinct h3.id_base_type
    FROM inheritance h3, type cD3, type cB3
    WHERE h3.type_inheritance = 'IMPLEMENTS'
    AND h3.id_base_type = cB3.id_type
    AND h3.id_derived_type = cD3.id_type
    AND cB3.classification = 'INTERFACE'
```

```
    AND cD3.classification = 'ENUM'
  )
  AND h1.id_base_type not in (
    SELECT distinct id_type FROM generic_parameter
  );
```

Base Implementação Genérica

Listing A.39: Quantidade de interfaces genéricas implementadas por somente concretas

```
SELECT count(distinct h1.id_base_type)
  FROM inheritance h1, type cD1, type cB1
  WHERE h1.type_inheritance = 'IMPLEMENTS'
  AND h1.id_base_type = cB1.id_type
  AND h1.id_derived_type = cD1.id_type
  AND cB1.classification = 'INTERFACE'
  AND cD1.classification = 'CONCRETE'
  AND h1.id_base_type not in (
    SELECT distinct h2.id_base_type
    FROM inheritance h2, type cD2, type cB2
    WHERE h2.type_inheritance = 'IMPLEMENTS'
    AND h2.id_base_type = cB2.id_type
    AND h2.id_derived_type = cD2.id_type
    AND cB2.classification = 'INTERFACE'
    AND cD2.classification = 'ABSTRACT'
  )
  AND h1.id_base_type not in (
    SELECT distinct h3.id_base_type
    FROM inheritance h3, type cD3, type cB3
    WHERE h3.type_inheritance = 'IMPLEMENTS'
    AND h3.id_base_type = cB3.id_type
    AND h3.id_derived_type = cD3.id_type
    AND cB3.classification = 'INTERFACE'
    AND cD3.classification = 'ENUM'
  )
  AND h1.id_base_type in (
    SELECT distinct id_type FROM generic_parameter
  );
```

Listing A.40: Quantidade de interfaces genéricas implementadas por somente abstratas

```
SELECT count(distinct h1.id_base_type)
  FROM inheritance h1, type cD1, type cB1
  WHERE h1.type_inheritance = 'IMPLEMENTS'
  AND h1.id_base_type = cB1.id_type
  AND h1.id_derived_type = cD1.id_type
  AND cB1.classification = 'INTERFACE'
  AND cD1.classification = 'ABSTRACT'
  AND h1.id_base_type not in (
    SELECT distinct h2.id_base_type
    FROM inheritance h2, type cD2, type cB2
    WHERE h2.type_inheritance = 'IMPLEMENTS'
    AND h2.id_base_type = cB2.id_type
    AND h2.id_derived_type = cD2.id_type
    AND cB2.classification = 'INTERFACE'
    AND cD2.classification = 'CONCRETE'
  )
  AND h1.id_base_type not in (
```

```
    SELECT distinct h3.id_base_type
    FROM inheritance h3, type cD3, type cB3
    WHERE h3.type_inheritance = 'IMPLEMENTS'
    AND h3.id_base_type = cB3.id_type
    AND h3.id_derived_type = cD3.id_type
    AND cB3.classification = 'INTERFACE'
    AND cD3.classification = 'ENUM'
  )
  AND h1.id_base_type in (
    SELECT distinct id_type FROM generic_parameter
  );
```

Listing A.41: Quantidade de interfaces genéricas implementadas por somente enum

```
SELECT count(distinct h1.id_base_type)
  FROM inheritance h1, type cD1, type cB1
  WHERE h1.type_inheritance = 'IMPLEMENTS'
  AND h1.id_base_type = cB1.id_type
  AND h1.id_derived_type = cD1.id_type
  AND cB1.classification = 'INTERFACE'
  AND cD1.classification = 'ENUM'
  AND h1.id_base_type not in (
    SELECT distinct h2.id_base_type
    FROM inheritance h2, type cD2, type cB2
    WHERE h2.type_inheritance = 'IMPLEMENTS'
    AND h2.id_base_type = cB2.id_type
    AND h2.id_derived_type = cD2.id_type
    AND cB2.classification = 'INTERFACE'
    AND cD2.classification = 'CONCRETE'
  )
  AND h1.id_base_type not in (
    SELECT distinct h3.id_base_type
    FROM inheritance h3, type cD3, type cB3
    WHERE h3.type_inheritance = 'IMPLEMENTS'
    AND h3.id_base_type = cB3.id_type
    AND h3.id_derived_type = cD3.id_type
    AND cB3.classification = 'INTERFACE'
    AND cD3.classification = 'ABSTRACT'
  )
  AND h1.id_base_type in (
    SELECT distinct id_type FROM generic_parameter
  );
```

Listing A.42: Quantidade de interfaces genéricas implementadas por somente concreta e abstrata

```
SELECT count(distinct h1.id_base_type)
  FROM inheritance h1, type cD1, type cB1
  WHERE h1.type_inheritance = 'IMPLEMENTS'
  AND h1.id_base_type = cB1.id_type
  AND h1.id_derived_type = cD1.id_type
  AND cB1.classification = 'INTERFACE'
  AND cD1.classification = 'ABSTRACT'
  AND h1.id_base_type in (
    SELECT distinct h2.id_base_type
    FROM inheritance h2, type cD2, type cB2
    WHERE h2.type_inheritance = 'IMPLEMENTS'
    AND h2.id_base_type = cB2.id_type
    AND h2.id_derived_type = cD2.id_type
    AND cB2.classification = 'INTERFACE'
    AND cD2.classification = 'CONCRETE'
```

```
  )
  AND h1.id_base_type not in (
    SELECT distinct h3.id_base_type
    FROM inheritance h3, type cD3, type cB3
    WHERE h3.type_inheritance = 'IMPLEMENTS'
    AND h3.id_base_type = cB3.id_type
    AND h3.id_derived_type = cD3.id_type
    AND cB3.classification = 'INTERFACE'
    AND cD3.classification = 'ENUM'
  )
  AND h1.id_base_type in (
    SELECT distinct id_type FROM generic_parameter
  );
```

Listing A.43: Quantidade de interfaces genéricas implementadas por concreta e enum

```
SELECT count(distinct h1.id_base_type)
  FROM inheritance h1, type cD1, type cB1
  WHERE h1.type_inheritance = 'IMPLEMENTS'
  AND h1.id_base_type = cB1.id_type
  AND h1.id_derived_type = cD1.id_type
  AND cB1.classification = 'INTERFACE'
  AND cD1.classification = 'ENUM'
  AND h1.id_base_type in (
    SELECT distinct h2.id_base_type
    FROM inheritance h2, type cD2, type cB2
    WHERE h2.type_inheritance = 'IMPLEMENTS'
    AND h2.id_base_type = cB2.id_type
    AND h2.id_derived_type = cD2.id_type
    AND cB2.classification = 'INTERFACE'
    AND cD2.classification = 'CONCRETE'
  )
  AND h1.id_base_type not in (
    SELECT distinct h3.id_base_type
    FROM inheritance h3, type cD3, type cB3
    WHERE h3.type_inheritance = 'IMPLEMENTS'
    AND h3.id_base_type = cB3.id_type
    AND h3.id_derived_type = cD3.id_type
    AND cB3.classification = 'INTERFACE'
    AND cD3.classification = 'ABSTRACT'
  )
  AND h1.id_base_type in (
    SELECT distinct id_type FROM generic_parameter
  );
```

Listing A.44: Quantidade de interfaces genéricas implementadas por somente abstrata e enum

```
SELECT count(distinct h1.id_base_type)
  FROM inheritance h1, type cD1, type cB1
  WHERE h1.type_inheritance = 'IMPLEMENTS'
  AND h1.id_base_type = cB1.id_type
  AND h1.id_derived_type = cD1.id_type
  AND cB1.classification = 'INTERFACE'
  AND cD1.classification = 'ENUM'
  AND h1.id_base_type in (
    SELECT distinct h2.id_base_type
    FROM inheritance h2, type cD2, type cB2
    WHERE h2.type_inheritance = 'IMPLEMENTS'
    AND h2.id_base_type = cB2.id_type
    AND h2.id_derived_type = cD2.id_type
    AND cB2.classification = 'INTERFACE'
```

```
    AND cD2.classification = 'ABSTRACT'
  )
  AND h1.id_base_type not in (
    SELECT distinct h3.id_base_type
    FROM inheritance h3, type cD3, type cB3
    WHERE h3.type_inheritance = 'IMPLEMENTS'
    AND h3.id_base_type = cB3.id_type
    AND h3.id_derived_type = cD3.id_type
    AND cB3.classification = 'INTERFACE'
    AND cD3.classification = 'CONCRETE'
  )
  AND h1.id_base_type in (
    SELECT distinct id_type FROM generic_parameter
  );
```

Listing A.45: Quantidade de interfaces genéricas implementadas por concreta, abstrata e enum

```
SELECT count(distinct h1.id_base_type)
  FROM inheritance h1, type cD1, type cB1
  WHERE h1.type_inheritance = 'IMPLEMENTS'
  AND h1.id_base_type = cB1.id_type
  AND h1.id_derived_type = cD1.id_type
  AND cB1.classification = 'INTERFACE'
  AND cD1.classification = 'ABSTRACT'
  AND h1.id_base_type in (
    SELECT distinct h2.id_base_type
    FROM inheritance h2, type cD2, type cB2
    WHERE h2.type_inheritance = 'IMPLEMENTS'
    AND h2.id_base_type = cB2.id_type
    AND h2.id_derived_type = cD2.id_type
    AND cB2.classification = 'INTERFACE'
    AND cD2.classification = 'CONCRETE'
  )
  AND h1.id_base_type in (
    SELECT distinct h3.id_base_type
    FROM inheritance h3, type cD3, type cB3
    WHERE h3.type_inheritance = 'IMPLEMENTS'
    AND h3.id_base_type = cB3.id_type
    AND h3.id_derived_type = cD3.id_type
    AND cB3.classification = 'INTERFACE'
    AND cD3.classification = 'ENUM'
  )
  AND h1.id_base_type in (
    SELECT distinct id_type FROM generic_parameter
  );
```

Derivada Implementação Não Genérica

Listing A.46: Quantidade de subclasses não genéricas por classificação implementando interfaces

```
SELECT cD.classification, count(distinct id_derived_type)
  FROM inheritance h, type cD, type cB
  WHERE type_inheritance = 'IMPLEMENTS'
  AND h.id_derived_type = cD.id_type
  AND h.id_base_type = cB.id_type
  AND cD.classification in ('CONCRETE', 'ABSTRACT', 'ENUM')
```

```
  AND cB.classification = 'INTERFACE'
  AND h.id_derived_type not in (SELECT distinct id_type FROM generic_parameter)
  group by cD.classification;
```

Derivada Implementação Genérica

Listing A.47: Quantidade de subclasses genéricas por classificação implementando interfaces

```
SELECT cD.classification, count(distinct id_derived_type)
  FROM inheritance h, type cD, type cB
  WHERE type_inheritance = 'IMPLEMENTS'
  AND h.id_derived_type = cD.id_type
  AND h.id_base_type = cB.id_type
  AND cD.classification in ('CONCRETE', 'ABSTRACT', 'ENUM')
  AND cB.classification = 'INTERFACE'
  AND h.id_derived_type in (SELECT distinct id_type FROM generic_parameter)
  group by cD.classification;
```

Tipos Genéricos

Listing A.48: Total de tipos não genéricos

```
SELECT count(distinct t.id_type) FROM type t
  WHERE t.id_type not in (
    SELECT distinct g.id_type FROM generic_parameter g
  );
```

Listing A.49: Total de tipos genéricos

```
SELECT count(distinct g.id_type) FROM generic_parameter g;
```

Listing A.50: Quantidade de tipos genéricos por classificação

```
SELECT t.classification, count(distinct t.id_type)
  FROM generic_parameter g, type t
  WHERE g.id_type = t.id_type
  group by t.classification;
```

Listing A.51: Quantidade de parâmetros genéricos por restrição

```
SELECT discriminator, count(distinct b.id_parameter_bound)
  FROM bound b
  group by discriminator;
```

Listing A.52: Quantidade de tipos por invocação

```
SELECT b.discriminator, count(distinct b.id_inheritance)
  FROM bind b
  group by b.discriminator;
```

Listing A.53: Número de parâmetros genéricos por tipo.

```
SELECT nogp, count(id_type) FROM type
  WHERE nogp is not null
  group by nogp
  order by nogp;
```

CRTP

Listing A.54: Quantidade de CRTP por classificação

```
SELECT c.classification, count(distinct c.id_type)
  FROM inheritance h, bind b, type c
  WHERE b.id_inheritance = h.id_inheritance
  AND h.id_base_type = c.id_type
  AND b.id_type = h.id_derived_type
  AND c.classification in ('ABSTRACT', 'CONCRETE', 'INTERFACE', 'ENUM')
  group by c.classification;
```

Listing A.55: Quantidade de CRTP explícito por classificação

```
SELECT classification, count(distinct pg.id_type)
  FROM generic_parameter pg, bound b, type c
  WHERE b.id_parameter_bound = pg.id_generic_parameter
  AND c.id_type = pg.id_type
  AND b.id_type = pg.id_type
  group by c.classification;
```

Listing A.56: Quantidade de CRTP implícito por classificação

```
SELECT c.classification, count(distinct c.id_type)
  FROM inheritance h, bind b, type c
  WHERE b.id_inheritance = h.id_inheritance
  AND h.id_base_type = c.id_type
  AND b.id_type = h.id_derived_type
  AND c.classification in ('ABSTRACT', 'CONCRETE', 'INTERFACE')
  AND c.id_type not in (
    SELECT distinct pg.id_type FROM generic_parameter pg, bound b
        WHERE b.id_parameter_bound = pg.id_generic_parameter
        AND b.id_type = pg.id_type
  )
  group by c.classification;
```