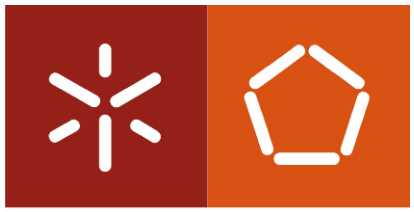

Universidade do Minho
Departamento de Informática

Rui Miguel Silva Couto

Inferência de PSM/PIM e padrões de concepção a partir de código Java

Dissertação de Mestrado
Mestrado em Engenharia Informática

Trabalho realizado sob a orientação do
Professor Doutor António Nestor Ribeiro

Outubro de 2011

Agradecimentos

A elaboração desta dissertação não tinha sido possível sem a ajuda de diversas pessoas às quais gostaria de agradecer.

Ao meu orientador, Prof. António Nestor Ribeiro por todo o trabalho de supervisão e disponibilidade prestada sem os quais este trabalho não seria possível.

Agradeço ainda ao Prof. António Costa por toda a disponibilidade e conselhos, bem como ao Prof. Yann-Gaël Guéhéneuc pela sua prestação.

Aos meus colegas, em especial ao Nuno Abreu e ao Nuno Azevedo pelo companheirismo, ajuda e conselhos, ao Jaime Neto e ao Pedro Branco por toda a disponibilidade e auxílio.

Um agradecimento especial à Isabel Santos por todo o apoio sem o qual esta tarefa teria sido muito mais complicada.

Por fim mas não menos importante à minha família por toda a compreensão e paciência para mim, em especial aos meus pais e irmão.

Pelo apoio que todas estas pessoas me deram directa ou indirectamente lhes dedico este trabalho.

Resumo

Devido ao constante crescimento do número de plataformas e linguagens disponíveis a quem desenvolve software, estamos a atingir elevados níveis de complexidade. Este facto levou à necessidade de criar novas técnicas de desenvolvimento de software que permitam facilitar o processo, abstraindo a complexidade que lhe é subjacente. O *Object Management Group* (OMG) apresentou uma solução para esse problema definindo a *Model Driven Engineering* (MDE). A MDE baseia o seu processo de desenvolvimento na definição e transformação de modelos, nomeadamente modelos independentes da computação (CIM), independentes da plataforma (PIM) e dependentes da plataforma (PSM). A *Unified Modelling Language* (UML) permite a criação de *Platform Specific Models* (PSM) e *Platform Independent Models* (PIM), ou até mesmo diagramas mais específicos como diagramas de classe.

Alguns anos antes de surgir a MDE, Erich Gamma catalogou um conjunto de boas formas de produzir software. Estas formas denominam-se padrões de concepção e a sua importância já foi amplamente reconhecida. Estes padrões são úteis não só no desenvolvimento, mas também no processo de análise de software.

Baseado em programas Java, apresenta-se neste documento a viabilidade de abstrair código fonte em modelos do MDE. O código será transformado em diagramas PIM e PSM, nos quais serão inferidos *Design Patterns* (ou padrões de concepção). Para tal será especificada uma ferramenta que pretende disponibilizar essas funcionalidades. Implementada sob a forma de *plugin*, baseia-se no mapeamento de informação num metamodelo como representação intermédia da informação. Posteriormente disponibiliza a abstracção de informação por transformação de modelos PSM em PIM com base nessa representação. A inferência de padrões é possível devido à representação de informação contida no metamodelo em factos Prolog, que serão a base para o processo de pesquisa de padrões. Esta ferramenta de engenharia reversa permite que o processo seja iniciado em código fonte (e não em modelos, como descrito pelo MDE).

Palavras-Chave: Java, MDA, MDE, Padrões de Concepção, PIM, PSM, UML.

Abstract

Due to the constant increase in the number of platforms and languages available to software developers, we are reaching high levels of complexity. To abstract the complexity that underlies it, the development of new techniques is needed. A solution to this problem was presented by the Object Management Group (OMG) by specifying the Model Driven Engineering (MDE). The MDE bases its development process in models definition and transformation, specifically Computation Independent Models (CIM), Platform Independent Models (PIM) and Platform Specific Models (PSM). The Unified Model Language (UML) allows to create Platform Specific Models (PSM) and Platform Independent Models (PIM), or even more specific diagrams as class diagrams.

Some years before the MDE appearance, Erich Gamma *et al.* catalogued a set of good practices to produce software. These means are called design patterns, and its importance has already been widely recognized. These patterns are not only useful in software developing, but also in the software analysis process.

Based on Java programs, this document presents the feasibility to transform source code on MDE models. This code will be transformed into PIM and PSM diagrams, in which will be inferred design patterns. As such, a tool which implements these functionalities will be specified. Implemented as a plugin, it maps the information on a metamodel to obtain an intermediate information representation. Based on that representation it provides information abstraction, by transforming PSM on PIM models. The design patterns inference is possible due to the representation of information contained in the metamodel as Prolog facts, which will be the basis for the design pattern search. Being a reverse engineering process, it allows the process to be started from the source code (and not in models, as predicted by MDE).

Keywords: Java, MDA, MDE, Design Patterns, PIM, PSM, UML.

Conteúdo

1 Introdução **1**

1.1 Contexto 2

1.1.1 A *Model Driven Engineering* 3

1.1.2 A *Model Driven Architecture* 4

1.1.3 *Design patterns* ou Padrões de concepção 5

1.2 Desafios de implementação 6

1.3 Motivação 6

1.4 Objectivos 8

1.5 Estrutura da dissertação 8

2 Estado da arte e trabalho relacionado **11**

2.1 Introdução 11

2.2 *Design patterns* ou Padrões de concepção 11

2.3 A *Model Driven Architecture* 12

2.3.1 A *framework* MDA 12

2.3.2 Artefactos da MDA 14

2.3.3 Processo de desenvolvimento segundo a MDA 17

2.3.4 O processo MDA 20

2.3.5 Afinações no processo de conversão 21

2.3.6 MDA inverso 22

2.3.7 Implementações da MDA 23

2.4 Inferência de código e modelos 25

2.5 Inferência de padrões 27

2.6 Análise das ferramentas disponíveis 30

2.6.1 Inferência de modelos 31

2.6.2 Inferência de padrões de concepção 34

2.7 Resumo 35

3 Análise das ferramentas **37**

3.1 Introdução . . . 37
3.2 ArgoUML . . . 38
3.2.1 Análise de código . . . 38
3.2.2 Reconhecimento dos elementos . . . 38
3.2.3 Outras funcionalidades . . . 40
3.3 Fujaba (Reclipse) . . . 40
3.3.1 Análise de código . . . 41
3.3.2 Reconhecimento dos elementos . . . 42
3.3.3 Outras funcionalidades . . . 43
3.4 jGRASP . . . 43
3.4.1 Análise de código . . . 44
3.4.2 Reconhecimento dos elementos . . . 44
3.4.3 Outras funcionalidades . . . 45
3.5 Ptidej . . . 45
3.5.1 Análise de código . . . 46
3.5.2 Reconhecimento dos elementos . . . 46
3.5.3 Outras funcionalidades . . . 47
3.6 Visual Paradigm . . . 48
3.6.1 Análise de código . . . 48
3.6.2 Reconhecimento dos elementos . . . 49
3.6.3 Outras funcionalidades . . . 49
3.7 Resumo . . . 49

4 Desafios e cenários de utilização 51
4.1 Introdução . . . 51
4.2 Enquadramento do problema . . . 51
4.2.1 Código fonte para PSM . . . 52
4.2.2 PSM para PIM . . . 54
4.2.3 Inferência de padrões num PSM . . . 56
4.3 Cenários de utilização . . . 59
4.3.1 Produção de um PSM . . . 59
4.3.2 Abstracção de um PIM . . . 60
4.3.3 Inferência de padrões . . . 62
4.4 Resumo . . . 63

5 A ferramenta MapIt 65
5.1 Introdução . . . 65
5.2 Proposta de resolução dos problemas . . . 65

5.2.1 Código fonte para PSM 66
5.2.2 PSM para PIM 72
5.2.3 Inferência de padrões num PSM 76
5.2.4 Interface gráfica 79
5.2.5 Interligação dos diferentes módulos 81
5.3 Proposta de implementação 82
5.3.1 *JavaParser Simple Interface* (JPSI) 82
5.3.2 *GNUProlog Java Simple Interface* (GPJaSI) 84
5.3.3 *System Analysis Interface* (SAI) 85
5.3.4 *Plugin* NetBeans 88
5.4 Resumo 93

6 Caso de estudo **95**
6.1 Introdução 95
6.2 O caso em estudo 96
6.2.1 Agenda 96
6.2.2 JHotDraw 96
6.3 Descrição detalhada da aplicação da ferramenta 98
6.3.1 Importar ficheiros Java 98
6.3.2 Pré-processamento dos ficheiros 99
6.3.3 Diagrama de Classes (PSM) 101
6.3.4 Diagrama PIM 102
6.3.5 Inferência de padrões 104
6.3.6 Detalhes de implementação 112
6.4 Resultados obtidos 112
6.4.1 Resumo dos módulos 113
6.4.2 Comparação com outras ferramentas 114
6.5 Resumo 118

7 Conclusões **121**
7.1 Introdução 121
7.2 Avaliação do trabalho realizado 122
7.3 Trabalho futuro 124
7.3.1 Extensão a outras linguagens 124
7.3.2 Extensão do catálogo de padrões 124
7.3.3 Extensão da ferramenta a *round-trip* 124
7.3.4 Implementação do *plugin* em outros suportes 125
7.3.5 Melhoramento da representação visual 125

A Padrões de concepção **137**
A.1 Catálogo de padrões . . . 137
A.2 Padrões de concepção . . . 137
A.2.1 Abstract Factory . . . 138
A.2.2 Adapter . . . 138
A.2.3 Bridge . . . 139
A.2.4 Builder . . . 140
A.2.5 Chain of Responsability . . . 140
A.2.6 Command . . . 141
A.2.7 Composite . . . 141
A.2.8 Decorator . . . 142
A.2.9 Facade . . . 143
A.2.10 Factory Method . . . 143
A.2.11 Flyweight . . . 144
A.2.12 Interpreter . . . 145
A.2.13 Iterator . . . 145
A.2.14 Mediator . . . 146
A.2.15 Memento . . . 146
A.2.16 Observer . . . 147
A.2.17 Prototype . . . 147
A.2.18 Proxy . . . 148
A.2.19 Singleton . . . 149
A.2.20 State . . . 149
A.2.21 Strategy . . . 150
A.2.22 Template Method . . . 150
A.2.23 Visitor . . . 151

B Catálogo personalizado de padrões **153**
B.1 Catálogo . . . 153

Acrónimos

API – *Application Programming Interfaces*

AS – *Action Semantics*

CASE – *Computer-Aided Software Engineering*

CIM – *Computation Independent Model*

CWM – *Common Warehouse Metamodel*

GPJaSI – *GNUProlog Java Simple Interface*

IDE – *Integrated Development Environment*

JDI – *Java Debug Interface*

JPSI – *JavaParser Simple Interface*

JRE – *Java Runtime Environment*

MDA – *Model Driven Architecture*

MDE – *Model Driven Engineering*

MFE – *Model Filter Engine*

MOF – *Meta Object Facility*

MapIt – *Model and Patterns Inferring Tool*

OCL – *Object Constraint Language*

OMG – *Object Management Group*

ORM – *Object Relational Mapping*

PIM – *Platform Independent Model*

PSM – *Platform Specific Model*

QVT – *Query Views and Transformations Standards*

SAI – *System Analysis Interface*

SOUL – *Smalltalk Open Unification Language*

SQL – *Structured Query Language*

UML – *Unified Modelling Language*

XMI – *XML Metadata Interchange*

XML – *Extensible Markup Language*

Lista de Tabelas

5.1 Resumo das regras de transformação PIM em PSM, com base em *MDA Explained* [49]. 91
5.2 Resumo das regras de transformação PSM em PIM, com base na Tabela 5.1. 91

Lista de Figuras

1.1 Simplificação do processo MDA, modelado em UML 4
1.2 Padrão de concepção Composite, adaptado de [25]. 5

2.1 Processo MDA. 13
2.2 Relações entre metamodelos, metalinguagem, modelo e linguagem. 15
2.3 A *framework* MDA. 15
2.4 Processo de desenvolvimento tradicional. 18
2.5 Processo de desenvolvimento segundo a MDA. 19

3.1 Diagrama de classes do software Agenda. 38
3.2 Elementos UML carregados pela ferramenta ArgoUML. 39
3.3 Diagrama de classes gerado pela ferramenta ArgoUML. 39
3.4 Propriedades da classe `Main` inferidas pela ferramenta ArgoUML. . 40
3.5 Editor de código fonte da ferramenta ArgoUML. 40
3.6 Diagrama de classes inferido pelo Reclipse. 41
3.7 Diagrama de classes inferido pelo Fujaba. 42
3.8 Padrões de concepção inferidos pelo Fujaba. 43
3.9 Outros padrões inferidos pelo Fujaba. 44
3.10 Diagrama de classes gerado pelo jGRASP. 44
3.11 Parte do código da classe `Main`, visualizado no editor do jGRASP. 45
3.12 Listagem de entidades reconhecidas pelo Ptidej. 46
3.13 Diagrama de classes inferido pelo Ptidej. 47
3.14 Diagrama de classes inferido pelo Visual Paradigm. 48

4.1 Representação do processo de conversão de código fonte para PSM. 52
4.2 Exemplo de transformação simples. 55
4.3 Representação do processo de inferência de padrões. 57

5.1 Excerto da classe JMethod. 66
5.2 Metamodelo Java utilizado. 67

5.3 Excerto da classe SystemBuilder, *facade* para o JPSI. 68
5.4 Factos gerados durante o processo de análise. 70
5.5 Multiplidade $1-1$ (à esquerda) e $1-*$ (à direita). 71
5.6 Algoritmo de iteração de regras, para variáveis anónimas. 72
5.7 Metamodelo do PIM. 73
5.8 Interface de transformação de modelos. 76
5.9 Um dos métodos de análise de padrões. 79
5.10 Excerto da classe das propriedades visuais. 80
5.11 Excerto da classe de análise da informação de classes. 83
5.12 Visão geral do GPJaSI. 85
5.13 Visão geral do SAI. 87
5.14 Formato de definição de padrões. 87
5.15 Catálogo de padrões utilizado. 88

6.1 Imagens do software JHotDraw. 97
6.2 Funcionalidade extra no menu de contexto e na barra de ferramentas. 99
6.3 Janela de pré-processamento. 100
6.4 Diagrama de classes UML (PSM) para a Agenda. 101
6.5 Detalhe de um elemento UML, `Agenda`. 102
6.6 Diagrama de classes UML (PSM) para projecto mais complexo. . 103
6.7 Diagramas PSM (em cima) e PIM (em baixo) gerados. 104
6.8 Diagrama PSM (em cima) e PIM (em baixo) gerados para um subconjunto das classes de JHotDraw. 105
6.9 Escolha de padrões na janela de pré-processamento. 106
6.10 Listagem com um padrão Composite identificado. 106
6.11 Listagem de padrões inferidos em JHotDraw. 107
6.12 Visualização de informação sobre padrões em JHotDraw, de forma simplificada. 108
6.13 Explicação da definição de padrões. 108
6.14 Identificação do mesmo padrão várias vezes. 110
6.15 Dois padrões no mesmo diagrama: Decorator (em cima) e Composite (em baixo). 110
6.16 Duas representações do padrão Composite: normal (em cima) e simplificada (em baixo). 111

A.1 O padrão Abstract Factory, adaptado de [25]. 138
A.2 O padrão Adapter, adaptado de [25]. 138
A.3 O padrão Bridge, adaptado de [25]. 139

A.4 O padrão Builder, adaptado de [25]. 140
A.5 O padrão Chain of Responsability, adaptado de [25]. 140
A.6 O padrão Command, adaptado de [25]. 141
A.7 O padrão Composite, adaptado de [25]. 142
A.8 O padrão Decorator, adaptado de [25], adaptado de [25]. 142
A.9 O padrão Facade, adaptado de [25]. 143
A.10 O padrão Factory Method, adaptado de [25]. 143
A.11 O padrão Flyweight, adaptado de [25]. 144
A.12 O padrão Interpreter, adaptado de [25]. 145
A.13 O padrão Iterator, adaptado de [25]. 145
A.14 O padrão Mediator, adaptado de [25]. 146
A.15 O padrão Memento, adaptado de [25]. 146
A.16 O padrão Observer, adaptado de [25]. 147
A.17 O padrão Prototype, adaptado de [25]. 148
A.18 O padrão Proxy, adaptado de [25]. 148
A.19 O padrão Singleton, adaptado de [25]. 149
A.20 O padrão *State*, adaptado de [25]. 149
A.21 O padrão Strategy, adaptado de [25]. 150
A.22 O padrão Template Method, adaptado de [25]. 150
A.23 O padrão Visitor, adaptado de [25]. 151

Capítulo 1

Introdução

Desde cedo no desenvolvimento de software o processo de modelação tem mostrado muita importância dados os benefícios que oferece. Actualmente, este processo serve principalmente para definir as características de um sistema durante a sua concepção, ao mesmo tempo que serve de documentação para o mesmo. Existem já utilizações mais especializadas para estes modelos como a integração activa no processo de desenvolvimento, embora esta não seja uma abordagem frequente. Dada a sua importância e aumento da utilização da modelação, o *Object Management Group* (OMG) definiu uma linguagem padrão para este processo, a *Unified Modelling Language* (UML) [21].

Apesar de todos os benefícios, este processo de modelação consome muito tempo pois tem de ser o mais completo e específico possível, para permitir que o sistema implementado com base nos modelos gerados seja correcto e completo. Este processo é então um ponto de partida para a fase de implementação, e uma vez terminado raramente volta a ser foco de atenção. Por este motivo, com o passar do tempo e à medida que vão sendo feitas correcções e alterações na implementação, estes modelos vão-se tornando desactualizados e acabam por se tornar obsoletos. Desta forma o processo de modelação acaba por ser trabalho não aproveitado e sem importância. No final do processo de implementação eles servem apenas de documentação. Muitas vezes e devido a este problema, ao processo de modelação não é dada a importância necessária nem são utilizadas as suas potencialidades [49].

Do OMG, que definiu a UML, surge uma nova forma de criar sistemas de software, aproveitando o trabalho de modelação e solucionando os problemas atrás mencionados. Esta nova forma de criar software é a *Model Driven Architecture* (MDA), uma abordagem de desenvolvimento de software orientado a modelos. A MDA prevê a criação de sistemas de software, com base na definição e transformação de modelos (em modelos e em código fonte). Desta forma todo o foco da criação de sistemas de software será colocado no processo de modelação do software, pois todo o trabalho de modelação será trabalho útil e reaproveitado posteriormente. No final do processo de desenvolvimento este modelo estará coerente com a implementação obtida.

A MDA especifica o processo desde a criação de modelos até à obtenção de código fonte. No entanto, o processo inverso de código para modelos não é definido. Alguns autores mostraram que a MDA é viável e que pode ser utilizada com a tecnologia que dispomos hoje em dia [69, 56]. Os autores reconheceram ainda a importância do processo inverso, por exemplo para manter a coerência nos modelos quando ocorre uma alteração no código fonte [49].

Assim, o processo inverso ao especificado pela MDA é o foco desta dissertação. Será abordada a possibilidade da inferência de informação em diagramas de alto nível. Com base nestes diagramas serão tidos em conta os padrões definidos por Erich Gamma no seu livro, para mais facilmente perceber o comportamento de um sistema e permitir uma visão geral mais completa [25].

1.1 Contexto

Quando consideramos o processo de desenvolvimento de software, a abstracção tem sido significado de maior facilidade. Se olharmos para trás na história do processo de desenvolvimento de software vemos que este tem evoluído no sentido da abstracção. A título de exemplo tivemos o assembly [78] que nos abstraiu do binário e dos componentes de um computador. Posteriormente a linguagem C [47] evoluiu do assembly, o C++ [77] do C. Mais recentemente a linguagem Java [28] evoluiu do C++ [56]. Esta evolução natural das linguagens e ferramentas resulta em maior abstracção que é significado de maior protecção da complexidade e problemas específicos. Esta abstracção tem tornado o processo de desenvolvimento de software mais fácil, permitindo que menos tempo seja gasto em pormenores de implementação que em nada contribuem para a solução final [49].

O desenvolvimento baseado em modelos é uma abstracção de ordem superior, onde no limite temos abstracção total do processo de implementação. Quanto a este tipo de desenvolvimento a primeira tentativa foi feita pelas abordagens *Computer-Aided Software Engineering* (CASE) [22], permitindo a geração de código baseado em diagramas. CASE consiste num conjunto de ferramentas, que entre outras funcionalidades, permite a criação de diagramas de alto nível. Contudo demonstrou-se demasiado avançado para a sua época devido à falta de qualidade de serviço por parte dos sistemas operativos (por exemplo tolerância a faltas, segurança, etc.), entre outras limitações. Para além disso era muito limitado em vários aspectos como por exemplo na integração de código, falta de especificação e integração em outros ambientes. Algumas destas limitações eram culpa da tecnologia da altura, outras do facto do CASE ser demasiado genérico. A junção de todos estes factores resultou num baixo impacto na indústria [61].

As plataformas e linguagens de programação evoluíram muito desde os tempos do CASE resolvendo algumas das limitações que ele tinha. O uso de linguagens orientadas a objectos é um dos grandes avanços, permitindo a reutilização de código por um lado, e por outro lado (em alguns casos) fornecem independên-

cia de plataformas (como em Java). A maturação das linguagens relativamente às funcionalidades disponibilizadas a quem desenvolve software permite também uma maior abstracção por fornecer um maior conjunto de funcionalidades de alto nível, como é o caso da gestão automática de memória em Java. Apesar de todos estes avanços existem alguns problemas e impedimentos que continuam a existir. O maior de todos eles é talvez a incapacidade das linguagens ocultarem o crescimento da complexidade das plataformas. As plataformas existentes estão em evolução e contêm cada vez mais classes e funcionalidades. Por outro lado novas plataformas surgem constantemente no mercado oferecendo novas possibilidades.

Outro grande problema que enfrentamos é o facto da maioria do código existente ser escrito e mantido manualmente, isto é, sem auxílio de ferramentas. Isto reflecte-se num grande desperdício de tempo e esforço por parte de quem o faz. A combinação destes problemas resulta na complexidade tecnológica que encontramos na indústria de software, onde existem muitas plataformas, com *Application Programming Interfaces* (API) complexas. Os programadores perdem muito tempo a aprender estas API e padrões de desenvolvimento para estas plataformas. Desta forma os programadores conseguem apenas aprender a usar correctamente um pequeno subconjunto de tecnologias [61].

É neste contexto que a MDA aparece como um novo degrau no desenvolvimento de software. Agora que as linguagens já estão mais maduras, as plataformas mais estáveis e os sistemas operativos mais completos, a utilização da MDA aparece como uma promessa de resolução desses problemas. É sobre este processo MDA na sua forma inversa que vai incidir este trabalho, explorando as possibilidades de facilitar o processo de implementação, migração e evolução de soluções de software.

1.1.1 A *Model Driven Engineering*

A *Model Driven Engineering* (MDE) é uma metodologia de desenvolvimento baseada em modelos. A primeira tentativa de utilização foi feita por CASE como foi descrito atrás. O desenvolvimento baseado em modelos e na transformação dos mesmo já mostrou ter vários benefícios, principalmente por ocultar a complexidade dos sistemas. A ideia base é que existe apenas a necessidade de especificar o modelo de um sistema, sendo que por transformação destes modelos chegaríamos a uma implementação concreta.

Outra vantagem que esta metodologia apresenta é manter a coerência entre a implementação gerada e a especificação do modelo. Por outro lado permite reduzir a carga de código programado manualmente. Tendo uma especificação sob a forma de um modelo é possível que ele seja verificado por meios formais, evitando erros prematuros. Existem muitas tecnologias que disponibilizam funcionalidades que permitem o controlo de erros (por abstracção de funcionalidades complexas), como *middlewares*, API, etc que disponibilizam interfaces mais

simplificadas. Elas geram código mais simples, logo menos sujeito a erros [61].

Para que esta metodologia possa ser amplamente utilizada e desenvolvida é preciso definir formas padrão de criar e desenvolver ferramentas compatíveis com a mesma. Para tal, o OMG em 2001 definiu o processo MDA, baseado nos conceitos MDE.

1.1.2 A *Model Driven Architecture*

A desculpa tradicional para a não especificação de modelos, é que eles são "apenas papéis", e o esforço da escrita poderia ser centrado na escrita de código. Em contraste, a MDA faz com que os modelos sejam úteis pois permitem gerar código. O esforço investido nos modelos pode ser reutilizado várias vezes [49]. Permite gerar documentação actual, de acordo com a aplicação em vigor, e não uma foto do estado do software após a sua planificação inicial. A MDA é apropriada para desenvolvimento de boas soluções nos ambientes multi-plataforma de hoje em dia.

A MDA é a definição da forma padrão de fazer um desenvolvimento baseado em modelos, com a metodologia MDE. Este processo foi definido e é suportado pelo OMG e encontra-se representado na Figura 1.1. A MDA prevê que o destaque da modelação resida no facto de separar as funcionalidades da sua implementação e pormenores de plataforma, para um sistema em específico. Este processo afirma ser o próximo passo no desenvolvimento de software, que será independente de linguagens, plataformas, *middlewares* e *frameworks*. Esta apresenta-se então como um novo paradigma de desenvolvimento de software. Com esta abordagem quem desenvolve software pode focar os seus esforços em funcionalidades, evitando as repetições bem como repetição de modelações, conseguindo interoperabilidade e portabilidade [54]. Outras ferramentas poderão tratar do processo de gerar código e executáveis com base nestes modelos. A MDA representa a mudança de paradigma, sendo que o esforço de programação será agora depositado na fase de modelação de funcionalidades [49].

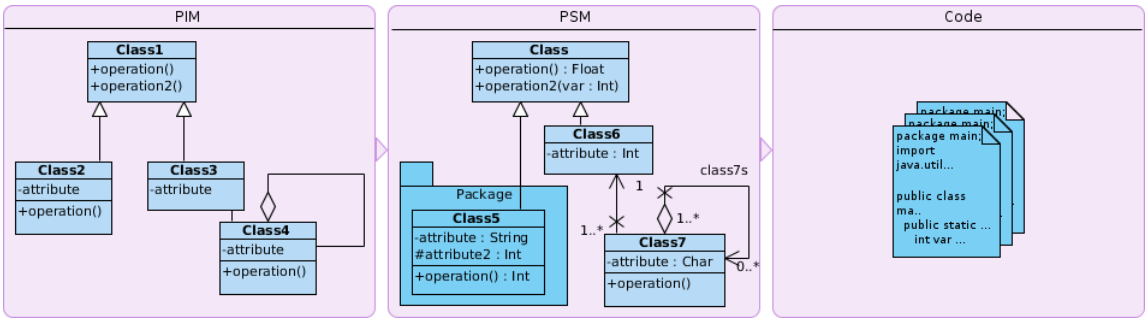

Figura 1.1: Simplificação do processo MDA, modelado em UML

O processo MDA vem então revolucionar a forma como a fase de modelação

é abordada e aproveitada. Vêm dar um novo ênfase a esta fase, fazendo com que todo este trabalho se torne útil, e mais que isso, que se torne o suficiente para a produção de sistemas de software.

1.1.3 *Design patterns* ou Padrões de concepção

Para compreender o processo de inferência de padrões é importante saber o que é um padrão de concepção (*Design Pattern*), bem como os padrões que devem ser tidos em conta. Christopher Alexander descreveu um padrão de concepção como o núcleo de uma solução para um problema que ocorre muitas vezes [25]. Solução essa que pode ser usada para resolver esse problema várias vezes sem a necessidade de se repetir a resolução [1]. Esta afirmação é relativa a padrões na construção de edifícios, contudo considera-se que também é válida no contexto da engenharia de software [25]. Um padrão de concepção (ou simplesmente padrão) é descrito por quatro componentes:

Nome O nome de um padrão identifica-o;

Problema Num padrão, o problema designa o que ele é capaz de solucionar;

Solução A solução é a descrição de como o padrão resolve um problema;

Consequências As consequências descrevem o custo e os benefícios de usar esse padrão.

A Figura 1.2 mostra um exemplo de um padrão de concepção, neste caso o padrão Composite. Este padrão define de que forma se pode organizar objectos em estruturas, por composição de objectos do mesmo tipo. A utilização deste padrão permite por um lado representar hierarquias, e por outro lado ignorar a diferença entre um objecto singular ou composição de padrões.

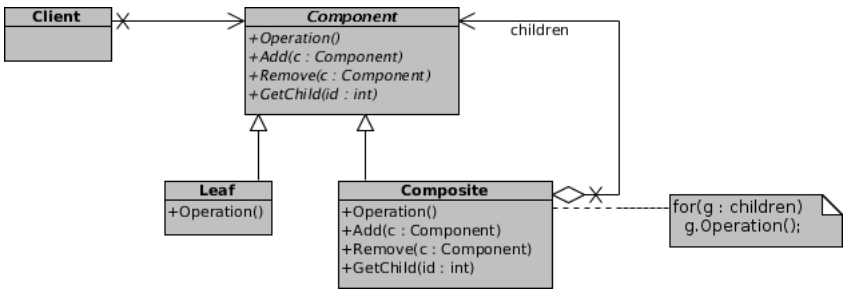

Figura 1.2: Padrão de concepção Composite, adaptado de [25].

Um padrão de concepção descreve uma solução simples e elegante para problemas específicos e bem conhecidos. Mais do que apenas uma solução, está provado que representam boas soluções amplamente usadas. Tendo estas soluções catalogadas, seria interessante tirar partido disso quando construímos

aplicações. Estas soluções são orientadas a objectos. As linguagens orientadas a objectos têm muitas vantagens como serem reutilizáveis, modulares e com código flexível. Estes factores fazem com que os padrões de concepção se enquadrem correctamente neste contexto [25].

1.2 Desafios de implementação

Neste documento é proposta a elaboração de uma ferramenta de software que irá entrar no âmbito das abordagens propostas. Na elaboração desta ferramenta existem alguns desafios que são de esperar à partida. A problemática MDA e mais em específico o processo inverso são áreas que já foram objecto de estudo tendo resultado mesmo em alguns trabalhos por parte de vários autores. Assim, com base no trabalho já feito por eles e no estudo sobre os seus resultados há algumas dificuldades que são já consideradas.

Para analisar os desafios propostos, podemos seguir o processo que a ferramenta proposta terá de efectuar para solucionar os problemas propostos. Para começar, existe a problemática da análise do código fonte. O código fonte surge em vários tipos (texto, *bytecode*) e de várias formas, com diferentes indentações e tipos de organização. Para além disso, e pela flexibilidade que o código proporciona, Java é por si só uma linguagem algo complexa em termos de sintaxe.

Derivado do problema anterior, mais especificamente da complexidade da linguagem Java, existe o problema da representação da informação extraída. A informação lógica contida nos ficheiros Java, tem de ser representada de alguma forma, bem como as relações algo complexas como são o caso os *packages*, as *inner classes*, as relações entre classes, etc.

A representação desta informação tem de possibilitar uma fácil análise para permitir a inferência de informação sobre ela, mais concretamente os padrões de concepção. Ainda em relação aos padrões, existe a necessidade de ter alguma forma de os catalogar, identificar e representar.

A abstracção de um modelo vai exigir que seja estudada a forma como o processo MDA é feito, para prever o seu funcionamento inverso. Se um mapeamento *Platform Independent Model* (PIM) para *Platform Specific Model* (PSM) gera artefactos, o processo inverso terá de ser capaz de os remover. Um artefacto é um qualquer resultado obtido do processo de desenvolvimento de software.

De uma forma mais geral, estes serão os desafios esperados durante a elaboração deste trabalho e implementação da ferramenta.

1.3 Motivação

O aumentar do número de ferramentas, linguagens e plataformas por um lado, a necessidade de abstracção e automatização por outro, são os problemas que

o processo de desenvolvimento de software está a encontrar neste momento. Nos dias de hoje, se quisermos desenvolver uma aplicação *web*, as possibilidades são imensas: PHP [30], Ruby on Rails [37], Java EE [79], ASP [8] entre muitas outras. As tecnologias disponíveis e meios para os fazer também são bastantes: desde diferentes *Integrated Development Environment* (IDE), editores de texto, editores gráficos, bem como diferentes *frameworks* de desenvolvimento como por exemplo Hibernate [41], Java Server Faces [12], struts [20] e muitos outros. Caso esta aplicação necessite de uma base de dados a escolha volta a ser muita, as tecnologias disponíveis são imensas: Oracle [14], MySql [13], PostgreSQL [29], Sql Server [10], sendo que esta listagem poderia ser muito mais extensa. No momento de fazer a publicação da aplicação a variedade da oferta continua pois os servidores *web*/aplicacionais aparecem também em grande número: Apache [19], JBoss [42] Tomcat [19], Glassfish [11], Rails [37], Sinatra [60], ISS [9], etc.

Como se pode constatar, por mais simples que uma aplicação seja a oferta em termos de tecnologias, *frameworks* e linguagens é imensa. Seria inviável alguém ser capaz de conhecer todas estas vertentes na perfeição. Ou existe uma especialização numa área específica (linguagem, *framework* e/ou tecnologia) onde se tem conhecimentos a fundo e se adquiriu a capacidade de desenvolver correctamente nessa área, ou então temos um conhecimento mais genérico e superficial sobre todas as áreas. A primeira opção limita as capacidades que quem desenvolve poderia aplicar, a segunda faz com que não conhecendo as tecnologias a fundo não se possa tirar verdadeiro partido delas. Este é um problema de certa forma previsível, uma vez que as ferramentas tenderiam naturalmente a evoluir em quantidade e qualidade. Diferentes visões resultariam em diferentes abordagens e consequentemente diferentes ferramentas. A solução para este problema de heterogeneidade é o mesmo que surgiu por exemplo, para a heterogeneidade de hardware: a abstracção.

Esta diversidade e heterogeneidade leva-nos a outro problema. Depois de modelado um sistema é necessário efectuar a sua implementação. No caso de uma aplicação comum com interface, camada de negócio e camada de dados será então necessário implementar todos os elementos relevantes do modelo para cada um dos componentes, e este trabalho tem de ser feito manualmente. Este é o problema que a MDA tenta solucionar, necessitando apenas da definição de um modelo, e uma única vez que sejam definidas as regras de transformação.

O desenvolvimento baseado em modelos é uma solução que irá surgir naturalmente, mas a um certo ponto poderemos ter a necessidade de fazer uma migração do software existente. Deste modo há uma necessidade de ter ferramentas que sejam capazes de inferir modelos com base no código existente actualmente. A abstracção oferece também uma visão de mais alto nível sobre um sistema de software. As vantagens dessa visão são variadas, como por exemplo perceber mais facilmente os constituintes de um sistema, localizar os pontos mais críticos de um sistema, perceber a dimensão e o modo como os constituintes interagem, entre muitas outras vantagens. A inferência de modelos, baseada em código já existente é não só uma necessidade, mas também uma mais-valia

na análise de soluções existentes. Estas necessidades serão a motivação para o desenvolvimento de uma ferramenta de inferência de modelos baseada no código fonte. Também deverá ser capaz de inferir os padrões de concepção no modelo, conceito que será apresentado mais à frente.

1.4 Objectivos

É objectivo desta dissertação resolver vários problemas dos apresentados atrás. De forma mais genérica os objectivos vão ao encontro ao desenvolvimento de uma ferramenta capaz de extrair e analisar informação de código Java. Propõe-se ainda comparar esta ferramenta com outras existentes na mesma área, o que permitirá obter algumas conclusões. Assim, destacam-se os objectivos relativos para a ferramenta proposta:

- Analisar e extrair informação do código fonte de programas Java;
- Inferir e representar diagramas de classes UML com base na informação extraída;
- Inferir padrões de concepção nessa representação;
- Permitir abstrair modelos PSM a PIM.

Cumpridos estes requisitos é de esperar então uma ferramenta essencial para o processo MDA na medida que vai fornecer uma funcionalidade importante, a de inferir informação do código fonte (Java) e representá-la a um nível mais abstracto. Por outro lado irá ser interessante na medida em que permitirá integrar sistemas já existentes no processo MDA.

1.5 Estrutura da dissertação

Esta dissertação está organizada em capítulos bem definidos que separam a informação em partes distintas. No primeiro capítulo foi apresentado e introduzido o problema em causa, bem como os objectivos a solucionar.

No segundo capítulo é feito o levantamento do estado da arte, o que corresponde a uma análise do estado actual dos elementos, tecnologia e soluções mais relevantes para a temática. São também analisadas algumas ferramentas onde estas soluções são aplicadas.

Ao longo do terceiro capítulo são apresentadas e analisadas algumas das ferramentas mais importantes. Estas ferramentas foram seleccionadas pela importância que representam tanto no mercado de trabalho como em ambiente académico. Serão apresentadas as suas principais características, funcionalidades e alguns resultados obtidos da sua utilização.

Será apresentado no capítulo quarto o problema em causa de forma mais detalhada, começando por enquadrar o problema, passando de seguida aos cenários práticos que a aplicação pretende resolver.

No capítulo quinto é apresentada a solução proposta. É apresentado de que modo se irá resolver cada um dos problemas propostos no capítulo quarto, bem como irá ser implementada cada uma das funcionalidades previamente propostas. Por fim é feito um balanço do trabalho efectuado.

No sexto capítulo é apresentado um caso de estudo aplicado à ferramenta desenvolvida. É seleccionado um conjunto de programas e utilizado o seu código fonte para serem objecto de estudo de modo a concluir sobre a implementação obtida.

Por fim no sétimo capítulo é feita uma conclusão sobre todo o trabalho desenvolvido, finalizando assim este documento com o capítulo de conclusões.

Capítulo 2

Estado da arte e trabalho relacionado

2.1 Introdução

A elaboração de um projecto de investigação necessita que seja feito um estudo prévio na sua área, para que este seja elaborado com base em princípios correctos e actuais. É necessário analisar o estado da arte, isto é, o que se encontra de mais actual neste ramo de investigação. Este estudo é importante para que se possa iniciar o processo de desenvolvimento com ideia do que já existe a ser feito na área e com uma visão global do estado actual das tecnologias e metodologias. Neste capítulo será mostrado o estado actual das tecnologias, metodologias e feitas as considerações necessárias para que se possa proceder à implementação da solução.

2.2 *Design patterns* ou Padrões de concepção

Grande parte da atenção dada aos padrões de concepção foi conseguida graças a Erich Gamma [25] pelo trabalho que desenvolveu nesta área. O seu principal contributo foi a catalogação de um conjunto de padrões de concepção, documentados no livro *Design Patterns - Elements of Reusable Object-Oriented Software* [25].

Depois da catalogação levada a cabo, outros autores começaram de imediato a identificar novos padrões. No entanto muitos destes padrões eram apenas variações de padrões já detectados. Outros não eram realmente padrões, ou não tinham qualquer relevância. Actualmente continuam a surgir estudos que identificam novos padrões, variações de padrões, micro-arquitecturas ou até anti-padrões [76]. O desenvolvimento de software tem necessidade destes padrões, necessidade esta que se expressa na quantidade de padrões presente no software que é desenvolvido hoje em dia. Estando estes padrões espalhados ao longo do

código e no caso de não estarem documentados é necessário uma fase de engenharia reversa para os conseguir identificar, embora nem sempre a recuperação seja garantida [31].

Os padrões considerados neste trabalho serão um subconjunto dos que Erich Gamma *et al.* catalogaram por estes se mostrarem actuais e úteis. A problemática de novos padrões poderem surgir será tida em conta. Para tal, será considerada a possibilidade do módulo de inferência de padrões funcionar com base em catálogos, catálogos estes que podem ser definidos pelos utilizadores. O problema da sua identificação não é de todo um problema novo. Este problema já foi abordado por vários autores previamente e é considerado um problema complexo [31].

Pretende-se desta forma solucionar a problemática da heterogeneidade de ideias em volta dos padrões. Para além disso será considerado um catálogo pré-definido para que o utilizador tenha um ponto de partida.

2.3 A *Model Driven Architecture*

Para conseguir alcançar resultados satisfatórios face aos objectivos propostos, é necessário perceber o funcionamento do processo MDA. Também é importante saber em que ponto se encontra o desenvolvimento de certos artefactos da MDA, bem como esta pode e está a ser utilizada. Desta forma será analisado mais em detalhe este processo, seus artefactos e funcionalidades ao longo das próximas secções.

2.3.1 A *framework* MDA

Como foi referido previamente, a MDA prevê a definição de modelos para dar início ao processo de desenvolvimento. Estes modelos existem a vários níveis de abstracção, começando pelos modelos de negócio, que não têm informação sobre o software, os designados *Computation Independent Model* (CIM). Contudo, o processo MDA define apenas dois termos, o PIM e o PSM. Qualquer que seja o modelo, terá de se enquadrar numa destas categorias, embora alguns autores considerem que seja difícil definir uma separação clara entre eles. Estes autores argumentam que apenas se pode dizer que um tipo de modelo é mais específico que outro, sendo o conceito de especificidade considerado relativo [49]. Outros autores ignoram os modelos PSM por considerarem que estes são apenas uma representação intermédia entre um PIM e o código final, portanto sem relevância [56].

Para que sejam feitas as transformações entre os modelos são necessárias ferramentas. Uma ferramenta de transformação utiliza um PIM e com ele gera um (ou mais) PSM. Posteriormente esse modelo será processado por uma ferramenta, que pode ser a mesma. Essa ferramenta utilizará então o PSM e irá

gerar o código fonte. Estas transformações são a essência do processo MDA e as ferramentas a sua concretização. A definição da transformação é o que vai permitir que este processo ocorra de forma automatizada.

Uma definição de transformação consiste num conjunto de regras de transformação. Essas regras são especificadas de forma clara e inequívoca. Elas fornecem a informação de como um modelo (ou parte) pode ser transformado em outro modelo (ou parte). Em *MDA Explained* [49], uma transformação é definida da seguinte forma:

> *"A transformation is the automatic generation of a target model from a source model, according to a transformation definition."* [49]

Uma das características mais importante no processo de transformação é a preservação da consistência entre modelos e com o código fonte. Consideramos uma transformação como um conjunto de regras de transformação. Por sua vez uma regra de transformação é uma descrição de como um ou mais elementos na linguagem fonte podem ser mapeados em um ou mais elementos na linguagem destino. É nestes mapeamentos que deve ser focada a atenção quando se quer garantir a preservação de consistência.

A Figura 2.1 representa de uma maneira simplificada de que forma a interligação de definições com ferramentas processam e transformam modelos, em modelos cada vez mais refinados até ao código final.

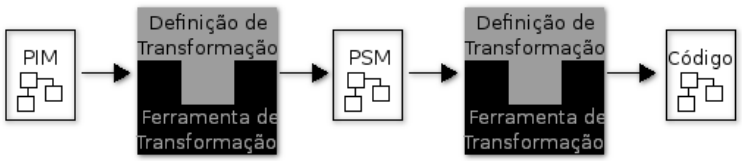

Figura 2.1: Processo MDA.

Estes dois tipos de modelos PIM e PSM são os elementos mais importantes para quem desenvolve software segundo a MDA (sobretudo o PIM). Eles são o foco da atenção no desenvolvimento porque a sua transformação é automatizada por ferramentas, que utilizam as regras de transformação para ir refinando estes modelos. Ambos possuem importância por fornecerem dois níveis de informação distintos. Um PIM pode ser útil (para além do que já foi referido) para mostrar por exemplo ao cliente final, que não necessita de muitos detalhes sobre o modo como a implementação está feita.

As regras de transformação possuem uma forma inversa, o que faz com que uma transformação possa ocorrer no sentido inverso. Para que seja possível abstrair o código em modelos, é então necessário compreender o funcionamento das regras no sentido directo (modelos para código).

2.3.2 Artefactos da MDA

Existem vários termos que contribuem para o processo MDA [59] e estão relacionados com ele. Assim, para compreender o funcionamento do processo MDA e a sua arquitectura é essencial compreender estes termos, pelo que irão ser analisados em maior detalhe. Estes termos são os componentes constituintes da MDA que definem as suas normas, úteis para criação de ferramentas por exemplo. A interligação destes componentes constitui a especificação da *framework* MDA.

Um dos componentes mais importantes quando falamos da especificação da MDA é o metamodelo. Segundo a MDA, um modelo tem de ser escrito numa linguagem "bem definida", no entanto não foi explicado ainda como definir uma linguagem que cumpra esse requisito. O mecanismo que nos permite definir esse tipo de linguagens é o metamodelo. Este consiste num modelo que define uma linguagem, ou, um modelo de modelos. Define a forma padrão para a interoperabilidade, uma vez que os metamodelos representam modelos bem definidos que usam API padrão e assim podem ser trocados e migrados [71].

Metadados são os componentes de interoperabilidade (via migração de modelos). A troca de metadados é importante porque o processo de transformação de metamodelos pode ser conseguido por meios formais e entendido por todos os componentes que participam no processo [49]. Estes modelos são genéricos com capacidade de completude semântica. O facto destes modelos serem genéricos assegura a compreensão por parte de todos os participantes, a completude assegura que o modelo pode ter um alto nível de especificação. A metamodelação requer uma linguagem formal, um formato de troca, modelo para acesso e descoberta, formato para publicação e mecanismos de extensão para que possa ser entendido por todos esses componentes [71].

Outro termo importante na MDA é o *Meta Object Facility* (MOF) [64], uma definição padrão do OMG. O MOF consiste na especificação de uma linguagem de definição de linguagens de modelação. Esta especificação é útil, por exemplo, para que possa existir a construção de ferramentas para definição linguagens de modelação. O modo como o MOF funciona é por meio de um interface que permite criar repositórios de modelos. Este interface é escrito numa linguagem que permite a sua utilização em diversos ambientes (CORBA-IDL)[67]. O MOF também define a forma de troca de modelos. Quando uma linguagem de modelação é definida com o metamodelo do MOF, este define uma forma padrão de gerar um formato de troca: o *XML Metadata Interchange* (XMI) [65]. O papel do MOF na MDA é fornecer conceitos e ferramentas para operar sobre linguagens de modelação. Usando a definição do MOF, pode-se definir transformações entre linguagens de modelação. Assim, o MOF, é uma linguagem especial para definir todas as outras linguagens. Assegura que as ferramentas serão capazes de ler e escrever todas as linguagens padronizadas pelo OMG [49].

Uma metalinguagem é a linguagem em que permite escrever metamodelos. Tal como as linguagens de programação são escritas em linguagens "bem definidas", a linguagem "bem definida" para modelar metamodelos é a metalinguagem.

A relação entre metamodelo, metalinguagem, modelo e linguagem é mostrada na Figura 2.2.

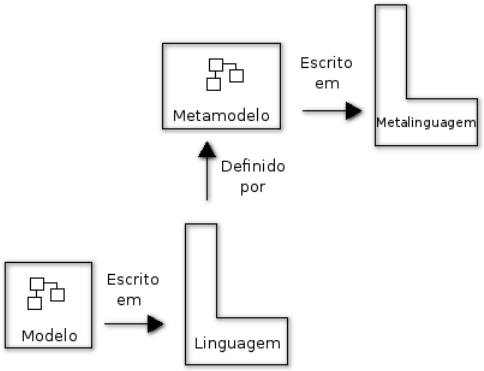

Figura 2.2: Relações entre metamodelos, metalinguagem, modelo e linguagem.

As ferramentas de transformação são outro elemento essencial à *framework* e são estas que permitem automatizar o processo.

Por fim, a linguagem é a expressão da definição de um metamodelo. A linguagem é o elemento que permite a escrita de modelos concretos.

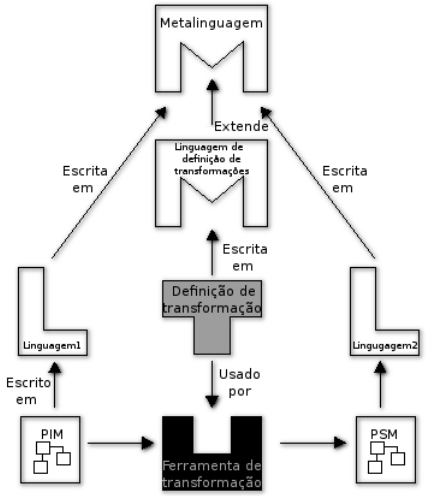

Figura 2.3: A *framework* MDA.

Estes elementos combinados constituem a *framework* MDA. Esta *framework* é apresentada na Figura 2.3 onde estão representados os elementos MDA bem como as relações entre eles.

Existem ainda outros termos relevantes no contexto MDA, que estão relacionados directamente ou indirectamente com a *framework* MDA.

O termo *Query Views and Transformations Standards* (QVT) é relativo a um norma que vai definir como as transformações entre linguagens definidas utilizando o MOF vão ser efectuadas. Esta norma vai ser parte integrante do MOF.

A UML é a linguagem padrão de modelação de diagramas, sendo que o metamodelo UML é uma instância do modelo MOF. O metamodelo UML tem a mesma estrutura do modelo MOF. Contém mais meta-classes que o MOF por ser utilizado para modelar mais do que simplesmente metamodelos. A UML pode ser utilizada para modelar o sistema em questão e também para modelar as transformações entre modelos.

A linguagem UML não é a linguagem obrigatória de modelação. A única restrição imposta pelo OMG é que os modelos precisam de uma linguagem bem definida, tal como uma linguagem tem de ser em definida para que possa ser interpretada por um compilador. Contudo, a UML acaba por ser a linguagem escolhida maioritariamente por vários motivos, como ser a principal linguagem de modelação utilizada nesta área cientifica bem como o largo número de ferramentas que lhe dão suporte [4].

A *Object Constraint Language* (OCL) [66] representa uma linguagem de definição de restrições por meio de expressões e pode ser utilizada em modelos MOF e UML. A sua utilização aumenta a precisão dos modelos. Estas expressões OCL indicam os valores esperados mas não especificam como devem ser calculados. Estas expressões podem ser traduzidas para linguagens de programação (como Java). A OCL tem vindo a ser usado em UML para especificar restrições. Geralmente definem restrições de invariância, pré-condições e pós-condições. No âmbito MDA, um modelo mais específico significa um resultado final mais completo. A utilização de OCL com UML permite a construção de PIM com maior qualidade, completando a consistência dos diagramas UML. O uso de UML com OCL também não especifica completamente o comportamento, mas gera PSM mais completos.

O *Common Warehouse Metamodel* (CWM) [63] é a linguagem de modelação desenhada para modelar aplicações de *data warehousing*. Como tal, tem muito em comum com o metamodelo UML, mas contém mais algumas meta-classes, como modelação de bases de dados relacionais. Como os metamodelos CWM são todos modelados utilizando MOF, podem ser todos usados como fonte ou destino da MDA.

Existe um grande número de linguagens definidas pelo OMG que permitem a modelação de modelos PIM e PSM. A mais conhecida e utilizada delas é a UML. A OCL é a linguagem de *query* e expressão para a UML. O *Action Semantics* (AS) [56], linguagem definida pelo OMG para especificar comportamento, não é considerada a forma ideal para escrever modelos PIM, por falta da abstracção necessária. As linguagens usadas no processo MDA têm de ter definições formais, sendo que esta restrição é necessária para que seja possível transformações automáticas por ferramentas. Todas as linguagens padronizadas

pelo OMG cumprem esse requisito.

Um perfil é um mecanismo de especialização, sendo parte da UML. É definido por um conjunto de estereótipos, restrições e *tagged values*. Como resultado, os perfis definem metamodelos especializados. A alternativa seria a definição de um novo metamodelo. Estes perfis acabam por ter muita importância na transformação de um PIM em PSM [59, 72].

Um PIM deve ser completo, consistente e inequívoco para permitir gerar um PSM completo. A UML é especialmente útil para modelação de aspectos estruturais via diagramas de classes. No entanto a UML não é suficiente pois não modela a parte comportamental/dinâmica, isto é não tem uma definição suficientemente formal para permitir gerar um PSM. Com esta abordagem grande parte do código teria de ser complementada manualmente. Em *Executable UML* [56], é descrito como gerar um PSM completo a partir de um PIM, incluindo a parte dinâmica. A componente dinâmica é definida com a combinação de UML com AS. A AS, que é a linguagem definida para descrever as máquinas de estado, não é uma linguagem de muito alto nível (pelo que se considera que escrever AS tem pouca vantagem sobre a escrita directa em PSM). Também a linguagem AS não é padronizada ou concreta, o que não permite a escrita de definições de uma forma padronizada. O AS necessita que os utilizadores definam as suas próprias especificações. A UML modela aspectos estáticos, a máquinas de estado são um ponto partida definir comportamento, que é completado com AS [56]. Esta representa uma solução possível e funcional, embora não seja perfeita.

2.3.3 Processo de desenvolvimento segundo a MDA

O processo de desenvolvimento tradicional está representado na Figura 2.4. Neste processo o desenvolvimento e produção de modelos ocorrem desde a fase 1 até à fase 3. Os produtos destas fases (modelos) são geralmente considerados "papeis" e nada mais por normalmente não terem parte activa no processo de desenvolvimento [49]. Assim que a fase de desenvolvimento (em 4) começa, a ligação com os modelos começa a desaparecer devido ao atalho frequentemente utilizado. As alterações no código fonte não costumam ser propagadas para os modelos por falta de tempo para tal. A escrita de código é considerado produtividade, a escrita de modelos não, mesmo que parte dos modelos fosse aproveita (pois parte do código tem sempre de ser escrita).

Devido ao facto de novas tecnologias emergirem com grande frequência e trazerem avanços e melhorias face às existentes, frequentemente as empresas têm necessidade de migrar o seu software para estas tecnologias. Existe um elevado custo aliado à migração de tecnologias, mas o cenário é ainda pior quando as tecnologias emergentes perdem compatibilidade com versões anteriores (e em utilização). Por diversos motivos o código pode necessitar de ser migrado e os factos apresentados permitem perceber à partida os custos que advêm deste processo. Durante o processo de desenvolvimento a documentação acaba por ser

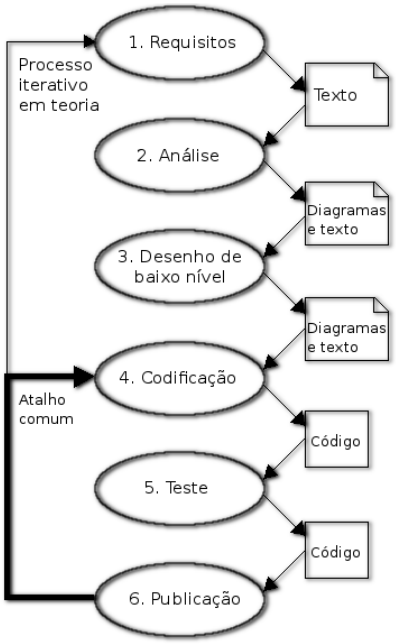

Figura 2.4: Processo de desenvolvimento tradicional.

deixada para segundo plano por diversos motivos. Esta traz um custo acrescido ao processo de desenvolvimento em tempo e esforço, não existe uma motivação forte suficiente para a fazer e para além disso raramente é verificada por pessoas externas ao mesmo projecto. A falta de motivação faz também com que a mesma não seja mantida actualizada quando surgem alterações no código. A falta de motivação para escrita da documentação juntamente com a necessidade de migração ou alterações no código já existente é um problema que frequentemente resulta em custos elevados e processos de migração complexos.

O processo MDA, como se pode ver na Figura 2.5, propõe uma ligeira alteração ao processo tradicional. A principal diferença está em que a MDA assenta em transformações automatizadas entre passos, executadas por ferramentas. Mesmo a transformação de PIM em PSM é automatizada, que é onde os benefícios óbvios da MDA se mostram.

A principal vantagem que a MDA proporciona é permitir a concentração de esforços na fase de modelação, gerando modelos PIM. Posteriormente, todos os modelos específicos (PSM) serão gerados a partir destes. A definição das regras exactas de transformação entre modelos têm de ser escritas manualmente. Esta tarefa é difícil e especializada, mas tem de ser feita uma só vez, ao contrário do que aconteceria se estas transformações fossem feitas manualmente. Tendo estas definições disponíveis, as mesmas podem ser utilizadas no desenvolvimento de vários projectos. Por um lado o processo de desenvolvimento é focado no desenvolvimento do modelo PIM de alto nível, sem necessidade de especificar detalhes de implementação. Por outro lado pode ser dada mais atenção aos

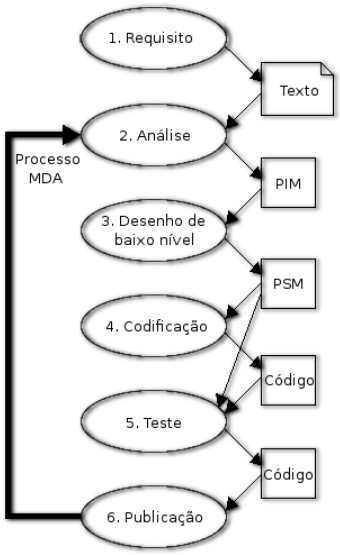

Figura 2.5: Processo de desenvolvimento segundo a MDA.

pormenores do sistema para resolver o problema em questão. O sistema resultante esperado é um sistema que encaixa melhor com as necessidades do cliente, feito com mais qualidade, mais funcionalidades e menos tempo. Este ganho será apenas conseguido com o uso de ferramentas que automatizem por completo o processo de transformação de modelos. Para além da geração de PSM existe também a necessidade de gerar as comunicações entre estes modelos, o que assegura *cross-platform* [49].

Em contraste com o processo tradicional de desenvolvimento de software, a MDA distingue-se pelo não abandono do PIM depois de escrito. Quando houver alterações no código, as ferramentas deverão ser capazes de reflectir essas alterações nos modelos. Assim, os modelos serão consistentes com o código, e estarão disponíveis para documentação. Contudo, existem requisitos impossíveis de capturar que terão de ser escritos manualmente.

Também é necessário ter em conta as metodologias suportadas pelo MDA. As metodologias de desenvolvimento são o que gerem o desenvolvimento de software dentro das empresas. Para que a MDA possa ser adoptada, existe necessidade de saber a que metodologias de desenvolvimento de software ele se adapta. Em *Executable UML* [56] é explicado que a MDA é bastante flexível, e como tal pode ser adaptada a qualquer metodologia. Nas metodologias mais ágeis permite resultados mais rapidamente (assim que parte do PIM esteja completo), nos modelos iterativos permite um desenvolvimento mais faseado (especificando completamente o PIM antes da geração de código). Esta flexibilidade na escolha do processo de implementação faz com que possa haver testes em qualquer momento, uma vez que os modelos não têm de ser completamente especificados para poderem ser compilados [56].

2.3.4 O processo MDA

O processo MDA é constituído por dois principais elementos. Um desses elementos consiste numa linguagem de modelação de domínio específico. Esta linguagem especifica um sistema em particular, representando as suas entidades, relações, etc. O outro elemento constituinte do processo MDA são as ferramentas de transformação que permitem transformar modelos (definidos nessa linguagem) em outros modelos ou até em código fonte, simulações, etc.

O processo de desenvolvimento da MDA é um processo iterativo, semelhante ao processo de desenvolvimento tradicional e bem definido (Figura 1.1). Começa com a definição de um (ou mais) PIM para uma problema específico. Esse PIM, como seria esperado contém apenas funcionalidades e comportamento de negócio. Poderá haver um diagrama ainda mais abstracto que o PIM, denominado CIM que são modelos com um nível de abstracção muito superior, contudo não possuem relevância para o problema em questão.

Um PIM consiste num diagrama visual que representa abstractamente os elementos de um sistema (Figura 1.1, à esquerda). Estes modelos contêm a informação mais abstracta do sistema (sem considerar modelos CIM). Estes diagramas não podem conter nenhum elemento da plataforma destino, como por exemplo a notação Java para atributos públicos e privados, caso contrário deixariam de ser genéricos.

Este PIM vai representar os dados de entrada para a ferramenta de transformação que irá gerar um PSM. Por vezes é considerado um PIM de segundo nível que é um passo intermédio entre um PIM de alto nível e um PSM [54]. Tal como um PIM, um PSM consiste num diagrama visual, contudo contém mais informação do que um PIM (Figura 1.1, ao centro). Um PSM contém a notação específica da linguagem destino. Por exemplo no caso de transformação de um PSM para Java, teríamos a transformação de atributos PSM, num atributo Java privado com os métodos `get` e `set`. Também a noção de método (como em "método Java") é uma especificação da plataforma destino.

Importa ainda referir que as ferramentas responsáveis pelo processo de transformação necessitam de dois tipos de informação específica. O primeiro tipo de informação necessário são os modelos de nível superior (neste caso, o PIM). O outro tipo de informação que necessitam são detalhes relativos aos modelos que vão produzir, como é o caso da plataforma destino. A informação relativa aos modelos destino é necessária devido ao facto dos modelos produzidos serem específicos para uma plataforma.

O passo final consiste na criação do código fonte. Neste passo é necessário uma ferramenta de transformação bem como todos os detalhes da plataforma destino. O resultado esperado é código pronto a compilar bem como todos os ficheiros necessários [54]. Thomas *et al.* [58] considera ainda dois passos adicionais ao processo. Um deles já foi referido, a criação de um CIM. O outro é a fase de criação do resultado, que consiste na compilação do código gerado (e a sua possível publicação, como no caso de uma aplicação *Web*).

A transformação de PIM para PSM ainda não é trivialmente conseguida, por falta da existência de ferramentas ideais para o conseguir. A transformação de PSM para código é possível por vezes, devido à especificidade destes diagramas e proximidade entre estes dois formatos. Por fim, PIM para código é possível com limitações e restrições devido às limitações da especificação da componente dinâmica. Neste momento as condições não são as ideais para o uso da MDA com sucesso e em larga escala. Embora ainda não seja possível tirar partido de todos os benefícios da MDA, as ferramentas de hoje em dia já permitem ter algumas das vantagens da MDA e antever as grandes vantagens que poderá trazer num futuro próximo [49]. Com algumas restrições e limitações, por vezes de forma não padronizada, já é possível obter software baseado em transformação de modelos [69, 56].

2.3.5 Afinações no processo de conversão

Existem variações e alterações que podem ser feitas durante o processo de transformação. Este processo não é uma simples transformação automatizada, e pode necessitar da interacção do utilizador para produzir melhores resultados. A afinação da transformação consiste na definição de pormenores relativos a essa transformação. Para perceber estas afinações é necessário especificar alguns conceitos. A rastreabilidade é a capacidade de manter o rasto de um elemento entre modelos, bem como recuperá-lo do código gerado. A consistência incremental significa que alterações feitas em transformações de PIM, são preservadas em novas iterações do processo. Por fim, a bidireccionalidade significa que as alterações podem ser propagadas tanto do modelo para o resultado como do resultado para o modelo. A transformação de modelos pode ser vista como um processo de instanciação e como tal deve manter informação das transformação entre modelos, mantendo sempre a consistência. Estes termos representam melhorias que podem ser introduzidas no processo de transformação. As transformações podem também conter parâmetros que correspondem a afinações neste processo. Estas afinações podem ser de várias naturezas como por exemplo definição de tamanhos de campos, tipo de variáveis, precisão das variáveis entre outros.

As linguagens destino das transformações têm necessidades diferentes, logo diferentes regras de transformação são necessárias. Os perfis UML [51] permitem satisfazer estas necessidades, pois são afinações da conversão. Um modelo criado em UML com perfis, consiste num PSM específico de um problema. Representa um PSM refinado e adaptado a uma situação particular (portanto uma afinação).

A definição formal das regras de transformação é um processo difícil e complexo, pois necessita de perfeição em todos os detalhes. É uma tarefa que consome muito tempo e necessita de ferramentas de verificação de consistência. Este argumento é normalmente usado para criticar o processo MDA. Contudo, é necessário ter em conta que sem aplicar a MDA, estas transformações sempre foram feitas manualmente, sem qualquer automatização, e sendo repetidas para cada ferramenta que fosse implementada. Para além disso, uma vez definidas

estas transformações poderão ser usadas tantas vezes quantas necessárias, o que é uma enorme vantagem.

2.3.6 MDA inverso

Actualmente, para uma qualquer implementação que não contenha um modelo coerente e completo, a MDA tal como definida não apresenta grandes vantagens. Para conseguir usufruir das suas vantagens é necessário inverter parte do processo MDA, com base no código dessa aplicação. O código tem de ser analisado para perceber as funcionalidades e organização do software em questão. De seguida o software tem de ser transformado num modelo abstracto. Estes passos necessitam de uma análise detalhada e precisa, bem como da extracção dos elementos suficientes para perceber o problema. Por fim é criado o modelo abstracto. Todos estes passos são custosos em tempo e repetitivos mas podem ser automatizados.

A inferência de modelos baseada em código é um processo importante para se poder utilizar a MDA em software já existente, feito com base em processos tradicionais. A maior vantagem é a capacidade de analisar uma aplicação sem necessidade de ler o código e perceber as suas particularidades bem como todos os detalhes. Se conseguirmos reverter o passo final (código para modelo) vamos então ter um modelo genérico contendo apenas os elementos de lógica de negócio mais importantes para a compreensão do problema. A preservação da consistência entre modelos e código fonte será também um ponto onde esta funcionalidade se poderá mostrar interessante. Para além disso existe também a capacidade de detectar erros em padrões de concepção. Tendo estas capacidades podemos então ter todas vantagens oferecidas pela MDA, como a migração, troca, evolução de modelos, bem como troca de plataformas e tecnologias mantendo o mesmo modelo, baseado numa aplicação já existente. Neste documento propõe-se uma ferramenta capaz de solucionar os problemas apresentados atrás, no que diz respeito ao processo MDA inverso.

Olhando para trás vemos que houve uma evolução no processo de desenvolvimento ao longo dos anos. O desenvolvimento tem sido feito cada vez mais a alto nível e desta vez, é a MDA que propõe uma nova mudança. Neste momento, o foco do desenvolvimento é feito no código fonte, com a abordagem MDA será de esperar que passe para os modelos. Em *MDA Explained* [49] o autor acredita que a MDA vai alterar a forma como no futuro o desenvolvimento de software vai ser feito, uma vez que esta abordagem é cada vez mais utilizada.

Para além disso a olhando para a história das ciências da computação vemos que linguagens imperativas substituíram o *assembly*, com cepticismo por parte dos programadores. Os programadores estavam preocupados com questões de eficiência, contudo com o passar do tempo os compiladores mostraram-se tão eficientes que estas preocupações desapareceram. Os cépticos tinham parte da razão: houve de facto uma perda de eficiência, tal como seria de esperar na

evolução para a MDA. Contudo, os benefícios fornecidos são muito superiores ao custo, como mais facilidade no desenvolvimento de projectos de larga escala e menos dificuldade de manutenção. Como será de esperar também, com o passar do tempo a tecnologia tenderá a diminuir este perda de performance. Com a MDA é esperado um tipo de revolução semelhante, onde inevitavelmente os PIM podem vir a ser compilados em código fonte, por exemplo. Este é no entanto um processo que poderá ainda demorar algum tempo até ser largamente utilizado, porque como se pode ver alguns anos após a especificação no livro *MDA Explained* [49] muito continua ainda por fazer. A migração de aplicações existentes para processos MDA é algo que não agrada a quem já tem aplicações em produção, e o processo seria custoso. A falta de ferramentas também tem sido um entrave à adopção da MDA. Esta será também uma motivação para a elaboração deste projecto.

2.3.7 Implementações da MDA

Existem diversos autores que propõem implementações da MDA que resultam por vezes em ferramentas que concretizam essas implementações. Estas ferramentas são feitas quer pelos autores, quer por empresas que utilizam essas implementações. Dois exemplos de ferramentas implementadas são as ferramentas propostas por Stephen Mellor *et al.* [56] e por Oscar Pastor *et al.* [69], em que algumas das implementações já estão em prática.

Regra geral, a forma como estes autores especificam as implementações do MDA é pela definição de um perfil UML. Juntamente com este perfil (subconjunto/adaptação da UML) são necessárias linguagens de definição de restrições bem como de especificação de comportamento.

O processo de transformação de modelos é completamente definido. É necessário que seja definida a semântica, detalhes da plataforma e tudo o que é abstraído do processo de modelação [56]. Em todos os métodos apresentados pelos autores, como seria de esperar isto é fundamental.

Em *Executable UML* [56] o autor descreve o modelo com *use cases*. De seguida implementa os modelos, em que o componente estrutural é descrito com diagramas de classes. Os ciclos de vida dos componentes são especificados com diagramas de estado. O processo de desenvolvimento é um processo iterativo de escrita de modelos, escolha do compilador, teste do resultado e caso não seja satisfatório, volta-se à fase de escrita dos modelos, para os refinar. Para definição de comportamento em *MDA Explained* [49] adoptado o AS, em *MDA in practice* [69] é utilizada a linguagem OASIS.

No livro *MDA in practice* [69] é definido o processo MDA baseado em quatro modelos. Estes modelos são baseados em quatro tipos de especificações: funcionais, comportamentais, de comunicação e decomposição. Estes são os componentes considerados essenciais para modelar um software. Os modelos que as concretizam são de objectos, de dinâmica, funcionais e de apresentação. É de

notar que a interacção dos utilizadores com o sistema também é contemplada na fase de modelação.

Em *MDA in Practice* [69] os modelos são também definidos num subconjunto de UML. É também utilizado OCL como linguagem de restrições juntamente com OASIS. Também o OO-Method descrito neste livro permite a geração de código pronto a compilar em soluções a partir de modelos. Um ponto central desta abordagem é a linguagem OASIS uma vez que permite obter as vantagens do formalismo de uma forma facilitada.

Ainda em *Executable UML* [56] é introduzido o conceito de modelo de um domínio, que é uma abstracção de parte do problema. Estes domínios em conjunto formam o modelo que representa a abstracção do sistema. Estes domínios são representados sob a forma de diagramas de classes UML, onde é usado OCL para especificar as restrições. As restrições são incluídas nas classes com os símbolos '{' e '}', sendo descritas algumas restrições como as que encontramos na linguagem *Structured Query Language* (SQL), como chave primária, chave estrangeira, etc.

O compilador é o componente final essencial ao processo. Deve ser adequado à plataforma, ambiente e desempenho pretendido. Pode ser comprado ou desenvolvido por quem concebe os modelos. O importante é que este compilador mapeia as entidades abstractas em entidades da linguagem destino.

O código gerado pelo compilador é denominado arquétipo. Segundo Stephen Mellor *et al.* [56] este código não tem que ser legível e mais que isso não deve ser lido e alterado, porque todas as alterações devem ser feitas ao nível do modelo. Esta abordagem é incompatível com a já descrita implementação incremental, onde era permitida a alteração do código.

Para que a MDA seja mais usada é ainda necessário que seja feita uma padronização de alguns conceitos, como das linguagens que permitam modelação completa. Na teoria a MDA está pronta para ser usada, e exemplo disso é o facto de já estar em uso no mercado, e provando que a perda de performance dos modelos compilados não é um grave problema [56]. Na prática é preciso ainda algum trabalho para que esta seja mais utilizada.

Estes processos descritos pelos autores podem ser, na maioria dos casos, directamente mapeados nos elementos constituintes da MDA, com algumas limitações (como por exemplo em *Executable UML* [56] onde é ignorado o PSM). Regra geral todas as abordagens têm uma representação directa dos modelos, das ferramentas, das linguagens e do código produzido.

Nestas abordagens um dos temas analisado foi a integração do processo MDA com software já existente. Um dos autores que aborda esta temática fá-lo de uma forma prática mas limitativa. O *OO-Method* descreve o processo de integração de software existente denominado *Legacy Systems*. Estes modelos são abstraídos em *Legacy Views*, onde os componentes do software são representados em classes. Estes componentes tornam-se então parte integrante do sistema a ser modelado. Estes componentes, quando abstraídos como classes têm proprie-

dades e métodos. Dos *Legacy Systems* apenas é possível aceder às propriedades que eles expõem, uma vez que por definição não podem ser alterados. A *Legacy View* é a modelação orientada a objectos de um sistema pré-existente. A sua definição é semelhante à de outras classes uma vez que também são uma abstracção estrutural e comportamental. Enquanto representação gráfica estes elementos são marcados com o estereótipo `legacy`. São então tratados e interligados com as outras classes. Passam a fazer parte do problema ao mesmo tempo que contribuem para a solução final.

Esta solução de integração de soluções existentes pode não ser a ideal, uma vez que não permite a alteração do sistema existente. Ele passa a ser como uma caixa negra, onde apenas se podem usar alguns componentes que são os que ele expõe ao sistema existente. Por outro lado a recuperação completa do modelo de um sistema pré-existente tem muitas mais vantagens, como por exemplo a alteração do mesmo. Isto poderá ganhar mais interesse ainda, se falarmos por exemplo na fase de manutenção de uma aplicação da qual não exista um modelo. Assim, esta solução proposta aparece mais como uma atenuação ao problema do que uma resolução.

2.4 Inferência de código e modelos

O processo de engenharia reversa de código e inferência dos modelos é já objecto de estudo por vários autores. De seguida serão apresentadas algumas alternativas a abordagens que já estão a ser utilizadas. Também serão apresentadas ferramentas envolvidas no processo. A maioria das ferramentas foi criada em ambiente académico, contudo existem ferramentas comercias de grande valor.

Existem dois tipos de abordagens de análise de código para a inferência dos modelos: abordagem estática e abordagem dinâmica. A primeira abordagem consiste na análise da informação estática do código, permitindo recuperar informação estrutural, como diagrama de classes e as suas relações, bem como variáveis e métodos disponíveis nessa classe. Este tipo de análise vai ser a principal preocupação para a ferramenta proposta. Os resultados esperados desta abordagem são diagramas de classes, PIM e PSM. A segunda abordagem trata da informação dinâmica, analisando o comportamento do software. Consegue prever à partida situações problemáticas e de erro. Esta abordagem é baseada no comportamento e interacção entre as classes por análise dos métodos. Normalmente produz diagramas de estado, de colaboração entre outros. Este tipo de abordagem pode se mostrar necessário para permitir a inferência de determinados padrões. A título de exemplo, Yann-Gaël Guéhéneuc na sua tese [31] propõe a análise estática para a inferência de modelos a vários níveis e utiliza apenas a análise dinâmica para complementar a informação estática no processo de identificação dos padrões. Face a este resultado a análise estática pode ser considerada uma boa abordagem para a inferência de modelos.

Na abordagem estática é importante referir que o código é a base de toda

a análise. Os elementos mais relevantes são extraídos do código, que vai ajudar a gerar modelos abstractos e concisos. O código contém demasiados detalhes da implementação, como variáveis (por exemplo) que não são necessários para os modelos abstractos e necessitam de ser excluídos de alguma forma [6]. A abordagem proposta consiste na recolha de tanta informação quanto possível do código fonte e modelá-la [50]. De seguida é processada a filtragem dessa informação seleccionando apenas os elementos mais importantes [2], eliminando elementos irrelevantes. Nesta fase é onde a interacção com o utilizador se pode tornar importante, sendo que o utilizador vai decidir qual o nível de detalhe que o modelo deve ter e eliminar possíveis componentes irrelevantes para a análise, mesmo que eles sejam importantes para o sistema em causa [74].

Existem vários problemas a ter em conta quando é feita a inferência de modelos a partir de código fonte Java. O maior problema consiste na análise de informação e relações porque para conseguir um diagrama abstracto e preciso é necessário ter em conta vários factores [34]. Os *classifiers* a ter em conta são atributos, métodos públicos abstractos, métodos públicos e métodos *overloaded*. Estes elementos podem todos ser recuperados do código fonte. A mesma abordagem de análise de código fonte e extracção dos componentes pode ser feita para classes, interfaces, *bound elements*, tipos de dados e enumerações.

A recuperação deste tipo de dados é a parte mais fácil porque se baseia apenas na análise de código fonte. A parte complicada é a inferência de relações entre elementos (para além da componente comportamental). Essas relações são elementos muito importantes definidos em UML para a análise abstracta de um modelo. A principal dificuldade na identificação de relações vem não só da falta de especificação na implementação (no código fonte), mas também da falta de algoritmos precisos de recuperação [34].

Também devem ser considerados vários tipos de relações e dependências, o que não facilita o processo [48]. Algumas técnicas para identificar esses componentes foram já objecto de estudo e serão apresentadas de seguida. A informação relevante para as associações binárias (existência de uma invocação entre duas classes) e alvo de associação (representação visual das associações) pode ser extraída directamente do código fonte.

A multiplicidade por outro lado é especialmente complicada de inferir quando as instâncias não são nem *arrays* nem colecções. Então, a multiplicidade a considerar será $0..*$. As associações N-árias, nomeadamente agregações e composições podem também ser recuperados do código fonte [34]. Nas relações "uma para muitos" existem dois passos a ser considerados:

- Recuperar os elementos disponíveis no código Java;
- Remover os detalhes não necessários para perceber as relações (a UML apenas necessita de associações simples) [27].

A informação das dependências para diagramas de classes UML não pode ser directamente extraída do código fonte, tem de ser inferida de uma representação

simplificada [34, 45]. Os restantes elementos UML são: atributos, operações (métodos no caso de Java) e tipos podem também ser directamente extraídos do código fonte [45].

Uma grande parte dos elementos dos diagramas de classes UML podem ser extraídos directamente do código fonte desde que todas as classes estejam presentes. Um diagrama de classes pode ser construído apenas por análise do código. Contudo, nem todos os componentes serão recuperados. Por outro lado alguns dos componentes são praticamente impossíveis de recuperar, enquanto que outros necessitam de um algoritmo eficiente para ser recuperados.

Para que se perceba a dificuldade da inferência de diagramas completos, Yann-Gaël Guéhéneuc [34] concluiu que a ferramenta mais precisa é Ptidej [32] com a capacidade de recuperar 62% dos constituintes UML, após um estudo sobre as melhores ferramentas de inferência de modelos.

Existe ainda outra decisão que tem de ser tomada para decidir a forma como a inferência de modelos vai ser feita: fazer uma análise baseada em código fonte ou *bytecode*. A decisão será sobre a primeira abordagem, por vários motivos. Primeiro o código fonte representa o sistema tal como ele é [2], sem optimizações do compilador e contendo todas as relações. Depois o *bytecode* representa código compilado, por isso necessita de trabalho extra de análise para perceber o funcionamento da *Java virtual machine*, e para perceber isso é necessário um estudo mais aprofundado da especificação da mesma [52]. A análise baseada em *bytecode* poderia fornecer a possibilidade de usar *Java reflection*, contudo este método de análise de classes não permite (entre outras limitações) a análise do código fonte das classes. Esta limitação é bastante impeditiva para a análise de certos padrões.

De seguida vão ser apresentadas as metodologias mais comuns, baseado num estudo sobre trabalho feito na área. O primeiro passo consiste na análise do código (fonte ou *bytecode*), e extracção da informação relevante para construir uma representação como árvore sintáctica [18, 85] ou um grafo direccionado, como proposto por Larry Barowski *et al.* [2]. A representação obtida possui um nível de detalhe muito elevado e necessita de ser simplificado [6, 80] mantendo contudo os elementos mais importantes. Por fim e dependendo do diagrama pretendido, é necessário saber o que fazer com essa representação. Neste caso específico o próximo passo consiste na inferência de padrões de concepção.

2.5 Inferência de padrões

A inferência de padrões de concepção num projecto de software pode ter várias finalidades. Pode ser, por exemplo, utilizada como medida de qualidade, obtenção de informação extra, entre outras possibilidades. A identificação de padrões pode também surgir no contexto de manutenção de um projecto. Nesta fase de manutenção de um projecto, quando é preciso fazer correcções no mesmo, surge a necessidade de analisar detalhadamente o software em questão. Por norma para

perceber o software a alto nível a única solução é a análise da documentação, que normalmente é obsoleta e incompleta, tornando-se inútil. Resta então fazer uma análise manual, por leitura do código fonte. Uma solução a este problema passa por obter o modelo do projecto e fazer a análise de padrões presentes no código [31].

A capacidade de inferência de padrões é o segundo componente da ferramenta proposta. Depois da criação de diagramas de classes ou PIM/ PSM com base no código, a ferramenta deverá ser capaz de inferir possíveis padrões de concepção existentes nesse software. Este processo pode ser feito de várias formas distintas e específicas. O resultado final (em quantidade e qualidade) vai depender da abordagem feita ao problema [16].

Como descrito por vários autores, a ideia base consiste em analisar o código e recolher os elementos relevantes. Normalmente esses elementos são classes, relações, métodos e inovações. Com os dados recolhidos, é criada uma representação intermédia. Outro componente da ferramenta responsável por identificar e mostrar os padrões processa esta representação. Alguns autores demonstraram que para este processo, existem diversas abordagens razoáveis [17, 75, 36, 44].

Existem três abordagens a ter em conta no processo de análise: estrutural, comportamental e semântica, sendo a última a menos usada. A análise estrutural foca-se na estrutura dos componentes de um software (em Java para as classes, interfaces, etc.). A análise comportamental observa o comportamento desses componentes, na forma como interagem bem como resultados que produzem (objectos). Por fim a análise semântica analisa o interior de cada um desses componentes. A abordagem mais usada é a estrutural baseada na representação interna, por vezes combinada com a comportamental. De acordo com Jing Dong *et al.* [16], a análise estrutural permite a obtenção de bons resultados. Combinado a análise estrutural com a comportamental podem ser melhorados os resultados. Normalmente a adição da análise semântica não trás melhorias significativas.

A forma como estas ferramentas fazem a sua representação interna é muito distinta em vários aspectos. As abordagens podem ser separadas em categorias: representação como grafo (ou árvore) onde a hierarquia dos elementos é preservada, representação como matriz, representação numa forma sintáctica que pode ser analisada e processada e representação numa linguagem de programação (como Prolog [15]). De acordo com Jing Dong *et al.* [16], as matrizes e grafos levam a bons resultados podendo ser obtidos a partir das representações de diagramas de classes, o que é importante neste contexto. Para além disso, a representação nessas formas é facilmente manipulável e já provou ser uma boa abordagem [33]. Outras abordagens têm vantagens e desvantagens quando comparadas com estas, mas como a ferramenta proposta tem como objectivo ter uma representação prévia como diagrama de classes, a abordagem como grafo (ou árvore) parece ser adequada.

Depois de obtida a representação interna, o próximo passo consiste na in-

ferência dos padrões de concepção. Também aqui existem diversas abordagens possíveis, que serão de seguida apresentadas. A ideia base consiste na comparação entre padrões de concepção e uma representação interna, onde ambos têm de estar na mesma linguagem. Nesta comparação são procuradas similaridades entre ambas as representações [16]. Todas as alternativas consistem em variações desta abordagem. Uma abordagem em específico consiste em representar os padrões em diagramas de classes, e de seguida procurar por similaridades estruturais [17, 70]. DeMIMA, uma ferramenta de análise de padrões, específica um processo em três passos: modelar o código fonte, enriquecer o modelo com relações e especializar o modelo [36]. Existe também a possibilidade de representar os padrões em matrizes e de seguida utilizar um algoritmo de "pontuação" (*scoring*) onde é testada a similaridade entre duas matrizes, uma contendo a representação do código, outra a representação de um padrão. De seguida é calculado uma "pontuação" (*score*) que indica qual o grau de similaridade destas matrizes [83]. Uma abordagem diferente consiste em *fingerprinting* onde o código é analisado pelos atributos externos, que foram catalogados em padrões de concepção previamente. Para estas abordagens importa ainda referir que algumas levam à identificação de padrões exactos, enquanto que outras apenas extraem aproximações de padrões [16].

Em DeMIMA [36] é proposta uma ligeira alteração ao processo, introduzindo o conceito de multi-camada. Este processo consiste em três níveis no processo: analisar o código e gerar um modelo, representar o modelo de acordo com um idioma, e por fim descrever esse modelo na mesma linguagem dos padrões [36]. Esta variante não prova trazer melhorias significativas aos resultados obtidos.

Já Andrea *et al.* [53] descreve um processo com análise de baixo nível que permite bons resultados. Para tal, o código é analisado e a informação estrutural com os métodos e suas invocações é extraído. A análise estrutural cria candidatos a padrões e com esta informação auxiliada da informação de métodos e invocações, uma análise de baixo nível é efectuada onde os padrões são então inferidos.

Um problema conhecido na análise de padrões é a possibilidade de falsos positivos. Este problema consiste na detecção de padrões onde na verdade não existem. Algumas ferramentas, como Fujaba sugerem uma abordagem *top-down* após extraídos os padrões, para fazer evitar falsos positivos [23]. Outras ferramentas optam por fazer uma selecção mais rigorosa ao identificar padrões, sendo que esta abordagem pode levar a falsos negativos ignorando possíveis padrões [75].

Em Ptidej [31] é feita uma abordagem com algumas diferenças que serão descritas de seguida. Para começar é definido um metamodelo, e são definidos três níveis de abstracção: idiomático, de concepção e implementação. O nível de implementação corresponde ao código fonte do programa em questão. O nível de concepção é o nível mais alto de abstracção, correspondente ao PSM. O nível idiomático é um novo nível definido pelo autor, como um nível intermédio entre

os níveis anteriores. O nível idiomático contém informação estática (Java) e dinâmica (para o comportamento, execuções, afectação, etc.), e o autor descreve um metamodelo para representar esta informação. O autor considera ainda que as propriedades fundamentais para a identificação de padrões residem nas relações entre as classes, e considera que a ambiguidade na especificação destas em UML são um problema na sua identificação. Assim, define detalhadamente estas relações e algoritmos para demonstrar a sua tese. A identificação de padrões é feita com análise estática e dinâmica. Para a análise dinâmica recorre à linguagem Prolog, onde cria uma base de conhecimento de forma dinâmica que posteriormente consulta [31].

A inferência de padrões com base em factos numa base de conhecimento não é uma abordagem nova. Foi já proposta uma abordagem similar com a geração de factos para um software, juntamente com *Smalltalk Open Unification Language* (SOUL). Neste caso o que é feito é a geração de factos (semelhantes a factos Prolog) com base no código fonte de um software. São também definidas regras que representam padrões de concepção. Posteriormente é utilizado o SOUL para fazer a inferência dos padrões existentes com base nos factos e nas regras definidas [87]. Existe pelo menos uma implementação deste conceito e que prova que esta é uma boa abordagem e que permite a sua utilização em contexto real, sendo mais do que um mero conceito [57].

Propõe-se assim uma ferramenta com base nas alternativas apresentadas. Dado um software, este será mapeado num metamodelo adequado previamente definido. Este metamodelo deverá contemplar não só informação estática, mas também alguma informação dinâmica como é o caso de invocação de métodos. Dessa representação será extraída a informação num formato externo de representação de factos. Recorrendo a uma ferramenta externa, serão analisados estes factos (que constituem a base de conhecimento) em busca de padrões de concepção. Para permitir a inferência de padrões de concepção será previamente estabelecido um conjunto de regras que definem esses padrões. Esta abordagem é similar a algumas apresentadas nesta secção, uma vez que estas se mostraram boas abordagens.

2.6 Análise das ferramentas disponíveis

Ao longo desta secção serão analisadas as ferramentas disponíveis relacionadas com os problemas apresentados. Estas são ferramentas de inferência de modelos (com base em código fonte), IDEs e ferramentas de inferência de padrões. Com esta análise pretende-se em primeiro lugar ter uma visão geral sobre o estado de ferramentas existentes nesta área. Em segundo lugar saber quais as funcionalidades e limitações destas ferramentas. Em terceiro lugar é importante saber até que ponto estar ferramentas conseguem alcançar os objectivos a que se propõem. Por fim será comparada a qualidade da ferramenta face à sua implementação

2.6.1 Inferência de modelos

O largo conjunto de ferramentas disponíveis pode ser divido em dois grupos: ferramentas de engenharia reversa e ferramentas de *round-trip*. As ferramentas de *round-trip* são capazes de reverter código, criar representações modificáveis e gerar novamente código a partir dessas representações. As ferramentas de engenharia reversa por outro lado, concentram os seus esforços em analisar código, capturando de forma eficiente os requisitos e detalhes de uma solução de software. Na ferramenta proposta o principal esforço será ir de encontro às capacidades de engenharia reversa. O processo MDA propõe, como analisado, o processo de transformação de modelos em modelos, e em código. Assim esta não será a principal preocupação uma vez que seguindo a especificação outra ferramenta poderá usar os diagramas inferidos para gerar código. Por outro lado o objectivo proposto da inferência de informação poderá ser conseguido apenas com engenharia reversa.

Existem actualmente várias ferramentas disponíveis para ajudar no processo de inferência do código. De seguida as ferramentas mais relevantes e reputadas serão apresentadas. Algumas destas ferramentas são de *round-trip*, outras de engenharia reversa. As principais vantagens e desvantagens serão apresentadas também, juntamente com uma pequena descrição.

ArgoUML É uma ferramenta bastante complexa e completa [5]. É um IDE com capacidades de *round-trip*. É capaz de ler código fonte, gerar representações de diagramas de classes, suportar alterações no diagrama e gerar novamente código.

Dali A ferramenta Dali [46] teve origem como uma ferramenta de análise de código fonte e geração de diagramas de alto nível. Esta ferramenta é um *plugin* para o IDE Eclipse e evoluiu para uma ferramenta mais focada na problemática *Object Relational Mapping* (ORM). Para além disso destaca-se por possuir uma base de conhecimento sobre os factos e indicações fornecidos pela análise [46].

Fujaba Significa *From Java and Back Again*, e como o nome indica é uma ferramenta de *round-trip* [50] e é cada vez mais a base dos estudos feitos nesta área, para a linguagem Java. É a base para ferramentas como Reclipse. Está preparado para suportar funcionalidades de suporte para Java, diagramas UML e padrões de concepção [80]. Serve de base a outras ferramentas devido às suas capacidades de extensibilidade (*Plugins*).

Idea *IntelliJ Idea* é uma ferramenta comercial focada no processo de re-documentação de tecnologia Java [43]. O processo de engenharia reversa é estático, e é capaz de gerar diagramas de classes [50].

jGRASP Oferece mais funcionalidades que apenas engenharia reversa. jGRASP é um IDE, com suporte para escrita de código, compilação e execução,

bem como capacidades de inferência de diagramas de classes UML a partir de código Java (texto ou *bytecode*), sendo a última a funcionalidade mais relevante para este projecto [40]. Para mostrar os diagramas UML é utilizada a ferramenta *Flexible Graph Drawing Library*. Esta ferramenta utiliza *Java Debug Interface* (JDI) para aceder ao estado da máquina virtual Java para análise dinâmica [38]. Esta ferramenta é maioritariamente utilizada em contexto académico e tem actualizações regulares.

Ptidej Significa *Pattern Trace Identification Detection and Enhancement in Java* [31]. Começou como uma ferramenta de geração de código com base em padrões de concepção e tornou-se uma ferramenta importante de engenharia reversa. Esta ferramenta é capaz de detectar padrões de concepção e os seus possíveis defeitos [61]. Por análise estática esta ferramenta é capaz de detectar classes, interfaces e relações recorrendo ao *bytecode*. Para fazer esta análise é utilizado um módulo chamado PADL que infere as relações de herança, interfaces e instanciações por iteração sobre o código das classes. O método formal de inferência das relações e multiplicidade é analisado por Yann-Gaël Guéhéneuc [33].

Reclipse É uma ferramenta dinâmica de detecção de padrões de concepção com base em código fonte, que é baseada na ferramenta Fujaba [23]. No processo estático esta ferramenta analisa o código, e cria um grupo de candidatos com base na semelhança desses padrões com modelos previamente representados. A representação interna é feita por meio de um grafo sintáctico. A análise estrutural é feita sobre essa representação [85]. Os autores de Fujaba deixaram de desenvolver a vertente de engenharia reversa da ferramenta Fujaba, desenvolvendo agora a ferramenta Reclipse.

Reveal É uma ferramenta de engenharia reversa (como a ferramenta proposta) [55]. Os autores afirmam que a ferramenta tem os resultados mais precisos no processo de inferência, quando comparada com Superwomble, Rose e Together. Esta ferramenta difere das outras por possuir um *full parser*. Isso significa que todo o código é analisado, não apenas os elementos chave da linguagem. Os autores suspeitam que as outras ferramentas não fazem uma análise completa do código. A ferramenta de análise de código usada é Keystone, e a ferramenta para criar representação visual é Graphviz [55].

Superwomble É uma ferramenta de inferência de diagramas de classes [86]. Esta ferramenta é referenciada por ser ponto de comparação por parte da ferramenta Reveal. É uma ferramenta capaz de criar diagramas de classes simples (um subconjunto dos diagramas de classes UML). Superwomble não é uma ferramenta muito completa [55].

Rigi É uma ferramenta [62] semi-automatizada de engenharia reversa. Permite identificação automática e interactiva (com auxilio do utilizador) para inferir diagramas. Contém um *parser* e um editor visual (Regiedit). É no

entanto uma ferramenta genérica e precisa de ser configurada para funcionar com uma linguagem específica [80]. A sua abordagem consiste em analisar o código, identificar artefactos e relações e por fim extrair o modelo abstracto. Devido a estas características muitas ferramentas usam o Rigi para apresentar e manipular representações.

Rose É uma ferramenta comercial de engenharia reversa que oferece visualização de classes, interfaces e associações numa representação como árvore [39]. Tem também a capacidade de gerar diagramas de classes. É uma ferramenta comercial semelhante a Together também comercial, pela IBM.

Shimba Permite a engenharia reversa de aplicações e *Java applets*, com inferência de informação estática e dinâmica [81]. Opera directamente sobre Java *bytecode*. Através de análise estática infere classes, interfaces, métodos, construtores e variáveis. É capaz de inferir também as relações de extensão, implementação, agregação, composição, invocação, acesso e atribuição [81, 80]. A sua abordagem é a tradicional: analisar o código fonte Java, criar uma representação interna como grafo e carrega essa representação para Rigi. Adicionalmente utiliza o JExtractor (analisador de *bytecode*) e Rigi como auxiliares.

Together Oferece inferência de classes Java, representando-as em diagramas de classes UML [7]. É uma ferramenta comercial da Borland [50].

Visual Paradigm É uma ferramenta comercial com capacidades de *round-trip*, capaz de inferir e gerar diagramas a partir de código fonte [68]. Suporta um largo número de diagramas UML, desde classes, estados, sequência, casos de uso, entre outros.

Em relação às ferramentas comerciais não foi possível uma análise mais profunda por não serem ferramentas *open-source*, e os seus algoritmos internos não estarem disponíveis. Existem outras ferramentas similares às apresentadas, com o mesmo objectivo, contudo por não serem significativamente diferentes, ou por serem antigas e desactualizadas não foram tidas em conta.

Baseado no trabalho de outros autores, de todas as ferramentas apresentadas as que se destacam são Fujaba, jGRASP e Ptidej. São as ferramentas com mais trabalho de investigação devido às suas capacidades e expansibilidade. Ptidej afirma-se como sendo a ferramenta com maior precisão nos elementos UML que é capaz de recuperar [32]. Contudo Ptidej não oferece suporte a diagramas de mais alto nível como PIM e PSM. Ainda assim Ptidej será uma boa base de estudo e comparação. jGRASP é mais que uma ferramenta de inferência, é um IDE contudo sem funcionalidades de padrões (quer detecção, quer correcção), por isso esta ferramenta é também um bom ponto de comparação para a ferramenta proposta, principalmente na parte da representação da informação, bem como a funcionalidade de edição de código. Fujaba é um pouco diferente do objectivo da ferramenta proposta, por ser uma ferramenta de *round-trip*.

Depois do estudo das abordagens que estas ferramentas fazem ao processo de abstracção de código em modelos, do modo de análise estática e suas particularidades, juntamente com a análise das soluções existentes há algumas considerações a fazer (face à ferramenta proposta). Todas as ferramentas apresentadas afirmam ser capazes de reverter código Java (ou *bytecode*) para diagramas de classes. Contudo o nível de detalhe não é muito preciso nem completo. Mais que isso, quase todas as ferramentas conseguem gerar apenas diagramas de classes. Mesmo sendo capazes de gerar código a partir dos diagramas, não conseguem fazer operações nem análises em padrões, com a excepção do Ptidej e do Fujaba. Sendo algumas destas ferramentas baseadas em outras, elas herdam todos os possíveis problemas e limitações que possam existir.

Independentemente da ferramenta o processo de inferência é na sua essência o mesmo. As ferramentas começam por analisar o código (texto ou *bytecode*) e extrair a informação necessária. Depois de uma filtragem (ou selecção rigorosa), é criada uma representação interna. Esta representação é processada de alguma forma, e mostrada ao utilizador. Baseado nestas ferramentas e nesta informação, pode-se então concluir que dada a similaridade da ferramenta é viável seguir passos semelhantes. Algumas destas ferramentas tiram partido de outras para fazer parte do processo (seja reverter modelos, representar informação, etc.). Neste caso é necessário ter em conta a necessidade de integrar código, bem como as possíveis limitações. Concluindo, na ferramenta proposta existem duas grandes decisões a ser tomadas. Primeiro decidir se usar alguma ferramenta existente para fazer parte do processo. Segundo, decidir que representação interna utilizar para os dados recolhidos. Estas decisões são importantes por terem influência em todo o processo por um lado, e por outro lado não podem ser decididas com base numa ferramenta específica pelo facto que a ferramenta proposta tem um objectivo diferente das ferramentas analisadas.

2.6.2 Inferência de padrões de concepção

De seguida serão apresentadas ferramentas que põem em práticas os conceitos teóricos apresentados previamente relativamente à inferência de padrões.

DEPAIC++ Embora seja uma ferramenta de análise de código C++, o relevante serão os seus fundamentos teóricos [17]. Implementa uma abordagem baseada na análise de diagramas de classes [17].

Fujaba Previamente analisada, importa referir que se distingue pela sua análise *bottom-up* e *top-down* para melhores resultados.

Hedgehog É uma ferramenta que lê a definição de um sistema de uma representação SPINE [3]. Este tipo de representações necessita que o utilizador defina as relações entre classes. Esta ferramenta usa análise estrutural, semântica e comportamental [75].

Jochen Seemann [**73**] Descreve teoricamente uma ferramenta bastante competente baseada em análise estrutural com representação interna como grafo. Esta ferramenta tem uma abordagem similar à pensada para a ferramenta proposta, o que mostra que o tipo de análise com a representação combinam para um bom resultado.

PINOT É uma ferramenta completa e automatizada para inferência de padrões [75]. Garante ser capaz de reconhecer todos os padrões combinando análise estática e dinâmica para melhores resultados, de acordo com o autor [75]. Utiliza um compilador de Java *open-source* com um módulo de análise de padrões. Esta ferramenta utiliza uma tabela de símbolos como representação intermédia.

Estas ferramentas não serão objecto de comparação por diferirem tanto quer na representação interna, quer nos algoritmos usados. Muitas vezes são também baseados em outras ferramentas para análise do código fonte [16]. Assim torna-se difícil encontrar um ponto de comparação entre todas estas ferramentas, bem como com a ferramenta proposta. Então, o que importa referir é o processo comum a todas as ferramentas. Este processo consiste na representação como modelo, e analisar o modelo com um algoritmo de comparação. O algoritmo vai procurar similaridades em representações previamente definidas.

A inferência de padrões de concepção não é de todo um processo trivial. Mesmo que os autores afirmem que as suas ferramentas conseguem inferir padrões eficientemente, nenhuma os consegue identificar completamente e de forma eficiente. Algumas ferramentas conseguem inferir um maior número de padrões, outras conseguem inferi-los com maior precisão e detalhe. Existe ainda a sugestão do uso de anotações, por exemplo, para uma inferência eficiente. Contudo esta proposta funciona apenas para software que seja construído do zero, ou editando código fonte existente o que não é o objectivo desta ferramenta.

Quanto à ferramenta proposta as abordagens a ser consideradas serão a análise estática (juntamente com alguma informação dinâmica) para o código, gerando os diagramas de classes. A escolha destas abordagens justifica-se pelos dois principiais objectivos a que a ferramenta se propõe, tornando assim mais fácil combinar essas funcionalidades. Sabendo as limitações da análise estática, será sempre considerada, ou no mínimo deixado em aberto a possibilidade de ter análise dinâmica.

2.7 Resumo

Cada vez mais a MDA tem vindo a ganhar importância. É a promessa de ser uma alteração de paradigma no desenvolvimento de software, que já convenceu muitos autores. O OMG fez um grande trabalho no processo de especificação da MDA, *frameworks* e componentes para interagir nesta abordagem. O processo MDA foi apresentado em maior detalhe neste capítulo.

Muitos autores começaram de imediato a trabalhar em ferramentas consistentes com essa especificação chegando a ferramentas importantes e interessantes, algumas *open-source* como Ptidej, outras comerciais como Visual-Paradigm. Os padrões de concepção catalogados por Erich Gamma [25] encaixam perfeitamente no processo MDA por serem facilmente representáveis e identificáveis em diagramas de classes UML e complementarem o processo de abstracção de informação. As três funcionalidades descritas existem dispersas nas ferramentas descritas atrás, com os pontos fortes e fracos já identificados. Não existe no entanto nenhuma ferramenta que combine as funcionalidades descritas num único ambiente de desenvolvimento sendo que as ferramentas mais próximas deste objectivo são Ptidej, Fujaba e JGrasp. Estas e outras ferramentas foram apresentadas ao longo deste capítulo.

Todas as ferramentas descritas permitem atingir os objectivos a que se propõem, e algumas são bastante caras pelo seu valor produtivo para as organizações onde são usadas. Contudo, nenhuma destas ferramentas é uma solução completa, pelo que a ferramenta proposta acaba por ser um pouco a integração destas funcionalidades destas ferramentas em uma só. Mesmo assim a ferramenta proposta acaba por não ser uma solução completa por não oferecer funcionalidades de *round-trip*, contudo tenciona ser uma ferramenta que execute completamente o processo MDA inverso com analise de padrões. Como esta ferramenta se tenta aproximar da MDA, outras ferramentas poderiam usar os modelos gerados para criar outros modelos e mesmo código. Este resultado foi obtido graças à comparação descrita neste capítulo.

Este capítulo termina concluindo que o trabalho de análise de ferramentas, metodologias e implementações existentes é muito importante. Com base neste trabalho foi possível tirar algumas conclusões que permitiram tomar decisões relevantes para a fase de implementação. Permitiu também ter uma ideia de que nível de desenvolvimento se pode começar a elaboração da ferramenta. Também com este estudo foi possível enumerar à partida alguns problemas que serão de esperar encontrar. Permite ainda saber à partida que resultados são realistas e possíveis de atingir. Após o levantamento do estado da arte é então possível passar à próxima fase e começar a especificar a ferramenta que será implementada com base no estudo efectuado e nos objectivos propostos.

Capítulo 3

Análise das ferramentas

3.1 Introdução

A análise das ferramentas actualmente existentes vai de encontro ao objectivo do levantamento do estado da arte. Desta forma será feita uma análise das ferramentas mais importantes disponíveis neste contexto actualmente. É objectivo deste estudo tirar algumas conclusões quanto às funcionalidades, desempenho, arquitectura, etc. que serão úteis para posteriormente tomar decisões em relação aos detalhes da ferramenta a implementar.

Neste capítulo serão analisadas as ferramentas mais relevantes das que foram descritas no capítulo anterior. Para tal será utilizado um programa exemplo em Java denominado Agenda que contém os elementos mais relevantes a ser inferidos, como classes, interfaces, atributos, relações, entre outros. De seguida este programa será analisado com cada uma das ferramentas em análise recorrendo às funcionalidades que elas oferecem. Na Figura 3.1 está representado o diagrama de classes que representa a Agenda. A Agenda é um software que representa uma agenda electrónica. Nesta agenda é possível criar eventos com participantes, em que os participantes poderão ser alunos ou professores. A agenda permite também criar eventos complexos com sub-eventos, bem como persistir a sua informação. Destaca-se a existência de um padrão de concepção neste código. Este software servirá de base à análise das capacidades das ferramentas em análise.

O programa exemplo representa uma agenda pessoal com gestão de eventos e é um exemplo meramente ilustrativo. Neste código é possível ver a existência de um padrão de concepção, o padrão Abstract Factory, representado à direita na área destacada.

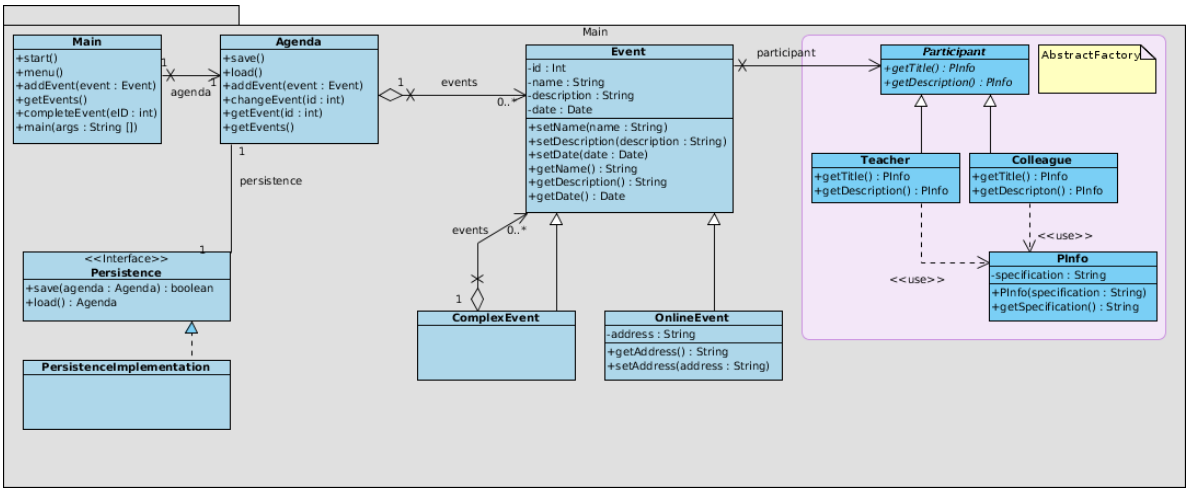

Figura 3.1: Diagrama de classes do software Agenda.

3.2 ArgoUML

Esta ferramenta consiste numa ferramenta *standalone* (independente) que apenas necessita do *Java Runtime Environment* (JRE) para poder ser executada [5]. Ela suporta a importação de classes Java directamente a partir do interface principal. Para fazer a análise iniciou-se a ferramenta e seleccionou-se a opção de importar código das classes Java. A ferramenta tratou do processo de engenharia reversa e foi então possível analisar o resultado gerado.

3.2.1 Análise de código

Esta ferramenta reconheceu com sucesso o código fonte Java da aplicação exemplo como se pode ver na Figura 3.2. Foi capaz de reconhecer e representar todas as classes e interfaces bem como analisar o código fonte. A ferramenta gerou automaticamente o diagrama de classes assim que terminou a análise dos ficheiros.

3.2.2 Reconhecimento dos elementos

A ferramenta em análise reconheceu com sucesso todos os elementos da aplicação exemplo. Os elementos foram correctamente apresentados em notação UML, embora de uma forma simplificada. Contudo nem as relações nem os elementos das classes são mostrados correctamente no diagrama apesar de estarem contidos na descrição das classes, como se pode ver pela Figura 3.3. A informação sobre as relações entre as classes é visível nas propriedades das classes o que mostra que é inferida, contudo no diagrama UML essas relações não são apresentadas. A Figura 3.4 mostra as propriedades da classe `Main`, onde existe um atributo

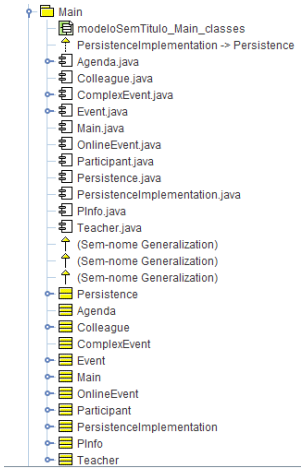

Figura 3.2: Elementos UML carregados pela ferramenta ArgoUML.

`Agenda`. As relações de agregação, composição, uso e implementação (no caso testado) não foram reconhecidas nem representadas.

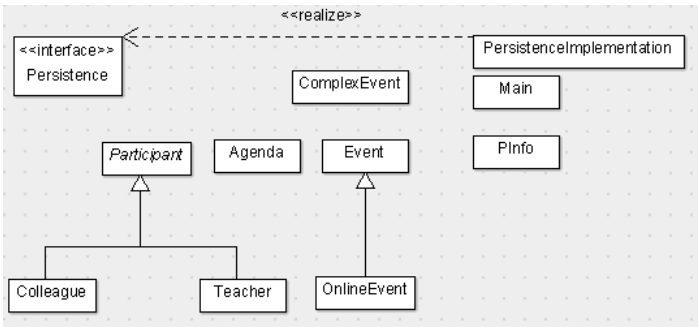

Figura 3.3: Diagrama de classes gerado pela ferramenta ArgoUML.

O diagrama de classes que representa a aplicação exemplo produzido pelo ArgoUML é relativamente fraco, como se pode ver na Figura 3.3. A ferramenta não foi capaz de representar muitas das relações entre elementos, mesmo tendo as reconhecido. Na representação visual não estão presentes quaisquer informações adicionais como variáveis, métodos, etc. Pode se então concluir que no reconhecimento e representação de elementos UML, esta ferramenta não é muito precisa nem rigorosa, não sendo mesmo capaz de representar alguns dos elementos mais básicos. Este não pode ser considerado um diagrama de classes UML nem qualquer outro diagrama MDA de alto nível por não conter a informação mínima essencial para ter utilidade.

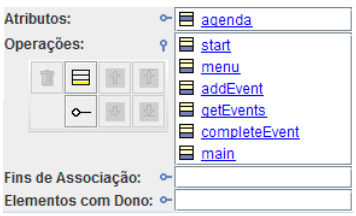

Figura 3.4: Propriedades da classe `Main` inferidas pela ferramenta ArgoUML.

3.2.3 Outras funcionalidades

Esta ferramenta destaca-se por possuir bastantes funcionalidades extra. É capaz de gerar código a partir de diagramas de classes, não só em Java mas também em outras linguagens. Contudo, no processo reverso, o programa mantém o código fonte original (como se pode ver na Figura 3.5), e gera código Java a partir do original. Esta ferramenta fornece ainda capacidades de edição de diagramas UML, incluindo diagramas de classes, estados, caso de uso, sequência, entre outros.

Após a análise e teste desta ferramenta conclui-se que o principal ênfase não foi a engenharia reversa dos projectos, mas sim a concepção de diagramas de classes e geração do código fonte. Embora seja uma ferramenta bastante interessante do ponto de vista de desenvolvimento de novos diagramas e projectos, não é tão interessante quando são consideradas as funcionalidades reversas. Importa ainda dizer que esta ferramenta não tem qualquer funcionalidade relativa a padrões.

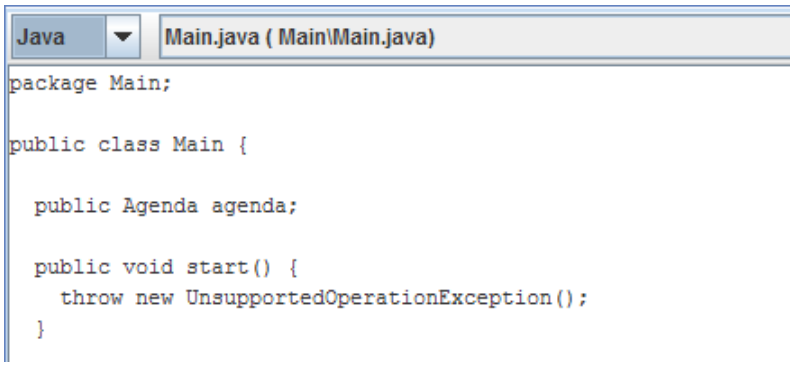

Figura 3.5: Editor de código fonte da ferramenta ArgoUML.

3.3 Fujaba (Reclipse)

A ferramenta denominada Fujaba responsável pela engenharia reversa do código Java e inferência dos padrões foi migrada para o *plugin* Reclipse [23]. Como a

ferramenta Fujaba com o módulo de engenharia reversa continua a existir, foram testadas estas duas implementações da ferramenta. O Reclipse é um *plugin* para o IDE Eclipse e para o testar foi necessário instalá-lo no mesmo. Por outro lado Fujaba necessita apenas do JRE.

Enquanto *plugin* esta ferramenta é a menos intuitiva de utilizar. A sua integração com o IDE poderia ser melhor conseguida. A integração num IDE é uma mais valia uma vez que permite ter outras funcionalidades bastante poderosas como por exemplo a edição de código conseguida a custo zero, ou ainda a edição gráfica fornecida pelo Eclipse. Por outro lado a ferramenta independente é mais simples de utilizar.

Esta ferramenta tem uma boa reputação na comunidade académica, principalmente pela expansibilidade que permite sob a forma de *plugins*. Contudo existe uma falta de suporte a esta ferramenta, não existindo uma comunidade ou uma *wiki* por exemplo para auxiliar na sua utilização.

3.3.1 Análise de código

O resultado da inferência do diagramas de classes com o *plugin* Reclipse não foi o melhor. A falha mais grave é a incapacidade desta ferramenta reconhecer algumas classes. No exemplo em questão as classes `Agenda` e `ComplexEvent` não foram inferidas como se pode ver na Figura 3.6.

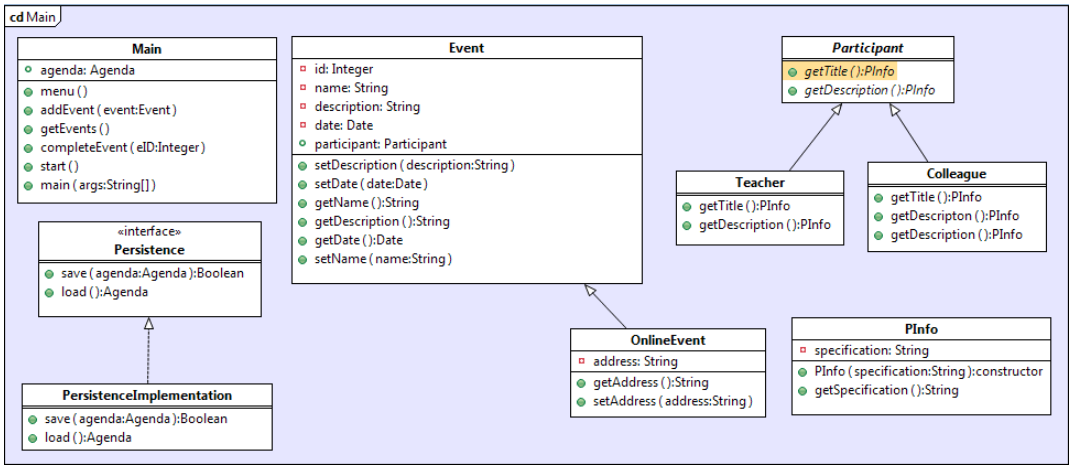

Figura 3.6: Diagrama de classes inferido pelo Reclipse.

Quando testada a ferramenta Fujaba tool suite nas mesmas condições, foi incapaz de reconhecer as mesmas classes (`Agenda` e `ComplexEvent`) e colecções com tipos (por exemplo, `ArrayList<Tipo>`). Esta impossibilidade foi observada ao testar o programa e tentar inferir o diagrama de classes. Esta é uma falha relativamente impeditiva, uma vez que não reconheceu na totalidade as classes do programa, como evidencia a Figura 3.7.

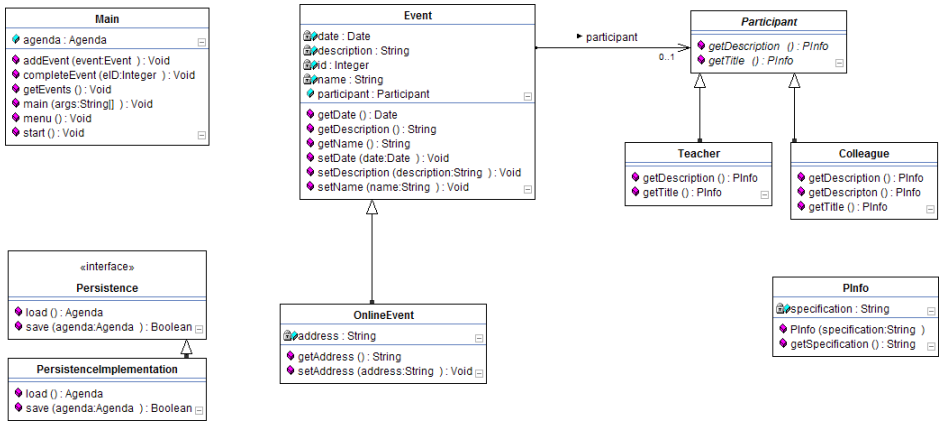

Figura 3.7: Diagrama de classes inferido pelo Fujaba.

O *plugin* Reclipse possui uma funcionalidade de reconhecimento de padrões de software. A funcionalidade de detecção de padrões necessita que um catálogo seja fornecido. Devido à falta de informação em relação à utilização da ferramenta não foi possível encontrar nenhum catálogo para fazer testar esta funcionalidade. Assim no *plugin* Reclipse foi impossível testar a detecção de padrões.

A ferramenta Fujaba, por outro lado, possui um *plugin* de inferência de padrões. A ferramenta foi capaz de reconhecer com sucesso o padrão de concepção contido na aplicação (Figura 3.8). Também reconheceu outro tipo de padrões como mostra a Figura 3.9. Para o padrão reconhecido foi ainda capaz de identificar um nível de certeza em percentagem. A identificação de padrões é no entanto um pouco confusa, identificando várias vezes o mesmo padrão por exemplo, e misturando com outros padrões.

3.3.2 Reconhecimento dos elementos

Nesta ferramenta os elementos Java não foram perfeitamente reconhecidos. Apenas classes e interfaces sem colecções foram reconhecidos. As agregações simples por composição também não foram inferidas.

Assim, após vários testes e tentativas com esta ferramenta em várias versões, conclui-se que para programas simples, com código seguindo uma notação comum esta ferramenta mostra várias dificuldades em cumprir a sua tarefa. Esta ferramenta é descrita como bastante poderosa, pelo que os testes efectuados são uma pequena amostra que não é conclusiva para todas as funcionalidades. A versão *plugin* mostrou-se bastante complexa de utilizar, não chegando mesmo a testar todas as funcionalidades. A versão *tool suite* por outro lado mostrou-se mais simples e capaz, sendo contudo necessário instalar à parte os *plugins* necessários. Esta ferramenta é um bom ponto de partida em termos de funcionalidades, contudo não contém todas as funcionalidades da ferramenta proposta.

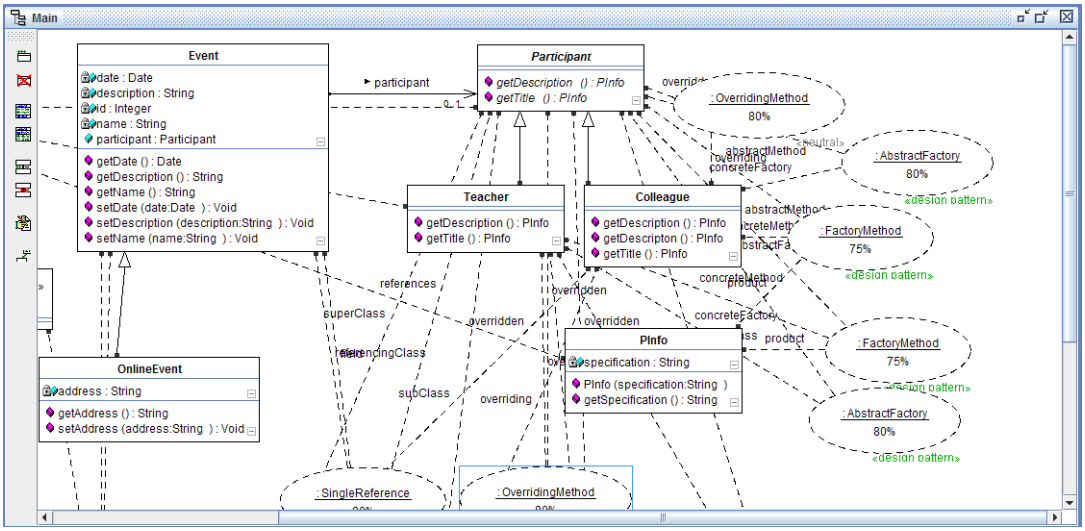

Figura 3.8: Padrões de concepção inferidos pelo Fujaba.

3.3.3 Outras funcionalidades

Esta ferramenta é largamente extensível por adição de *plugins* desenvolvidos por outros autores, o que estende as funcionalidades desta ferramenta. Desta forma existe muitas outras funcionalidades que são adicionadas de acordo com as necessidades do utilizador. Neste estudo apenas foram testadas as funcionalidades descritas atrás por serem as relevantes para o problema.

3.4 jGRASP

A ferramenta jGRASP [40] é um IDE não só capaz de analisar código, mas também de oferecer funcionalidades de edição e geração de código. Um dos seus principais focos é a análise de software para geração de diagramas. Quanto a diagramas, permite gerar *Control Structure Diagrams*, diagramas de classes e *Complexity Profile Graphs* para Java (e também alguns destes diagramas para outras linguagens). Permite ainda a análise de alguns dados de execução.

A ferramenta necessita apenas do JRE para executar. Devido ao facto de ser um *standalone*, algumas funcionalidades oferecidas como a edição de código são de baixa qualidade aparecendo por vezes de forma muito básica, o que não acontece em outras ferramentas que são integradas num IDE.

Esta ferramenta teve uma forte aceitação pela comunidade em geral, sendo usada por vezes em contextos académicos. Este facto fez com que esta ferramenta fosse analisada para perceber os seus pontos fortes.

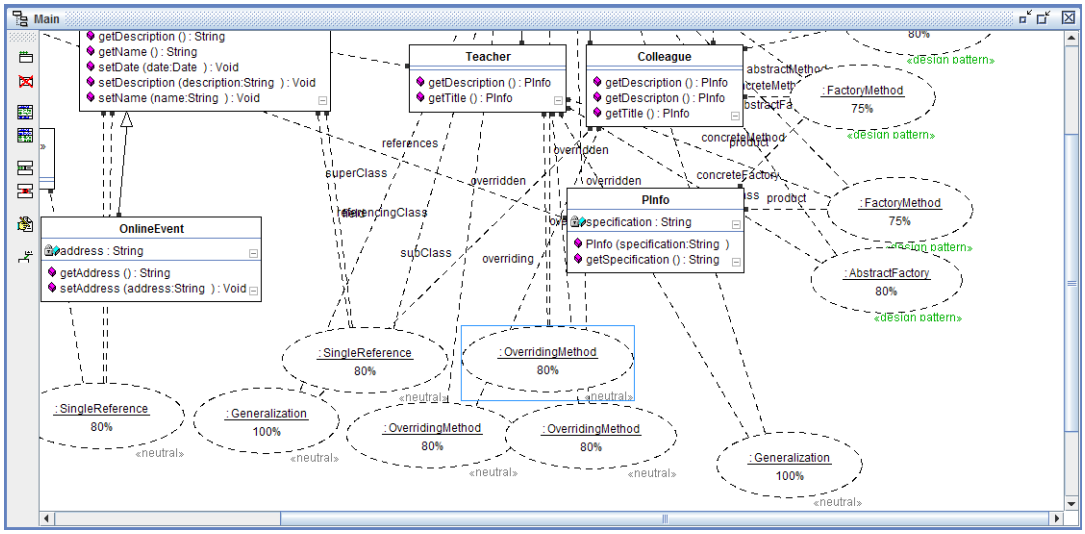

Figura 3.9: Outros padrões inferidos pelo Fujaba.

3.4.1 Análise de código

A ferramenta analisou de forma eficaz os ficheiros Java, bastando indicar a sua localização. Todo o processo foi automatizado, apresentando de imediato um diagrama de classes. Foi capaz de reconhecer todos os elementos Java, contudo apenas representa os elementos de forma muito simplificada no diagrama. Como se pode ver na Figura 3.10 o diagrama apresentado é algo incompleto e simplificado. Apenas são apresentados os nomes das classes e representados como a forma de um rectângulo. Este diagrama acaba por ter muito pouca informação e nada de útil adicionar ao problema.

3.4.2 Reconhecimento dos elementos

Esta ferramenta possui várias funcionalidades UML, como um ambiente de criação de diagramas de classes. Contudo nesta ferramenta o processo de engenharia reversa limita-se a inferir as classes e interfaces. Nenhuma informação adicional foi recuperada, nem relações entre elementos, nem informação sobre as classes, como se pode ver na Figura 3.10.

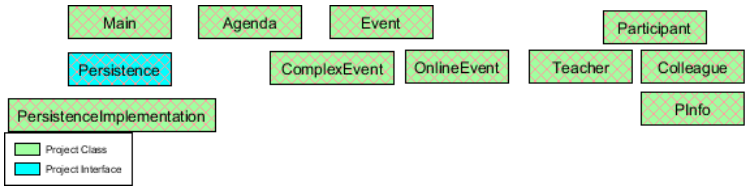

Figura 3.10: Diagrama de classes gerado pelo jGRASP.

Esta ferramenta é contudo capaz de reverter o código fonte das classes, como se pode ver na Figura 3.11. No entanto esta informação não é utilizada na representação da informação. Este comportamento da aplicação mostra que não foi colocada relevância no processo de engenharia reversa e inferência de informação, mas sim na produção de software.

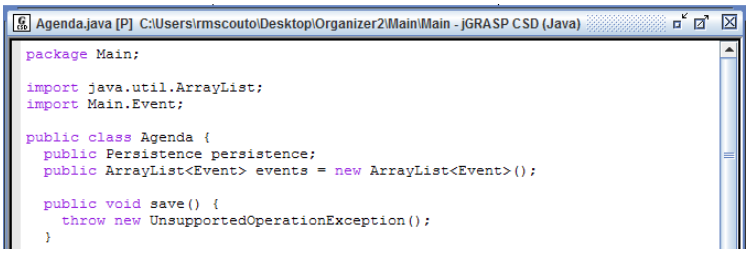

Figura 3.11: Parte do código da classe `Main`, visualizado no editor do jGRASP.

3.4.3 Outras funcionalidades

À semelhança do ArgoUML esta ferramenta também tem a capacidade de gerar código. Adicionalmente permite criar, compilar e executar código, bem como fazer operações de depuração. Quanto ao processo de engenharia reversa, a forma como esta ferramenta gera código é com base no código inferido, que é guardado aquando da análise dos ficheiros. Já no que diz respeito a diagramas, os diagramas de classes são os únicos elementos que podem ser gerados a partir do código fonte.

Esta ferramenta não é definitivamente centrada em engenharia reversa. O seu foco parece estar no desenvolvimento de software, com a funcionalidade adicional de mostrar diagramas de classes. Esta ferramenta parece adequada para uso académico uma vez que fornece um ambiente de desenvolvimento com auxilio de UML. A principal funcionalidade que poderá vir a ter interesse para a ferramenta proposta é a edição de código que a ferramenta oferece. Não existindo quaisquer tipos de operações com modelos ou padrões.

3.5 Ptidej

Esta ferramenta é neste momento desenvolvida com a cooperação de vários investigadores que contribuem com código e ideias que a vão melhorando [35]. As versões disponíveis no site (tool suite e black-box) encontram-se desactualizadas. A versão tool suite responsável por identificar padrões não foi capaz de reproduzir qualquer resultado no exemplo proposto.

A versão mais actual está presente nos repositórios do Ptidej. O acesso aos repositórios é limitado, e apenas atribuído para fins de investigação. Assim, esta

ferramenta não encaixa no ramo das ferramentas *open-source* por não ser de código aberto e acesso público. Por outro lado não é uma ferramenta comercial por permitir o acesso ao código fonte e não ter custos associados.

A obtenção e colocação da ferramenta em execução é um processo algo complicado, uma vez que as informações de instalação fornecidas pelo autor não são muito completas. Até conseguir executar esta ferramenta foi um processo algo custoso e trabalhoso, conseguindo apenas executar o *standalone*. Estes factos fazem com que a utilização da ferramenta não seja fácil nem trivial.

3.5.1 Análise de código

Esta ferramenta analisa o código na forma de *bytecode*. Para iniciar o processo é necessário indicar o arquivo Java a analisar. A ferramenta inicia então o processo de análise do ficheiro e extrai as informações, mostrando de seguida um diagrama de classes UML.

A ferramenta em questão foi capaz de analisar um arquivo Java (jar) e reconheceu todos os elementos (classes e interfaces). As entidades são correctamente identificadas e listadas, como se pode ver na Figura 3.12. Um diagrama de classes UML é mostrado assim que o processo de análise termina.

Figura 3.12: Listagem de entidades reconhecidas pelo Ptidej.

3.5.2 Reconhecimento dos elementos

Todos os elementos presentes no projecto em análise são identificados e representados com notação UML como evidencia a Figura 3.13. Apenas as relações de implementação e herança estão presentes no diagrama. Quanto às relações de agregação, composição e associação apenas algumas representações são mostradas, concluindo portanto que não foram inferidas.

A ferramenta apresenta sobretudo funcionalidades de identificação e análise de padrões e "micro-arquitecturas". O módulo de detecção de padrões possui um catálogo pré-definido de padrões de concepção, correspondentes ao catálogo de Gamma. Para além disso possui ainda um conjunto de padrões chamados

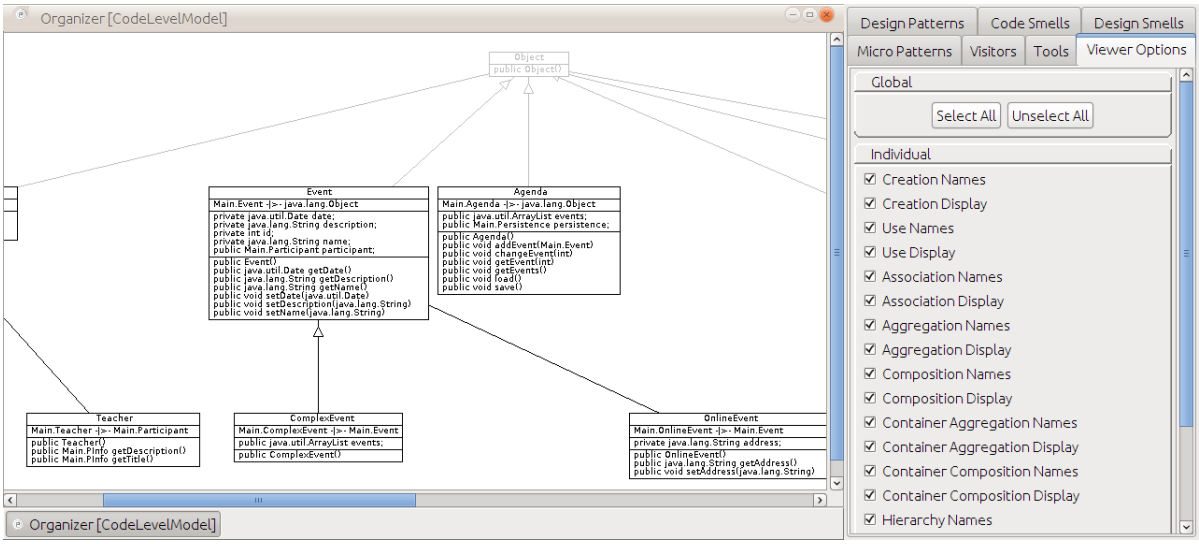

Figura 3.13: Diagrama de classes inferido pelo Ptidej.

"micro-padrões" que podem ser identificados. A ferramenta conta ainda com um conjunto de outros módulos de análise de código embutidos. Foi então analisado o projecto Agenda nesta ferramenta. Foi importado o código e utilizadas algumas das suas funcionalidades. Contudo, não foi possível chegar a qualquer conclusão. O programa não produziu qualquer resultado quando utilizadas as funcionalidades de análise de padrões e "micro-arquitecturas". Quando utilizadas, algumas análises davam um erro como *output*, outras mostravam mensagens que pouca informação davam mas nunca produziam resultados específicos. Desta forma não é possível concluir quanto à eficiência da ferramenta e dos seus diferentes módulos. Apenas foi possível concluir quanto à funcionalidade de representação de diagramas de classes UML, interface e tipo de funcionalidades disponíveis.

3.5.3 Outras funcionalidades

Para além da identificação de padrões que é a funcionalidade mais relevante, existem outras funcionalidades. Todas as outras funcionalidades são em volta da análise do código e padrões. Não são existem quaisquer funcionalidades de edição ou visualização do código, bem como de recompilação (na versão testada). Esta corresponde à versão *standalone*. Mais uma vez estas funcionalidades não produziram resultados visíveis que permitam concluir. Os diagramas apresentados são estáticos e não permitem interacção nem alteração da sua disposição gráfica.

3.6 Visual Paradigm

O Visual Paradigm é uma ferramenta comercial com excelente reputação. É um IDE de modelos com funcionalidades de edição, engenharia reversa e produção de código. Permite criar e inferir um largo número de diagramas (classes, estado, *use case* entre muitos outros). A sua interface é bastante complexa e disponibiliza um grande conjunto de operações. Funciona não só para programas criados de base mas também para programas inferidos.

Também esta ferramenta disponibiliza funcionalidades de *round-trip*. Gera não só código Java, mas também C++, C, C#, PHP entre muitas outras linguagens. O código analisado e gerado não tem de ser necessariamente orientado a objectos. Permite ainda analisar código nas mesmas linguagens que gera. Esta é uma ferramenta muito completa o que faz com que o seu preço seja relativamente elevado.

3.6.1 Análise de código

Sendo esta uma ferramenta comercial é compreensível que tenha os melhores resultados, e de facto é o que acontece [68]. Com o código gerado por esta ferramenta para o modelo proposto na Figura 3.1, ela é capaz de inferir o seu diagrama de classes, que é apresentado na Figura 3.14. Este diagrama inferido é o mais completo de todos os gerados pelas ferramentas analisadas. Sendo que esta ferramenta não é *open-source*, não é possível obter informações sobre detalhes de implementação da mesma. Contudo serve este diagrama como base de comparação para o que seria esperado por parte das outras ferramentas.

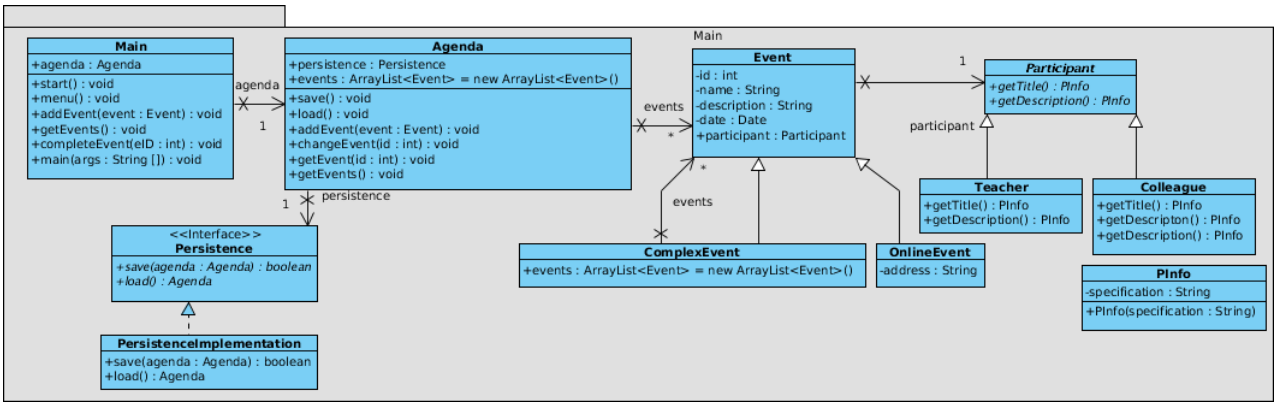

Figura 3.14: Diagrama de classes inferido pelo Visual Paradigm.

3.6.2 Reconhecimento dos elementos

Como se pode ver pela Figura 3.14 esta ferramenta inferiu correctamente todos as classes, interfaces, métodos e atributos, bem como as relações entre classes e interfaces. Apesar de não inferir um diagrama exactamente igual ao que gerou, infere um diagrama correcto e completo.

3.6.3 Outras funcionalidades

Esta ferramenta possui muitas outras funcionalidades, contudo não são relevantes para o problema em questão uma vez que esta é uma ferramenta comercial. As funcionalidades mais relevantes são as de construção e edição de modelos, geração de código e engenharia reversa e todas as funcionalidades que se encontram em torno dessas. Importa referir por exemplo a edição avançada de modelos, geração de outros tipos de modelos com base em diagramas de classes entre outras funcionalidades relacionadas.

3.7 Resumo

Após o estudo das ferramentas feito durante este capítulo foi possível tirar algumas conclusões que serão apresentadas de seguida.

Em geral as ferramentas disponíveis fornecem muitas funcionalidades úteis. A funcionalidade de inferência de diagramas de classes UML existe em várias ferramentas, contudo em poucas é bem conseguida. Este será uma questão a abordar na ferramenta em questão, tentando maximizar a qualidade dos diagramas inferidos.

A execução de operações em modelos mostrou-se bastante limitada, muitas vezes possibilitando apenas a geração de código ou alteração dos dados dos diagramas. Apenas uma ferramenta das analisadas permitiu inferência de padrões, e nenhuma delas permitiu níveis superiores de abstracção (PIM).

Também os interfaces das aplicações nem sempre se mostraram os melhores. Existiram dificuldades em perceber o funcionamento de algumas ferramentas, por vezes por serem muito simplistas, outras vezes por serem demasiado complexos. Aconteceu também faltar uma resposta por parte do programa: por vezes os programas não mostravam qualquer reacção positiva nem negativa após alguma acção, outras vezes paravam a execução devido a uma excepção sem qualquer explicação. Houve ainda casos em que as integrações em outras ferramentas foram relativamente limitadas.

A ferramenta proposta passa um pouco pela junção das funcionalidades que estas ferramentas apresentam de forma distribuída. Algumas das funcionalidades serão alvo de maior importância (como a inferência de padrões), outras melhoradas (como os interfaces e interacção). Também funcionalidades novas serão incluídas. É importante distinguir quais as funcionalidades que merecem mais

atenção por serem um maior contributo. Por exemplo a inferência de diagramas de classes à algo “trivial” encontrado num grande conjunto de ferramentas.

Capítulo 4

Desafios e cenários de utilização

4.1 Introdução

Será ao longo deste capítulo apresentado e aprofundado o problema em causa. Irá ser feita uma análise mais detalhada dos problemas específicos que se concretizam em problemas que são de esperar encontrar ao longo do processo de desenvolvimento da ferramenta proposta. Este capítulo resume os problemas que a ferramenta se propõe solucionar e serve de base para a fase de implementação.

Para cada um dos problemas que se pretende abordar será feita uma análise mais detalhada dos problemas subjacentes. Esta análise será mais concreta e detalhada, servindo já de orientação ao processo de implementação.

Outro ponto abordado serão os cenários específicos em que esta ferramenta se tornará útil. Será feita uma descrição detalhada de cenários existentes em geral, e que subconjunto desses cenários serão solucionados, bem como o motivo para tal se verificar. Serão assim apresentadas as condições necessárias para o programa funcionar correctamente.

Neste capítulo ficarão especificados os detalhes para cada um dos problemas em questão. Só desta forma é possível pensar sobre a fase de implementação e alcançar resultados satisfatórios. Apresentam-se ainda possíveis cenários de utilização que serão também indicações importantes para a fase de implementação.

4.2 Enquadramento do problema

As funcionalidades propostas para o projecto correspondem no fundo à resolução de três problemas específicos. Para compreender estes problemas e de que forma se tentarão solucionar, eles serão analisados em maior detalhe ao longo das próximas secções. Os problemas identificados correspondem aos três seguintes itens:

- Efectuar o processo MDA inverso desde o código fonte até ao PSM.

- Efectuar o processo MDA inverso desde PSM fonte até ao PIM.
- Efectuar a inferência de padrões de concepção num modelo PSM.

De seguida estes problemas serão analisados e especificados em maior detalhe.

4.2.1 Código fonte para PSM

Este problema da transformação de código em modelos enquadra-se, como já referido, no âmbito da MDA. De uma forma mais específica centra-se na problemática da transformação automática de modelos (em código fonte), neste caso inversa. O problema em causa assenta na funcionalidade de elevar o grau de abstracção do software num processo por passos. O processo começa pelo nível do código fonte (neste caso Java) que será abstraído para um diagrama de classes (correspondente ao PSM). Este problema ocorre com alguma regularidade no desenvolvimento de software e representa a necessidade de abstrair um sistema complexo, desactualizado ou desconhecido (ou parte dele) para uma representação mais útil.

Quando este problema é abordado do ponto de vista da implementação é necessário ter em conta vários factores, e permite concluir que se está perante um problema composto por três principais itens. Esses três itens correspondem à análise, representação intermédia e representação visual do diagrama.

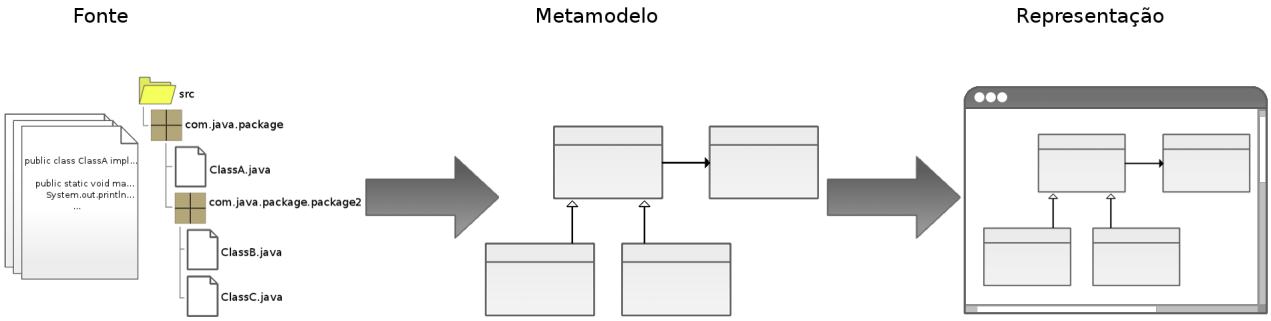

Figura 4.1: Representação do processo de conversão de código fonte para PSM.

1. O primeiro ponto do problema consiste na análise do projecto. Este processo prevê dois passos distintos: a análise da estrutura do projecto (análise da hierarquia de ficheiros) e a análise concreta de cada um dos ficheiros Java. A análise da estrutura do projecto (como conjunto de ficheiros) leva à necessidade de percorrer toda esta hierarquia, seleccionando os ficheiros relevantes ao mesmo tempo que se ignoram os ficheiros irrelevantes ao problema em questão. A análise concreta de cada um dos ficheiros leva

à necessidade de fazer uma análise semântica aos mesmos por via de um *parser*. Esta análise permite obter e extrair toda a informação expressa textualmente nesses ficheiros. A implementação da funcionalidade de iteração dos ficheiros juntamente com a implementação ou utilização de um *parser* para extracção da informação de cada um dos elementos são as tarefas necessárias para solucionar este problema. Este passo está representado do lado esquerdo na Figura 4.1, onde os ficheiros são analisados.

2. O segundo ponto deste problema é dependente do primeiro. Este consiste na problemática da representação intermédia da informação analisada no passo anterior. Depois de analisado um projecto é necessário preservar os dados extraídos de forma eficiente, onde não seja perdido nenhum detalhe relevante. Para tal é necessário ter uma forma que seja adequada para essa representação de informação. A MDA prevê uma forma para o fazer por intermédio de um metamodelo. Para solucionar este problema é necessário definir um metamodelo adequado não só à linguagem Java mas também que suporte o processo MDA inverso. Os dados extraídos no ponto anterior serão então mapeados no metamodelo definido. Este metamodelo deve ser completo, concreto e preciso ao mesmo tempo que não deve estar sobrecarregado de informação. Apesar de não ser necessário preservar toda a informação, é importante que ele esteja preparado para tal. Se o metamodelo permitir representar alguma informação que não é estritamente necessária (como por exemplo implementação dos métodos) poderá mais tarde permitir a extensão da ferramenta (por exemplo a uma ferramenta de *round-trip*). Resumidamente, a resolução deste problema necessita da definição de um metamodelo adequado ao problema em causa (considerando possibilidades de expansão). Necessita ainda de uma forma de mapear correctamente esses elementos no metamodelo. Na Figura 4.1 ao centro está representado este passo, que consiste no mapeamento no metamodelo.

3. O terceiro e último ponto a considerar é dependente da resolução dos anteriores. Este ponto apresenta necessidade de fazer a representação visual da informação do projecto, já representada no metamodelo. O que isto significa é que os elementos do metamodelo são transformados em elementos UML, ou seja, para cada elemento do metamodelo vai haver um elemento (visual) UML que será representado visualmente ao utilizador. Nem todos os elementos do metamodelo possuirão um mapeamento directo para UML o que levará à necessidade de processar previamente esse metamodelo. Uma forma eficiente de auxiliar o processo de representação é por adição de informação extra ao metamodelo. Esta informação refere-se mais em específico à informação visual para um elemento que contém a posição, cor, dimensões, entre outros. Apesar desta informação não estar directamente relacionada com o metamodelo, por motivos de eficiência será considerado a inclusão no mesmo. Este ponto fica apenas solucionado

quando for definido concretamente como são representados estes elementos, isto é, de que modo serão apresentados ao utilizador. Um requisito essencial é que a representação visual dos componentes seja interactiva (em que os elementos aparecem dispostos no ecrã e o utilizador interage, podendo até rearranjar os mesmos). Resta apenas decidir em que contexto serão apresentados: dentro de outra ferramenta (via *plugin*), de forma independente (aplicação Java), num formato externo (imagem), entre outras representações possíveis. De uma forma mais concreta, depois de definido esse formato é necessário fazer uma travessia pelo metamodelo, ler a informação visual associada a cada elemento o pré processamento. Finalmente calcular uma disposição para os elementos e proceder à sua representação. Resumidamente, este problema necessita que exista um mapeamento entre as entidades do metamodelo, as entidades UML, e uma forma visual de representação. De seguida necessita de uma forma de ajustar as propriedades visuais de acordo com os elementos e fazer a sua disposição (visual). Finalmente os elementos serão representados visualmente no formato escolhido. Mostra-se este passo na Figura 4.1 à direita, onde é exemplificada a representação visual.

4.2.2 PSM para PIM

A abstracção de modelos está normalmente associada à simplificação e/ou redução dos mesmos. Esta simplificação tem de ser um processo controlado e com lógica, caso contrário poderemos remover elementos essenciais ao programa tornando essa abstracção inútil. O processo de abstracção de modelos tem elevada importância na análise de um sistema. É mais fácil obter informação de um sistema com menos componentes, pois quanto maior for o nível de abstracção, menos informação específica existe para analisar. Se pensarmos em concreto num projecto de software Java vemos que o primeiro passo para compreender o que um projecto faz é normalmente pelo seu nome. A leitura de um resumo sobre esse mesmo projecto (por exemplo num manual) dá nos uma visão global sobre o projecto. Só depois de uma contextualização se analisa a informação mais específica. No caso da análise de modelos o mesmo se aplica. Na análise de um projecto uma abordagem possível é começar a um mais alto nível, neste caso PIM, e ir aumentando o nível de informação com o aumentar da especificidade dos diagramas (diminuindo a abstracção) passando para o PSM. Este é a motivação que leva ao problema de abstracção de um PIM em PSM.

A abstracção de um modelo PSM para PIM (neste caso) é parte do processo MDA. Neste caso estamos perante o aumentar do nível de abstracção de um modelo. Isto é conseguido por redução do número de elementos, da informação contida nos elementos ou até mesmo alteração dessa informação. Este problema é apenas composto por este ponto, que corresponde a abstracção de um modelo mais específico e com mais informação, o PSM num outro modelo mais genérico, que proporciona um ponto de vista de mais alto nível, o PIM.

Existem três requisitos a ter em conta. Em primeiro lugar é necessário ter um método de abstracção que pode ser um processo de filtragem, ou um processo de transformação. Em segundo lugar é necessário considerar a necessidade de ter uma representação visual, tal como existe para o PSM. O terceiro e último requisito é a implementação das funcionalidades de modo a que permitam a extensibilidade das funcionalidades no futuro.

Das funcionalidades previstas, a abstracção é a que requer maior atenção e especificação. Esta funcionalidade necessita que várias decisões sejam tomadas previamente. Em primeiro lugar é necessário definir qual o significado de um PIM neste contexto, isto é, que elementos, relações e atributos fazem parte deste modelo e de que forma. Depois de especificado o metamodelo do PIM é necessário definir como vão ser feitas as transformação dos elementos do PSM para o respectivo PIM. Estas transformações podem ser conseguidas de várias formas e variam de acordo com as abordagens. Uma abordagem consiste num processo interactivo em que o utilizador selecciona quais os elementos que pretende preservar no PIM destino. De seguida, de forma automatizada serão ajustadas as ligações e dependências necessárias [84]. Uma outra abordagem é a automatização completa do processo de transformação. Esta abordagem surge com base na especificação feita em *MDA Explained* [49]. Devido à natureza das transformação de PIM para PSM definidas por este autor percebe-se que naturalmente pode existir uma forma inversa para as mesmas. Esta abordagem permite uma automatização completa do processo de transformação. A forma como estas transformações ocorrem tem de ser bem definida para que o processo ocorra com sucesso. Definindo uma transformação como um método que recebe um elemento e retorna um (novo) elemento do mesmo tipo pode ser uma abordagem demasiado simplista.

```
public Jelement formatElement(JElement in) {
  if (in.getClass() == JInterface.class)
    return null;
  else
    return in;
}
```

Figura 4.2: Exemplo de transformação simples.

Pensando em qualquer regra aplicada a uma classe que seja dependente de uma outra classe, vemos que um método que receba uma classe e retorne uma outra classe, não vai ser suficiente. Este método não é suficiente para resolver o problema uma vez que o seu contexto é limitado à classe que está a analisar e não consegue aceder à informação de outras classes no sistema. Podemos ver um exemplo de uma regra simples e insuficiente na Figura 4.2, onde facilmente se percebe que a transformação está limitada a transformações no

elemento que é passado como argumento ao método (`JElement in`). Desta forma é possível ver que um dos problemas a abordar será a decisão de como proceder às transformações dos elementos, de forma a que o contexto da transformação seja global, e não relativo apenas ao elemento a transformar. Por fim é necessário ter em conta que o processo de transformação deve ser implementado de forma a que possa facilmente ser refinado, estendido ou alterado, caso necessário.

O problema da representação visual será em tudo similar ao problema da representação visual de um PSM, problema esse já apresentado. Uma das diferenças neste caso é que é possível aproveitar algumas propriedades visuais do PSM caso estas tenham sido processadas previamente. Se o processo de computação das propriedades visuais for bem implementado vai permitir que essa funcionalidade seja utilizada para processar as propriedades visuais do metamodelo PIM.

A necessidade da extensibilidade deste componente é baseada sobretudo no processo de transformação. A relevância de permitir a extensão deste componente baseia-se na necessidade do utilizador poder refinar ou alterar este processo. Este problema surge por poder existir a necessidade do utilizador querer uma forma de abstracção diferente, de propor um PIM diferente, ou querer um maior nível de abstracção no PIM gerado. Assim, este problema, consiste na existência da necessidade de o utilizador poder definir as suas próprias regras de transformação. Dessa forma este componente pode a qualquer altura ser alterado e estendido de acordo com as necessidades do utilizador não ficando limitado à especificação inicial.

4.2.3 Inferência de padrões num PSM

O problema da inferência de padrões num modelo não é novo, pelo contrário, é um problema já abordado por vários autores, havendo até várias ferramentas capazes de o fazer sendo a sua utilidade já reconhecida. Resumidamente é um nível de abstracção superior que irá contribuir para o processo de análise. Esta abstracção permite perceber quais as "peças" que constituem um software. Percebendo de que forma essas "peças" interagem é possível perceber qual o objectivo de um software, a sua organização conceptual e ainda alguns aspectos sobre o seu funcionamento. Essas "peças" correspondem no fundo a padrões de concepção que existem ao longo código fonte de um software. As abordagens descritas por outros autores identificaram alguns dos problemas encontrados neste processo, contudo esta funcionalidade no contexto apresentado vai diferir das abordagens apresentadas. Esta variação reside no facto de que este reconhecimento vai ser feito com base em informação já inferida e utilizada para outros fins nomeadamente gerar o diagrama PIM. Outros autores propõem que a análise de padrões seja feita directamente a partir do código fonte sem qualquer relação com modelos abstractos. Contudo, de forma a tirar proveito do trabalho já feito, a inferência de padrões será feita com base nos modelos inferidos. Face a esta especificação, o problema da inferência de padrões modelo PSM contará com

quatro componentes. O primeiro será o modo como a análise do código, ou mais em específico do modelo será feita. O segundo componente será a definição da forma como a pesquisa dos padrões será feita. O terceiro componente a analisar será de que forma a identificação de padrões poderá ser flexível e facilmente extensível. O quarto e último componente a analisar será o já abordado problema da representação visual.

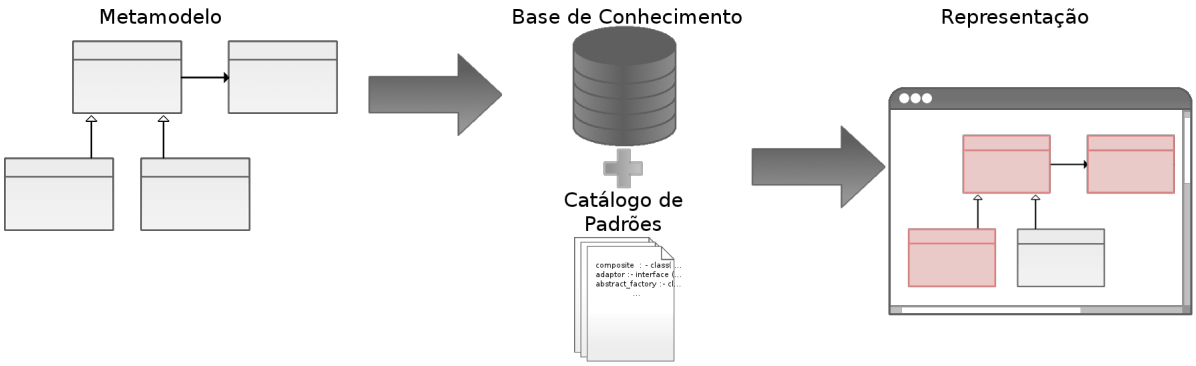

Figura 4.3: Representação do processo de inferência de padrões.

O primeiro passo do processo de identificação de padrões consiste em criar uma base de conhecimento com a informação contida no projecto, representado na Figura 4.3 à esquerda. Só desta forma é possível começar a análise sobre os dados contidos no modelo. Este problema será apresentado e detalhado mais à frente. Depois de definido concretamente um padrão de concepção a nível computacional, fica especificado também quais os elementos têm de ser extraídos da representação do projecto. É possível concluir que tendo o metamodelo dos padrões definido, a sua identificação depende de dois aspectos: em primeiro lugar de uma definição correcta e precisa dos padrões, e em segundo lugar de uma base de conhecimentos (com factos) que permita a pesquisa de padrões. Existem diversas abordagens sugeridas por diferentes autores que permitem concluir que a mais adequada depende sobretudo do contexto do problema.

A representação intermédia do projecto, da base de conhecimento e dos padrões estão relacionadas entre si. Existe uma forma de transformação directa entre elas na ordem apresentada. A base de conhecimento com informação sobre o programa é o ponto de partida para solucionar o problema da identificação de padrões. É proposto por alguns autores utilizar o metamodelo do PSM como base de conhecimento. Desta forma é possível reaproveitar o trabalho já feito, juntamente com uma comparação estrutural como processo de pesquisa de padrões. Uma outra abordagem é a utilização de uma tecnologia externa. Utilizando uma tecnologia externa é possível tirar partido de todas as suas funcionalidades porém necessita de uma adaptação dessa tecnologia ao contexto em questão. Neste caso é necessário também definir qual a tecnologia em questão (e uma ferramenta que a implemente) bem como as regras de transformação de

informação para esta tecnologia. Por fim será necessário desenvolver uma ferramenta que implementa as funcionalidades de mapeamento, interacção e troca de informação.

Uma das necessidades encontradas é a de ter uma forma concreta de definir um padrão. Esta definição tem uma elevada importância pois só assim o processo de identificação será bem definido e controlado. De uma boa definição e implementação desta funcionalidade depende a extensibilidade do projecto e a sua utilidade prática. Uma definição estática ao nível de implementação dos padrões identificáveis resulta numa funcionalidade rígida, dificilmente extensível e facilmente fica desactualizada. Uma definição de padrões num formato parametrizável será em princípio uma tarefa mais complexa para analisar e importar estas definições. Apesar da parametrização ser mais complicada, é a que permite que este módulo seja posteriormente estendido e adaptado a contextos específicos. Tendo em conta as limitações da primeira alternativa, a segunda será o problema escolhido para resolver. Resumido, é necessário implementar um módulo que permita reconhecer definições de padrões externas, importar essas definições e aplicá-las ao projecto em análise, utilizando uma ferramenta externa para fazer a pesquisa de padrões. Na Figura 4.3 ao centro é mostrado de que forma este catálogo entra no processo de identificação de padrões.

O formato externo onde são definidos os padrões é chamado "catálogo de padrões". Este catálogo consiste na definição de um conjunto de padrões (possivelmente com alguma informação adicional). Só desta forma o utilizador pode definir um catálogo personalizado e identificar esses padrões no software em análise. Esta definição terá de ser feita num ficheiro externo e que possa facilmente ser editado pelo utilizador. Assim, é necessário especificar um formato de definição de padrões, uma forma de importar e interpretar esse catálogo e por fim utilizar essas regras na pesquisa de padrões.

O último problema a abordar é a representação visual sendo que esta será feita de forma análoga à representação do PIM e do PSM. Comparando as Figuras 4.1 e 4.3 (à direita) podemos constatar que a representação é similar em ambos os casos, tendo uma ligeira alteração no caso dos padrões. Isto permite perceber que o mesmo módulo, devidamente ajustado, poderá ser usado em ambas as representações. Desta forma será vantajoso tirar partido do trabalho feito para a representação dos outros modelos, fazendo apenas as alterações estritamente necessárias. O ideal seria reaproveitar todo o trabalho feito porém existem algumas alterações que são realmente necessárias, nomeadamente na representação dos padrões, uma vez que este problema não foi tido em conta previamente. Uma abordagem possível a este problema seria ter uma representação igual à dos outros modelos (PIM e PSM), realçando os elementos que pertencessem a um padrão. Desta forma seria apenas necessário fazer uma pequena alteração na representação, que permitisse marcar (por exemplo) os elementos que sejam parte de um padrão. Uma variação a esta abordagem consiste em reduzir a informação mostrada neste diagrama, ignorando por exemplo informação de métodos e atributos. Esta variação pode ser útil pois abstrai pormenores irrelevantes quando

o utilizador pretende apenas analisar os padrões.

4.3 Cenários de utilização

Esta secção faz uma apresentação mais específica e detalhada do problema em causa no que diz respeito à utilidade prática da ferramenta em questão. Serão descritos possíveis cenários em que a utilização desta ferramenta se torna útil e proveitosa. Estes cenários consistem em situações hipotéticas que facilmente podem ser encontrados na vida real. Os cenários descritos serão separados de forma a salientar cada uma das funcionalidades da ferramenta. Serão ainda apresentados os problemas subjacentes a cada um dos cenários propostos.

4.3.1 Produção de um PSM

De seguida será descrito um cenário que permite perceber a utilidade deste módulo. Podemos considerar um programador que vai continuar a implementação de um projecto Java. Este projecto não é um projecto novo, vai já conter código desenvolvido, classes, interfaces, etc. Obviamente que o programador não vai saber exactamente qual a arquitectura do projecto, vai necessitar de uma contextualização por não estar previamente envolvido no projecto. Este programador pretende fazer uma extensão das funcionalidades do projecto, refazer partes do projecto, implementar funcionalidades incompletas e até optimizar partes problemáticas. Esta adaptação não é um processo trivial que se possa resolver facilmente. O programador não vai querer perder tempo a ler o código implementado e perceber detalhes que lhes são completamente inúteis numa primeira fase. Contudo ele tem necessidade de saber o que se passa ao nível do código de uma forma abstracta. Vai querer perceber a organização dos componentes, as interacções e informação estrutural e organizacional. Este utilizador vai querer no fundo ter uma visão global e imediata do sistema em causa, ou de partes dele. Ele pode por exemplo analisar um pacote Java de cada vez, percebendo por partes o projecto. Dessa forma pode até perceber quais os pontos mais críticos e onde pode ser mais vantajoso optimizar ou começar a trabalhar. O utilizador vai desenvolver o projecto num IDE, neste caso NetBeans, e vai querer estas funcionalidades disponíveis a custo zero, sem necessidade de instalar novo software, migrar a aplicação entre outros processos, tendo os resultados dentro da aplicação.

Podemos considerar um outro cenário onde este módulo seria vantajoso. Neste caso, um programador vai integrar numa equipa de desenvolvimento e consideremos que esta possui um projecto em desenvolvimento. O novo programador não vai estar a par do processo de desenvolvimento e por esse motivo não vai conhecer a arquitectura do sistema em causa. Apesar de o utilizador desconhecer a implementação e os seus detalhes, ele tem necessidade de o perceber para que possa integrar a equipa e começar a produzir rapidamente. A primeira

abordagem em que se pensa seria entregar o código ao novo programador. Contudo, esta abordagem não seria viável porque iria demorar demasiado tempo, ao mesmo tempo seria uma tarefa difícil e custosa para ele. Para além disso, seria um processo pouco produtivo porque seria de esperar que no final o utilizador continuasse com dúvidas por não conseguir uma visão geral do projecto (para além de que esta tarefa não iria motivar o novo programador). Contudo existe uma outra alternativa mais vantajosa para todos que consiste em mostrar algo mais interessante que contém o nível de informação adequada ao novo programador (e que será mais do seu agrado). Para conseguir esta outra forma a equipa de desenvolvimento vai querer gerar diagramas de classes representativas da aplicação em causa. Com estes diagramas poderá fazer uma apresentação mais adequada ao novo programador e dar uma nova perspectiva de mais alto nível que certamente será mais útil, mais rápida e fácil para o programador de adquirir. Com esta abordagem o programador estará melhor integrado, terá uma melhor ideia do aspecto arquitectural da ferramenta o que faz com que ele se coloque mais rapidamente no local apropriado e comece a produzir resultados de acordo com o esperado. Mais uma vez, a condição colocada será que os programadores estejam a desenvolver o seu projecto no IDE NetBeans.

Estes são dois cenários que mostram como este módulo se pode facilmente tornar útil num ambiente de produção e desenvolvimento, por exemplo como mostrado numa equipa de desenvolvimento de software. Estes cenários levantam problemas reais para fins bem definidos que serão abordados e tratados de forma conveniente.

4.3.2 Abstracção de um PIM

Para perceber a utilidade deste tipo de diagramas, vamos considerar três novos cenários de utilização apresentados de seguida.

Consideremos um primeiro cenário onde um jovem empreendedor (líder da equipa) começou um projecto que será a base da sua empresa. De início considera que ele só consegue desenvolver a aplicação e dar resposta a todos os pedidos. Contudo, com o passar do tempo e devido ao seu sucesso, ele só não consegue produzir para acompanhar tantos pedidos de módulos adicionais. Assim, ele vê-se obrigado a pedir a alguém para ingressar na sua empresa, e dessa forma vai contactar alguns colegas. Rapidamente se apercebe que estando sozinho a desenvolver, e agora que o seu projecto já tem uma dimensão considerável, que não será fácil para alguém vindo de fora perceber toda a arquitectura e detalhes de implementação. Por outro lado também não é necessário que este novo elemento consiga perceber toda a estrutura do programa. No fundo o que é necessário é que este novo elemento tenha uma ideia geral da organização da aplicação, e a partir daí desenvolva o seu módulo. Este módulo será independente do resto da aplicação ainda que possa ser integrado, pelo que o utilizador não tem necessidade (nem vontade) de perceber os detalhes do resto da aplicação (e possíveis outros módulos). O que o jovem empreendedor necessita é de uma forma de

mostrar uma simplificação, de mais alto nível que um PSM (por este conter demasiados detalhes) para mostrar ao seu novo colega. O nível de abstracção ideal esconde os detalhes que lhes são indiferentes e apenas irão mostrar a informação relevante para que o novo colega não esteja completamente alheio ao projecto como um todo. Como líder não quer ter trabalho adicional de fazer diagramas (sujeitos ao erro humano), quer ter a possibilidade de reaproveitar trabalho. Com a utilização da ferramenta proposta, o custo de ter este diagrama seria o mesmo de gerar um PSM, com a diferença que o utilizador tem que seleccionar a opção que permite gerar PIM.

Um outro cenário que demonstra facilmente a utilidade deste módulo será descrito de seguida. Consideremos novamente um cenário de integração de um elemento numa equipa de software, com a implementação de um projecto a decorrer. Este novo utilizador será apenas responsável por uma das partes neste projecto. Para perceber exactamente o que está a tratar pode decidir gerar um PSM que irá analisar. Porém, se ele gerar um PSM para todo o sistema poderá ser informação a mais e fazê-lo perder demasiado tempo a analisar detalhes que lhe são indiferentes, tornando-se este processo contraproducente. Este programador precisa apenas de analisar ao detalhe a parte que lhe diz respeito (com um PSM), e ter uma visão de mais alto nível do resto do projecto (com um PIM). Da mesma forma outros elementos da equipa podem seguir a mesma abordagem para terem uma visão geral do projecto ao longo do seu desenvolvimento sem demasiados detalhes.

Um cenário final e distinto dos apresentados é o que vai ser descrito de seguida, que mostra uma outra possível utilização. Consideremos o caso de uma aplicação que já está implementada. Existe um produto gerado por este código fonte. Vai ainda existir um utilizador que é responsável por migrar esta aplicação para uma outra linguagem e plataforma, porque a plataforma actual está obsoleta (por exemplo). Este utilizador pode nem estar bem por dentro do âmbito da aplicação e não saber o que esta aplicação faz ao certo, logo não vai querer desenvolver uma aplicação de raiz (sendo que essa abordagem não seria uma migração). Assim este utilizador vai querer saber quais os elementos que são relevantes para o processo de migração. Vai querer também saber que elementos são específicos da plataforma (uma vez que está a proceder à migração para uma outra plataforma). Por fim sendo uma migração entre plataformas, que podem elas ser semelhantes, este utilizador vai ter a necessidade de definir o seu metamodelo PIM e escrever as suas próprias regras de transformação. Definindo o seu próprio metamodelo PIM como extensão de um metamodelo base pode ser vantajoso porque alguns detalhes podem migrar sem alterações entre estas duas plataformas, como é exemplo o conceito de classe, método ou atributo. Por escrever as suas próprias transformações pode aumentar a produtividade e até permitir a geração de um PIM com outros fins.

Com estes três cenários é possível perceber como a abstracção de um PSM para um PIM é útil em diversos cenários práticos. É possível perceber com alguma facilidade que esta ferramenta se torna útil tanto em cenário de desen-

volvimento, manutenção como na migração de aplicações.

4.3.3 Inferência de padrões

Nesta secção serão apresentados os últimos cenários de utilização, correspondentes ao terceiro componente da ferramenta proposta. Novamente serão propostos três cenários distintos onde a ferramenta será útil.

Comecemos por considerar um programador que está a fazer a análise de um software, e com auxilio dos módulos atrás já fez a análise estrutural do software. Contudo, a informação que retirou não é suficiente, que saber algo mais sobre o software, sobre a organização conceptual dos componentes. Quer ainda que esta informação lhe seja disponibilizada de forma abstracta sem necessidade de analisar o código fonte componente a componente. Esta informação mostra ainda de que forma os componentes que constituem o projecto se agrupam em padrões. O programador fica ainda a saber de que forma os elementos interagem e qual o seu papel específico no projecto. O utilizador vai querer uma representação semelhante à dos diagramas já descritos por já estar habituado a essa representação e dessa forma não tem de fazer um esforço adicional para os perceber.

Um cenário completamente diferente é o caso de um utilizador que está a iniciar o projecto de desenvolvimento. Este projecto tem como pré-requisito ter uma elevada qualidade e código bem construído. Para tal este utilizador define um conjunto de padrões de qualidade, isto é, formas de construir software que ele considera correctas e vantajosas. Ao longo do processo de desenvolvimento e com regularidade, o utilizador vai analisar o seu código fonte em pesquisa desses padrões. Com o aumentar de linhas de código, o aumentar do número de padrões encontrados vão ser um indicador do aumentar de qualidade do código fonte, ou por outro lado, a sua ausência um sinal de que algo está errado. Estes padrões podem assim ser uma medida de qualidade relacionada com o número de padrões que surgem com o evoluir do projecto. Caso este projecto fosse elaborado por mais que uma pessoa, o responsável poderia utilizar esta ferramenta para controlar a qualidade do software desenvolvido à medida que o projecto avança. Esta funcionalidade permite ainda que quem supervisionar o projecto (seja o próprio utilizador ou outra pessoa), consiga ter uma visão do sentido em que o processo de desenvolvimento está a avançar, isto é, quais os padrões mais utilizados, menos utilizados, em que cenário é utilizado cada um deles, etc.

É descrito de seguida o último cenário proposto. Considere-se um docente que está a receber trabalhos de programação avançada em Java. Este docente tem de receber todos os projectos de uma turma, analisar o programa e ver se está tudo correcto, isto é, se o projecto está correctamente implementado. Para além disso precisa de uma medida de qualidade do código (para além de ver o programa a executar). O docente estará naturalmente saturado de ver projectos e analisar código seria a última coisa que lhe passaria pela cabeça.

Assim, precisa de algo que o ajude a ter alguma medida de qualidade. Para tal, definiu o que considera de "padrões de qualidade", que consistem no fundo a formas de elaborar código que o docente considera serem fundamentais para a obtenção de um código fonte com qualidade. Estes são os padrões que ele espera encontrar no código dos alunos por serem formas correctas de fazer as formas (e de forma que o docente leccionou). De forma análoga pode definir "padrões de má qualidade", que são o inverso, isto é, formas erradas de fazer as coisas que não deveriam constar no projecto. Este número de padrões encontrados, poderia por exemplo, ajudar o docente a fazer o ajuste da nota de um aluno sem ter necessidade de olhar para o código do mesmo. Como restrição o docente só tem necessidade que os alunos desenvolvam os projectos em NetBeans e tenham o *plugin* instalado, uma vez que o catálogo é carregado na altura da análise.

Estes cenários demonstram de que forma é possível obter um benefício prático com esta ferramenta em diversos ambientes. Em primeiro lugar num ambiente de manutenção e análise de software já existente. Num segundo cenário estamos perante um processo de desenvolvimento iniciado a partir do zero. Por fim encontramos um caso de avaliação de projectos recorrendo à análise de padrões no software.

4.4 Resumo

Ao longo deste capítulo foi apresentado mais em detalhe o problema em causa. Cada módulo proposto foi apresentado detalhadamente, especificando as propriedades que caracterizam os problemas subjacentes a cada um deles. A especificação dos problemas permite pensar concretamente em cada um dos problemas e propor soluções para os mesmos.

Neste capítulo foram também apresentados diversos cenários de utilização, onde se mostra a utilidade da aplicação em contextos reais. Para cada um dos módulos foram apresentadas várias propostas, mostrando assim várias possibilidades de utilização da ferramenta proposta.

Esta análise dos problemas a abordar é importante para permitir perceber mais em detalhe os problemas relacionados com as funcionalidades propostas e de que forma se concretizam em problemas na fase de implementação. Os casos de estudo apresentados indicam implicitamente um conjunto de detalhes que será importante ter em conta durante o processo de implementação.

Depois de detalhados os problemas de cada funcionalidade, é então possível pensar sobre a implementação da solução. Ao longo da próxima secção será apresentada uma proposta de solução para cada um dos problemas apresentados. Depois de detalhadas as propostas poderão então ser implementadas como prova de conceito, para mostrar as implicações das decisões tomadas.

Capítulo 5

A ferramenta MapIt

5.1 Introdução

É neste capítulo que vão ser apresentadas as decisões conceptuais tomadas que são relevantes para a implementação da solução. O resultado esperado é uma ferramenta denominada *Model and Patterns Inferring Tool* (MapIt) com diversas funcionalidades que solucionam os problemas previamente propostos.

A ferramenta MapIt que implementará as funcionalidades propostas como prova de conceito será constituída por vários módulos. Estes módulos estão organizados hierarquicamente e deverão interagir entre si para atingir o objectivo proposto. Como resultado propõe-se quatro módulos distintos. Dois desses módulos serão independentes, um deles abstrairá os dois anteriores, e o quarto abstrairá os módulos inferiores, fornecendo interacção com o utilizador. Estes módulos serão disponibilizados como bibliotecas que poderão posteriormente ser incorporadas por projectos.

O resultado final proposto é um *plugin* que pode ser adicionado ao IDE NetBeans, tirando partido das suas funcionalidades. Ao mesmo tempo demonstrará que esta ferramenta pode facilmente ser utilizada e produzir bons resultados.

Ao longo deste capítulo serão apresentadas as decisões tomadas em relação a cada problema encontrado, que servirão de base para o processo de desenvolvimento de cada um dos módulos da ferramenta MapIt.

5.2 Proposta de resolução dos problemas

Apresentados atrás os problemas específicos que a ferramenta MapIt irá solucionar, será agora apresentada para cada um desses problemas uma proposta de solução. Estas decisões foram tomadas com base no estudo feito no estado da arte e deduções baseadas nos objectivos propostos. Os cenários de utilização deram também indicações na altura de tomar as decisões. As propostas apresentadas são as decisões conceptuais consideradas mais adequadas ao contexto

dos problemas em questão.

5.2.1 Código fonte para PSM

A transformação de código fonte num modelo PSM será o primeiro ponto a abordar. A primeira tarefa corresponderá à análise e abstracção do código fonte num modelo específico de linguagem (PSM). Este PSM será descrito num metamodelo definido para esta linguagem. O metamodelo representado na Figura 5.2 é adequado à linguagem Java e tendo em conta os detalhes da funcionalidade proposta (a representação visual e a capacidade de manipulação dos dados do modelo). O metamodelo descrito contempla todos os elementos da linguagem Java necessários à representação de um projecto. Para além disso foi pensado de forma a permitir gerar código fonte com base no mesmo. Esta extensão do metamodelo foi pensada com a possibilidade de extensibilidade da ferramenta MapIt em mente, como se pode constatar na Figura 5.1, onde é mostrado um excerto de código do metamodelo que evidencia a possibilidade de incorporar o corpo de um método no mesmo.

```
public class JMethod {
  private String name;
  private int modifier;
  private String returnType;
  private String body;
  ...
```

Figura 5.1: Excerto da classe JMethod.

O primeiro passo consiste na análise do código fonte e é feito por meio de um *parser*. A forma de análise de código escolhida for a análise do código fonte textual. Esta decisão faz sentido se pensarmos no contexto desta aplicação. Primeiro, esta aplicação pretende incidir sobretudo ou no processo de desenvolvimento e implementação de código, ou numa fase de manutenção onde há necessidade de entender e analisar projectos e o código fonte está disponível. Face a isto, a decisão de analisar o código no seu formato textual parece adequada. Para além disso, a analise feita em outro formato, nomeadamente em *bytecode*, seria uma análise feita em código que já tinha sofrido optimizações por parte do compilador com possíveis alterações que tivessem impacto directo na estrutura e informações do código em questão. Tomada esta decisão é então necessário adoptar um *parser* para a linguagem Java. Foi escolhido o JavaParser [26], que permite analisar todos os elementos de uma classe Java fornecida. Esta escolha foi tomada por permitir analisar todos os elementos, classe a classe, extraindo apenas a informação necessária. Se posteriormente existir necessidade de alterar a informação que é mapeada no metamodelo, será ao nível do módulo

que abstrai o *parser* que as alterações têm de ser feitas. Desta forma é feita uma análise estrutural completa a todos os ficheiros de um projecto. As funcionalidades do *parser* são utilizadas dentro de um mecanismo de iteração de ficheiros de um projecto.

O próximo passo será a decisão do formato intermédio da representação dos dados extraídos. A representação escolhida foi a utilização de um metamodelo, decisão esta tomada com base no estudo prévio feito nesta àrea. Vários autores utilizam a metamodelação como forma de representação intermédia de uma aplicação. Devido às funcionalidades a desenvolver sobre os dados extraídos, um metamodelo da linguagem Java será a forma ideal [31]. O metamodelo deve contemplar todos os elementos da linguagem Java bem como a hierarquia de ficheiros num projecto, nomeadamente os packages. Este metamodelo tenta ser o mais completo possível para permitir à partida a sua extensibilidade. O diagrama do metamodelo encontra-se representado na Figura 5.2 na forma UML.

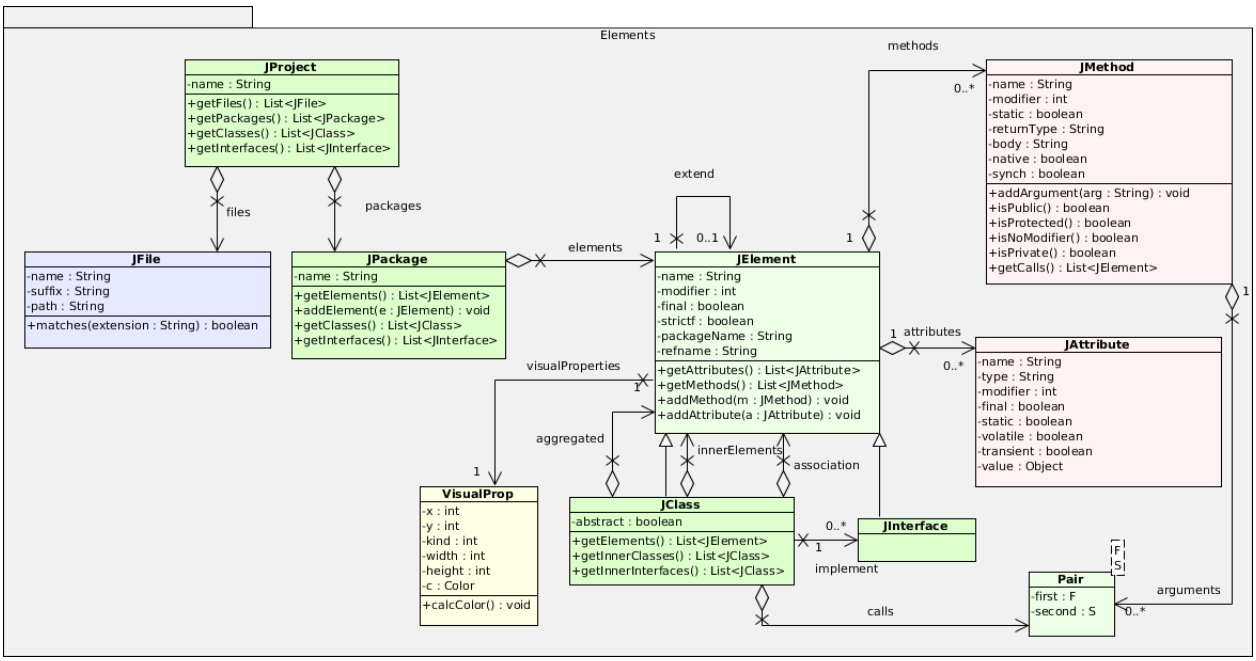

Figura 5.2: Metamodelo Java utilizado.

A decisão do tipo de ferramenta a desenvolver foi por uma ferramenta de engenharia reversa que permitirá inverter parte do processo MDA. A alternativa a esta abordagem seria uma ferramenta de *round-trip* que permite não só a inversão do processo, mas também o processo directo. Como o objectivo principal da ferramenta MapIt é suportar o processo MDA inverso, a funcionalidade *round-trip* não a prioridade (embora seja deixado espaço para tal). A implementação de *round-trip* iria consumir tempo e trabalho a implementar, quando não é o foco deste trabalho.

A linguagem escolhida é a linguagem Java. Esta foi escolhida da mesma

forma que poderia ter sido escolhida uma outra qualquer linguagem de programação. Esta é uma linguagem sólida, robusta e madura. É amplamente utilizada no mercado de trabalho e devido à familiaridade com a mesma será a linguagem para análise adoptada. A escolha da linguagem de programação não é uma escolha permanente, pelo que outra linguagem poderá ser adaptada à ferramenta MapIt desde que ela seja representável pelo metamodelo (que seja orientada a objectos, basicamente). Nesta alteração apenas o módulo de *parsing* tem de ser alterado. Na Figura 5.3 mostram-se os principais métodos disponibilizados pelo módulo que implementa a funcionalidade de gerar um PSM. A alteração da implementação destes métodos é transparente para as aplicações que utilizam este módulo, pelo que podem ser alterados para serem utilizados com outras linguagens. Fica assim aberto espaço a expansibilidade no que diz respeito à linguagem alvo da ferramenta.

```
public class SystemBuilder {
  private PackageBuilder pb;
  private ArrayList<JFile> filelist;
  private String path;
  private File root;
  private String pname;

  //Cria uma instância de um metamodelo, para um conjunto de ficheiros
  public void buildSystem(ArrayList<String> files, String pname) { ...

  //Cria uma instância de um metamodelo, dado o caminho de um projecto
  public void buildSystem(String path) { ...

  //Devolve a implementação do metamodelo
  public JProject getProject() { ...
```

Figura 5.3: Excerto da classe SystemBuilder, *facade* para o JPSI.

A técnica escolhida para a transformação de modelos (transformação MDA inversa) foi a automatização completa do processo. Duas abordagens foram previamente referidas, a automática e a interactiva. A abordagem interactiva poderia produzir resultados mais refinados uma vez que o utilizador participa no processo, mas por outro lado exige a interacção do utilizador, consumindo tempo e trabalho. Como um dos objectivos do trabalho é proporcionar a automação completa do processo, a interacção com o utilizador será reduzida ao mínimo possível, resumindo-se a escolher quais as classes que serão processadas bem como os parâmetros do processo. A escolha desta abordagem foi feita com base no objectivo proposto e apresenta uma ferramenta automatizada de inversão do processo MDA.

Para que seja possível automatizar todo o processo de transformação do código em modelos é necessário ter as regras de transformação bem definidas. Estas regras estão correctamente identificadas e descritas sendo elas que permitem re-

alizar o processo de abstracção. Estas regras serão descritas de seguida com uma abordagem *bottom-up*, isto é, começando pelos elementos mais específicos. O mapeamento descrito é directo entre cada elemento Java e o correspondente elemento do metamodelo.

Atributo Um atributo Java é mapeado num JAttribute, que é parte de um JElement. Para cada atributo é analisada a sua informação, como nome e os seus atributos, sendo essa informação preservada no metamodelo.

Método Um método Java é mapeado num JMethod também ele parte de um JElement. É analisado primeiro o seu nome e propriedades, seguindo-se a sua informação de invocação de métodos, que é mantida de forma simbólica e posteriormente concretizada em relações ao nível do metamodelo.

Classe Uma classe Java é mapeada num JClass, elemento este representante de uma classe. Este elemento é uma concretização de um JElement. Para cada classe é extraída a informação exterior como o nome e as propriedades. Posteriormente é analisado cada método e atributo (processo já descrito).

Interface Um interface Java é mapeado num JInterface, elemento representante de um interface. Também este elemento concretiza um JElement. O método de análise de um interface é o mesmo da análise de uma classe, com a diferença de que existem invocações de métodos no caso da classe.

Elemento (Classe ou Interface) Um JElement é a abstracção de um elemento Java que pode ser um interface ou uma classe. Estes elementos contém vários atributos que são partilhados entre si, por isso foi criada esta abstracção que contém os elementos partilhados entre ambos. Um JElement não deverá existir na representação final de um JProject, isto é, todos os elementos são JClass ou JInterface. Este elemento não é explicitamente utilizados nem possui uma correspondência com um elemento concreto da linguagem Java.

Pacote Um pacote (package) Java é representado num JPackage que aglomera todos os JElement de um projecto. Não é considerada uma hierarquia explicita de pacotes, contudo o seu nome é suficiente para permitir deduzir esta hierarquia. Quando uma classe é analisada, é extraída a informação do pacote ao qual pertence. Se um pacote já existir a classe é adicionada a esse pacote. Caso contrário é criado um novo pacote ao qual será adicionada a classe.

Projecto Um projecto Java é representado por um JProject que corresponde a um conjunto de pacotes Java e outros ficheiros. A análise de ficheiros iterativa vai permitir a análise de cada um dos ficheiros e fazer a sua classificação de acordo com a sua funcionalidade: se é um elemento Java (classe ou interface), ou se é um ficheiro externo, mantendo uma referência para o mesmo.

Na Figura 5.2 está representado o metamodelo previamente descrito, sob a forma de diagrama de classes. De uma forma mais fácil podemos perceber quais os elementos que pertencem ao metamodelo e de que forma se organizam.

O modo de análise do código escolhido foi a análise estática. Esta escolha baseia-se na adequação ao problema em questão, conclusão tomada no levantamento do estado da arte. Esta é uma boa abordagem para inferir informação sobre aspectos arquitecturais e estruturais do projecto a analisar, permitindo posteriormente bons resultados no processo de inferência de padrões. O resultado desta análise estática é informação representada na forma de factos Prolog (a base de conhecimento). Estes factos vão conter toda a informação extraída e vão ser adicionados à base de conhecimento para posteriormente serem utilizados na pesquisa de padrões [31]. Mostra-se na figura 5.4 um excerto dos factos gerados na pesquisa de padrões, produzidos pela ferramenta MapIt durante esse processo.

```
%%%Classes
class(colormap).
class(commandmenu).
...
%%%Contains
contains(commandbutton,command).
contains(clipboard,clipboard).
...
%%%Implements
implements(commandmenu,actionlistener).
implements(palettelayout,layoutmanager).
...
%%%Extend
extends(colormap,object).
extends(commandmenu,jmenu).
...
```

Figura 5.4: Factos gerados durante o processo de análise.

Mais à frente será mostrado que a informação levantada no processo de análise estática e modelados em factos Prolog é suficiente para permitir a inferência de padrões.

Durante a análise do código fonte todos os dados são recolhidos e filtrados. O processo de *parsing* é completo e cobre todos os elementos do código fonte e para cada elemento todos os seus constituintes. Durante o processo de *parsing* os dados são filtrados, preservando apenas a informação relevante para a representação do sistema. A análise de todos os elementos de um projecto, seguido de uma filtragem de elementos foi uma das abordagens já analisadas. Este tipo

de abordagem permite que o processo possa ser refinado posteriormente no caso de ser necessário extrair outros dados do código fonte. Esta abordagem em duas fases é mais flexível e permite fazer alterações no processo de análise de forma simplificada e eficiente, apenas onde necessário. Para que este processo possa ser feito em duas fases existe primeiro o *parsing*, seguindo-se de um processamento da informação extraída. Este processamento permite alterar alguma informação, bem como concretizar a informação simbólica recolhida na fase de *parsing*.

Considerando o processo de análise em duas fases resta especificar cada uma delas. Na primeira é analisado o código fonte e extraída toda a informação estática do projecto. Normalmente nesta fase não é possível concretizar as relações entre as classes e informação como invocação de métodos. Nesta primeira fase é concretizada a informação possível e anotada a que não poder ser analisada de forma simbólica (representada por um nome, por exemplo). Assim que termina a primeira fase todos os elementos estarão já representados no metamodelo. Neste momento pode começar a segunda fase onde as informações simbólicas serão concretizadas em relações entre os elementos do metamodelo. Assim no final da segunda fase a representação intermédia concretizada pelo metamodelo estará completa e disponível para ser utilizada.

A questão da cardinalidade é um outro assunto a resolver que também já foi abordado por alguns autores. Esta é uma propriedade especialmente complicada de inferir por existirem muitos tipos de cardinalidade nas relações entre elementos. Na implementação da ferramenta MapIt existem dois tipos de cardinalidade considerados. Primeiro existe a associação, que representa a relação de "um para um" $(1-1)$, unidireccional que se pode ver na Figura 5.5 à esquerda.

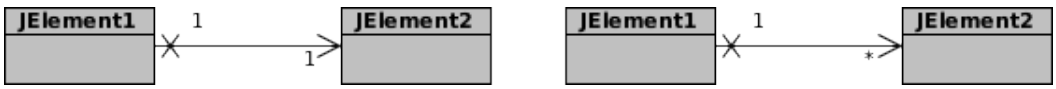

Figura 5.5: Multiplidade $1-1$ (à esquerda) e $1-*$ (à direita).

Neste caso é uma relação simples de um JElement para um outro JElement. O outro tipo de relação existente é de agregação/composição. Este tipo de relação é caracterizado pela cardinalidade "um para vários" $(1-*)$ que pode ser observada na Figura 5.5 à direita. Tanto a agregação como a composição são tratadas da mesma forma por ser indiferente neste caso a sua distinção. A relação "um para vários" $(1-*)$ é uma relação mais abrangente do que "um para um" ou "um para dois", logo cobre todos os casos de relações possíveis. Embora perca alguma especificidade, esta não é relevante para o problema em questão, e será esta a cardinalidade considerada. No metamodelo estão no entanto previstos distintos graus de cardinalidade e facilmente poderá ser estendido, caso se mostre necessário ou útil no futuro. É desta forma que fica definido como será considerada e tratada a questão da cardinalidade das relações num projecto de software para o metamodelo considerado.

Existe uma outra informação que foi adicionada ao metamodelo. Esta informação corresponde à invocação de um método por parte de uma classe, numa outra classe. Esta informação foi adicionada por um lado de modo a permitir uma mais fácil inferência de padrões (a informação de invocação de métodos é frequentemente utilizada), e por outro lado para fornecer mais informação nos diagramas apresentados ao utilizador.

Um aspecto abordado pela ferramenta Fujaba é o grau de certeza na identificação de um padrão. De forma análoga pode ser definido um grau de aproximação a um padrão. Esta funcionalidade pode ser facilmente adaptada a este módulo. Para tal propõe-se a utilização de variáveis Prolog anónimas. A ideia consiste em pesquisar os padrões, alternando cada uma das suas variáveis com uma variável anónima. Para uma regra com N variáveis, de 1 até N, será feita uma invocação dessa regra na base de conhecimento. Em cada iteração (i) dessa invocação, a variável na posição i será substituída por uma variável anónima. Mostra-se o algoritmo proposto, numa pseudo-linguagem similar a Java, na Figura 5.6.

```
String rule = "calls"; //regra prolog
String[] args = new String[]{"A","B","C"}; //variáveis da regra
String[] pArgs;
for(int i=0;i<args.size();i++) { //iterar as variáveis
   pArgs = args.clone();
   pArgs[i] = "_"; //colocar varíavel índice i anónima
   generateFacts(rule,pArgs);//gerar factos para novas variáveis
}
```

Figura 5.6: Algoritmo de iteração de regras, para variáveis anónimas.

Adicionalmente poderiam ser feitas permutações nas variáveis das regras. Desta forma serão apresentadas todas as aproximações a padrões, mostrando que conjunto de classes formam padrões parciais.

Neste momento foram já tomadas as decisões conceptuais que permitem implementar uma das funcionalidades previstas, neste caso a "análise e extracção de informação do código fonte de programas Java". Estas decisões permitem implementar um módulo que extrai a informação de um projecto Java e faz a sua representação com base no metamodelo apresentado. Partindo destas decisões é possível iniciar o processo de desenvolvimento deste módulo.

5.2.2 PSM para PIM

A segunda tarefa considerada para esta ferramenta é a possibilidade de transformação de um modelo específico (PSM) num modelo de mais alto nível (PIM). Esta tarefa consiste numa transformação em que dado um modelo de origem

(neste caso PSM) e por meio de regras bem definidas, obtemos um outro modelo que abstrai o anterior (o PIM). Tal como na inferência do código fonte num modelo PSM, a metamodelação será essencial para perceber a diferença entre estes modelos e que regras serão necessárias. Dessa forma foi definido um metamodelo do PIM que se pode obter partindo da linguagem Java (e respectivo metamodelo) (na Figura 5.2).

É importante ter em conta algumas considerações em relação ao PIM definido. Quando é referido PIM no contexto da ferramenta MapIt não está a ser considerado um PIM exacto, isto é, não estamos perante um modelo completamente independente da plataforma. Apesar do modelo proposto ser uma abstracção de um nível superior ao PSM e remover muitos elementos específicos da linguagem Java não consegue remover todos estes componentes. Devido à natureza do processo de transformação inversa de modelos (abstracção de informação) é de esperar que os modelos estejam algo relacionados com a linguagem da qual foram inferidos. Este PIM acaba por ser um modelo mais abstracto, contudo alguns detalhes da linguagem Java ficam presentes no modelo destino. Apesar de não ser um PIM puro, é um PIM com alguma informação da plataforma destino. A utilidade e propósito deste modelo mantém se o mesmo contudo é necessário clarificar ao que se refere exactamente um PIM no contexto desta ferramenta.

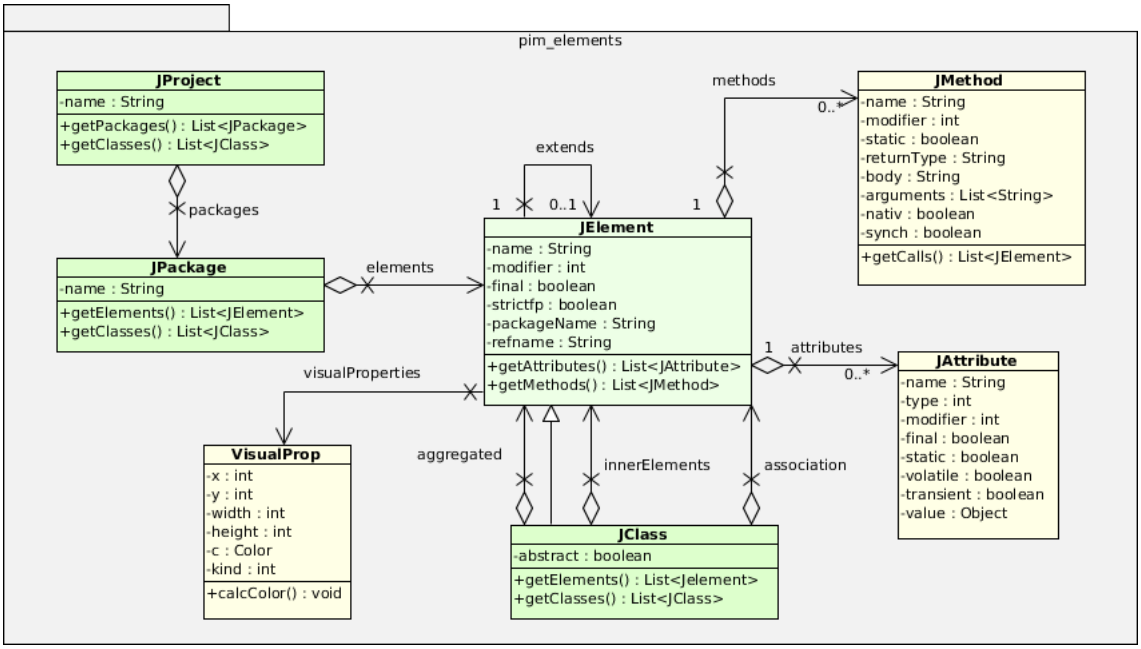

Figura 5.7: Metamodelo do PIM.

Para tornar possível este processo é necessário ter os dois modelos, quer o PIM, quer o PSM bem definidos. Por bem definidos deve-se entender que estão representados numa notação específica (neste caso UML), e que todos

os seus elementos e relações estão bem especificados [49]. Dizer que os modelos estão bem especificados corresponde no fundo a dizer que existe um metamodelo completo que especifica completamente estes dois tipos de modelos. Apenas com estes modelos bem especificados é possível definir as regras de transformação dos modelos. Nas Figuras 5.2 e 5.7 é possível ver a definição do metamodelo PSM e PIM respectivamente. Neste caso em específico, o metamodelo do PIM é baseado no metamodelo do PSM por remoção de elementos específicos da linguagem Java. É desta forma conseguido um modelo que se torna independente da linguagem, apesar de alguns elementos específicos persistirem, contudo importância está no aumentar de abstracção e o processo si. O grau de restrição das regras de transformação vai definir o nível de abstracção do modelo obtido. As regras específicas consideradas serão apresentadas mais à frente.

O processo de transformação de modelos é parte do processo MDA inverso. Este processo consiste na transformação de modelos por meio de regras bem definidas, como descrito em *MDA Explained* [49]. Estas regras definem de que modo um elemento de um dos modelos é mapeado em outro modelo com um diferente nível de abstracção. Esta transformação é concretizada na funcionalidade essencial que dá suporte ao processo MDA: a transformação de modelos e aumento do nível de abstracção (neste caso). Em *MDA Explained* [49] é definido um conjunto de regras de transformação que permitem a partir de um modelo PIM chegar até um modelo PSM. Estas regras especificam um conjunto de transformações a serem aplicadas a cada um dos elementos do PIM original, de modo a aumentar a informação que irá conter esse elemento no modelo PSM. Para a implementação da ferramenta proposta o que foi feito foi um inverter destas regras, obtendo assim um conjunto de regras de transformação que elevam o nível de abstracção. As regras especificadas por transformação inversa não são condição necessária e suficiente para obter um PIM puro, contudo como já foi referido permitem o aumentar de abstracção (que poderá ser refinado a qualquer altura) gerando um modelo de mais alto nível.

O processo de transformação é implementado sob a forma de um processo de remoção de artefactos, ou por outras palavras, filtragem de um modelo. Esta transformação é conseguida processando todos os elementos de um modelo específico. No final conseguimos um modelo PIM que é derivado do PSM original e consistente com ele por estas regras estarem bem definidas. De seguida serão apresentadas as regras específicas de transformação que foram desenvolvidas para este contexto. Elas permitem perceber de que forma irá funcionar concretamente o processo de transformação.

Atributo A um atributo é removida a informação de tipos. Para além disso, será tornado público se for privado e a classe à qual pertence contiver os métodos `get` (ou `is`) e `set` públicos para esse atributo. Os métodos devem se chamar `getNome` ou (`isNome`) e `setNome`, onde `Nome` corresponde ao nome do atributo com a primeira letra capitalizada. A propriedade de visibilidade é um componente específico da linguagem (como de Java,

C++, etc.), e como tal será removida. O encapsulamento é um elemento específico das linguagens orientadas por objectos, que será removido.

Método A um método é removida a informação de tipos. É removida informação dos tipos de retorno e dos tipos dos argumentos. Para além disso alguns métodos em específico são removidos, como `get`, `is` e `set`. Todos os métodos serão públicos. Estas restrições são relativas às mesmas propriedades de visibilidade e encapsulamento, específicas da plataforma, como já foi referido.

Classe Numa classe são apenas processados os seus elementos. Poderá ainda ser feito um ajuste nas suas relações, por exemplo as informações de agregação, composição e associação não serão tratadas de forma diferenciada.

Interface Um interface é um elemento específico de algumas linguagens orientadas a objectos, como tal será removido. Este é um elemento específico e abstracto, apenas devem existir as suas concretizações.

Pacote Um pacote não sofre qualquer alteração, é um elemento de organização da informação, poderia possivelmente mudar de designação.

Projecto A um projecto são removidos todas as referências a ficheiros, e será processado cada um dos seus elementos de acordo com as regras descritas atrás.

Este conjunto de regras definem uma possível transformação para um modelo. Elas poderão ser estendidas, refinadas ou alteradas de acordo com as necessidades. Estas especificação de regras provam que é possível definir transformações para cada elemento de um modelo, que permitem gerar um outro modelo com um nível de abstracção superior.

As transformações definidas são feitas por meio de um interface de transformação. Este interface consiste num conjunto de métodos, que correspondem ao conjunto de regras a aplicar a cada elemento. Para cada um dos elementos a processar, existe um método que vai receber esse elemento e no fim o vai retornar modificado ou (preferencialmente) uma cópia desse elemento processado. Este mesmo interface é bastante dinâmico e pode ser utilizado para fins completamente opostos como por exemplo adicionar informação a um elemento. O interface proposto (sendo na verdade uma classe abstracta que deve ser estendida) é apresentado na Figura 5.8. A implementação deste interface concretiza uma especificação de transformação de um PSM. Todas as regras não especificadas pelo interface, são feitas automaticamente pela ferramenta implementada.

Para que estas transformações de elementos afectem (ou sejam afectadas) por outros elementos é necessário considerar um detalhe no processo de transformação. Estas transformações serão feitas por intermédio de um método (que recebe argumentos e retorna um elemento) e o detalhe a considerar nas transformações consiste em passar como argumento a um método de transformação não

```
public abstract class AbstractTransformation {
 public abstract JFile formatFile(JFile in, JProject jp);
 public abstract JElement formatElement(JElement in, JPackage jp, JProject jpr);
 public abstract JInterface formatInterface(JInterface in, JPackage jp, JProject jpr);
 public abstract JMethod formatMethod(JMethod in, JElement elem, JPackage jp, JProject jpr);
 public abstract JAttribute formatAttribute(JAttribute in, JElement elem, JPackage jp,
JProject jpr);
}
```

Figura 5.8: Interface de transformação de modelos.

só o elemento a processar, mas também todos os elementos que lhe são hierarquicamente superiores. Isto significa que por exemplo ao processar um método, é passado o JElement, o pacote e o projecto aos quais ele pertence. Desta forma é possível ter mais controlo no processo de transformação e definir regras que afectem todo o projecto, como é o caso da regra de transformação `formatMethod`, apresentada na Figura 5.8.

Tendo este interface bem definido e entendendo a forma como as transformações são feitas, é possível perceber então que a alteração das regras de transformação é simples de fazer. O utilizador necessita apenas de programar o método que faz a transformação do elemento, retornando o novo elemento no fim (ou um valor nulo, caso pretenda ignorar o elemento). Assim está definida uma forma padrão para permitir fazer a transformação de elementos implementando assim uma transformação MDA reversa.

Devido à utilização de regras definidas para cada um dos componentes é possível implementar o processo de transformação de modelos como previsto pelo MDA, mas de uma forma inversa. Pela forma como se propõe esta funcionalidade é possível também a qualquer altura alterar este processo de acordo com o do modelo PIM pretendido. Fica assim especificado o conjunto de decisões conceptuais tomadas para resolver um dos problemas previamente propostos, neste caso a "abstracção de diagramas PSM a PIM".

5.2.3 Inferência de padrões num PSM

O último componente a implementar é a inferência de padrões num modelo PSM. Esta análise é feita por via de análise estática estrutural do diagrama de origem (PSM), que por sua vez foi inferido com base no código fonte. O processo que implementa esta funcionalidade inicia com uma análise estática da informação contida no modelo. Durante a mesma são retiradas as informações relevantes por meio de análise estrutural em todos os elementos. De seguida esta informação é representada internamente e é sobre ela que irá ocorrer o processo de inferência de padrões.

Para implementar a funcionalidade de inferência de padrões é proposta a integração da linguagem Prolog no módulo que concretiza esta funcionalidade. O que se propõe é a integração de um motor de inferência Prolog neste projecto que

juntamente com a informação inferida permitirá identificar os padrões presentes num software. A base de conhecimento do Prolog vai ser utilizada para guardar a representação de toda a informação necessária à identificação de padrões que for extraída do projecto. Assim, a conjunção dos factos Prolog com a informação estrutural do PSM permite ter uma outra representação do projecto.

A análise estática estrutural feita sobre o projecto vai produzir factos Prolog que representam as propriedades do software em análise. Estes factos gerados vão criar uma base de conhecimento, essencial para todo o processo de identificação. Existem diversas formas de obter esta base de conhecimento. Uma possibilidade é análise do código fonte e gerar factos durante esse processo. Uma outra possibilidade é a geração de factos com base numa outra representação da informação intermédia (o PSM) [31]. A segunda alternativa mostra-se mais vantajosa por dois principais factores e foi por isso a escolhida. Em primeiro lugar a análise ao código fonte seria refazer trabalho já feito durante inferência do PSM. Se por outro lado fosse aproveitado o processo de *parsing* para gerar os factos resultaria na perda de independência entre estes dois módulos. Em segundo lugar seria necessário voltar a fazer uma análise e filtragem da informação do código fonte (processo já feito para obter o PSM). Será por fim será demonstrado que a utilização da informação de mais alto nível do que código fonte é suficiente para permitir a identificação de padrões num software. Esses factos são resultado das informações representadas no PSM e serão descritos de seguida.

class/1 Este facto representa a informação de que existe uma classe com o dado nome. Para cada classe do software vai existir um facto deste tipo na base de conhecimento. Por exemplo: `class(nome)`.

interface/1 Este é o facto que representa a informação de que existe um interface com um dado nome. Para cada interface do software vai existir um facto deste tipo na base de conhecimento (analogamente a uma classe). Por exemplo: `interface(nome)`.

contains/2 Neste facto é representada a informação de agregação, composição ou associação entre duas classes ou uma classe e um interface. Este facto vai existir quando uma elemento (classe) contém um outro (classe ou interface). Por exemplo, para duas classes A e B, A contendo B: `contains(a,b)`.

extends/2 A representação da relação de herança é feita com este facto. Quando um elemento estende um outro (é subclasse), é gerado este facto para estas duas classes. Se tivermos as classes A e B em que A estende B, temos por exemplo: `extends(a,b)`.

implements/2 A informação sobre a implementação de um interface é representável com este facto. Para uma relação de implementação, em que tenhamos por exemplo uma classe A que implementa um interface B temos o facto: `implements(a,b)`.

calls/3 A informação da invocação de métodos de uma classe em outra também é representável. Para tal existe este facto, que representa a invocação de métodos de um elemento em outro, bem como o nome do método. Por exemplo para a classe A que invoca o método m na classe ou interface B temos: `calls(a,b,m)`.

Estes são os factos necessários para poder identificar os padrões de concepção considerados neste estudo, o que será demonstrado de seguida. O componente responsável pela geração de factos pode ser facilmente estendido e refinado para gerar factos ajustados às necessidades. Existe uma descrição de uma abordagem similar à apresentada, contudo não para Prolog mas sim para uma linguagem semelhante denominada SOUL [87]. A escolha da tecnologia Prolog e não uma outra linguagem específica de padrões (como SOUL, por exemplo) é uma mais valia uma vez que permite ter uma maior flexibilidade que é fornecida pela ferramenta, permitindo integrar conceitos de inteligência artificial na pesquisa, por exemplo. Outro motivo para a escolha de Prolog é o facto desta ser uma linguagem madura e bastante utilizada, sobretudo nas áreas de inteligência artificial. Isto torna esta linguagem numa boa escolha.

Uma vez preenchida a base de conhecimento com os factos atrás descritos é possível questionar a base de conhecimento para pesquisa de padrões. Para tal o processo começa com a interpretação do catálogo de padrões, ficando disponível a sua representação como regras Prolog. A ferramenta MapIt irá então solicitar ao Prolog que invoque a base de conhecimento, procurando conjuntos de factos que satisfaçam as restrições impostas por um determinado padrão. O problema é no fundo reduzido à satisfação de um conjunto de propriedades para uma dada regra, numa dada base de conhecimento. É possível ver que a definição de um novo padrão é um processo tão simples como a definição de uma nova regra Prolog, que será adicionada ao catálogo. Do ponto de vista da implementação, o módulo que permite a pesquisa de padrões disponibilizará um método de pesquisa de padrões, mostrado na Figura 5.9. Por sua vez este módulo irá interagir com o módulo que gera o PSM e com o módulo que interage com o Prolog.

A especificidade das regras contidas no catálogo (mais restritivas ou mais abrangentes) vai determinar a qualidade dos resultados obtidos. Definindo regras mais rigorosas, a quantidade de padrões encontrados será menor, contudo terão tendência a ser mais precisos e os resultados mais correctos. Se por outro lado as regras forem mais abrangentes, irá resultar num maior número de padrões (que serão apresentados ao utilizador, sendo distinguidos com um número identificador). Como estes padrões são encontrados em maior número a sua qualidade será naturalmente inferior, podendo até ser encontradas aproximações de padrões. Com um catálogo mais abrangente pode ainda acontecer serem encontrados falsos positivos, isto é, serem identificados conjuntos de elementos que não correspondem a nenhum padrão.

Depois de apresentado o modo de funcionamento do módulo de inferência de padrões baseado na linguagem Prolog e num motor de inferência, resta ape-

```
public static ArrayList<Pair<String,ArrayList<ArrayList<String>>>>
 findPatterns(ArrayList<String> files, String catalogPath,String pname) throws Exception {
  SystemBuilder sb = new SystemBuilder(); //Interacção com módulo de parsing
  sb.buildSystem(files, pname); //Instanciação de metamodelo
  QueryEngine qe = new QueryEngine(); //Interacção com módulo prolog
  PatternCatalog pc = new PatternCatalog(); //Instanciação de catálogo
  pc.load(new File(catalogPath)); //carregar catálogo
  FactLoader fl = new FactLoader(sb.getProject(),qe);
  fl.loadFacts(pc.isUsingClasses(), pc.isUsingInterfaces(), pc.isUsingSubclasses(),
    pc.isUsingCalls(), pc.isUsingContains(), pc.isUsingImplement());
  fl.loadPatterCatalog(pc);
  DataQuery dq = new DataQuery(qe); //Questionar módulo prolog
  ArrayList<Pair<String,ArrayList<ArrayList<String>>>> result = dq.findPatterns(pc);
  qe.stop();
  return result;
}
```

Figura 5.9: Um dos métodos de análise de padrões.

nas definir o já referido catálogo de padrões. Para que este módulo seja útil é necessário que o utilizador possa definir os seus próprios padrões. Como já foi referido, novos padrões e regras surgem constantemente e por vezes necessitam de ser adaptados a contextos específicos. Para permitir a parametrização dos padrões foi definida uma forma importar catálogos definidos pelo utilizador. Estes catálogos consistem no fundo em um conjunto de regras Prolog que serão interpretadas pela ferramenta e posteriormente enviadas ao motor de inferência. Estes catálogos serão definidos num ficheiro externo (permitindo que o utilizador tenha por exemplo uma biblioteca de catálogos), num formato aproximado ao Prolog (com alguma informação extra). Quando o utilizador o pretender utilizar apenas terá de indicar a sua localização.

A integração do Prolog na implementação da ferramenta em Java fornece a conjugação de duas linguagens de grandes potencialidades. Esta integração permite obter todas as funcionalidades de inferência do Prolog dentro da ferramenta proposta. Fornece uma fácil extensibilidade e refinamento da funcionalidade de inferência de padrões, tirando partido do Prolog para este processo. A integração da linguagem na ferramenta permite também que seja facilmente trocada informação entre o Prolog e o módulo Java que o incorpora. Isto permite por exemplo aplicar técnicas de inteligência artificial para permitir a análise dinâmica ou a pesquisa de outros tipos de informação no código fonte [31].

5.2.4 Interface gráfica

Um componente comum a todas as funcionalidades é a representação visual da informação. Existem várias alternativas para a implementação deste componente como proposto por diferentes autores. Estas alternativas podem ser divididas em três opções principais. A primeira alternativa consiste na implementação de um módulo de representação independente responsável por mostrar ao utilizador a informação de forma visual. Esta é a alternativa mais trabalhosa

e básica, que possivelmente produz os resultados menos eficientes uma vez que o utilizador desenvolve completamente esse módulo. A segunda alternativa consiste na utilização de uma ferramenta externa para proceder à representação da informação. Esta é uma boa alternativa pois permite aproveitar trabalho feito previamente. Os resultados obtidos por esta alternativa serão mais vantajosos se o modulo de representação for independente dos restantes módulos por permitir reaproveitar trabalho já feito. No caso de integrar a ferramenta MapIt num IDE a melhor solução será implementar este módulo representação dentro do próprio IDE, com os meios que este fornece. Esta abordagem consiste na terceira alternativa, que surge como um meio termo entre a primeira e a segunda alternativas. Propõe-se então implementar esta funcionalidade comum a todos os módulos (por representar os diagramas PSM e PIM) de forma integrada no IDE destino.

Este componente é utilizado por todos os outros módulos no momento da representação de informação. Para tal necessita apenas de receber uma concretização de um metamodelo. Para cada elemento do metamodelo tem de haver uma "propriedade visual" que consiste num conjunto de propriedades relativas a esse elemento. Essas são as propriedades utilizadas pelo módulo responsável pela representação visual. A propriedade visual é concretizada numa classe que contém as essas propriedades. Esta é apresentada na Figura 5.10.

```
public class VisualProp {
  private int x;
  private int y;
  private int kind;
  private int width;
  private int height;
  private Color c;

  public VisualProp() { ...

  public VisualProp(int x, int y, int kind, int width, int height, Color c) { ...

  //Existem ainda os respectivos 'getters' e 'setters'
```

Figura 5.10: Excerto da classe das propriedades visuais.

As representações visuais consistem em diagramas UML, que permitem também obter informação detalhada para cada elemento do diagrama. Para cada um dos modelos descreve-se de seguida a forma como utilizam este componente de representação visual.

PSM Esta é a representação da informação de um software no formato UML com possíveis informações complementares. É feito um mapeamento dos elementos Java para a linguagem UML (processo conhecido e documentado). Assim que o componente de inferência de PSM termina a sua análise, envia o pedido para este módulo que trata de fazer a representação da informação obtida.

PIM Este caso é em tudo similar ao anterior do PSM. O módulo de abstracção de PSM em PIM trata do processo de transformação. Assim que termina invoca o módulo de representação. Também os diagramas seguem uma notação UML.

Padrões A representação dos padrões de concepção é feita de forma semelhante aos outros modelos. Contudo nesta representação existe uma variação. A abordagem escolhida passa por representar todas as classes envolvidas no processo, e quando seleccionado um padrão, as classes que compõem esse padrão são realçadas. Adicionalmente poderá ser solicitado um diagrama simplificado onde é removida toda a informação das classes, ficando estas representadas apenas pelo seu nome.

Apenas quando este módulo estiver implementado, está concluído o processo de implementação dos restantes. Os módulos de inferência de PIM, PSM e padrões são dependentes destes para permitir a obtenção de resultados práticos. Terminada a descrição das decisões subjacentes a este módulo, está concluída a especificação das decisões por trás de cada um dos módulos propostos.

5.2.5 Interligação dos diferentes módulos

Depois de descritas as principais decisões sobre cada um dos módulos falta apenas especificar de que modo estes componentes interagem entre si. É necessário definir uma forma padrão de comunicação para permitir por um lado uma comunicação robusta entre componentes e por outro lado permitir uma forma fácil de alteração de componentes. O primeiro ponto é necessário para obter um projecto de boa qualidade e que garanta que no final todos os componentes serão capazes de interagir e cooperar correctamente. O segundo ponto é importante para que permitir e extensibilidade ou refinação da ferramenta em questão, por alteração de componentes específicos da ferramenta MapIt.

Para que os componentes interajam facilmente é definido um conjunto de métodos e classes importantes que devem estar acessíveis externamente. Estas classes (contendo métodos) abstraem a informação e funcionalidades fornecidas pelas classes inferiores. Adicionalmente os módulos serão compilados em bibliotecas a ser importadas por outros módulos. Quando uma funcionalidade for necessária, basta importar a biblioteca respectiva e aceder às classes públicas que a implementam.

A fácil alteração de componentes é baseada na definição de classes que servem de interface a um módulo. Desta forma quando um módulo tiver necessidade de ser alterado ou melhorado apenas tem de manter esse interface de comunicação. Esta forma de implementar a independência entre módulos é a concretização do padrão de concepção facade. É desta forma que o programa fica implementado de forma modular e bem estruturada. Permite a alteração, melhoramento

ou substituição de componentes individualmente, isto é, um componente pode ser completamente alterado sem qualquer consequência nos restantes.

Desta forma ficam especificadas as decisões tomadas para permitir a implementação da interligação entre os diferentes módulos. Depois de especificadas todas as decisões conceptuais para cada um dos módulos, é possível iniciar a especificação dos detalhes de implementação dos respectivos módulos.

5.3 Proposta de implementação

Na secção anterior foram apresentadas as decisões conceptuais que solucionam os problemas previstos. Estas decisões foram tomadas para os problemas apresentados no capítulo do estado da arte e do problema. Nesta secção serão apresentadas as concretizações das decisões tomadas na secção anterior. Serão apresentadas propostas concretas para cada um dos problemas identificados de modo a solucionar os problemas já apresentados, mantendo-se tão próximo das decisões conceptuais tomadas quanto possível. Este processo será essencial para guiar a fase de implementação.

5.3.1 *JavaParser Simple Interface* (JPSI)

O *JavaParser Simple Interface* (JPSI) será o módulo responsável pelo *parsing* e representação da informação extraída sobre um projecto. Contemplará dois componentes principais que são o *parser* e o metamodelo de Java.

O processo de análise de informação e mapeamento para um metamodelo será feito em duas fases. Primeiro a informação é extraída tal como representada. Nesta primeira fase não é possível concretizar por exemplo as relações de agregação e composição. Isto acontece porque um elemento pode necessitar de informação de um outro que ainda não foi analisado. Para tal é utilizado um campo que guarda o nome do elemento destino. A segunda fase que começa assim que a primeira termina quando todos os elementos já foram analisados. Nesta fase é quando os nomes são substituídos pela referência adequada que eles representavam. No final a representação está concluída, onde todos os elementos e relações entre si foram mapeados no metamodelo.

O JavaParser é uma ferramenta de código aberta disponibilizada sob LGPL[1]. Esta ferramenta permite analisar um ficheiro Java e extrair toda a informação relevante desde métodos, atributos, código fonte entre outras informações. Esta ferramenta será incorporada numa funcionalidade que permite percorrer uma árvore de ficheiros, indicando qual a raiz do projecto. O JavaParser disponibiliza um método para visitar os elementos de uma classe ou interface que lhe seja passada como parâmetro, o que é suficiente para o tipo de análise pretendido. Mostra-se um excerto do código que utiliza esta funcionalidade disponibilizada

[1]Disponível em `http://code.google.com/p/javaparser/`

por esta ferramenta na Figura 5.11. Combinado o JavaParser com a funcionalidade de travessia descrita é possível analisar todos os elementos da hierarquia de um projecto. Resumidamente, para cada nodo da árvore será analisado cada um dos ficheiros Java presentes. Com o método disponibilizado pelo JavaParser será extraída a informação contida. É extraída informação sobre métodos (nomes, declaração, propriedades), atributos (nomes, propriedades), tipo de elemento (classe ou interface) e ainda informação de invocação de métodos.

```
private class MethodVisitor extends VoidVisitorAdapter {
  //Analisa um método - é invocado várias vezes
  @Override
  public void visit(MethodDeclaration n, Object arg) { ...
  //Analisa um atributo - é invocado várias vezes
  @Override
  public void visit(FieldDeclaration n, Object arg) { ...
  //Analisa o elemento - é invocado uma só vez
  @Override
  public void visit(ClassOrInterfaceDeclaration n, Object arg) { ...
  //Analisa uma invocação - é invocado várias vezes
  @Override
  public void visit(MethodCallExpr n, Object arg) { ...
```

Figura 5.11: Excerto da classe de análise da informação de classes.

A informação sobre agregação, composição e invocação de métodos enquanto analisada será representada de forma temporária pelo nome do elemento contido. Esta informação pode apenas ser concretizada quando estiver disponível toda a informação estrutural do projecto, isto é, quando a primeira fase de análise estiver concluída. Para tal existe uma classe responsável por fazer o tratamento dessa informação após a extracção inicial de informação, que concretiza os valores temporários prévios. É desta forma que fica concluída a análise e extracção da informação sobre o projecto em causa.

Para representar a informação extraída de um projecto é necessário criar um metamodelo da linguagem Java. Este é segundo componente do módulo JPSI e permite representar a informação extraída pelo JavaParser. O metamodelo necessário terá de ser desenvolvido de raiz com base na especificação da linguagem Java. Também serão considerado outros metamodelos existentes como foi o caso do proposto em Memoj [82]. O metamodelo proposto necessita de ser desenvolvido de raiz devido às diferentes funcionalidades que pretende suportar, que não são suportadas por outros metamodelo. Na Figura 5.2 está a representação do metamodelo proposto em UML. No metamodelo estão representados:

JProject Projecto composto por elementos Java e outros ficheiros relevantes.

JFile Ficheiro não directamente relevante para o projecto, podendo ser um ficheiro de configuração, *Extensible Markup Language* (XML), entre outros, que frequentemente são encontrados em projectos.

JPackage *Package* Java que permite organizar as classes e interfaces em hierarquias.

JElement Elemento abstracto de Java, que pode ser um interface ou uma classe. Foi considerado este elemento pela quantidade de informação semelhante partilhada entre as suas duas concretizações.

JClass Representação de uma classe Java, incluindo invocações, associações e agregações.

JInterface Representação de um interface Java.

JMethod Representação de um método Java, bem como todas as suas propriedades.

JAttribute Representação de um atributo Java e suas propriedades, com possibilidade de incluir também o seu valor.

VisualProperties Não directamente relacionado com Java, representa as informações visuais associadas a um JElement.

Pair Útil para representar pares de valores, como no caso das invocações, agregações, etc.

A informação extraída pelo *parser* será mapeada no metamodelo apresentado. A metamodelação é a técnica mais adequada para representação de informação e posterior inferência de padrões [31]. Assim, depois de analisado um projecto temos a sua representação que concretiza o metamodelo, pronta a ser utilizada por outros módulos. Todo o processo de análise, extracção e representação de informação será abstraído com um *facade*, e disponibilizado sob a forma de biblioteca jar para ser utilizado por outros projectos. Ficará assim disponível um interface de análise de projectos que fornece as funcionalidades básicas para os objectivos pretendidos.

5.3.2 *GNUProlog Java Simple Interface* (GPJaSI)

O *GNUProlog Java Simple Interface* (GPJaSI) é um módulo de abstracção da interacção com o programa GNUProlog [2]. O GNUProlog é um compilador e interpretador de Prolog usado como motor de inferência. O módulo permitirá a interacção total com o programa permitindo a asserção e remoção de factos, envio de questões ao motor de inferência e pesquisa de resultados na base de conhecimentos.

Este módulo será implementado com base em três *threads* principais que interagem entre si. Uma primeira *thread* contém o processo GProlog, outra thread de leitura de resultados produzidos, ainda outra thread de escrita para

[2]Disponível em `http://www.gprolog.org/`

o *input* do processo. Este módulo funciona como um interface para o processo que contém o GNUProlog em execução. Permite receber pedidos de uma forma específica que processa e envia para o processo.

Os resultados obtidos serão analisados, processados e representados internamente. Para tal foram definidos novos tipos de representação de informação Prolog, nomeadamente PIBooleanResult para resultados booleanos, PIDataResult para resultados de dados, PIExceptionResult para excepções e PIResult para todos os outros tipos de resultados.

Por fim este módulo permitirá controlar o ciclo de vida do processo. Permite iniciá-lo, terminá-lo e enviar-lhe questões Prolog directamente. O diagrama de classes deste módulo é o representado na Figura 5.12.

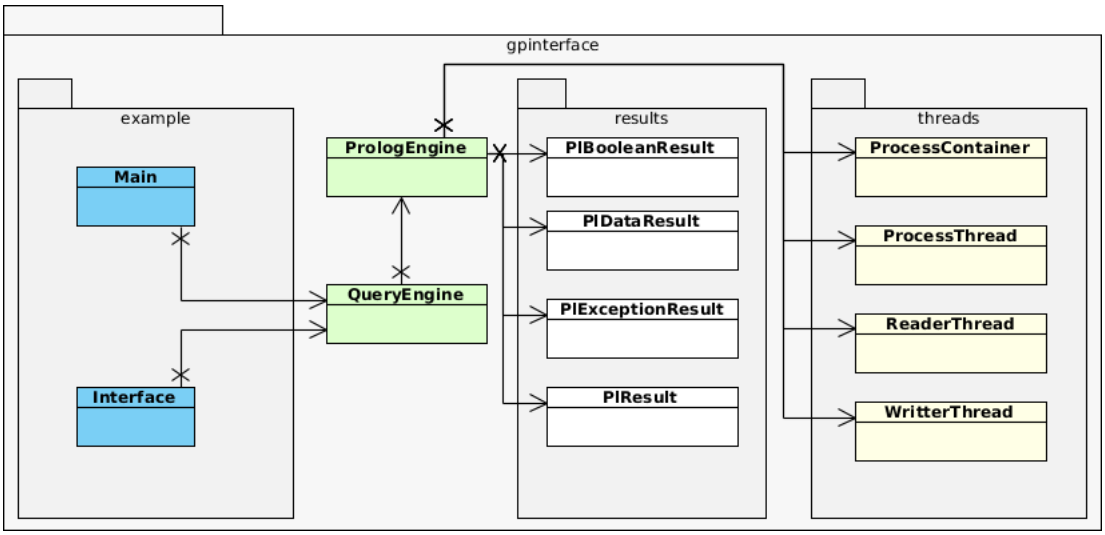

Figura 5.12: Visão geral do GPJaSI.

Mais uma vez é utilizado o padrão *facade*, pela disponibilização deste módulo como uma biblioteca que pode ser importada por outros projectos, abstraindo assim o processo de comunicação com o motor de inferência. Para a utilização desta biblioteca será necessário ter instalada a ferramenta GProlog.

5.3.3 *System Analysis Interface* (SAI)

O módulo *System Analysis Interface* (SAI) abstrairá e simplificará a utilização do GPJaSI e o do JPSI, bem como a interacção entre estes componentes como se pode ver na Figura 5.13. Este será o módulo que permite a interrogação da base de conhecimento, que carrega o catálogo de padrões e ainda cria os dados representativos para enviar para o motor de Prolog. Basear-se-à por um lado no JPSI para criar a representação, fazer travessias e processar a informação representada no metamodelo para gerar factos Prolog. Por outro lado utilizará o GPJaSI para criar a base de conhecimento fazer operações sobre ela.

Existirá um componente deste módulo responsável por criar os factos Prolog representantes dos elementos do projecto em análise. Neste mapeamento são considerados os seguintes elementos, que poderão ser utilizados para criar catálogos de padrões:

class/1 Representa uma classe Java, tem como argumento o nome da classe.
Exemplo: `classe(image).`

interface/1 Representa um interface Java, e como argumento utiliza o nome do interface.
Exemplo: `interface(slider).`

extends/2 Representa a relação de herança. Recebe como argumento o nome da classe "filho" e classe "pai" respectivamente.
Exemplo: `extends(triangle,image).`

calls/3 Representa a invocação de um método. Tem como argumentos o nome da classe de origem, de destino e o nome do método, respectivamente.
Exemplo: `calls(slider,triangle,draw).`

contains/2 Representa uma classe que contém uma outra por associação, agregação ou composição. Os argumentos são a classe mais abrangente e a classe contida, respectivamente.
Exemplo: `contains(triangle,line).`

implements/2 Representa a relação de implementação de um interface. Como argumentos recebe a classe e o interface, respectivamente.
Exemplo: `implements(image,slider).`

O catálogo de padrões será tratado por este módulo. Existirá um componente que é responsável por carregar, interpretar e utilizar o mesmo. Um catálogo de padrões é um ficheiro de texto com regras, num formato semelhante a Prolog mas com algumas restrições de formatação. Este formato foi desenvolvido de raiz utilizando as regras Prolog como guia. O formato o é apresentado em 5.14.

O `nome_padrão` representa o nome do padrão em questão e tem de ser o mesmo da regra em Prolog representada em `regra_prolog`. Esta será uma regra Prolog comum. O campo `n_argumentos` representa a aridade da regra Prolog. Estes padrões devem estar num ficheiro com uma estrutura específica. A estrutura prevê que seja definido um padrão por linha. O formato permite comentários, em que a linha comentada deverá começar por '`%`'. Para além disso a primeira linha não comentada deve começar por '`!`' e conter um conjunto de caracteres representantes aos tipo de factos a gerar. Os caracteres que poderão estar nessa linha irão identificar quais os factos que serão necessários. Os caracteres disponíveis para tal são '`c`' para classes, '`i`' para interfaces, '`C`' para invocações, '`k`' para conter (agregação, composição ou associação), '`e`' para

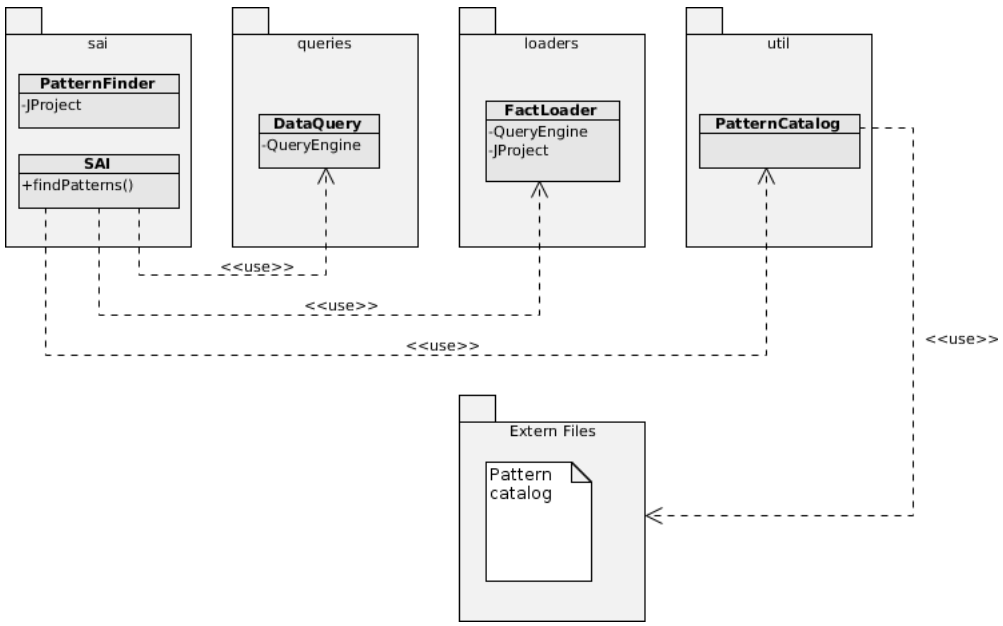

Figura 5.13: Visão geral do SAI.

```
%comentário
!CceiIk
nome_padrão/n_argumentos#(regra_prolog)
```

Figura 5.14: Formato de definição de padrões.

herança (ser subclasse) e 'I' para implementação. Utilizando factos Prolog e seguindo as restrições impostas, é possível criar catálogos de padrões. O catálogo será carregado por este módulo e utilizando para detectar padrões.

Para melhor perceber de que modo é possível implementar um catálogo de padrões, apresenta-se de seguida um exemplo de um catálogo. Este catálogo, representado na Figura 5.15 é o que foi utilizado em diversos testes, e contém a regra necessária para identificar o padrão Composite. Na regra deste padrão começa-se por especificar a informação relativa a classes e suas relações. De seguida é apresentada a informação que evita que os elementos unifiquem com eles próprios.

A interacção com a base de conhecimento é conseguida com este módulo. Em primeiro lugar ele criará a base de conhecimento com os factos necessários. De seguida criará as questões para enviar à base de conhecimento (com o `findall`), baseando-se no catálogo. Implementadas estas funcionalidades de geração de factos e questões, é disponibilizada uma forma de aceder às mesmas por meio de um interface. O processo de geração de dados bem como pesquisa e consulta de padrões fica assim abstraído e disponibilizado a outros módulos.

```
% c- classe, i-interface, C-calls, k- contains, e- extends, I- implements
!cikeI
composite/4#(composite(Component,AbstractComponent,Composite,Leaf) :-
(((class(Component),extends(Composite,Component),extends(Leaf,Component));

(interface(Component),class(AbstractComponent),
implements(AbstractComponent,Component), extends(Leaf,AbstractComponent),
extends(Composite,AbstractComponent))), class(Composite),class(Leaf),
contains(Composite,Component),

Component \== AbstractComponent, Component \== Composite, Component \==
Leaf, AbstractComponent \== Composite, Composite \== Leaf,
AbstractComponent \== Leaf))
```

Figura 5.15: Catálogo de padrões utilizado.

5.3.4 *Plugin* NetBeans

O *plugin* NetBeans será o interface com o utilizador. Comunica com o módulo inferior SAI (que por sua vez comunica com os módulos inferiores), e utiliza o metamodelo de JPSI. O *plugin* é assim o mais alto nível de interacção, que utiliza os outros módulos por inclusão das suas bibliotecas (jar) que são por si só interfaces de funcionalidades mais específicas.

Este *plugin* é constituído por vários componentes que vão ser descritos de seguida. Estes componentes correspondem a NetBeans components. A interacção entre estes componentes permite a obtenção das funcionalidades propostas.

Componente *PreprocessTopComponent*

Este será o componente visual mostrado ao utilizador assim que o processo inicia. Corresponde a um ecrã específico dentro do IDE. Neste ecrã vai ser apresentada a lista de ficheiros a serem analisados juntamente com um conjunto de opções de parametrização. É neste componente que é definido o catálogo de padrões bem como o nome do projecto. Quando iniciado o processo, este módulo pré-processa a lista de ficheiros (caso necessário) e invoca os módulos respectivos, passando como argumento os dados e parâmetros previamente estabelecidos.

A forma como os ficheiros são seleccionados para análise é descrito em *contextAction*. Os componentes invocados por este são: *DiagramTopComponent* responsável por representar diagramas de classes, *PimTopComponent* responsável por representar diagramas PIM, *PatternsTopComponent* responsável por representar padrões e o *PatternsToolboxTopComponent* que é um interface para listagem de padrões. Este é portanto o componente de interacção com o utilizador, o mais alto nível da ferramenta MapIt. Disponibiliza de forma adequada todas as funcionalidades implementadas e representações visuais, fornecidas pelos módulos anteriormente apresentados.

Componente *DiagramTopComponent*

Este componente corresponde a um outro ecrã no IDE, que apresentará o diagrama de classes. A representação consiste num diagrama UML, com as classes, interfaces, relações, atributos e métodos do projecto em análise. Este componente permite também obter informação adicional sobre um dado elemento.

Para obter um diagrama é utilizada a representação intermédia de um projecto. Para cada elemento são analisadas as suas propriedades visuais, o nome, métodos e atributos e é feita a sua representação gráfica. Por fim são feitas as ligações que representam associação, agregação, composição, implementação e herança de acordo com as regras UML.

É desta forma concluída mais uma das tarefas propostas, neste caso da inferência e representação de um diagrama de classes para um dado projecto.

Componente *ContextAction*

O componente *ContextAction* é representado por uma nova acção que poderá ser executada no menu de contexto. Quando seleccionado um ficheiro Java e invocando o menu de contexto, ficará disponível esta nova acção. Corresponde à possibilidade de adicionar (um ou mais) ficheiros do tipo Java para que sejam integrados no processo de inferência. Assim que esta acção é concluída é mostrado o *PreprocessTopComponent*. A partir daí é possível continuar a adicionar ou remover ficheiros e por fim dar inicio ao processo de análise.

Este menu de contexto é útil uma vez que se integra e estende as funcionalidades fornecidas pelo IDE, demonstrando que é possível obter uma integração útil e prática desta ferramenta. Outras funcionalidades poderiam facilmente ser adicionadas ao menu de contexto caso necessário, da mesma forma que esta foi adicionada.

Componente *ProjectAction*

Existe uma outra forma de seleccionar os ficheiros para análise, que é por meio de um botão de acção. Este botão permitirá adicionar todos os ficheiros Java presentes no projecto com uma só acção. Para além de adicionar os ficheiros, abre o *PreprocessTopDiagram*. Desta forma é possível analisar mais facilmente todo um projecto com menos esforço. Na prática o comportamento é equivalente a seleccionar os ficheiros individualmente, é apenas uma medida de conveniência.

Componente *PatternsToolboxTopComponent*

Quando invocada a opção de analisar padrões, este componente será invocado com os respectivos argumentos (lista de elementos) a analisar.

Quando iniciado, este componente irá utilizar o JPSI para obter a concretização do metamodelo (que poderá ser parte de um projecto) onde serão

pesquisados os padrões. De seguida utilizará o SAI: primeiro para carregar o catálogo especificado, depois para inferir os padrões presentes nesse metamodelo. Por fim apresentará uma lista com os padrões encontrados nesse metamodelo, que o utilizador poderá consultar.

Quando um padrão é seleccionado da lista de padrões encontrados, é invocado o *PatternsTopComponent*, componente responsável pela sua representação.

Esta é outra das tarefas propostas que fica desta forma concretizada: a inferência de padrões num projecto Java. Este componente trata não só de analisar um projecto mas também de invocar a sua representação visual.

Componente *PatternsTopComponent*

Depois de analisados os padrões e apresentada a lista ao utilizador em *PatternsToolboxComponent*, é necessário uma forma de os representar. O componente responsável por isso é o *PatternsTopComponent*.

Este componente apresentará um diagrama de classe semelhante ao apresentado em *DiagramTopComponent*. Contudo, neste diagrama estão realçados os elementos que fazem parte de um padrão (o escolhido pelo utilizador). Desta forma o utilizador consegue ver nas classes seleccionadas onde se encontra o padrão encontrado.

Alternativamente, caso o utilizador solicite, esta representação poderá ser simplificada. Neste caso será mostrada uma simplificação, onde cada elemento aparece representado apenas pelo seu nome (mantendo-se as relações).

Este componente complementa o *PatternsToolboxTopComponent* e a funcionalidade de inferência, sendo responsável pela representação visual.

Componente *Util*

A abstracção de um diagrama de classes PSM para PIM é outro dos objectivos propostos. Este processo de abstracção consiste na redução da informação e número de componentes do diagrama de classes (ou PSM). No final será criado um novo modelo que é o PIM respectivo do PSM analisado. Esta funcionalidade será incluída num componente chamado Util.

Tendo em conta que a abordagem escolhida foi a completa automatização, vai ser descrito o processo para tal. Assim, foi definido um componente chamado *Model Filter Engine* (MFE) que automatizará as transformações. Este componente fará a transformação de cada um dos elementos em um elemento do mesmo tipo, porém processado, ou um valor nulo no caso deste ser eliminado. Para que as transformações se possam reflectir em todo o projecto, para cada método que processa um elemento, será passada a hierarquia de elementos superiores. Assim o processamento de elementos poderá causar alterações transversais a todo o projecto. Este novo modelo é definido de acordo com um novo metamodelo, que é uma simplificação do metamodelo de Java (tal como o PIM é

uma simplificação do PSM). Este metamodelo está representado na Figura 5.7.

Assim foi definido um conjunto de regras facilmente extensíveis, que definem de que forma um elemento será representado quando abstraído para PIM. Este componente permitirá assim a geração de uma nova representação do sistema sob a forma de um PIM. Este módulo e o *PimTopComponent* completam outro dos objectivos propostos: a abstracção de modelos PSM para PIM. Um exemplo de regras de transformação, que futuramente poderão ser facilmente estendidas são concretizadas na Tabela 5.2, partindo das regras da Tabela 5.1 (baseado nas regras descritas em *MDA Explained* [49]).

Elemento PIM	Elemento PSM
Classe PIM	Classe Java
Associação PIM	Associação Java
Classe de associação PIM	Duas classes Java com associação bidireccional
Atributo privado PIM	Atributo privado Java
Operação PIM	Método Java
Atributo público PIM	Atributo privado Java, mais métodos `get` e `set` públicos
Nomes PIM	Transformados em nomes Java

Tabela 5.1: Resumo das regras de transformação PIM em PSM, com base em *MDA Explained* [49].

Elemento PSM	Elemento PIM
Classe Java	Classe PIM
Associação Java	Associação PIM
Atributo público Java	Atributo público PIM
Método Java	Operação PIM
Atributo público Java	Atributo público PIM
Atributo privado Java, mais métodos `get` e `set` públicos	Atributo público PIM
Nomes Java	Transformados em nomes PIM
Interface Java	Não existente no PIM

Tabela 5.2: Resumo das regras de transformação PSM em PIM, com base na Tabela 5.1.

Como podemos ver a transformação corresponde a um processo de abstracção, com as regras já apresentadas de forma resumida. De seguida serão analisadas em maior detalhe as regras de transformação para cada elemento do metamodelo do PSM. Estas regras consistem num exemplo adequado aos objectivos propostos, o que significa que poderão ser alteradas futuramente. Este módulo será implementado de forma a poder ser facilmente alterado, permitindo que regras diferentes possam ser definidas. A definição de regras mais restritas

e complexas irá fazer com que o PIM obtido de aproxime mais de um modelo completamente independente do contexto. De seguida é descrito um conjunto de regras que permitem obter um PIM.

JFile Não existe no PIM, é retornado um valor nulo na sua transformação.

JClass É transformado em classe PIM.

JElement Se é JClass é transformado em classe PIM. Se é JInterfece é inexistente, retornando valor nulo.

JInterface Não é considerado no PIM, o valor nulo é retornado na sua transformação.

JMethod Se é um `getter` ou `setter` (*get*, *set* ou *is*) é removido.

JAttribute Se é um atributo privado e possui um `getter` e um `setter`, é tornado um atributo PIM público. Se é público é transformado num atributo PIM (no PIM todos os atributos são públicos).

Está desta forma apresentado o componente que é responsável pela abstracção de modelos PSM. Foram definidas as regras de transformação e mostrado de que forma poderão ser alteradas. Fica desta forma apresentado e especificado o módulo que soluciona o problema de abstracção de modelos PSM em PIM.

Componente *PimTopComponent*

Para que o PIM abstraído tenha uma utilidade existe este componente. Ele é o responsável pela representação visual de um PIM. A forma de funcionamento deste componente é semelhante à do *DiagramTopComponent*, porém recebe como argumento o PIM em vez do PSM. O diagrama gerado é também ele representado em notação UML e naturalmente semelhante ao gerado pelo *DiagramTopComponent*. A semelhança entre as representações de diversos componentes reside no facto de o elemento que faz o processamento visual ser comum a estes componentes, seguindo uma abordagem similar para os diferentes modelos.

A união deste componente com o MFE (do componente *Util*) mostra como é possível abstrair um modelo PSM a um PIM, e respectiva representação visual. Foi também provado que a integração destes componentes numa só ferramenta e disponibilização como *plugin* é um resultado possível de obter.

Componente *PropertiesTopComponent*

Uma funcionalidade extra adicionada ao projecto foi a possibilidade de consultar informação mais completa de um elemento Java com base no diagrama onde ele se encontra (seja ele PIM, PSM ou de padrões). Esta funcionalidade será concretizada pelo *PropertiesTopComponent*. Consiste num módulo de analise e

representação de informação, de forma útil ao utilizador. Para tal utilizará o metamodelo para mostrar toda a informação disponível sobre esse elemento. Esta funcionalidade é indispensável para que os diagramas sejam realmente úteis ao utilizador final, permitindo ver toda a informação não representada visualmente nos diagramas.

5.4 Resumo

Este é o capítulo que trata de apresentar soluções para os problemas propostos no capítulo anterior. Depois de apresentados os detalhes específicos de cada problema a solucionar, é necessário apresentar soluções para esses problemas. Primeiro é necessário tomar as decisões teóricas com base no estado da arte, sendo que estas decisões são de uma importância muito elevada para que os resultados práticos sejam bons. De seguida são tomadas as decisões práticas, onde é decidido exactamente de que forma serão implementadas as decisões teóricas que concretizam essas decisões.

Conforme referido, primeiro foram tomadas decisões conceptuais sobre os problemas. Para cada um dos módulos foram apresentadas as decisões mais relevantes tomadas. Foram apresentadas as soluções possíveis, escolhida uma delas e apresentados os argumentos para essa decisão. Todas as decisões são apresentadas detalhadamente para que todos os aspectos sejam especificados, desde cada módulo individualmente até à interligação entre eles.

Depois de especificada e justificada cada uma das decisões é altura de decidir como vão ser concretizadas na prática. Assim, para cada um dos problemas e módulos é apresentada uma forma de implementar concreta, especificando detalhes de tecnologia e plataforma que irão solucionar os problemas propostos anteriormente, com base nas decisões teóricas tomadas.

Neste capítulo foram apresentadas várias decisões. Primeiro foram tomadas as decisões necessárias para cada um dos componentes essenciais: análise e representação de informação, abstracção de modelos e por fim identificação de padrões. Em segundo lugar foram apresentadas as decisões relativamente à implementação específica de cada um dos componentes que implementam essas funcionalidades: o módulo de *parsing* (JPSI), o interface de Prolog (GPJaSI), o interface de análise (SAI) e por fim o *plugin* que incorpora os outros componentes.

Desta forma está concluído o capítulo que aborda a solução proposta para a implementação de cada uma das funcionalidades. Foi apresentada uma proposta detalhada de implementação para cada um dos módulos, que permitirão guiar o processo de implementação e chegar a uma ferramenta funcional (a MapIt) que soluciona os problemas apresentados.

Capítulo 6

Caso de estudo

6.1 Introdução

Fazendo o levantamento da situação vemos que neste ponto já estão definidas as principais decisões conceptuais. Foram já também especificadas estas decisões e respectivas concretizações. Desta forma foi possível iniciar o processo de implementação.

Depois de terminado o processo de implementação das especificações apresentadas, é possível concluir que os problemas propostos se encontram satisfeitos. O resultado obtido foi um *plugin* que pode ser instalado no IDE NetBeans, tal como previsto. Depois de instalado, as novas funcionalidades passam a estar disponíveis ao utilizador. Da forma como está integrado no IDE permite que a sua utilização seja simples e que facilmente possa fazer parte do processo de desenvolvimento.

Para utilizar a ferramenta implementada existe um requisito que é necessário mencionar. O requisito é que o programa gprolog esteja instalado no sistema operativo. Este requisito justifica-se pelo facto da ferramenta gprolog ser o motor de inferência utilizado pela ferramenta. Quanto as limitações, a ferramenta necessita apenas que o código seja feito em Java (1.6 preferencialmente) e sem erros sintácticos.

A ferramenta foi desenvolvida em na linguagem Java, organizada em módulos e está dividida em partes bem definidas. Encontra-se também documentada com Javadoc para permitir uma análise mais simples do código fonte.

Depois de implementada a ferramenta proposta é altura de a aplicar a um caso em específico e fazer o levantamento dos resultados obtidos na prática. Desta forma prova-se o conceito apresentado e mostra-se que pode ser implementada uma ferramenta que soluciona os problemas apresentados com as decisões conceptuais tomadas. Ao longo deste capítulo são apresentados dois casos de estudo distintos onde a ferramenta implementada foi posta à prova. Este será o capítulo onde é demonstrado o resultado da implementação (concretização das decisões previamente tomadas).

6.2 O caso em estudo

Para testar a ferramenta implementada será utilizado um caso de estudo com vários cenários. Este caso de estudo apresenta-se em duas fases. Primeiro para um software mais pequeno onde o processo será explicado em maior detalhe, mostrando passo-a-passo detalhadamente a utilização da ferramenta. De seguida será utilizado o código fonte de um software mais complexo e com maior número de classes que servirá para demonstrar que a ferramenta proposta irá funcionar num ambiente mais complexo.

6.2.1 Agenda

A Agenda é um programa exemplo que foi já mostrado atrás na secção da análise das ferramentas existentes na Figura 3.1, onde serviu de teste a essas ferramentas. Este software consiste numa agenda que permite organizar eventos e participantes. É uma simples agenda mas será o suficiente para mostrar o funcionamento das funcionalidades da ferramenta proposta. Será um cenário mais simplificado do ponto de vista da complexidade do código fonte, mas que requer o mesmo esforço da análise de um software maior e mais complexo por parte da ferramenta implementada.

A escolha deste software como exemplo mais detalhado deve-se ao facto de ser construído à custa de padrões por um lado, e por outro lado não conter demasiadas classes ao mesmo tempo que representa os elementos UML mais comuns (classes, interfaces, relações de herança, implementação, etc.). Estes dois factores fazem deste software um bom caso de estudo, simples de analisar mas com a complexidade de elementos suficiente para testar as funcionalidades. De forma mais fácil e eficiente é possível perceber em detalhe o resultado da utilização da ferramenta no software escolhido com todos os elementos do seu código fonte.

Com a análise do código deste software espera-se conseguir fazer uma demonstração do funcionamento de todas as funcionalidades da ferramenta desenvolvida. Espera-se ainda conseguir diagramas não demasiado complexos e ao mesmo tempo identificar e analisar nele os padrões de concepção. Estes são os motivos que fundamentam a escolha deste software para primeiro cenário do caso de estudo, para uma análise mais detalhada.

6.2.2 JHotDraw

O software escolhido para o cenário mais complexo foi o JHotDraw [24]. Este software consiste num editor de imagens por composição de elementos, representado na Figura 6.1. Permite criar elementos visuais como círculos, rectângulos, estabelecer relações visuais entre eles, texto, etc. Para além da edição e composição de imagens inclui outras funcionalidades como a edição das propriedades

visuais, animação dos elementos, por exemplo.

Os resultados práticos obtidos com este software não são a principal preocupação. Os resultados que permite obter são na verdade bastante limitados quando comparados com outras ferramentas de edição de imagens. O que torna este software interessante é a sua autoria e modo como foi implementado. É importante salientar que apesar das suas limitações, este software pode ser utilizado para fins práticos, uma vez que permite criar diagramas e imagens.

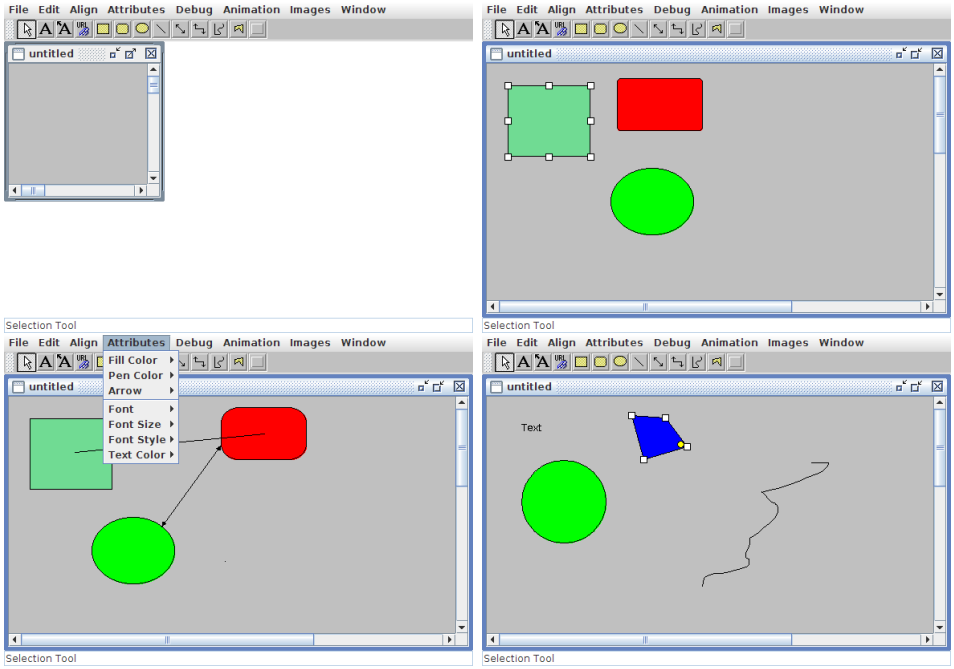

Figura 6.1: Imagens do software JHotDraw.

Existem vários factores que justificam a escolha deste software para além dos resultados práticos que produz. Primeiro é *open-source*, isto é, o seu código é de livre acesso e está disponível no site dos programadores. Em segundo este software tem uma dimensão considerável com cerca de 160 classes e 9000 linhas de código o que faz um bom caso de estudo de um ambiente real. Para além disso existe uma particularidade que distingue este software de muitos outros existentes. Essa característica é o facto do software ser desenvolvido por uma equipa que inclui Erich Gamma e como tal foi construído à custa de vários padrões de concepção conhecidos e documentados. Um dos objectivos que leva os autores a desenvolver este software é *"to identify new design patterns and refactorings"* [24]. O código fonte deste software foi já utilizado por outros autores para abordar a problemática de pesquisa de padrões [31]. A melhor forma de resumir o objectivo da utilização do software e os motivos que a levaram a essa escolha, é a citação dos autores no *website* do próprio software:

"JHotDraw is a Java GUI framework for technical and structured

> *Graphics. It has been developed as a "design exercise" but is already quite powerful. Its design relies heavily on some well-known design patterns. JHotDraw's original authors have been Erich Gamma and Thomas Eggenschwiler."* [24]

Os factos apresentados fundamentam a escolha deste software como caso de estudo de dimensão real. Na prática é de esperar que sejam encontrados facilmente vários padrões neste software. É também de esperar que os padrões surjam em contextos que façam sentido e quando identificados facilmente se perceba o porquê de estarem a ser utilizados. Não será feita uma análise tão detalhada como no primeiro cenário, mas serão analisados os resultados obtidos na sua generalidade mostrando o comportamento da ferramenta neste cenário mais complexo.

Importa ainda referir que algumas alterações foram feitas no código fonte para que o software pudesse funcionar correctamente na ferramenta implementada. Primeiro foram atribuídos tipos as colecções, como no caso do `ArrayList`, para que fossem correctamente inferidas as relações. Em segundo foi alterado o nome de algumas variáveis, nomeadamente `enum`, uma vez que na versão actual da especificação Java (1.6) esta é uma palavra reservada. De resto, todo o código permaneceu inalterado tal como está no disponível no *website* do software.

6.3 Descrição detalhada da aplicação da ferramenta

Será nesta secção apresentada a forma como esta aplicação pode ser utilizada na prática. Será também mostrado passo-a-passo como utilizar cada uma das funcionalidades bem como os resultados que são apresentados ao utilizador.

Com o IDE NetBeans (versão 6) instalado e funcional, o processo de instalação do *plugin* é muito simples, pois é feito pelo NetBeans tal como na instalação de um qualquer outro *plugin*. A partir daí o processo de utilização é trivial. Segue-se uma descrição de como utilizar a ferramenta passo-a-passo de modo a solucionar os problemas propostos.

6.3.1 Importar ficheiros Java

A importação ou selecção de ficheiros Java é o primeiro passo do processo, essencial para todas as operações. Este é o momento onde são definidos os elementos que irão ser parte do modelo a inferir. Neste momento o utilizador define como vai ser constituído o sistema em análise.

Agenda

Depois de instalado o *plugin*, uma nova opção fica disponível no menu de contexto para os ficheiros Java como se pode ver na Figura 6.2 à esquerda, ou um botão para analisar todo o projecto como se pode ver na Figura 6.2 à direita.

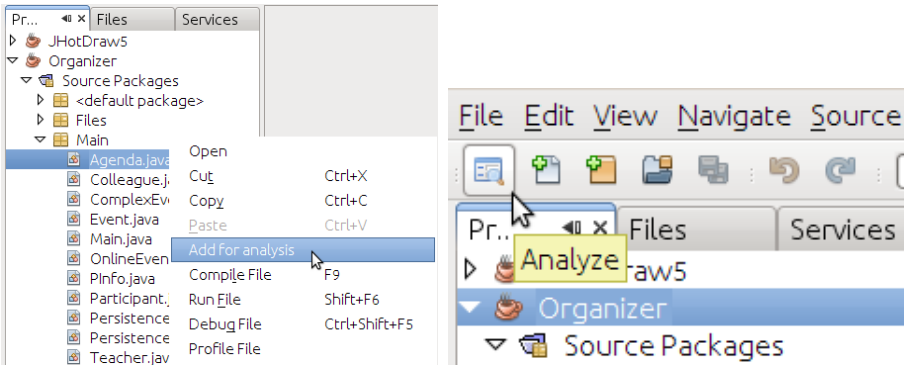

Figura 6.2: Funcionalidade extra no menu de contexto e na barra de ferramentas.

Se o utilizador pretender fazer uma selecção mais cuidada dos elementos a analisar, pode seleccionar um ficheiro ou um conjunto de ficheiros Java, e escolher a opção de adicionar os ficheiros para análise. Se por outro lado quiser analisar o projecto completamente, poderá seleccionar a opção de analisar todo um projecto. Estas duas formas de interacção dão mais flexibilidade ao utilizador na forma como faz a selecção de ficheiros.

JHotDraw

Seja qual for a dimensão do projecto, este processo ocorre da mesma forma. Para o projecto JHotDraw bastaria seguir os mesmos passos para importar os ficheiros Java. Porém, considerando a quantidade de ficheiros de software, talvez seja mais vantajoso utilizar o botão de análise uma vez que adiciona todos os ficheiros simultaneamente. A opção de seleccionar um conjunto de ficheiros será mais vantajosa quando se pretende analisar uma parte específica do projecto. Não existem quaisquer outras diferenças neste processo quando comparado com a Agenda.

6.3.2 Pré-processamento dos ficheiros

O pré-processamento dos ficheiros ocorre quando o utilizador parametriza o processo (que será posteriormente automatizado). É o momento onde são definidos os detalhes e requisitos, e o utilizador específica que resultados pretende obter do processo. Os restantes passos são completamente automatizados não necessitando de mais interacção do utilizador.

Agenda

Assim que o utilizador selecciona quais os ficheiros a analisar, o ecrã de pré-processamento é mostrado. Aqui são listados os ficheiros seleccionados, juntamente com um conjunto de opções que poderá escolher, representados na Figura 6.3.

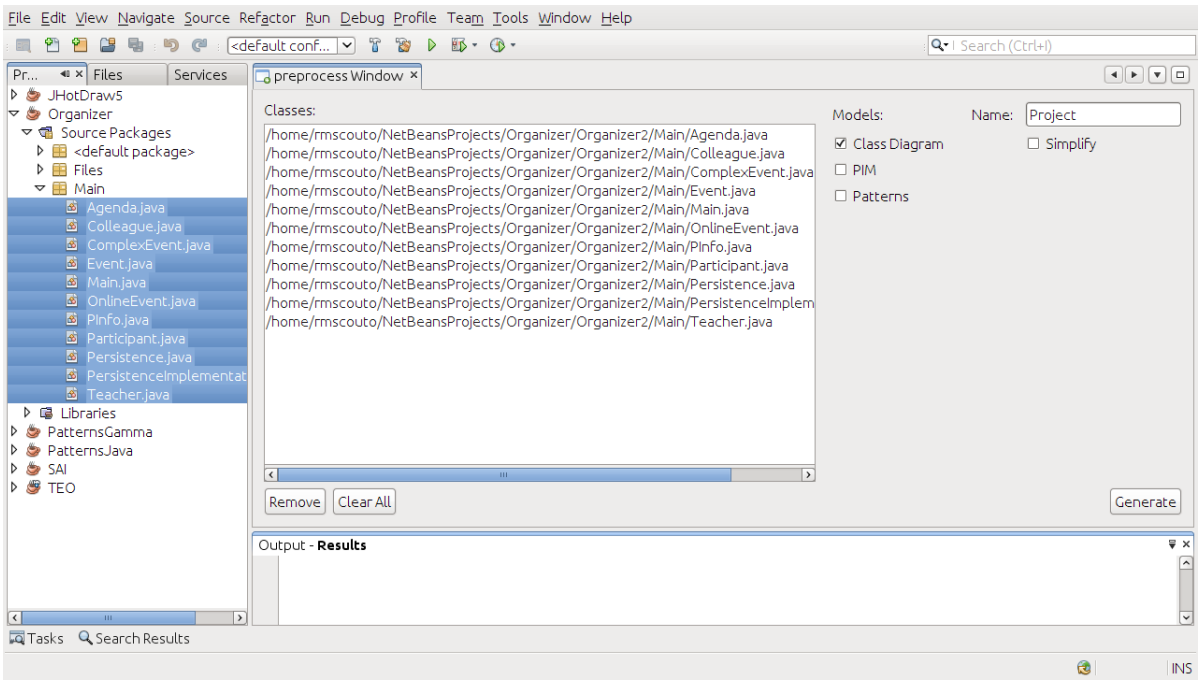

Figura 6.3: Janela de pré-processamento.

Depois de escolhidas as preferências e que diagramas pretende, o utilizador apenas tem de seleccionar Generate e a ferramenta mostrará os resultados pretendidos. A partir do momento que o utilizador solicita essa acção, é feito o processo de análise e representação intermédia da informação contida nos ficheiros a analisar. Essa análise soluciona o problema de "Analisar e extrair informação do código fonte de programas Java". De seguida serão descritas cada uma das funcionalidades disponíveis no menu de pré-processamento.

JHotDraw

Este processo também não difere seja qual for a dimensão do projecto a analisar. Existem contudo operações disponibilizadas, como é o caso de Clear All a pensar em grandes quantidades de ficheiros a analisar. A validade das funcionalidades para projectos pequenos é a mesma para projectos de maior dimensão.

6.3.3 Diagrama de Classes (PSM)

Representado sob a forma de diagrama de classes UML, o PSM é um dos diagramas objectivo da ferramenta proposta. Este processo gera resultados visuais úteis para análise de um software.

Agenda

Depois de seleccionados (e listados) os elementos o processo está pronto para iniciar. Nesse momento é apenas necessário seleccionar a opção Class Diagram e de seguida Generate. Neste caso em específico foram seleccionadas os elementos `Agenda`, `Colleague`, `ComplexEvent`, `Event`, `Main`, `OnlineEvent`, `PInfo`, `Participant`, `Persistence`, `PersistenceImplementation` e `Teacher`. Seleccionou-se a opção Generate e o diagrama UML obtido é o apresentado na Figura 6.4.

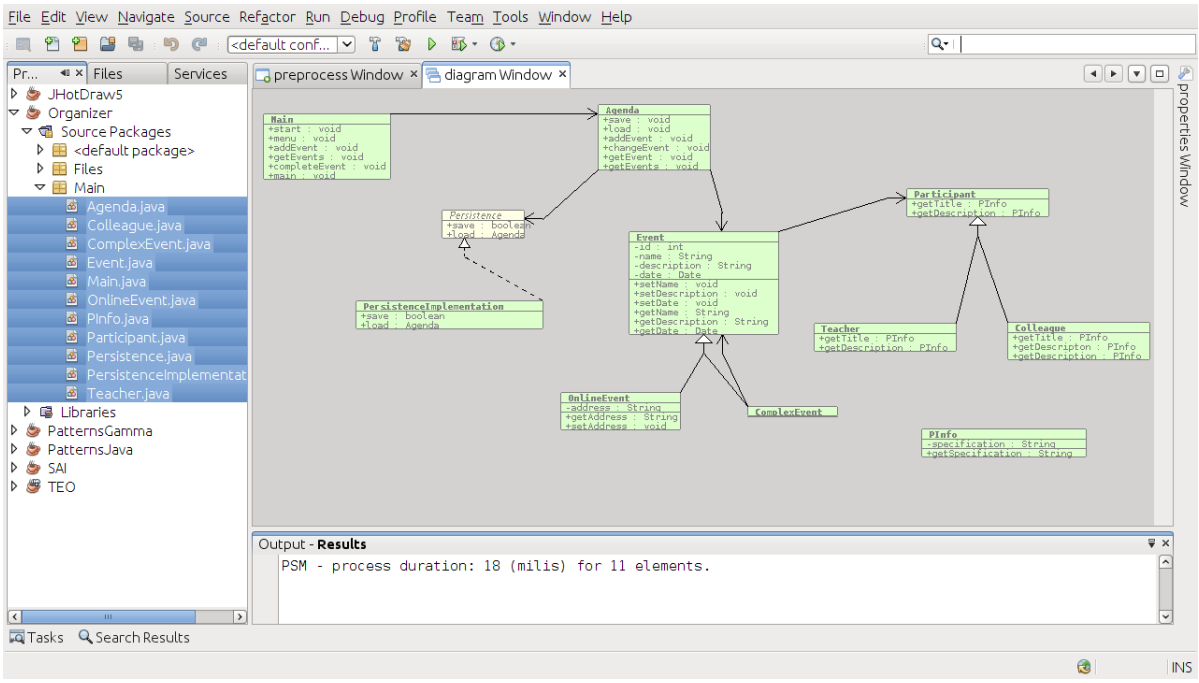

Figura 6.4: Diagrama de classes UML (PSM) para a Agenda.

Como se pode ver na Figura 6.4 o diagrama segue a notação UML. Todos os elementos previamente seleccionados e respectivas relações são representados visualmente. Neste caso podemos ver as classes e um interface, em que temos uma relação de implementação e várias de herança e uma composição. Clicando num desses elementos é possível ter informação mais detalhada sobre esse elemento, como é visível na Figura 6.5.

O diagrama apresentado é interactivo pelo que a disposição dos elementos pode ser alterada pelo utilizador. O problema de "Inferir e representar diagramas de classes UML com base na informação extraída" fica solucionado com a

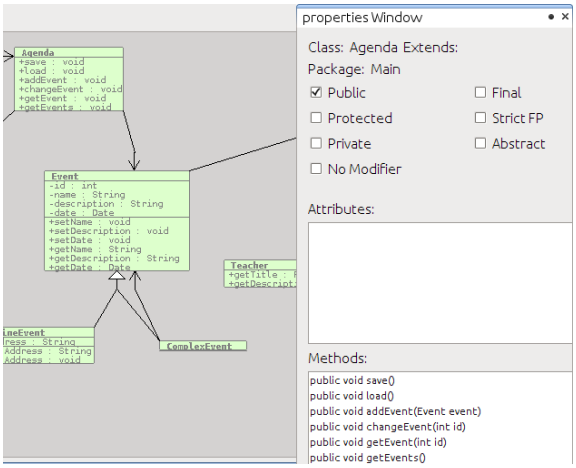

Figura 6.5: Detalhe de um elemento UML, `Agenda`.

implementação desta funcionalidade, que permite obter os resultados mostrados. Este processo é feito com base numa representação intermédia previamente calculada, como descrito atrás. O processo é completamente automatizado como era objectivo da ferramenta.

JHotDraw

Naturalmente que quanto maior a complexidade de um software, mais informação irá conter o diagrama PSM inferido. Tendo em conta a dimensão do software JHotDraw é de esperar que o diagrama gerado tenha uma maior complexidade. Como o diagrama tem mais elementos e relações entre eles, o diagrama é mais complexo, como mostra a Figura 6.6. O facto do diagrama gerado ser interactivo e permitir reorganizar os elementos visuais, facilita a análise dos diagramas mesmo que de grandes dimensões. Não existe uma restrição no número de elementos que a ferramenta suporta, desse modo esta funcionalidade é valida qualquer que seja o tamanho do software a analisar.

Para um software de grandes dimensões existe ainda outra opção que auxilia no processo de análise. Esta denomina-se Simplify e permite simplificar a representação visual de um diagrama. Para tal representa cada elemento do diagrama apenas pelo seu nome (removendo informação de métodos e atributos). Esta funcionalidade permite ter uma visão global mais simples sobre diagrama em causa.

6.3.4 Diagrama PIM

Este diagrama é também representado com a notação UML para diagramas de classes. Representa um diagrama de alto nível para o software em análise. Este tipo de diagramas é obtido pelo processo de transformação de diagramas PSM,

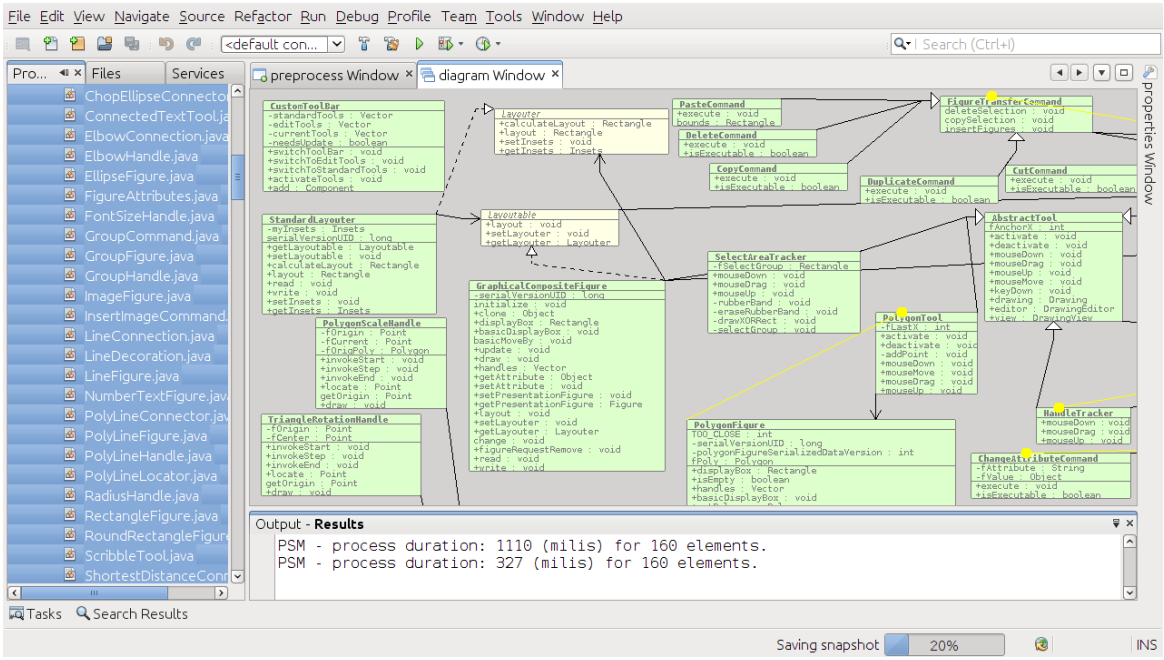

Figura 6.6: Diagrama de classes UML (PSM) para projecto mais complexo.

já apresentados.

Agenda

Partindo do mesmo ecrã de pré-processamento anteriormente mostrado (na Figura 6.3) é possível também inferir o diagrama PIM. O utilizador apenas tem de seleccionar a opção `PIM` e o diagrama é apresentado ao utilizador. Foi utilizado todo o código fonte do software em questão para permitir visualizar as diferentes transformações que ocorrem neste processo. Terminado o processo, mostra-se na Figura 6.7 na parte inferior o PIM resultante e na parte superior o diagrama de classes PSM original, para que se possa perceber as diferenças entre os dois diagramas.

Comparando os dois modelos na figura é possível perceber as várias transformações que ocorreram, vendo facilmente que o PSM foi abstraído a um PIM. Podemos ver métodos e elementos que foram eliminados e simplificados, tal como esperado. Estas transformações seguem as regras de transformação de PSM em PIM descritas previamente.

É desta forma que se demonstra a possibilidade de implementar a funcionalidade de abstracção de diagramas, que soluciona o problema de "Permitir abstrair os diagramas PSM a PIM". As decisões previamente tomadas mostram-se suficientes para permitir a implementação desta funcionalidade. Mais uma vez esta funcionalidade é completamente automatizada.

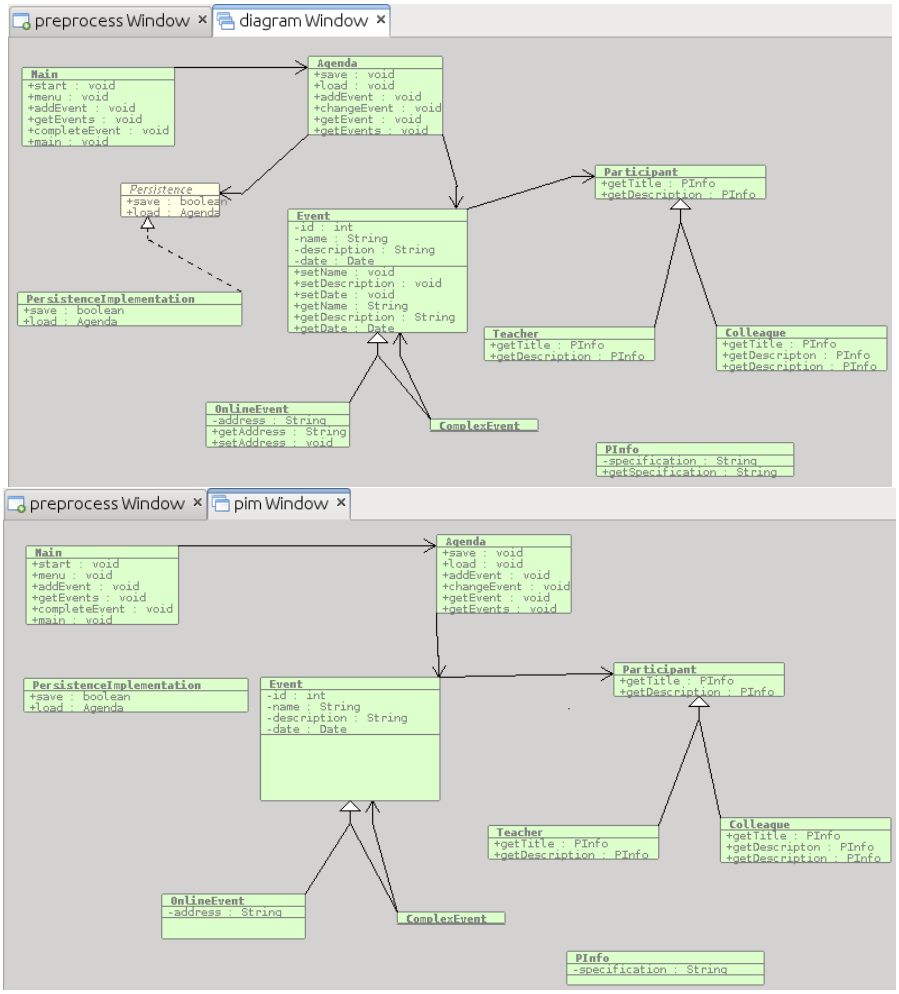

Figura 6.7: Diagramas PSM (em cima) e PIM (em baixo) gerados.

JHotDraw

Para projectos de maiores dimensões este processo é efectuado da mesma forma. Tal como no caso anterior do PSM, estes diagramas são dinâmicos o que facilita a análise de projectos de grandes dimensões. Esta funcionalidade é tão válida para projectos pequenos como para projectos de maiores dimensões. Na Figura 6.8 é possível ver estas mesmas transformações aplicadas a classes do projecto JHotDraw.

6.3.5 Inferência de padrões

Esta funcionalidade soluciona o último problema dos que a ferramenta se propõe a solucionar. Esta é a funcionalidade de análise de código de mais alto nível, que corresponde à possibilidade de inferência de padrões num software.

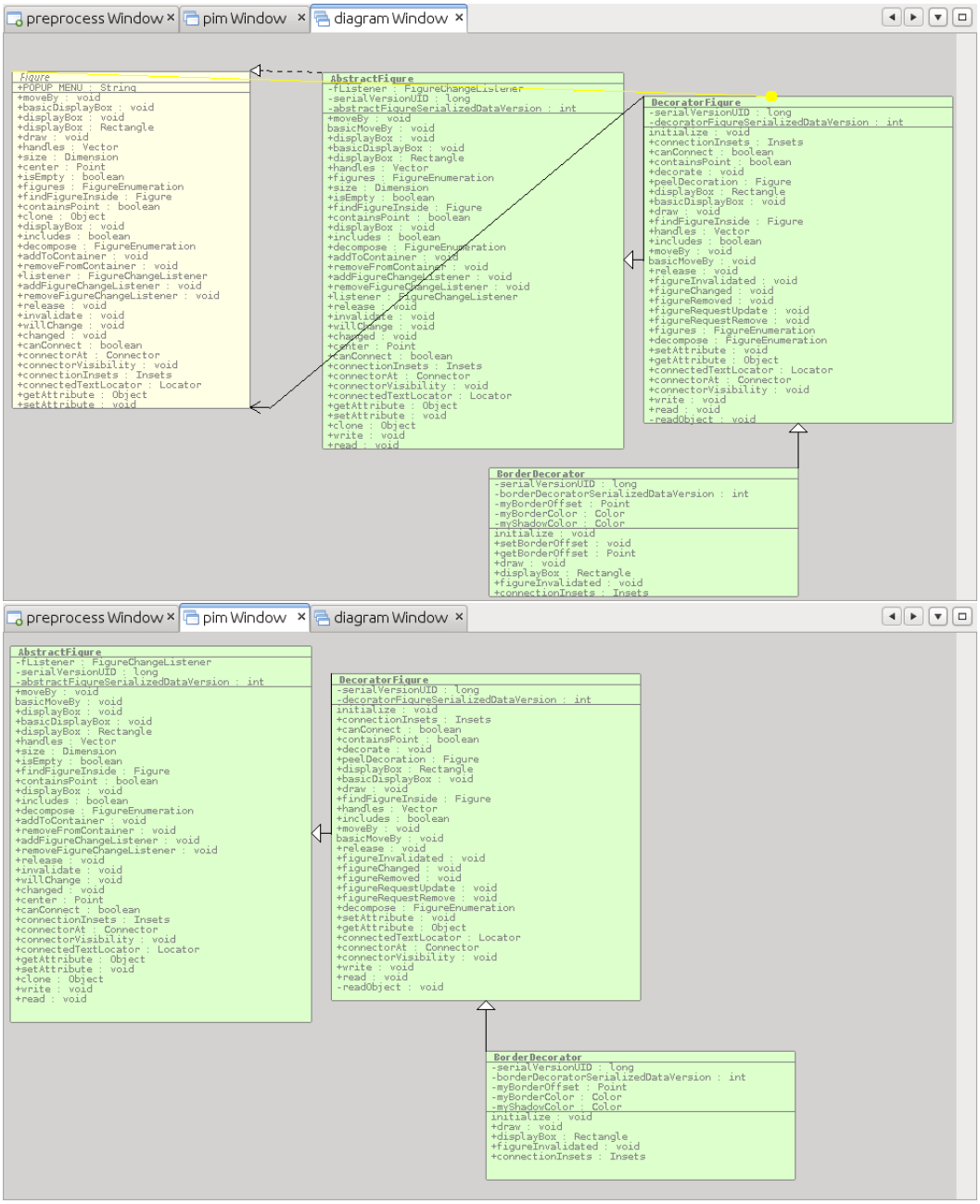

Figura 6.8: Diagrama PSM (em cima) e PIM (em baixo) gerados para um subconjunto das classes de JHotDraw.

Agenda

A última funcionalidade disponível no menu de pré-processamento é a inferência de padrões na representação intermédia. Para utilizar esta funcionalidade, a partir do ecrã de pré-processamento, o utilizador tem de seleccionar a opção Patterns. Depois tem de seleccionar uma de duas opções: usar o catálogo embu-

tido ou especificar a localização de um catálogo externo, como representado na Figura 6.9.

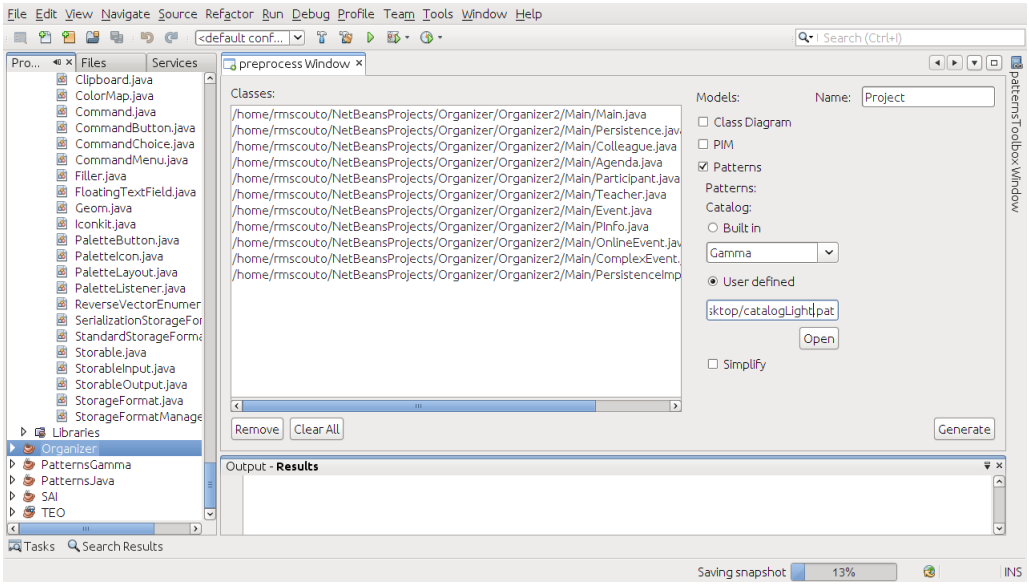

Figura 6.9: Escolha de padrões na janela de pré-processamento.

Caso o utilizador escolha o catálogo disponibilizado pela ferramenta o processo de detecção pode iniciar: identifica os padrões caso existam e faz a sua listagem, representado na Figura 6.10. O utilizador pode seleccionar um dos padrões listados e ele será representado visualmente.

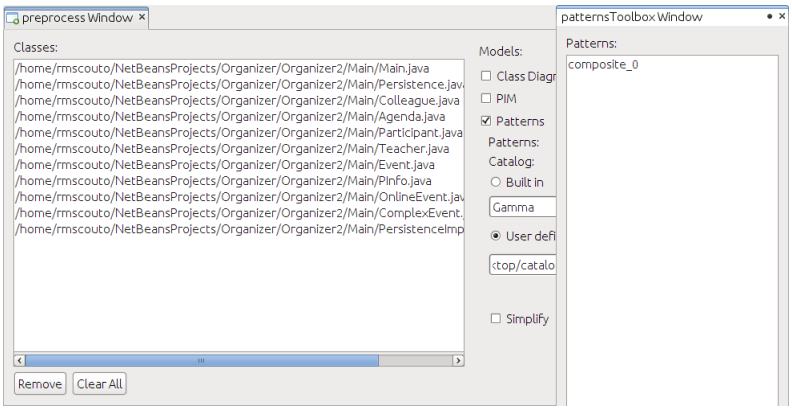

Figura 6.10: Listagem com um padrão Composite identificado.

Quando o utilizador seleccionar um catálogo externo terá de indicar o caminho para um ficheiro textual que contém a informação desse catálogo. Neste caso foi utilizado o catálogo apresentado na Figura 6.13 que contém o padrão Composite. Nessa mesma figura é mostrado um mapeamento entre o padrão descrito e a sua representação visual para que seja mais fácil perceber a definição

do padrão. Mais à frente será descrito de que modo é feito o mapeamento de cada um dos elementos do padrão a um elemento Prolog do catálogo.

JHotDraw

A dimensão do software em análise não é impedimento para a utilização deste módulo. Para software de maiores dimensões é normal que a quantidade de padrões encontrados seja maior (visível na Figura 6.11). A listagem de padrões inferidos mostra-se muito útil neste caso por organizar convenientemente os padrões.

Tal como nos diagramas PSM, também nas opções dos padrões está disponível a funcionalidade Simplify. Neste contexto esta funcionalidade permite ter uma visão mais global sobre os padrões existentes em softwares de grandes dimensões, como se pode ver na Figura 6.12.

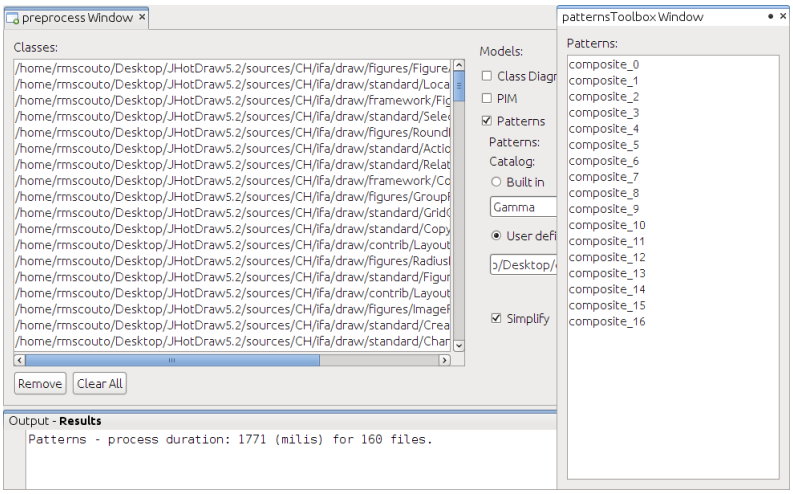

Figura 6.11: Listagem de padrões inferidos em JHotDraw.

Especificação de um padrão

Na Figura 6.13 está representado o padrão Composite. Aparece descrito por um lado sob a forma de factos Prolog que são no fundo uma entrada do catálogo (do lado esquerdo). Por outro lado é apresentada uma imagem representando graficamente esse padrão em notação UML (do lado direito). Para cada um dos mapeamentos será feita de seguida uma descrição.

1. Esta regra define que um elemento (`Component`) é uma classe. Esta regra é utilizada para definir as classes que existem num sistema.

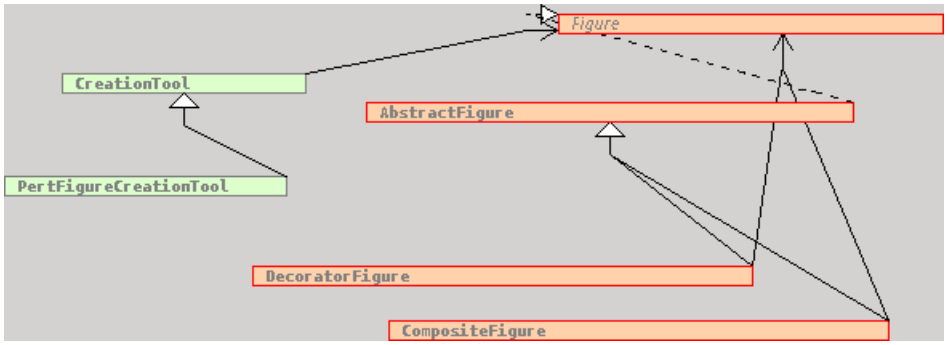

Figura 6.12: Visualização de informação sobre padrões em JHotDraw, de forma simplificada.

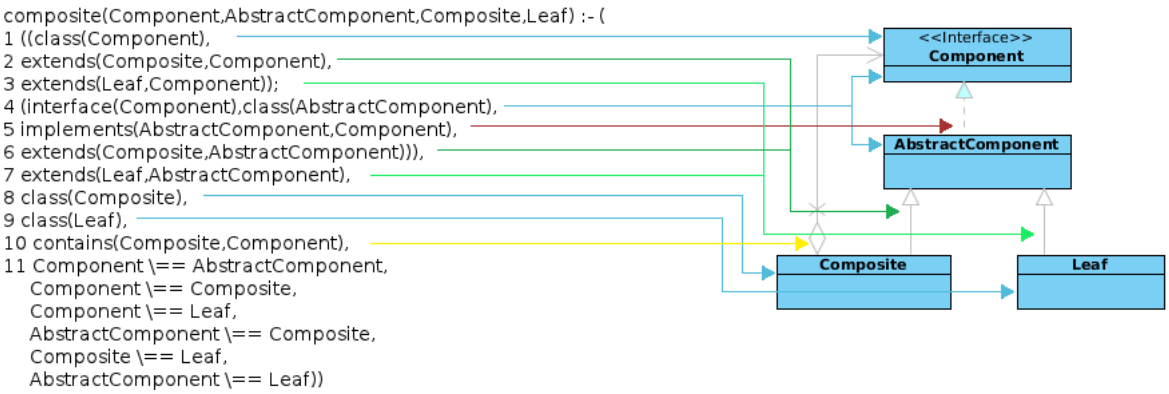

Figura 6.13: Explicação da definição de padrões.

2. 3. Esta é a regra que define a relação de herança entre dois elementos. Temos que `Composite` e `Leaf` herdam de `Component`. As relações de herança são especificadas desta forma.

4. Este conjunto de regras permite duas representações do mesmo padrão. Nesta regra permite que `Component` seja um interface concretizado por `AbstractComponent` , ao contrário do que acontece em 1 onde `Component` é uma classe.

5. Aqui é definida a regra que especifica a relação de implementação (em que `AbstractComponent` implementa o interface `Component`).

6. 7. Tal como em 2 e 3, esta regra define a relação de herança. Neste caso `Composite` e `Leaf` herdam de `AbstractComponent`.

8. 9. À semelhança de 1, esta regra define que um elemento (`Composite` e `Leaf`) é uma classe.

10. Esta regra define que um elemento `Composite` contém (um ou mais) `Component`. Esta regra define a existência de uma relação de agregação, composição ou associação entre dois elementos.

11. Esta regra define que os elementos não podem unificar com eles mesmos, por exemplo uma classe não pode ter simultaneamente o papel de `Component` e `Leaf`. É importante referir estas restrições, ou os resultados podem não corresponder ao esperado.

Esta é uma regra de exemplo e apresentada o padrão Composite. Demonstra a flexibilidade do formato de definição de padrões. As capacidades da linguagem Prolog fazem com que esta regra possa cobrir duas variações do padrão. A capacidade de representar as duas variações baseia-se no operando "ou", escrito em Prolog com `;`. Neste caso é definido o padrão Composite onde podemos ter `Component` como classe, ou `Component` como interface, e neste caso um `AbstractComponent` que implementa `Component`. Se `Component` for uma classe vai haver classes que estendem dela, como `Composite` e `Leaf`. Se por outro lado `Component` for um interface estas classes vão estender a sua implementação, `AbstractComponent`. Graças à utilização do operando "ou", vemos que facilmente se pode fazer uma definição mais genérica de um padrão que abrange duas variantes. Esta é a forma que permite ter uma maior ou menor restrição quando definimos uma regra, determinando assim se serão identificados mais ou menos padrões.

Se todas as restrições da regra forem satisfeitas, então estamos perante a presença de um padrão. Um catálogo poderá conter mais que uma definição, permitindo assim identificar assim vários padrões diferentes.

Se uma regra for demasiada restritiva, o mesmo padrão poderá aparecer várias vezes. Suponhamos um padrão que consiste apenas na relação de implementação, o padrão implements. Este padrão define-se como havendo um interface que pode ter uma ou mais implementações. Contudo, a regra do padrão está definida para um interface que possui uma só implementação:

```
implements(Interface,Classe) :- class(Classe),
interface(Interface), implements(Class,Interface).
```

Como esta regra é demasiado restrita, no caso de termos por exemplo três classes que implementam um interface, serão inferidos três padrões (Figura 6.14). Podemos também ver que em JHotDraw foram identificados dois padrões no mesmo conjunto de subclasses na Figura 6.15.

Depois de especificado o catálogo de padrões, é altura de fazer análise da representação intermédia de dados e mostrar ao utilizador que padrões foram inferidos nessa representação. No caso em questão foi pesquisado e identificado o padrão Composite, apresentado na Figura 6.10. Assim que o utilizador selecciona o padrão na lista, é mostrado num diagrama similar ao PSM, com os

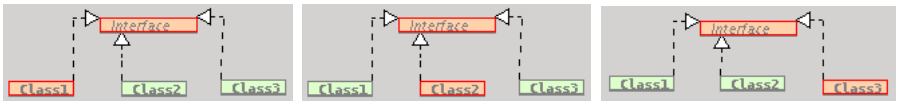

Figura 6.14: Identificação do mesmo padrão várias vezes.

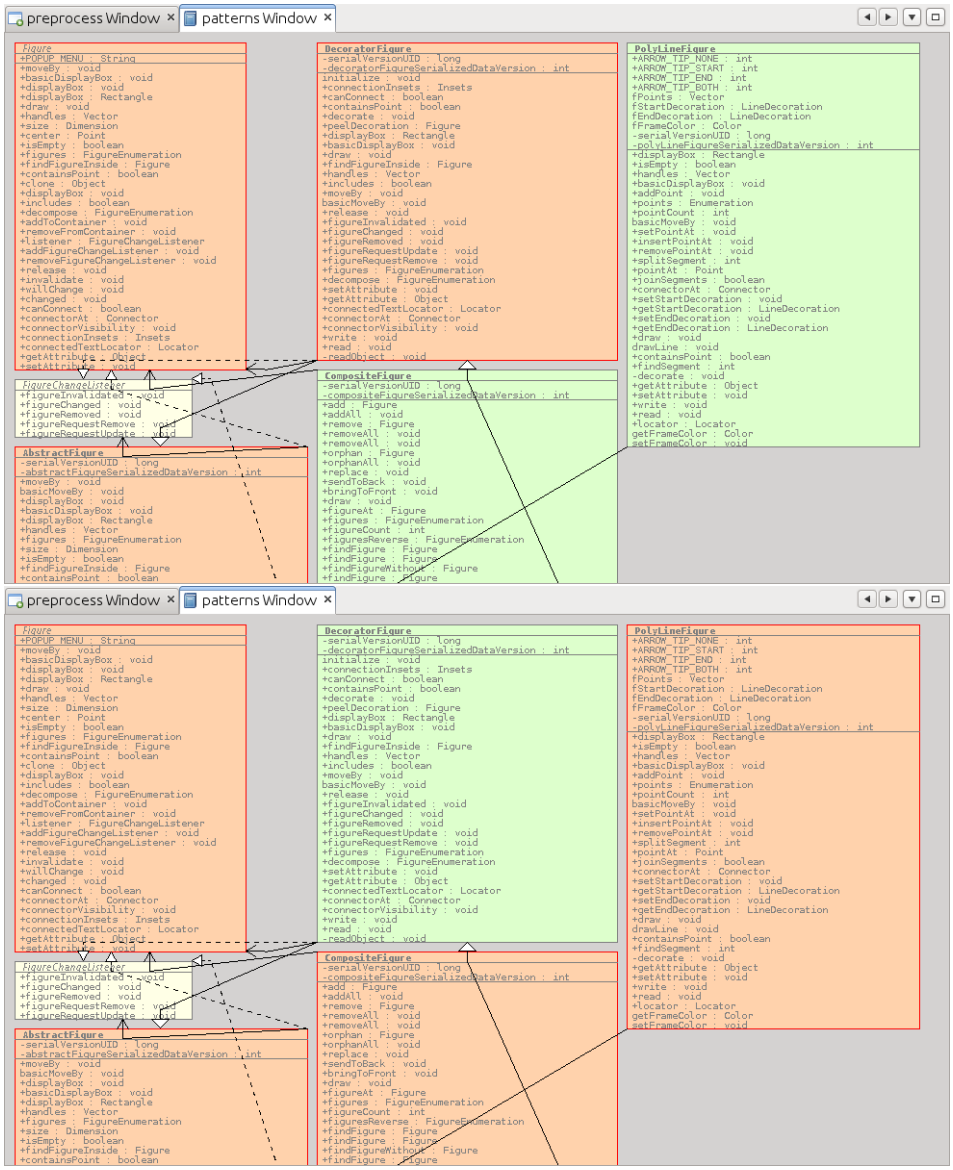

Figura 6.15: Dois padrões no mesmo diagrama: Decorator (em cima) e Composite (em baixo).

elementos que fazem parte do padrão realçados ou caso deseje, simplificado como apresentado na Figura 6.16.

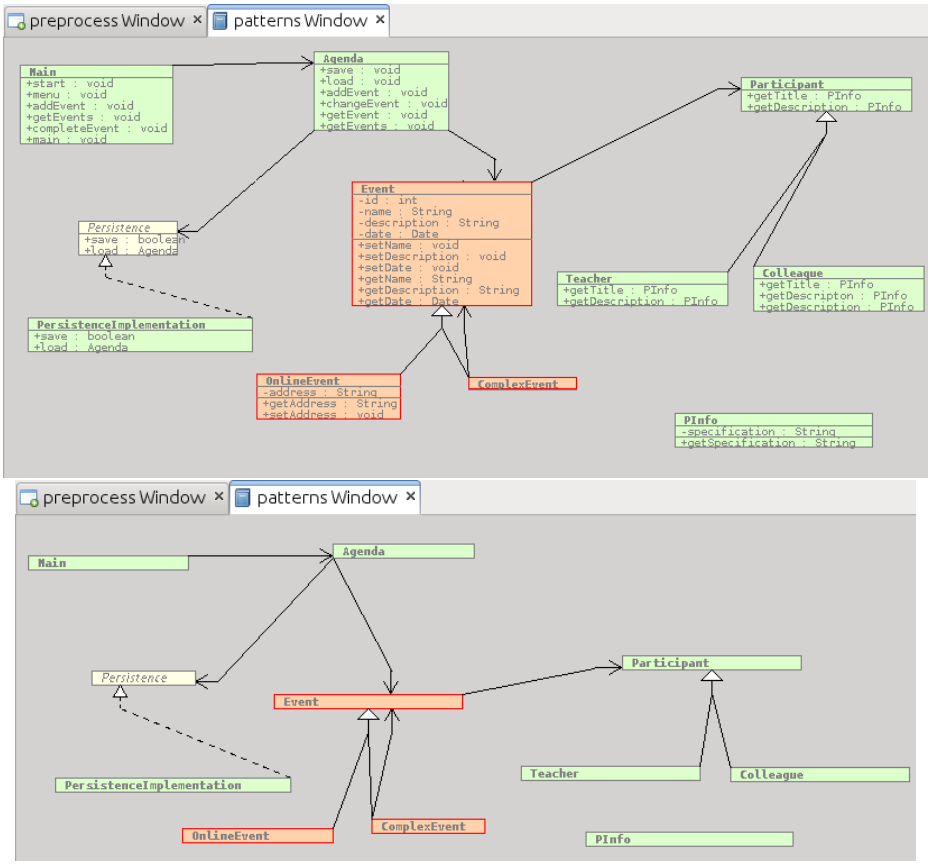

Figura 6.16: Duas representações do padrão Composite: normal (em cima) e simplificada (em baixo).

A implementação desta funcionalidade mostra que a integração da funcionalidade de inferência e representação de padrões de concepção num software pode ser integrada com as funcionalidades de inferência de modelos. Mostra-se também que é possível integrar uma tecnologia externa no processo, neste caso Prolog, e que esta se mostra uma mais valia obtendo resultados esperados. Foi ainda apresentada uma concretização de um catálogo de padrões que demonstra a facilidade da sua definição. Na implementação deste módulo foi reaproveitado algum do trabalho previamente feito, nomeadamente com a representação intermédia de dados e representação visual. Esta é a funcionalidade que soluciona o problema de, com base numa representação intermédia de um software "Inferir padrões de concepção nessa representação". Por fim é apresentado o interface que se mostra fácil de utilizar e não exige esforço por parte do utilizador para adaptação ao mesmo.

6.3.6 Detalhes de implementação

Depois de apresentado o caso de estudo e de que modo a ferramenta funciona afim de resolver com sucesso os problemas previamente propostos, é altura de referir alguns detalhes sobre o processo de implementação. O processo de implementação foi baseado nas decisões tomadas previamente, decisões essas que por sua vez foram baseadas nos problemas concretos a solucionar. Um problema encontrado ao longo do processo de desenvolvimento, com detalhes não previstos previamente foi a integração de um motor de inferência Prolog na ferramenta. Para além dos detalhes não previamente previstos, foi necessário definir uma ferramenta específica. Apesar do programa seleccionado para tal ser gratuito e de distribuição livre, impõe à partida uma dependência para a utilização desta ferramenta. Outro problema que necessitou de cuidado adicional foi o desenvolvimento dos diferentes módulos de forma independente, e respectivas comunicações. Planear o processo de desenvolvimento cuidadosamente e com ideia de todos os problemas foi essencial para o sucesso do processo de implementação.

O resultado do processo de implementação foi um *plugin* que é integrável no IDE NetBeans. Este é composto por quatro módulos: integração do motor de Prolog, *parsing*, o *facade* que interage dois módulos anteriores ao mesmo tempo que faz o processamento dos factos Prolog e por fim o *plugin* em si com as funcionalidades de representação visual. No final obtiveram-se cerca de 45 classes e mais de 4500 linhas de código. Para além disso o código foi desenvolvido no sentido de estar bem construído, documentado e estruturado permitindo uma fácil análise e compreensão.

Quando comparado o resultado com os objectivos propostos podemos tirar diversas conclusões. Primeiro, todos os objectivos propostos foram alcançados, isto é, todas as funcionalidades previstas foram implementadas e estão funcionais. Foi provado que as metodologias adoptadas são as adequadas para a resolução desses objectivos e solução de problemas. Depois, para cada um dos módulos havia objectivos propostos, como o aumentar do grau de usabilidade, interacção, melhoramento do processo de reconhecimento de elementos entre outros. Estes objectivos foram regra geral alcançados. Por fim a integração e combinação dos elementos de forma a cooperarem num só ambiente de desenvolvimento foi outro dos objectivos propostos. Este objectivo foi alcançado, como é possível concluir pelo caso de estudo, e a integração das três funcionalidades (mais módulo de visualização) foi conseguida numa só ferramenta.

6.4 Resultados obtidos

Depois de aplicada a ferramenta ao caso de estudo é altura de comentar os resultados obtidos. É necessário fazer algumas considerações quanto aos resultados obtidos no que diz respeito à qualidade dos resultados. Só desta forma é possível fazer um balanço sobre todo o projecto.

6.4.1 Resumo dos módulos

O aspecto visual da ferramenta já foi apresentado previamente na Figura 6.2, Figura 6.3, Figura 6.4, Figura 6.5, Figura 6.7, Figura 6.9, Figura 6.10 e Figura 6.16. Conclui-se que foi possível integrar completamente a ferramenta dentro do IDE. A integração da ferramenta fez uma extensão as funcionalidades que o IDE oferece. As acções ficam disponíveis em menus de contexto, em barras de ferramentas e separadores nas áreas comuns de desenvolvimento (por exemplo editores de código).

O correcto funcionamento das funcionalidades implementadas foi demonstrado no caso de estudo. Quando aplicada a funcionalidade da análise do código fonte de um programa, ele é reconhecido correctamente. A funcionalidade de análise do código fonte foi implementada com sucesso, recorrendo a um *parser* externo. Apesar de o metamodelo e o *parser* serem específicos para Java poderiam ser alterados para outras linguagens. Nesta ferramenta são específicos para uma linguagem e têm as restrições indicadas previamente. Para a linguagem Java o módulo proposto permite reconhecer elementos da linguagem de acordo com os objectivos.

A integração do motor de inferência Prolog na ferramenta proposta permite uma completa interacção. Fornece as funcionalidades necessárias para solucionar o problema proposto. Foi implementado com sucesso e permitiu obter os resultados esperados. Este módulo está preparado para que possa ser utilizado em outros projectos que necessitem destas funcionalidades, pois foi desenvolvido respeitando os princípios da modularidade. A restrição para o funcionamento é ter o programa gprolog instalado. Este módulo interage com o programa fornecendo a interacção com a ferramenta. Caso este módulo fosse reutilizado em outro projecto poderia existir a necessidade de refinar e estender algumas das funcionalidades oferecidas para permitir uma melhor interacção.

O módulo de análise, geração de factos e consulta de resultados foi implementado de acordo com o previsto. A sua interacção com os módulos inferiores foi bem conseguida e os resultados obtidos são satisfatórios como visto no caso de estudo. Este módulo abstrai de forma eficiente os módulos inferiores o que faz com que a qualquer altura possam ser substituídos, necessitando contudo que o metamodelo seja mantido. Uma alteração do metamodelo faz com que este módulo tenha de ser reescrito em parte. Os dados que preenchem a base de conhecimento são também gerados por este módulo e uma extensão dos factos, por exemplo, requer alterações neste módulo.

O módulo que permite o aumento do nível de abstracção de modelos foi completamente implementado e mostrado como permite solucionar o problema apresentado. Este módulo não é independente por estar integrado no *plugin*. Isto faz com que o mesmo não possa ser reutilizado, necessitando de trabalho extra para o migrar para um módulo independente. Quanto às regras de transformação, estas podem ser personalizadas, mas essa personalização é feita por alteração directa do código fonte, o que pode não ser prático para uma utilização

regular. Poderia ser definida uma forma de permitir carregar as transformações de um formato externo (à semelhança dos padrões). Isto poderia ser conseguido à custa da reflexão de Java, por exemplo. Como prova de conceito foi mostrado o funcionamento num contexto real com as regras propostas.

Como mostrado também nesta secção a representação gráfica de informação foi correctamente implementada e permite mostrar todos os tipos de diagramas. Os diagramas seguem a notação UML e são interactivos. Este módulo pode ser reaproveitado para outras tecnologias, mas é dependente do metamodelo e do IDE. O módulo de representação visual pode ser melhorado em muitos aspectos, como a usabilidade do mesmo, permitindo facilitar análise de sistemas complexos, por exemplo. Ainda relativamente a este módulo poderia ser implementada a funcionalidade de guardar e carregar diagramas por fazer sentido neste contexto.

Um balanço sobre os resultados do processo de implementação permite concluir que os módulos obtidos apresentam resultados satisfatórios. Funcionam de acordo com o esperado numa ferramenta deste tipo. Os resultados que produzem são a resolução dos problemas previamente apresentados e estudados. Existem contudo várias optimizações que poderiam ser feitas como foi descrito atrás para cada um deles.

6.4.2 Comparação com outras ferramentas

Será interessante fazer uma comparação desta ferramenta com as previamente analisadas. Esta comparação permite perceber por um lado em que categoria de ferramentas esta se enquadra, e por outro perceber que funcionalidades estão a ser comparadas. A comparação entre as funcionalidades seleccionadas das ferramentas analisadas serão apresentadas de seguida. Esta análise permite obter uma visão global sobre o enquadramento da ferramenta implementada face ao contexto das ferramentas deste tipo. Resta referir que as ferramentas em análise serão ArgoUML, Reclipse, Fujaba, jGRASP, Ptidej e o Visual Paradigm.

Quantidade de elementos UML

Nas ferramentas analisadas a quantidade de elementos foi variada, mas em geral baixa. Todas as ferramentas foram capazes de reconhecer os elementos Java (embora não na totalidade). No que diz respeito às classes e interfaces apenas o Reclipse/Fujaba não foi capaz de reconhecer todos os elementos. Por outro lado no que diz respeito às relações de implementação, agregação e associação houve várias ferramentas que não foram capazes de reconhecer correctamente as relações, como foi o caso de ArgoUML, jGRASP,e Ptidej. Outras ferramentas, nomeadamente ArgoUML e Reclipse não foram capazes de identificar colecções com tipos. Por último, Visual Paradigm apresenta os resultados mais satisfatórios como seria de esperar, reconhecendo todos os elementos e colecções correctamente.

Na ferramenta proposta os resultados de análise do código fonte foram bastante satisfatórios face às restantes ferramentas. Todos os elementos foram reconhecidos (tanto classes como interfaces), bem como as relações entre elas, com as restrições de as colecções terem tipos e serem declaradas como variáveis de classe.

Diagrama de classes

Todas as ferramentas oferecem a possibilidade de criar diagramas de classes. Os diagramas gerados variam em número, qualidade e quantidade de elementos representados. A ferramenta jGRASP foi a que apresentou resultados menos satisfatórios, com diagramas simplificados e com muita pouca informação, sendo resumidamente um diagrama em que cada elemento Java é representado por um rectângulo. Já ArgoUML é capaz de representar mais alguma informação do que jGRASP, contudo continua a não ser informação muito completa, adicionando apenas algumas relações entre elementos (herança e implementação). Em Reclise/Fujaba a qualidade dos diagramas melhora no que diz respeito ao detalhe dos elementos. Esta ferramenta apresenta os métodos e atributos em notação UML, contudo o número de relações diminui, reconhecendo apenas a relação de herança. Já em Ptidej os diagramas apresentam resultados mais satisfatórios, sendo que são representados todos os elementos em notação UML, desde atributos, métodos e relações embora nem todas as relações tenham sido apresentadas. Por outro lado os diagramas são estáticos não sendo possível, por exemplo, alterar a disposição dos elementos. Por fim Visual Paradigm tem os resultados mais completos com os diagramas mais detalhados e interactivos.

A ferramenta desenvolvida apresenta diagramas UML completos e detalhados, com os elementos, atributos, métodos e relações, sendo ainda possível obter informação extra clicando sobre eles. Os diagramas são interactivos e o utilizador pode alterar a disposição dos elementos. A representação visual não é tão elaborada como nalgumas das ferramentas visuais a nível estético, porém consegue cumprir com sucesso a tarefa de mostrar informação.

Inferência de padrões

A inferência de padrões apenas está disponível em algumas das ferramentas, nomeadamente no Reclipse/Fujaba e no Ptidej. Pelo que foi possível testar as ferramentas que foram capazes de inferir diagramas fizeram no com sucesso, realçando no diagrama os padrões encontrados. O catálogo de padrões para a ferramenta Fujaba era por via de um arquivo Java, sendo que seriam programados nessa linguagem, compilados e passados ao programa. Na ferramenta Ptidej os catálogos eram pré-definidos.

Na ferramenta desenvolvida a funcionalidade de inferência de padrões funciona como pretendido. A identificação é similar a outras ferramentas (realçando os elementos no diagrama), mas o formato de definição de padrões é mais simples

e acessível, mantendo ao mesmo tempo a flexibilidade fornecida pelo Prolog. Assim, esta ferramenta permite obter os resultados pretendidos, de forma similar a outras ferramentas.

PIM

A funcionalidade de abstracção de modelos não foi encontrada em nenhuma das ferramentas analisadas, pelo que não será possível obter uma conclusão por comparação.

Funcionalidades extra

As aplicações apresentadas possuíam várias funcionalidades extra não directamente relacionadas com as funcionalidades principais. Estas funcionalidades oferecidas contextualizam-se no objectivo de cada uma das ferramentas apresentadas.

A ferramenta ArgoUML permite utilizar as suas funcionalidades em outras linguagens. Tem ainda incluído um editor de texto. As funcionalidades oferecidas por esta ferramenta estão presentes no IDE onde a ferramenta desenvolvida é instalada. Assim, apesar da ferramenta desenvolvida funcionar apenas com a linguagem Java, as outras funcionalidades continuam disponíveis para outras linguagens.

A ferramenta Recplise é também um *plugin*, como tal tem as funcionalidades extra oferecidas pelo IDE Eclipse. As funcionalidades serão então similares às oferecidas pela ferramenta proposta. Já o Fujaba por outro lado permite estender as suas funcionalidades por instalação de *plugins*. Contudo a aplicação base é bastante limitada e não oferece funcionalidades extra.

O jGRASP oferece algumas funcionalidades sobre modelos como a geração de código fonte a partir de diagramas. Este tipo de funcionalidades não existe nem no IDE nem na ferramenta proposta, contudo esta funcionalidade no jGRASP é de relativa baixa qualidade e o código gerado não vai muito além da estrutura das classes e assinaturas de métodos.

A ferramenta Ptidej oferece um elevado número de funcionalidades extra da análise de alto nível em programas Java, sendo as únicas funcionalidades extra que oferece. Na ferramenta implementada a única funcionalidade deste género (análise de código) é apenas a inferência de padrões. No que diz respeito à análise de alto nível a ferramenta Ptidej tem um conjunto de funcionalidades que não são oferecidas por nenhuma outra ferramenta.

Por fim a ferramenta Visual Paradigm é uma ferramenta que centra as suas funcionalidades em edição de modelos. As funcionalidades que oferecem são desenvolvimento e edição de modelos, incluindo a geração de código. Oferece funcionalidades bastante desenvolvidas que não estão presentes em nenhuma das outras ferramentas. Existem contudo funcionalidades como a edição de código

que são bastante básicas e são oferecidas no IDE em que se integra a ferramenta proposta.

Ao desenvolver a ferramenta proposta sob a forma de *plugin* para o NetBeans foram conseguidas muitas funcionalidades extra de forma gratuita. O NetBeans é um IDE popular e poderoso que oferece muitas funcionalidades como edição de código fonte, identificação de erros, compilação e publicação de programas. Apesar de muitas das ferramentas analisadas também terem estas funcionalidades elas são de inferior qualidade (excepto nas ferramentas que são também *plugins*), uma vez que o IDE está especializado nessas funcionalidades. Assim, as funcionalidades extra são as que o IDE oferece e comparando com as funcionalidades oferecidas pelas outras ferramentas são, regra geral, de maior qualidade.

Depois desta análise e comparação de funcionalidades, pode-se concluir que as funcionalidades desenvolvidas cumprem os objectivos propostos, permitindo a integração de algumas funcionalidades que se encontram separadas nas ferramentas analisadas. Foi possível obter todo um ambiente de desenvolvimento e execução de aplicações juntamente com as funcionalidades desenvolvidas. Por um lado foi atingido o objectivo de desenvolver as funcionalidades que solucionavam os problemas propostos. Por outro lado existe um conjunto de funcionalidades extra que são fornecidas pelo IDE e que cobrem grande parte das funcionalidades extra apresentadas por outras ferramentas que são integradas de forma prática com a ferramenta em questão.

Extensão das funcionalidades

Devido ao modo como a ferramenta foi desenvolvida a extensão das suas funcionalidades é conseguida de maneira relativamente simples. A identificação de padrões como já foi referido, necessita apenas de um catálogo mais completo, bastando apenas criar novas regras que abranjam os padrões necessários. Para criar novos dados de população da base de conhecimento é necessário alterar o código fonte do módulo SAI.

O módulo de abstracção de diagramas pode também ser estendido, implementando diferentes transformações para os elementos. Esta alteração necessita da reescrita de parte do código fonte, editando uma classe específica. Apenas editando esta classe (que implementa interface de transformação) é possível redefinir as transformações. Contudo esta alteração ocorre numa só classe.

A linguagem suportada como já dito atrás é Java. Esta restrição baseia-se no facto de que o módulo de *parsing* é orientado à linguagem Java. A alteração da linguagem suportada pela ferramenta pode ter uma de duas consequências. Caso o metamodelo seja mantido (por ser adequado ou suficiente para a linguagem), apenas é necessário alterar ou substituir o módulo de *parsing*. Caso a linguagem não seja adequada ao metamodelo é necessário alterar não só o módulo de *parsing*, mas também o metamodelo.

O *plugin* em si e a representação gráfica são o componente hierarquicamente

mais alto da ferramenta. Isso faz com que este módulo possa ser facilmente alterado, substituído ou estendido sem qualquer alteração nos níveis hierárquicos inferiores. O refinamento dos elementos visuais, a adição de novas funcionalidades, a possibilidade de visualização por uma ferramenta externa ou até mesmo a exportação (para imagens por exemplo) é possível sem necessidade de nenhuma alteração extra.

O resultado do processo de implementação foi posto à prova com o teste na ferramenta JHotDraw e na Agenda ao longo deste capítulo. A utilização da ferramenta serviu para mostrar que os conceitos propostos para a resolução dos problemas encontrados são adequados. Foi possível demonstrar que as decisões propostas não são apenas ideias baseadas na teoria estudada, mas sim soluções válidas que permitem obter resultados reais. Os resultados finais obtidos do processo de implementação foram satisfatórios, não tendo havido necessidade de repensar funcionalidades após inicio do processo de implementação. O resultado final é no entanto uma prova de conceito, e várias coisas podem ser melhoradas e estendidas. Da mesma forma algumas funcionalidades ficam por implementar nesta versão, embora tenha já sido apresentado de que forma é possível fazer essa mesma extensão.

6.5 Resumo

Depois de implementada a ferramenta proposta, foi posta em prática para um caso de estudo específico, com duas vertentes. Ao longo deste capítulo foi criado todo um cenário que poderia corresponder a um cenário real, de modo a prever o que seria de esperar desta ferramenta. Foram então postas à prova as abordagens consideradas para cada um dos problemas. Para tal, foi seleccionada um programa (código fonte) bastante conhecido e utilizado na prática para ser objecto de estudo, e um de teste para mostrar detalhadamente como utilizar a ferramenta. Esses programas são o JHotDraw e a Agenda pelos motivos que foram apresentados. O código fonte destes programas foi então importado e utilizado na ferramenta MapIt para testar as funcionalidades.

Depois de importado o código fonte, a ferramenta MapIt foi posta em utilização para cada um dos programas, testando as diversas funcionalidades para provar que num contexto real seria possível obter os resultados esperados. Para cada um dos módulos foi feita uma descrição dos passos a tomar bem como do comportamento por parte da ferramenta. Demonstrou-se então que a ferramenta implementada soluciona os problemas propostos previamente com sucesso. O teste da ferramenta ocorreu em duas fases, primeiro mostrando o processo detalhadamente, de seguida mostrando o processo para um caso de maiores dimensões.

No final foram comentados os resultados obtidos pela ferramenta implementada, realçando detalhes que não são enquadrados nas outras secções. Foi feita também uma análise crítica do resultado do processo de implementação,

comparando as funcionalidades com outras ferramentas ao mesmo tempo que é apresentado de que forma é possível estender as funcionalidades implementadas. Desta forma fica concluído o capítulo do caso de estudo. Foi posta em prática a ferramenta implementada como prova de conceito para as decisões tomadas atrás para os problemas indicados.

Capítulo 7

Conclusões

Depois de terminado todo o processo de estudo, decisão e implementação da ferramenta bem como testes, é altura de fazer um balanço do mesmo. Para tal, neste capítulo será feito um resumo de todo o trabalho elaborado bem como as decisões mais importantes. No final serão apresentados algumas pistas para trabalho a realizar posteriormente.

7.1 Introdução

Da evolução natural das tecnologias resultou o aumento de complexidade que existe actualmente nos processos de desenvolvimento. Desta evolução surge a necessidade de uma mudança de paradigma, na forma como o processo de desenvolvimento de software ocorre. Para tal, a MDA surge como uma mudança de paradigma e aos poucos a sua importância tem vindo a crescer. A importância deste processo fez com que surgissem diversos estudos que resultaram na implementação de diversas ferramentas. Apesar da sua relevância o seu crescimento não tem sido tão grande quanto a sua importância. Desta forma concebeu-se uma ferramenta que será mais um contributo para o suporte do processo MDA por combinação de várias funcionalidades.

O objectivo principal deste trabalho consistiu na criação de uma nova ferramenta, disponibilizada sob a forma de *plugin* que tende a solucionar os quatro problemas apresentados previamente. Esses objectivos não são mais que a resolução de quatro problemas fundamentais:

- Análise e extracção de informação contida em código fonte Java;
- Inferência e representação de diagramas PSM, sob a forma de diagramas de classes UML;
- Inferência de padrões de concepção nos dados analisados;
- Abstracção dos diagramas PSM a PIM.

7.2 Avaliação do trabalho realizado

Neste documento foi apresentada uma ferramenta de suporte ao processo MDA. Esta proposta tem em vista permitir que sejam feitos os passos inversos ao que o processo MDA propõe. Desta forma é proposto o processo de gerar informação de alto nível partindo da informação mais específica disponível que é o código fonte. O objectivo foi ter uma ferramenta que permitisse automatizar todo o processo MDA de forma inversa, ao mesmo tempo que fornece a possibilidade do utilizador ter três pontos de vista diferentes sobre o seu software: modelos específicos, modelos independentes da plataforma e por fim organização do projecto enquanto conjunto de padrões de concepção.

A utilização desta ferramenta num contexto de desenvolvimento auxilia um utilizador, sobretudo quando este tiver necessidade de ter uma abstracção do seu software permitindo:

- Analisar o código fonte de um projecto de software Java, produzindo uma representação intermédia dessa informação analisada. Desta forma o código fonte é analisado e representado internamente de uma forma abstracta e mais flexível, que permitirá efectuar operações sobre ele.

- Representar de forma visual essa informação intermédia inferida com base no código fonte. Será assim então possível criar uma representação visual, nomeadamente um PSM que mostrará essa informação ao utilizador. Este PSM será concretizado na forma de um diagrama de classes UML.

- Inferir padrões de concepção nessa representação. Esta funcionalidade permite obter uma análise de mais alto nível ao software, uma vez que não está a fazer uma análise concreta ao nível do código fonte, mas sim uma análise ao projecto em global. É uma análise que tem em conta aspectos estruturais e funcionais, não é uma análise ao nível do código fonte que pouca informação abstracta poderá fornecer.

- Abstrair diagramas concretos a diagramas mais abstractos. Os diagramas PSM são diagramas que abstraem alguns detalhes de implementação, contudo são diagramas específicos de uma plataforma. Apesar disso, a informação que contém é ainda específica do contexto onde nos encontramos. O que é proposto é então uma forma de abstrair estes modelos a outros modelos de mais alto nível, neste caso modelos independentes da plataforma: PIM. Esta é outra forma de disponibilizar informação de alto nível ao utilizador.

- Disponibilizar estas funcionalidades num ambiente de produção, onde o utilizador possa facilmente começar a utilizar a ferramenta. Esta funcionalidade é importante para permitir que o utilizador possa tirar partido destas funcionalidades sem ter de alterar o ambiente onde desenvolve o software. Para tal é disponibilizado sob a forma de *plugin* para um IDE.

Estas são as principais contribuições da ferramenta proposta para o processo MDA. O enquadramento da ferramenta é possível em cenários distintos e poderão ser várias fases do processo de desenvolvimento. No suporte ao processo de desenvolvimento por modelos esta ferramenta pode ser utilizada em dois principais contextos. O primeiro corresponde à análise de sistemas antigos num contexto de manutenção de código já existente, sem documentação útil. O segundo contexto refere-se a ferramentas que necessitem de ser migradas e integrados em sistemas desenvolvidos de forma orientada a modelos. Já no processo de desenvolvimento tradicional também esta ferramenta é uma mais valia uma vez que permite auxiliar este processo. Permite ter diferentes visões do software que está a ser desenvolvido, mesmo que este seja desenvolvido de forma tradicional. Permite também a análise da qualidade do processo de desenvolvimento por análise dos padrões contidos nesse software.

É importante ainda comparar o resultado obtido não só face ao proposto, mas também face a outras ferramentas existentes. Fazendo esta comparação obtém-se uma análise sobre a parte mais prática desta ferramenta.

- A integração com o IDE destino foi um detalhe relevante. Outras ferramentas não conseguiram ter uma boa integração, sendo difíceis de utilizar ou demonstrando que não houve um esforço por parte dos autores. Na ferramenta desenvolvida esta integração foi feita de modo a disponibilizar uma utilização intuitiva. Considera-se que a sua integração foi bem conseguida, funcionando de forma similar a outras funcionalidades do IDE.

- A qualidade dos modelos UML produzidos por algumas ferramentas foi apontado como sendo de baixa qualidade. Em muitos casos faltavam informações essenciais como métodos e atributos, em outras relações entre elementos. Na ferramenta desenvolvida as representações são superiores às de muitas ferramentas e corrigem os problemas apontados nelas. Nos modelos obtidos é inferida muita da informação contida no código e correctamente representada em UML. Relativamente à representação visual dos mesmos, a qualidade visual não está ao nível de outras ferramentas (como da ferramenta Visual Paradigm, por exemplo). O facto dos modelos gerados serem manipuláveis é algo importante que não é disponibilizado por todas as ferramentas.

- A abordagem escolhida para a funcionalidade de abstracção de modelos permite que este processo seja parametrizável. Estas transformações permitem diferentes níveis de abstracção e diferentes resultados, o que se torna bastante útil e faz com que esta funcionalidade permita obter resultados práticos.

- A inferência de padrões destaca-se de outras ferramentas, por utilizar uma outra ferramenta e permitir a especificação fácil de padrões. A sua identificação permite obter uma grande amplitude de resultados dependendo

da especificidade do catálogo. Em outras ferramentas os catálogos são estáticos, em outras não são parametrizáveis. A inferência de padrões é contudo limitada à informação obtida na fase anterior, enquanto que outras ferramentas permitem a análise de outro tipo de informações, como micro-arquitecturas.

7.3 Trabalho futuro

Apenas no final da implementação da ferramenta proposta foi possível obter uma visão geral sobre as funcionalidades implementadas e qual o seu grau de desenvolvimento. Foi possível então concluir que existe um conjunto de funcionalidades que poderão ser estendidas no futuro, melhorando a ferramenta desenvolvida ou adicionando novas funcionalidades importantes no contexto MDA.

7.3.1 Extensão a outras linguagens

Como foi referido neste documento, a linguagem que a ferramenta é capaz de reconhecer é Java. Isto é a partida restritivo quanto pensamos num panorama mais geral e concluímos que existe um grande conjunto de outras linguagens de programação que poderiam tirar também partido desta ferramenta. As linguagens orientadas a objectos seriam as primeiras linguagens a ter em conta, uma vez que se enquadram bem no metamodelo desenvolvido. Outras linguagens não orientadas a objectos seriam as seguintes a ser abordadas. Esta extensão de funcionalidades iria naturalmente alargar as potencialidades da ferramenta.

7.3.2 Extensão do catálogo de padrões

O catálogo de padrões implementado serve apenas como prova de conceito. Os padrões que contém são apenas os necessários para demonstrar o funcionamento da ferramenta face a uma situação real, demonstrando que é uma forma viável de o fazer. Um possível ponto de trabalho seria a extensão do catálogo de padrões de forma a conter uma maior variedade de padrões, num catálogo mais completo. A análise de outros padrões que não os padrões de concepção identificados por Erich Gamma [25] para posterior adição a catálogos para a ferramenta é outro ponto possível a ter em conta como trabalho futuro.

7.3.3 Extensão da ferramenta a *round-trip*

Na sua forma actual a ferramenta implementada cumpre os objectivos propostos no âmbito da MDA. Esta ferramenta foi proposta no âmbito da engenharia reversa e como tal permite apenas efectuar o processo MDA inverso. Uma ideia que surge naturalmente é a extensão da ferramenta a *round-trip*, para que permita simultaneamente analisar código fonte gerando diagramas, e da mesma

forma gerar código fonte com base em diagramas. O metamodelo utilizado está já preparado para suportar a implementação desta funcionalidade.

Neste mesmo âmbito existe também a possibilidade de serem necessários outros tipos de diagramas para permitir gerar código fonte com base em modelos. Foi visto ao longo deste documento que existe uma necessidade de adicionar informação (mais em concreto informação comportamental) para que seja gerado código executável.

7.3.4 Implementação do *plugin* em outros suportes

A definição à partida da plataforma destino pode ser algo limitadora para o utilizador final. De momento, apenas utilizadores com o IDE NetBeans poderão utilizar esta ferramenta. Esta limitação é ao mesmo tempo um ponto de extensão de funcionalidades. Uma proposta de trabalho futuro será a migração desta ferramenta para um *plugin* de outros IDE, como Eclipse, Visual Studio entre outras. A ferramenta poderia até ser migrada para um programa *standalone*, perdendo no entanto a vantagem de ser *plugin* e beneficiar das funcionalidades do mesmo, mas ganhando vantagens de portabilidade, por exemplo.

No mesmo contexto e abrangendo a portabilidade, a alteração do motor de inferência é algo que fica também proposto como trabalho futuro. Apesar da ferramenta implementada utilizar o gprolog e ele ser grátis e de livre utilização, necessita no entanto que na plataforma destino onde se pretenda utilizar a ferramenta este esteja instalado. A alteração do módulo de Prolog (para um motor de Prolog embutido na ferramenta) é também algo em aberto como possibilidade de trabalho futuro.

7.3.5 Melhoramento da representação visual

A representação visual é parte da interacção com o utilizador. A interface gráfica é sempre algo que é trabalhoso e exige um esforço para que seja bem conseguida. Na ferramenta implementada não é excepção. Os diagramas gerados poderiam permitir um maior número de operações, como por exemplo guardar e carregar estes diagramas. A exportação destes modelos para imagens (ou outros formatos) tornaria estas representações independentes da ferramenta, permitindo aceder a estes diagramas sem a ferramenta instalada e até mesmo imprimi-los (por exemplo).

Bibliografia

[1] Christopher Alexander, Sara Ishikawa, Murray Silverstein, Max Jacobson, Ingrid Fiksdahl-King, and Shlomo Angel. *A Pattern Language*. Oxford University Press, New York, 1977.

[2] Larry Barowski and James Cross. Extraction and use of class dependency information for java. In *Proceedings of the Ninth Working Conference on Reverse Engineering (WCRE'02)*, pages 309–, Washington, DC, USA, 2002. IEEE Computer Society.

[3] Alex Blewitt, Alan Bundy, and Ian Stark. Automatic verification of design patterns in java. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, ASE '05, pages 224–232, New York, NY, USA, 2005. ACM.

[4] Heena Chandigarh and Ranjna Banur. A comparative study of uml tools. In *Proceedings of the International Conference on Advances in Computing and Artificial Intelligence*, ACAI '11, pages 1–4, New York, NY, USA, 2011. ACM.

[5] Software Freedom Conservancy. ArgoUML website, `http://argouml.tigris.org/`, July 2011.

[6] James Corbett, Matthew Dwyer, John Hatcliff, Shwan Laubach, Corina Pasareanu, Robby, and Hongjun Zheng. Bandera: extracting finite-state models from java source code. In *Proceedings of the 22nd international conference on Software engineering*, ICSE '00, pages 439–448, New York, NY, USA, 2000. ACM.

[7] Borland Software Corporation. Together website, `http://www.borland.com/us/products/together/`, July 2011.

[8] Microsoft Corporation. ASP website, `http://www.asp.net/`, July 2011.

[9] Microsoft Corporation. IIS website, `http://www.iis.net/`, July 2011.

[10] Microsoft Corporation. SQL Server website, `http://www.microsoft.com/sqlserver/`, July 2011.

[11] Oracle Corporation. Glassfish website, `http://glassfish.java.net/`, July 2011.

[12] Oracle Corporation. Java Server Faces website, `http://javaserverfaces.java.net/`, July 2011.

[13] Oracle Corporation. MySQL website, `http://www.mysql.com/`, July 2011.

[14] Oracle Corporation. Oracle website, `http://www.oracle.com/`, July 2011.

[15] Pierre Deransart. *Prolog: the standard*. Springer, Berlin, 1996.

[16] Jing Dong, Yajing Zhao, and Tu Peng. Architecture and design pattern discovery techniques - a review. *Proceedings of International Conference on Software Engineering Research and Practice (SERP)*, 1(1):621–627, 2008.

[17] Félix Agustín Castro Espinoza, Gustavo Núñez Esquer, and Joel Suárez Cansino. Automatic design patterns identification of c++ programs. In *Proceedings of the First EurAsian Conference on Information and Communication Technology*, EurAsia-ICT '02, pages 816–823, London, UK, 2002. Springer-Verlag.

[18] Liliana Favre. Formalizing mda-based reverse engineering processes. In *Proceedings of the 2008 Sixth International Conference on Software Engineering Research, Management and Applications*, pages 153–160, Washington, DC, USA, 2008. IEEE Computer Society.

[19] The Apache Software Foundation. Apace website, `http://www.apache.org/`, July 2011.

[20] The Apache Software Foundation. Struts website, `http://struts.apache.org/`, July 2011.

[21] Martion Fowler and Kendall Scott. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley, Boston, USA, 2004.

[22] Alfonso Fuggetta. A classification of case technology. *Computer*, 26:25–38, 1993.

[23] University of Paderborn Fujaba Tool Suite Developer Team. Fujaba website, `http://www.fujaba.de/`, July 2011.

[24] Erich Gamma and Thomas Eggenschwiler. JHotDraw website, `http://www.jhotdraw.org/`, July 2011.

[25] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Mass., 1995.

[26] Júlio Vilmar Gesser. JavaParser website, `http://code.google.com/p/javaparser/`, July 2011.

[27] Martin Gogolla and Ralf Kollmann. Re-documentation of java with uml class diagrams. In *Proc. 7th Reengineering Forum, Reengineering Week 2000*, pages 41–48, 2000.

[28] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Sun microsystems, Santa Clara, CA, USA, 2005.

[29] PostgreSQL Global Development Group. Postgres website, `http://www.postgresql.org/`, July 2011.

[30] The PHP Group. PHP website, `http://www.php.net/`, July 2011.

[31] Yann-Gaël Guéhéneuc. *Un cadre pour la tra caabilité des motifs de conception*. PhD thesis, Université de Nantes, 2003.

[32] Yann-Gaël Guéhéneuc. Abstract and precise recovery of uml diagram constituents. In *Proceedings of the 20th IEEE International Conference on Software Maintenance*, pages 523–, Washington, DC, USA, 2004. IEEE Computer Society.

[33] Yann-Gaël Guéhéneuc. A reverse engineering tool for precise class diagrams. In *Proceedings of the 2004 conference of the Centre for Advanced Studies on Collaborative research*, CASCON '04, pages 28–41. IBM Press, 2004.

[34] Yann-Gaël Guéhéneuc. A systematic study of uml class diagram constituents for their abstract and precise recovery. In *Proceedings of the 11th Asia-Pacific Software Engineering Conference*, APSEC '04, pages 265–274, Washington, DC, USA, 2004. IEEE Computer Society.

[35] Yann-Gaël Guéhéneuc. Ptidej website, `http://www.ptidej.net/`, July 2011.

[36] Yann-Gaël Guéhéneuc and Giuliano Antoniol. Demima: A multilayered approach for design pattern identification. *IEEE Trans. Softw. Eng.*, 34:667–684, September 2008.

[37] David Heinemeier Hansson. Ruby website, `http://rubyonrails.org/`, July 2011.

[38] T. Dean Hendrix, James H. Cross, II, and Larry A. Barowski. An extensible framework for providing dynamic data structure visualizations in a lightweight ide. *SIGCSE Bull.*, 36:387–391, March 2004.

[39] IBM. Rose website, `http://www-01.ibm.com/software/awdtools/developer/rose/`, July 2011.

[40] Dr. James H Cross II. jGRASP website, `http://www.jgrasp.org/`, July 2011.

[41] Red Hat Inc. Hibernate website, `http://www.jboss.com/products/hibernate/`, July 2011.

[42] Red Hat Inc. Jboss website, `http://www.jboss.com/`, July 2011.

[43] JetBrains. Idea website, `http://www.jetbrains.com/idea/`, July 2011.

[44] Kanit Jinto and Yachai Limpiyakorn. Java code reviewer for verifying object-oriented design in class diagrams. In *Information Management and Engineering (ICIME), 2010 The 2nd IEEE International Conference on*, ICIME '10, pages 471 – 475, Piscataway, NJ, USA, 2010. IEEE.

[45] Tetsuro Katayama and Yusuke Yabuya. Proposal of a method to support testing for java programs with uml. In *Proceedings of the 12th Asia-Pacific Software Engineering Conference*, pages 533–540, Washington, DC, USA, 2005. IEEE Computer Society.

[46] Rick Kazman and S. Jeromy Carrière. Playing detective: Reconstructing software architecture from available evidence. *Automated Software Engg.*, 6:107–138, April 1999.

[47] Brian W. Kernighan and Dennis M. Ritche. *The C programming language*. Prentice Hall, Upper Saddle River, NJ, USA, 1978.

[48] Martin Keschenau. Reverse engineering of uml specifications from java programs. In *Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, OOPSLA '04, pages 326–327, New York, NY, USA, 2004. ACM.

[49] Anneke Kleppe, Jos Warmer, and Wim Bast. *MDA Explained - The Model Driven Architecture: Practice and Promise*. Addison-Wesley, Boston, MA, USA, 2003.

[50] R. Kollman, P. Selonen, E. Stroulia, T. Systä, and A. Zundorf. A study on the current state of the art in tool-supported uml-based static reverse engineering. In *Proceedings of the Ninth Working Conference on Reverse Engineering (WCRE'02)*, pages 22–, Washington, DC, USA, 2002. IEEE Computer Society.

[51] François Lagarde, Huáscar Espinoza, François Terrier, and Sébastien Gérard. Improving uml profile design practices by leveraging conceptual domain models. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, ASE '07, pages 445–448, New York, NY, USA, 2007. ACM.

[52] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Sun Microsystems, Inc, Santa Clara, California, USA, 1999.

[53] Andrea De Lucia, Vincenzo Deufemia, Carmine Gravino, and Michele Risi. Design pattern recovery through visual language parsing and source code analysis. *J. Syst. Softw.*, 82:1177–1193, July 2009.

[54] Atif Mashkoor. Investigating model driven architecture. Technical report, Department of Computing Science Umea University, 2004.

[55] Sarah Matzko, Peter J. Clarke, Tanton H. Gibbs, Brian A. Malloy, James F. Power, and Rosemary Monahan. Reveal: a tool to reverse engineer class diagrams. In *Proceedings of the Fortieth International Conference on Tools Pacific: Objects for internet, mobile and embedded applications*, CRPIT '02, pages 13–21, Darlinghurst, Australia, Australia, 2002. Australian Computer Society, Inc.

[56] Stephen Mellor and Marc Balcer. *Executable UML: A Foundation for Model-Driven Architecture*. Addison-Wesley, Boston, MA, USA, 2002.

[57] Kim Mens, Tom Mens, and Michel Wermelinger. Maintaining software through intentional source-code views. In *Proceedings of the 14th international conference on Software engineering and knowledge engineering*, SEKE '02, pages 289–296, New York, NY, USA, 2002. ACM.

[58] Thomas O. Meservy and Kurt D. Fenstermacher. Transforming software development: An mda road map. *IEEE Computer Society*, 38:52–58, September 2005.

[59] Joaquin Miller and Jishnu Mukerji. *MDA Guide Version 1.0.1*. OMG, Needham, MA, USA, 2003.

[60] Blake Mizerany. Sinatra website, `http://www.sinatrarb.com/`, July 2011.

[61] Naouel Moha and Yann-Gaël Guéhéneuc. Ptidej and décor: identification of design patterns and design defects. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, OOPSLA '07, pages 868–869, New York, NY, USA, 2007. ACM.

[62] H. A. Müller and K. Klashinsky. Rigi-a system for programming-in-the-large. In *Proceedings of the 10th international conference on Software engineering*, ICSE '88, pages 80–86, Los Alamitos, CA, USA, 1988. IEEE Computer Society Press.

[63] Object Management Group, Needham, MA, USA. *Common Warehouse Metamodel (CWM) Specification*, 2003.

[64] Object Management Group, Needham, MA, USA. *Meta Object Facility (MOF) 2.0 Core Specification*, 2004.

[65] Object Management Group, Needham, MA, USA. *MOF 2.0/XMI Mapping, Version 2.1.1*, 2007.

[66] Object Management Group, Needham, MA, USA. *Object Constraint Language, Version 2.2*, 2010.

[67] Object Management Group, Needham, MA, USA. *Common Object Request Broker Architecture (CORBA) Specification, Version 3.1.1*, 2011.

[68] Visual Paradigm. Visual Paradigm website, `http://www.visual-paradigm.com/`, July 2011.

[69] Oscar Pastor and Juan Carlos Molina. *Model-Driven Architecture in Practice*. Springer, Berlin, 2007.

[70] Waldemar Pires, Franklin Ramalho, Anderson Ledo, and Dalton Serey. Checking uml design patterns in java implementations. In *Software Components, Architectures and Reuse (SBCARS), 2010 Fourth Brazilian Symposium on*, SBCARS '10, pages 120 – 129, Piscataway, NJ, USA, 2010. IEEE.

[71] John D. Poole. Model-driven architecture: Vision, standards and emerging technologies. In *In In ECOOP 2001, Workshop on Metamodeling and Adaptive Object Models*, 2001.

[72] Igor Sacevski and Jadranka Veseli. Introduction to model driven architecture. Seminar Paper, University of Salzburg, 2007.

[73] Jochen Seemann and Jürgen Wolff von Gudenberg. Pattern-based design recovery of java software. In *Proceedings of the 6th ACM SIGSOFT international symposium on Foundations of software engineering*, SIGSOFT '98/FSE-6, pages 10–16, New York, NY, USA, 1998. ACM.

[74] Shane Sendall and Wojtek Kozaczynski. Model transformation: The heart and soul of model-driven software development. *IEEE Softw.*, 20:42–45, September 2003.

[75] Nija Shi and Ronald A. Olsson. Reverse engineering of design patterns from java source code. In *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, pages 123–134, Washington, DC, USA, 2006. IEEE Computer Society.

[76] Ioannis Stamelos. Software project management anti-patterns. *J. Syst. Softw.*, 83:52–59, January 2010.

[77] Bjarne Stroustrup. *The C++ programming language*. Addison-Wesley, Murray Hill, NJ, USA, 1997.

[78] Sun microsystems, Palo Alto, CA, USA. *IA-32 Assembly Language Reference Manual*, 2000.

[79] Sun microsystems, Santa Clara, CA, USA. *Java Platform, Enterprise Edition (Java EE) Specification, v5*, 2006.

[80] Tarja Systä. *Static and Dynamic Reverse Engineering Techniques for Java Software Systems*. University of Tampere, Tampere, Department of Computer and Information Sciences, Finland, 2000.

[81] Tarja Systä, Kai Koskimies, and Hausi Müller. Shimba - an environment for reverse engineering java software systems. *Softw. Pract. Exper.*, 31:371–394, April 2001.

[82] Mircea Trifu. *Architecture-Aware, Adaptive Clustering of Object-Oriented Systems*. PhD thesis, Forschungszentrum Informatik Karlsruhe, 2003.

[83] Nikolaos Tsantalis, Alexander Chatzigeorgiou, George Stephanides, and Spyros T. Halkidis. Design pattern detection using similarity scoring. *IEEE Trans. Softw. Eng.*, 32:896–909, November 2006.

[84] Lionel Vigier and Andrey Sadovykh. Psm-to-Pim, a pragmatic way. In *ECMDA 2008 - Modernization Workshop*, 2008.

[85] Markus von Detten, Matthias Meyer, and Dietrich Travkin. Reverse engineering with the reclipse tool suite. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ICSE '10, pages 299–300, New York, NY, USA, 2010. ACM.

[86] Allison Leah Waingold. *Automated Extraction of Abstract Object Models*. PhD thesis, Massachusetts Institute of Technology, 2001.

[87] Roel Wuyts. Declarative reasoning about the structure of object-oriented systems. In *In Proceedings of the TOOLS USA '98 Conference*, pages 112–124. IEEE Computer Society Press, 1998.

Anexos

Anexo A

Padrões de concepção

A.1 Catálogo de padrões

O chamado "catálogo de padrões", consiste num livro escrito por Erich Gamma *et al.* denominado *Design Patterns - Elements of Reusable Object-Oriented Software* [25]. Neste livro estão catalogados, sob a forma de diagramas de classes resoluções para problemas bem conhecidos. Estes problemas são encontrados frequentemente quando programamos em linguagens orientadas a objectos.

Exemplo destes problemas é por exemplo a necessidade de adaptar uma classe a um interface que não corresponde com o seu. Neste caso existe um padrão chamado Adapter que descreve de que forma se soluciona este problema. A solução é apresentada como um diagrama de classes, juntamente com o contexto a que se adequa. Neste catálogo surge também a implementação na forma de exemplo.

Estes padrões são distinguidos em três tipos dependendo do seu objectivo final. As classes em que são agrupados são padrões de criação, estruturais e comportamentais.

Faz-se de seguida um resumo destes padrões para perceber melhor o seu enquadramento neste documento.

A.2 Padrões de concepção

De seguida vai ser feita uma análise aos padrões propostos. Será então analisado o propósito de cada padrão, a sua estrutura e comportamento. Por fim feito um pequeno resumo sobre a possibilidade de inferência bem como a metodologia proposta para o conseguir.

A.2.1 Abstract Factory

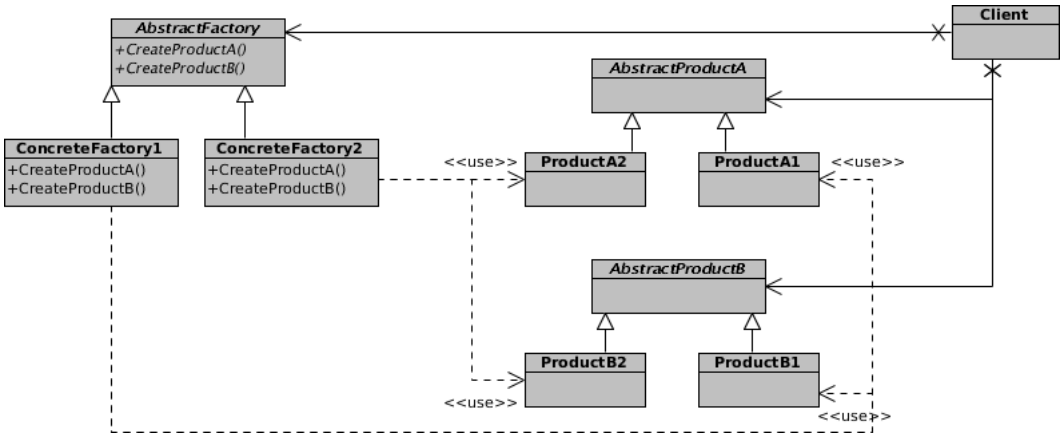

Figura A.1: O padrão Abstract Factory, adaptado de [25].

Este padrão (Figura A.1), define uma fábrica de objectos abstracta. Com a utilização deste padrão, através de uma classe comum, podem ser criados produtos concretos para um contexto específico.

Este padrão poderia ser reconhecido na pesquisa de classes abstractas (`AbstractFactory`) que tenham subclasses (`ConcreteFactory1`) que por sua vez contenham métodos comuns (`CreateProductA()`, por exemplo). Esses métodos teriam de ser invocados por subclasses (`ProductA1`) de outras classes abstractas (`AbstractProductA`). Este padrão sofre o risco de falsos positivos numa primeira análise.

Uma análise estrutural por comparação por uma especificação prévia do padrão poderia ser capaz de reconhecer com sucesso este padrão.

A.2.2 Adapter

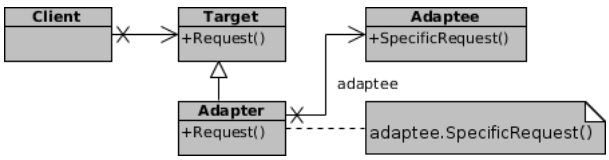

Figura A.2: O padrão Adapter, adaptado de [25].

Este (Figura A.2) é o padrão que descreve como adaptar uma classe não

adaptada a um contexto específico. Também pode ser usado de forma a criar classes que não contém interfaces compatíveis.

Este padrão seria detectado com a pesquisa de classes (`Target`) que contivessem uma subclasse (`Adapter`) com um ou mais métodos (`Request()`) iguais. A subclasse deveria incluir uma outra classe por composição (`Adaptee`). No método da subclasse (`Request`) deveria haver pelo menos uma chamada a um método (`SpecificRequest()`) da classe incorporada (`Adaptee`).

Uma análise estrutural deveria identificar este padrão. Seria necessária uma análise nos métodos das classes identificadas para perceber se estamos mesmo perante um padrão Adapter.

A.2.3 Bridge

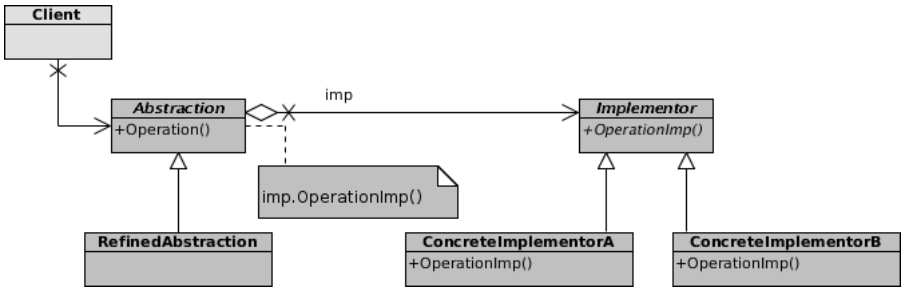

Figura A.3: O padrão Bridge, adaptado de [25].

Com este padrão (Figura A.3) é possível definir várias implementações para uma mesma abstracção. Define a possibilidade de fazer uma associação não permanente entre uma abstracção e uma associação específica. É especialmente importante para quando são previstas alterações na implementação sem que o cliente se aperceba disso.

A pesquisa deste padrão poderia ser feita por pesquisa de classes abstractas (`Abstraction`) com várias implementações abstractas (`Implementator`), que por sua vez possuam várias implementações (`ConcreteImplementatorA`). A classe abstracta (`Abstraction`) deveria então invocar um método (`OperationImp()`) na instanciação da implementação (`imp`).

A identificação deste padrão necessitaria de uma comparação estrutural por um lado. Depois de identificada a estrutura seria necessário verificar se existe uma cadeia de invocação de um método desde a abstracção até uma implementação concreta.

A.2.4 Builder

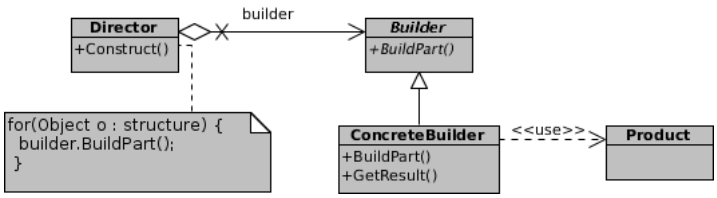

Figura A.4: O padrão Builder, adaptado de [25].

Quando se pretende ter diversas representações para um objecto que vai ser construído, este padrão (Figura A.4) é a solução. Permite criar independência entre o processo de criação e a implementação para um caso específico.

Seria então necessário pesquisar por uma classe (`Director`) que invocasse métodos (`BuildPart()`) de outra classe abstracta (`Builder`). Esses métodos deveriam então ser implementados por uma classe concreta (`ConcreteBuilder`).

A.2.5 Chain of Responsability

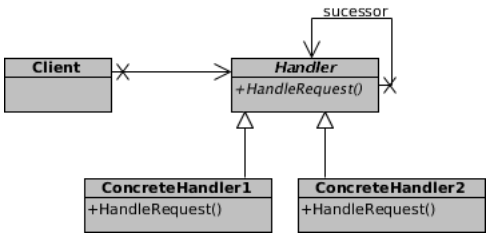

Figura A.5: O padrão Chain of Responsability, adaptado de [25].

Este padrão descreve, como o nome indica, uma cadeia de responsabilidade (Figura A.5). Soluciona a necessidade de ter vários objectos que podem atender um pedido. É também a solução para a necessidade de não saber à partida quem vai atender um pedido numa cadeia, ou até mesmo se quem vai atender o pedido for especificado dinamicamente.

A pesquisa deste padrão poderia ser feita por comparação estrutural. Nessa pesquisa seriam localizadas classes (`Handler`) que contivessem classes do mesmo tipo por composição (`successor`). A classe (`Handler`) deveria chamar um método (`HandleRequest`) que ela própria contenha, na instância da classe que possui. Estas classes poderão ter implementações concretas (`ConcreteHandler1`)

Uma pesquisa estrutural iria encontrar a hierarquia necessária. Uma pesquisa nos métodos iria eliminar possíveis falsos positivos, e deveria permitir a identificação com sucesso deste padrão.

A.2.6 Command

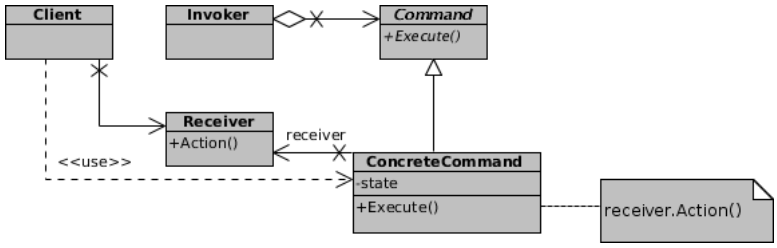

Figura A.6: O padrão Command, adaptado de [25].

Para permitir a parametrização de um cliente, com o pedido encapsulado como objecto deve utilizar-se este padrão (Figura A.6). Este comportamento permitirá definir por exemplo um `callback`, suportar funções de `logs` e acções reversíveis.

Este padrão seria o mais complicado de inferir pela sua natureza mais complexa. O cliente criaria um "comando" concreto (`ConcreteCommand`), que cumprirá a interface definida pelo "comando" abstracto (`Command`). Um outro elemento (`Receiver`) será o responsável por invocar o método específico (`Execute()`) no "comando". O "comando" da classe abstracta (`Command`) será invocada por um invocador (`Invoker`) externo.

A inferência deste padrão necessitaria de uma análise mais intensa a nível de comportamento. A detecção deste comportamento seria maioritariamente baseado em invocação de métodos, o que poderá tornar este padrão mais complicado de inferir.

A.2.7 Composite

A necessidade de organizar objectos em estruturas por composição de objectos do mesmo tipo, levou a necessidade de desenvolver este padrão (Figura A.7). A utilização deste padrão permite por um lado representar hierarquias, e por outro lado ignorar a diferença entre um objecto singular ou composição de padrões.

Uma classe (`Component`), que contenha uma outra classe (`Composite`) que herde dela, que por sua vez contenha uma lista (`Children`) de elementos da

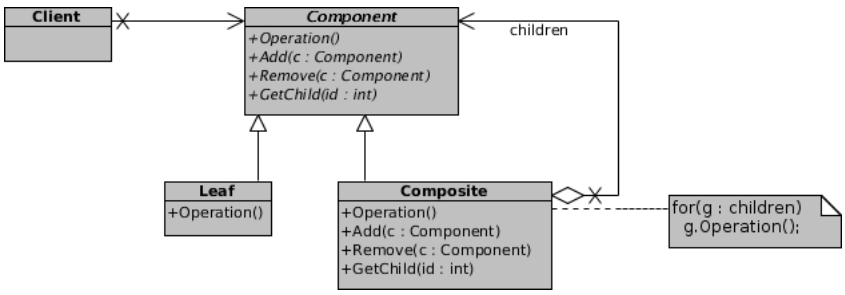

Figura A.7: O padrão Composite, adaptado de [25].

classe superior (`Component`) seria identificada como um padrão Composite. Na classe (`Composite`) que herda da principal (`Component`) deverá haver métodos (`Operation`) que invoquem métodos na classe superior (`Component`).

Este padrão seria em princípio relativamente simples de inferir devido a sua simplicidade. A análise mais importante seria estrutural. A análise dos métodos seria útil para confirmar se de facto se trata de um padrão Composite.

A.2.8 Decorator

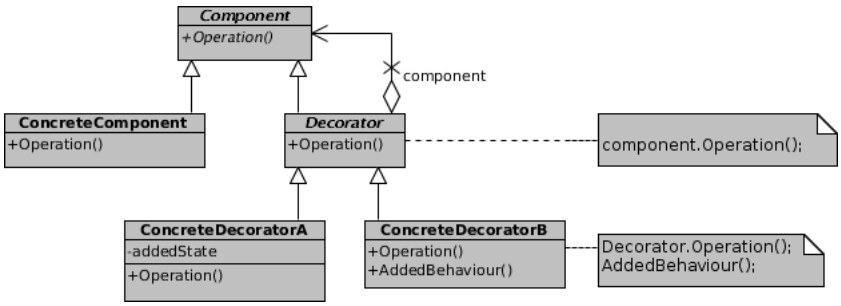

Figura A.8: O padrão Decorator, adaptado de [25], adaptado de [25].

Para quando existe a necessidade de adicionar (ou remover) responsabilidades dinamicamente a objectos, ou de fazer extensão de funcionalidades de forma flexível, temos o padrão Decorator (Figura A.8).

Deverá existir um interface (`Component`) que define os objectos que podem ter as responsabilidades, e um objecto (`ConcreteComponent`) ao qual podem ser adicionados as responsabilidades. Existindo também um componente (`Decorator`) em conformidade com o interface (`Component`) com subclasses (`ConcreteDecorator`) que adicionam as classes ao componente estamos perante a presença do padrão Decorator.

Este padrão poderia ser inferido com análise sobretudo estrutural. A análise dos métodos invocados também iria ajudar na confirmação deste padrão.

A.2.9 Facade

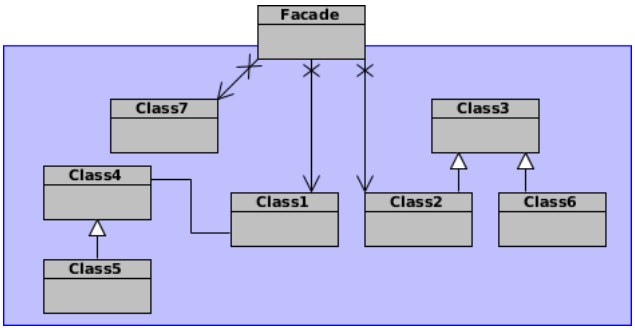

Figura A.9: O padrão Facade, adaptado de [25].

O padrão Facade (Figura A.9) fornece a capacidade de fazer uma abstracção de vários componentes, apresentando uma interface de acesso única. Permite por um lado abstrair, por outro alterar a implementação dos elementos que abstrai. Também pode servir de separador de camadas do sistema.

Este é um padrão algo subjectivo. Pode levar a muitos falsos positivos por um lado (no limite de consideramos uma hierarquia entre duas classes), ou ser muito difícil de detectar, caso se seja muito restritivo. A análise básica seria na pesquisa de classes (`Facade`) que contivessem um subconjunto de classes a que apenas elas invoquem métodos.

A pesquisa seria feita com base numa análise estrutural (na pesquisa da classe `Facade`), e com análise das invocações aos métodos das classes que o Facade abstrai.

A.2.10 Factory Method

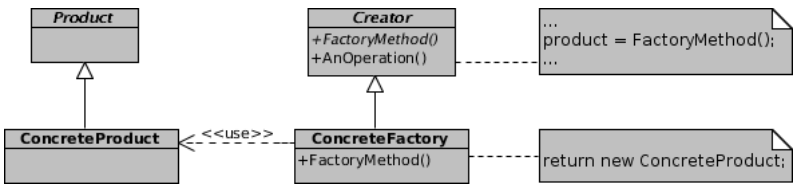

Figura A.10: O padrão Factory Method, adaptado de [25].

A utilização deste padrão (Figura A.10) é justificável quando houver a necessidade de ter uma interface para criar objectos, delegando às subclasses a decisão de qual objecto instanciar.

A sua inferência seria feita com uma pesquisa de um interface de criação de produtos (`Product`), que contenha uma implementação de um produto (`ConcreteProduct`). Também teria de haver também uma "fábrica" (`Cretor`) e uma implementação dessa "fábrica" (`ConcreteCreator`). A "fábrica" (`Creator`) irá instanciar um produto chamando o método de criação (`FactoryMethod`) da "fábrica" concreta.

Mais uma vez surge um padrão que necessitaria sobretudo de análise dos métodos, neste caso para perceber se a cadeia de invocações representa o comportamento de uma "fábrica" abstracta.

A.2.11 Flyweight

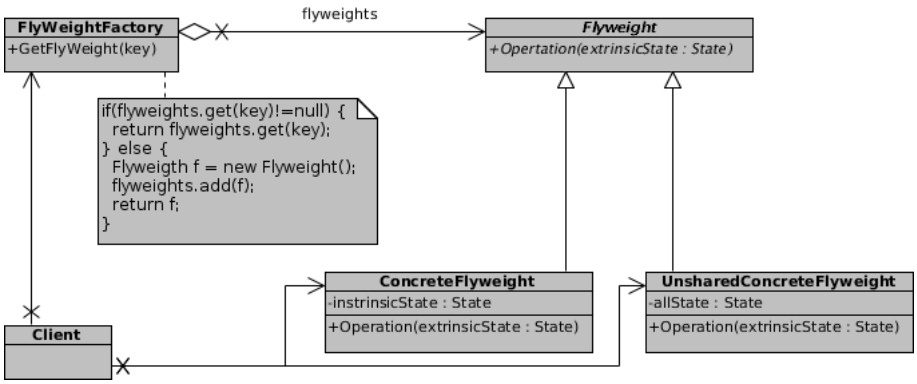

Figura A.11: O padrão Flyweight, adaptado de [25].

Com a necessidade de partilhar objectos muito refinados, surge este padrão (Figura A.11). Ele permite ter um grande número de elementos a um custo bastante baixo, através da partilha desses elementos, onde o seu estado é extrínseco.

Este padrão é bastante complexo em termos de inferência porque a sua estrutura é muito complexa. Existe a necessidade de ter um interface (`Flyweight`) definido (por um `ConcreteFlyweight`). Existirá então uma "fábrica" (`Flyweight-Factory`) que irá criar estes elementos. O maior problema está em que este método é baseado no comportamento da "fábrica". No momento de criar um objecto será necessário saber se já existe uma instância desse objecto antes de o criar. Por haver muitas formas de a definir, esta verificação poderá ser muito complicada ou mesmo impossível de inferir em certas condições.

A.2.12 Interpreter

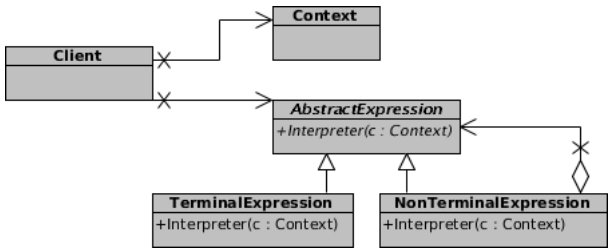

Figura A.12: O padrão Interpreter, adaptado de [25].

Este (Figura A.12) é um padrão de definição de linguagens e interpretação. Este padrão não é fácil de detectar, e é muito semelhante ao padrão Composite. O seu objectivo é também relativamente semelhante ao Composite.

A.2.13 Iterator

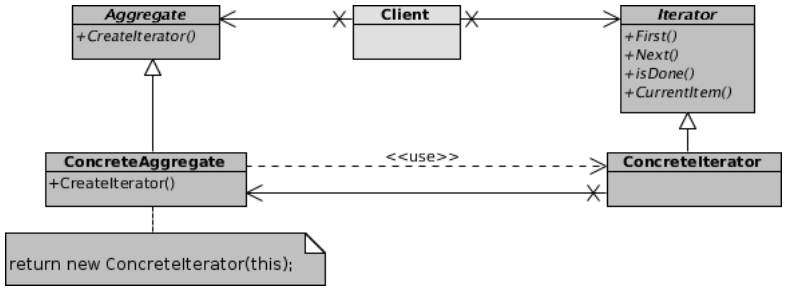

Figura A.13: O padrão Iterator, adaptado de [25].

Com o mesmo nome que a estrutura de Java `Iterator` (Figura A.13), este padrão representa isso mesmo. Define a forma como aceder sequencialmente a um objecto agregado sequencialmente sem expor a sua implementação.

A identificação deste padrão necessitaria de um elemento (`Iterator`) que define a interface de travessia. Deverá haver uma implementação (`ConcreteIterator`) desse interface. Por outro lado terá de haver uma (`Aggregate`) interface que define a criação de um objecto de iteração (`ConcreteAggregate`).

Por não possuir um comportamento que possa ser inequivocamente inferido, a detecção deste poderá levar a um elevado número de falsos positivos.

A.2.14 Mediator

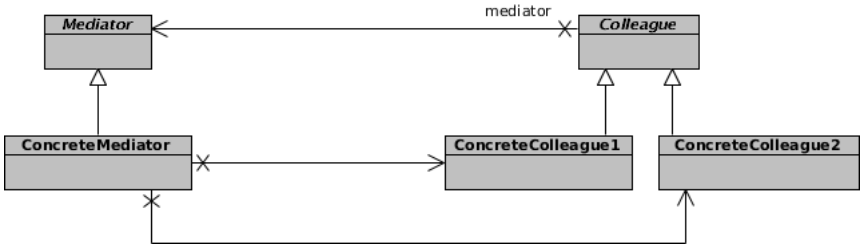

Figura A.14: O padrão Mediator, adaptado de [25].

Quando existe um elevado nível de complexidade entre objectos, e existe a necessidade de gerir as suas interacções usa-se o padrão Mediator (Figura A.14). Um mediator é então responsável por controlar o comportamento de um grupo de objectos.

A pesquisa necessitará de um interface (`Mediator`), que irá conter uma implementação (`ConcreteMediator`) onde está especificado o comportamento. Depois haverão objectos (`Colleague`), que devem fazer invocações nos métodos no `Mediator`. A implementação da classe principal (`ConcreteMediator`) invocará métodos das concretizações dos objectos a controlar (`Colleague`). Com estes elementos poder-se-á estar presente este padrão, contudo é muito difícil evitar falsos positivos. Em termos estruturais este padrão é bastante genérico e em termos de invocações este padrão não é muito específico.

A.2.15 Memento

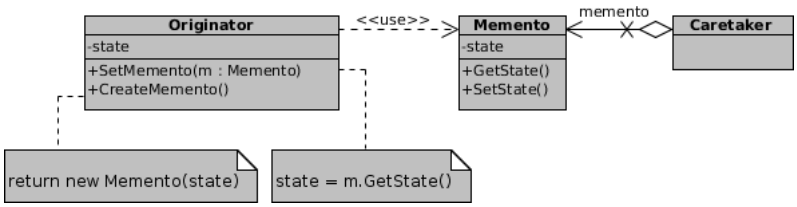

Figura A.15: O padrão Memento, adaptado de [25].

Quando se pretende preservar o estado de objectos, guardando o seu estado interno utiliza-se este padrão (Figura A.15). Este padrão respeita o encapsulamento. Permite regredir em versões de um objecto.

A sua identificação não será trivial. Requer a pesquisa por uma classe originária (`Originator`), que deverá conter os métodos de criação de estados

(`CreateMemento`) e de carregar estados (`SetMemento`). Deverá haver uma classe concreta (`Memento`) que responde a esses métodos.

A identificação deste padrão poderá ser algo complicada. Por um lado pesquisar uma classe que contenha dois métodos que sejam implementados por uma outra classe que inclua, poderá ser demasiado genérico, e levar a muitos falsos positivos. Por outro lado requerer que o método se chame `SetMemento` (por exemplo), não é uma solução viável. Mesmo com análise comportamental poderá ser algo complexo. Assim este padrão, apesar de importante, não poderá ser garantido à partida.

A.2.16 Observer

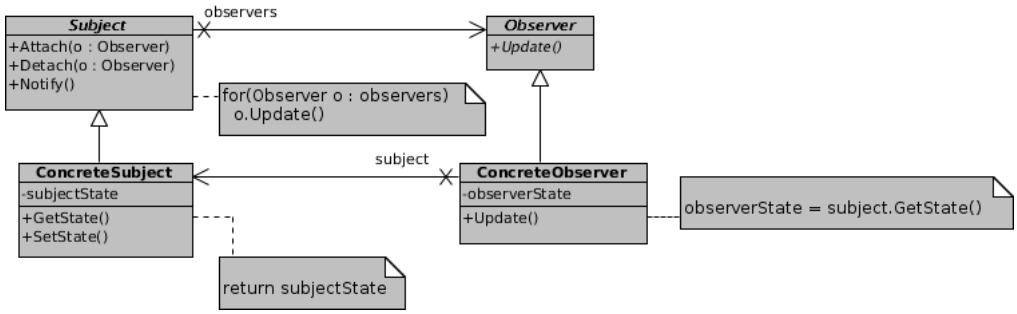

Figura A.16: O padrão Observer, adaptado de [25].

Observer (Figura A.16) define como se pode ter uma dependência um-para-muitos, para que quando o comportamento desse objecto muda, todos os dependentes sejam notificados.

Também a inferência deste padrão não é trivial. Em termos de estrutura, um interface (`Subject`) deve ter uma lista de interfaces de observadores (`Observer`). Esses observadores deverão responder a um método de actualização (`update`), invocado por um objecto concreto (`ConcreteSubject`). O resultado dessa invocação resultará em uma alteração de estado de um observador concreto (`ConcreteObserver`).

Também neste padrão uma análise demasiado ampla poderá resultar num elevado número de falsos positivos. Será necessário efectuar algum tipo de análise nos métodos dos elementos intervenientes.

A.2.17 Prototype

O padrão Prototype (Figura A.17) especifica de que forma se pode definir o tipo de objectos a produzir por meio de um protótipo. No processo de criação

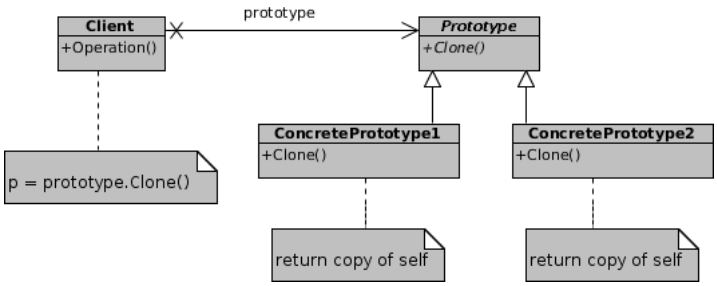

Figura A.17: O padrão Prototype, adaptado de [25].

esse protótipo será copiado. É útil para definição de tipos em *runtime* ou quando as instâncias de uma classe podem ter diferentes combinações de estados.

Para identificar este padrão seria necessário pesquisar uma classe (`Client`) que iria ter uma instância de um interface (`Prototype`). Este interface devetrá ter várias implementações concretas (`ConcretePrototype1`). Para evitar falsos positivos seria então necessário que a classe principal (`Client`) invocasse um método neste interface que será um `clone` dele mesmo.

Assim a identificação deste padrão será algo limitada. A estrutura terá de ser a indicada, e mais do que isso apenas funcionará caso sejam seguidos os padrões de Java utilizando o nome `clone` para o método de clonagem de objectos.

A.2.18 Proxy

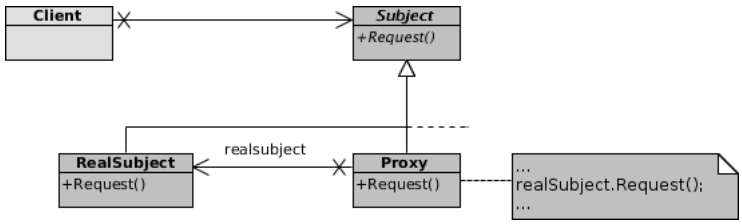

Figura A.18: O padrão Proxy, adaptado de [25].

Este padrão (Figura A.18) define a forma como um objecto acede a outro, e faz de intermediário nesse acesso. Este padrão tem várias vantagens, permitindo uma análise de um pedido antes que ele chegue ao destinatário. Permite então fazer optimizações, atrasos no pedido, registos, etc.

A análise deste padrão necessitaria da pesquisa por um interface (`subject`) que faz invocações a uma classe (`Proxy`) que tem uma referência para a classe destino (`RealSubject`). O comportamento para este padrão será a cadeia de

invocações a métodos começando no objecto principal (`Subject`), passando pelo `Proxy`, até à classe destino (`RealSubject`).

Para inferir este padrão seria então necessário uma análise estrutural e aos métodos. Este padrão é semelhante ao Adapter, contudo neste a cadeia de invocação de métodos requer que tenham o mesmo nome. Assim, seria necessária uma análise mais cuidada para evitar equívocos na inferência.

A.2.19 Singleton

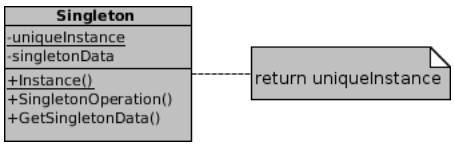

Figura A.19: O padrão Singleton, adaptado de [25].

O Singleton (Figura A.19) é um padrão bem conhecido, e representa um objecto que pode ser instanciado apenas uma vez. É importante ter essa garantia no caso de objectos partilhados, por exemplo.

Fazer a detecção deste padrão por análise estrutural seria praticamente impossível dado poder ter várias implementações possíveis. Para detectar este padrão seria necessário uma análise comportamental, comparando o resultado da invocação de um mesmo método duas vezes para comparar o objecto resultante.

A.2.20 State

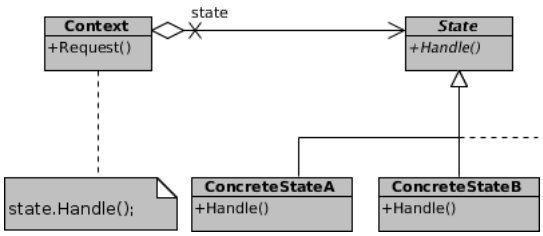

Figura A.20: O padrão *State*, adaptado de [25].

Este padrão (Figura A.20) define como alterar o comportamento de um objecto dependendo do seu estado interno de forma eficiente e flexível.

Seria pesquisada uma classe (`Context`) que contivesse uma instância de um interface (`State`). Por sua vez esse interface poderia ter várias instâncias (`ConcreteStateA`) diferentes. A classe principal (`Context`) deveria invocar um método (`Handle`) da interface, que por sua vez estaria nas implementações dessa interface.

A.2.21 Strategy

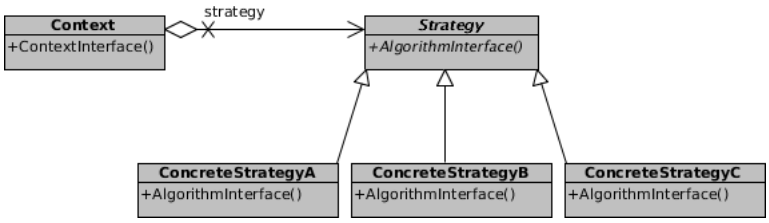

Figura A.21: O padrão Strategy, adaptado de [25].

A necessidade de encapsular uma família de algoritmos que possam ser trocados de forma eficiente, levou à criação do padrão Strategy (Figura A.21).

A pesquisa deste padrão seria feita por uma classe (`Context`) que contivesse uma outra classe abstracta (`Strategy`). Essa classe por sua vez teria várias implementações possíveis (`ConcreteStrategyA`). A classe principal (`Context`) iria invocar métodos da interface, realizados nas especificações. Este padrão é bastante semelhante com o padrão (`State`), e em termos de análise seria difícil distinguir ambos.

A.2.22 Template Method

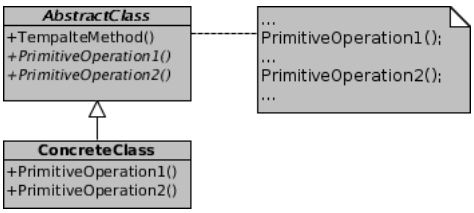

Figura A.22: O padrão Template Method, adaptado de [25].

Por vezes existe a necessidade de criar um algoritmo que em certo ponto possa ser expandido ou alterado, sem contudo alterar a estrutura do algoritmo. Este comportamento é reflectido no padrão Template Method (Figura A.22). Desta forma um algoritmo é definido, mas alguns dos passos podem ser diferidos para as suas sub-classes. Pode ser usado de forma a definir partes estáticas de algoritmos (invariantes) e deixar os comportamentos alteráveis às subclasses.

Em termos de estrutura é uma herança simples. Teríamos uma classe abstracta (`AbstractClass`) com um método (`TemplateMethod()`) que iria invocar todos os métodos (`PrimitiveOperation1`) presentes na classe. Deveria de existir uma classe concreta (`ConcreteClass`) que implementasse esses métodos.

Mais importante do que uma análise estrutural seria a análise dos métodos dos intervenientes. Para além disso este padrão corre o risco de falsos positivos.

A.2.23 Visitor

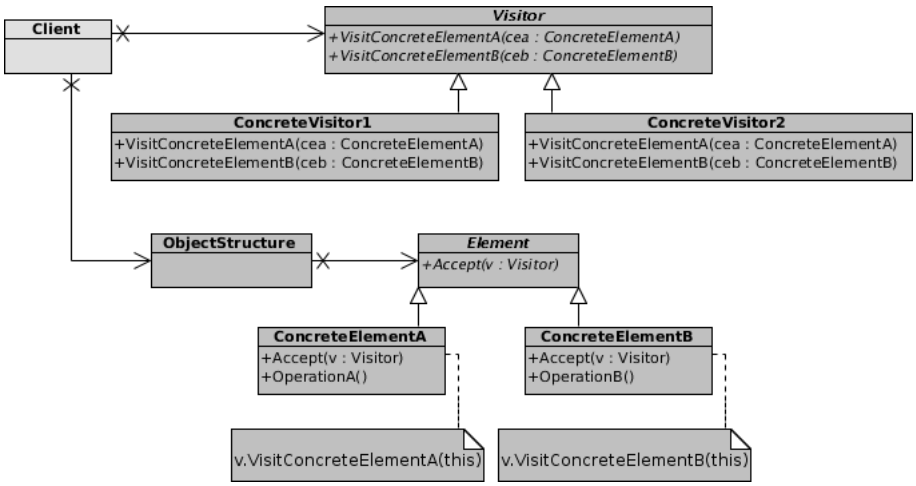

Figura A.23: O padrão Visitor, adaptado de [25].

Este padrão (Figura A.23) especifica de que forma é possível executar uma operação num grupo de objectos, com interfaces heterogéneos, dependendo da sua classe. Define como executar operações numa hierarquia de objectos sem alterar as classes dos elementos onde opera.

Este padrão poderia ser inferindo com uma pesquisa por uma classe abstracta (`Visitor`) que é o "visitante", que nas suas implementações (`ConcreteVisitor1`) contivesse métodos que recebam objectos (`VisitConcreteElementA (ConcreteElementA)`). Por sua vez, na estrutura de objectos, iria haver um método (`Accept`) que recebesse como argumento esse "visitante" (`Visitor`). Nesse mesmo método (`Accept`) seria necessário que seja invocado um método existente no "visitante" (`VisitConcreteElementA`).

Para identificar correctamente este padrão seria necessário combinar uma análise estrutural relativamente abrangente, com uma pesquisa mais refinada nos métodos, argumentos e invocações por parte dos participantes. Este padrão seria mais complicado de inferir pela existência de dois participantes distintos.

Anexo B

Catálogo personalizado de padrões

B.1 Catálogo

Este é o catálogo de padrões que contém uma regra para o padrão Composite. Foi utilizado para os testes com a ferramenta implementada.

```
% Pattern catalog
% c- classe, i-interface, C-calls, k- contains, e- extends, I- implements
!cikeI
composite/4#(composite(Figure,AbstractFigure,CompositeFigure,PolyLine) :-
(((class(Figure),extends(CompositeFigure,Figure),extends(PolyLine,Figure));
(interface(Figure),class(AbstractFigure),implements(AbstractFigure,Figure),
extends(PolyLine,AbstractFigure),extends(CompositeFigure,AbstractFigure))),
class(CompositeFigure),class(PolyLine),contains(CompositeFigure,Figure),
Figure \== AbstractFigure, Figure \== CompositeFigure, Figure \== PolyLine,
AbstractFigure \== CompositeFigure, CompositeFigure \== PolyLine,
AbstractFigure \== PolyLine))
```

Glossário

API (*Application Programming Interface*) é a definição de um conjunto de regras que definem como certa funcionalidade deve ser implementada. 3, 14

AS é uma linguagem de definição de comportamento de alto nível definida em *MDA Explained* [49]. 16, 17, 23

CASE representa um conjunto de ferramentas de desenvolvimento de software baseado em modelos, desenvolvido na década de 80. Este termo é vulgarmente utilizado para designar ferramentas de desenvolvimento baseadas em modelos. 2, 3

CIM (*Computation Independent Model*) são modelos de alto nível de modelação de um sistema, independentes do ambiente e propriedades de computação. 12, 20

CWM (*Common Warehouse Metamodel*) é a especificação de um conjunto de interfaces de troca de informação sobre modelos. 16

GPJaSI (*Gnu Prolog Java Simple Interface*) é o módulo que serve de interface de interacção dom a ferramenta gprolog. 84, 85

IDE (*Integrated Development Environment*) refere-se a um ambiente de desenvolvimento que integra um conjunto de ferramentas como edição de código, compilação, detecção de erros, depuração, etc. 7, 30, 31, 33, 41, 43, 48, 59, 60, 65, 80, 88, 89, 95, 98, 112–114, 116, 117, 122, 123, 125

JDI (*Java Debug Interface*) é um interface de depuração oferecido pela linguagem Java. 32

JPSI (*Java Parser Simple Interface*) é um dos módulos da ferramenta proposta neste documento. Refere-se ao módulo de interface com o *parser* de Java. 82, 83, 85, 88, 89

JRE (*Java Runtime Environment*) é o ambiente de execução de aplicações Java. 38, 41, 43

MapIt (*Model and Patterns Inferring Tool*) é a ferramenta de inferência de modelos e padrões, proposta nesta dissertação. 65–68, 70, 71, 73, 78, 80, 81, 88, 93, 118

MDA (*Model Driven Architecture*) é a especificação do OMG como desenvolver software baseado em modelos. 1–4, 6–8, 12–16, 18–24, 31, 35, 36, 39, 51–54, 67, 68, 74, 76, 121–124

MDE (*Model Driven Engineering*) é a aplicação na prática da especificação MDA. 3, 4

MFE (*Model Filter Engine*) é um dos módulos da ferramenta desenvolvida, responsável por fazer a filtragem de modelos, operação básica para a sua transformação. 90, 92

MOF (*Meta Object Facility*) consiste na especificação de uma linguagem de definição linguagens de modelação. 14, 16

OCL (*Object Constraint Language*) é uma linguagem declarativa de definição de restrições na linguagem UML. 16, 24

OMG (*Object Management Group*) é o grupo responsável pela definição dos padrões de desenvolvimento de software orientado a modelos. 1, 4, 14, 16, 17, 35

ORM (*Object Relational Mapping*) é uma técnica de programação que permite o mapeamento de objectos em bases de dados relacionais. 31

perfil UML é uma extensão da linguagem UML por adição de novos elementos, adaptando-o a um contexto especifico. 23

PIM (*Platform Independent Model*) define um modelo de computação que é independente da plataforma, e como tal pode ser reutilizado. 6, 8, 12, 13, 16–21, 23, 25, 28, 33, 49, 52, 54–56, 58, 61, 72–74, 76, 80, 81, 88, 90–92, 103, 121, 122

projecto consiste num conjunto maioritário de ficheiros de código fonte, mais alguns ficheiros de configuração como XML entre outros. 52, 54, 57–59

PSM (*Platform Specific Model*) define um modelo de computação específico de uma plataforma. É o nível de especificidade que se segue ao PIM. 6, 8, 12, 13, 16–21, 24, 25, 28, 29, 33, 51, 52, 54–58, 61, 66, 68, 72–78, 80, 81, 90–92, 101–104, 107, 109, 121, 122

QVT (*Query Views and Transformations Standards*) consiste num conjunto de linguagens de transformação definidas pela OMG. 16

SAI (*System Analysis Interface*) é o módulo da ferramenta responsável pela abstracção das operações de análise de um software, essencial para transformação de modelos. 85, 88, 90, 117

SOUL (*Smalltalk Open Unification Language*) é uma linguagem de unificação similar ao Prolog, construída em Smalltalk. 30, 78

SQL (*Structured Query Language*) é uma linguagem de consulta de informação, que permite pesquisa de dados numa base de dados. 24

UML (*Unified Modeling Language*) consiste numa linguagem padrão de definição de modelos. 1, 8, 16, 17, 21, 23, 24, 26, 27, 30–33, 36, 38–40, 44–47, 49, 53, 54, 67, 73, 80, 81, 83, 89, 92, 96, 101, 102, 107, 114, 115, 121–123

XMI (*XML Metadata Interchange*) é um formato de troca de dados para modelos baseado em XML. 14

XML (*Extensible Markup Language*) é um conjunto de regras que define um formato de ficheiros que contém informação organizada com etiquetas. 83