UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

EDUARDO STUDZINSKI ESTIMA DE CASTRO

K-Aspects: uma abordagem baseada em aspectos para implementação de sistemas de conhecimento

Dissertação apresentada como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação

Prof. Dr. Roberto Tom Price
Orientador

Profa. Dra. Mara Abel
Co-orientadora

Porto Alegre, maio de 2009.

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Castro, Eduardo Studzinski Estima de

K-Aspects: uma abordagem baseada em aspectos para implementação de sistemas de conhecimento / Eduardo Studzinski Estima de Castro – Porto Alegre: Programa de Pós-Graduação em Computação, 2009.

83 f.:il.

Dissertação (mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação. Porto Alegre, BR – RS, 2009. Orientador: Roberto Tom Price; Co-orientadora: Mara Abel.

1.Engenharia de Software. 2.Sistemas de Conhecimento 3.Projeto. I. Price, Orientador da. II. Abel, Co-orientador da. III. K-Aspects: uma abordagem baseada em aspectos para implementação de sistemas de conhecimento.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. Carlos Alexandre Netto
Vice-Reitor: Prof. Rui Vicente Oppermann
Pró-Reitor de Pós-Graduação: Prof. Aldo Bolten Lucion
Diretor do Instituto de Informática: Prof. Flávio Rech Wagner
Coordenador do PPGC: Prof. Álvaro Freitas Moreira
Bibliotecária-Chefe do Instituto de Informática: Beatriz Regina Bastos Haro

AGRADECIMENTOS

Gostaria de agradecer a minha família e a minha namorada, Gisele, pelo apoio na realização desse mestrado. Principalmente, por entender a minha ausência em alguns momentos para a realização do trabalho.

Gostaria de agradecer aos meus orientadores, Professor Tom Price e Professora Mara Abel pelo completo apoio prestado durante o desenvolvimento desse trabalho e pela riqueza do conhecimento adquirido ao longo desse mestrado.

Gostaria de agradecer aos amigos do Grupo BDI pela riqueza de idéias trocadas. Principalmente ao Sandro, pelas longas discussões teóricas e técnicas.

Gostaria de agradecer a ENDEEPER por ter apoiado esse trabalho.

Gostaria de agradecer a todo o corpo docente e funcionários do Instituto de Informática/UFRGS pelo esforço conjunto realizado ao longo dos anos possibilitando a pós-graduação estar em um excelente nível.

Por fim, gostaria de agradecer a todos aqueles que contribuíram de alguma forma para esse trabalho.

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS 6
LISTA DE FIGURAS 7
LISTA DE TABELAS 8
RESUMO 9
ABSTRACT 10
1 INTRODUÇÃO 11
1.1 Motivação 13
1.2 Objetivos 15
1.3 Organização dos Capítulos 17
2 CONCEITOS E TRABALHOS RELACIONADOS 19
2.1 Orientação a Aspectos 19
2.2 Orientação a Aspectos e Sistemas de Conhecimento 21
2.3 Anotações de Metadados 22
2.4 Orientação a Objetos e Sistemas de Conhecimento 24
2.4.1 Sistemas Orientados a Objetos para Processamento de Regras 24
2.4.2 XP.K 24
2.5 Análise Crítica dos Trabalhos Relacionados 27
3 K-ASPECTS: ASPECTOS PARA IMPLEMENTAÇÃO DO COMPONENTE CONCEITUAL 29
3.1 Processo de Uso de K-Annotations 30
3.2 Biblioteca de K-Annotations 32
3.3 Biblioteca de Tratamento de *Facets*, Axiomas e Regras 38
3.4 Biblioteca de K-Aspects do Componente Conceitual 39
3.5 Sumário do Capítulo 3 41
4 K-ASPECTS: ASPECTOS PARA IMPLEMENTAÇÃO DOS COMPONENTES DE TAREFA E INFERENCIAL 42
4.1 Biblioteca de K-Aspects para Componentes de Tarefa e Inferencial 43
4.2 Exemplo Didático 46
4.3 Sumário do Capítulo 4 49
5 VALIDAÇÃO DA PROPOSTA 50
5.1 Estudo de Caso: Sistema Petroledge 50

5.1.1 Componente Conceitual 52
5.1.2 Componentes de Tarefa e Inferencial 58
5.2 Solução Atual 61
5.2.1 Implementação do Componente do Modelo Conceitual sem K-Aspects 62
5.2.2 Implementação dos Componentes de Tarefa e Inferencial sem K-Aspects 65
5.3 Solução com K-Aspects 66
5.3.1 Implementação do Componente do Modelo Conceitual com K-Aspects 67
5.3.2 Implementação dos Componentes de Tarefa e Inferencial com K-Aspects 70
5.3.3 Benefícios da Implementação do Petroledge usando K-Aspects 71
5.4 Sumário do Capítulo 5 73
6 CONCLUSÃO 76
6.1 Trabalhos Futuros 77
REFERÊNCIAS 78
ANEXO GRAMÁTICA DO JEP (AXIOMAS) 82

LISTA DE ABREVIATURAS E SIGLAS

CKADS	CommonKADS
IA	Inteligência Artificial
JEP	Java Expression Parser
MC	Modelo de Conhecimento
OO	Orientação a Objetos
OA	Orientação a Aspectos
PSM	Métodos de Soluções de Problemas – Padrões Cognitivos
SC	Sistema de Conhecimento
SGBD	Sistema Gerenciador de Banco de Dados
UML	Unified Modeling Language

LISTA DE FIGURAS

Figura 2.1: Exemplo de Dispersão devido a Funcionalidades Transversais. 20
Figura 2.2: Exemplo de Aspecto para Log de Invocação de Método............................ 20
Figura 2.3: Código-fonte da Anotação *Log*... 23
Figura 2.4: Código-fonte da obtenção de anotações de uma classe via reflexão. 23
Figura 3.1: Processo de Uso de *K-Annotations*.. 31
Figura 3.2: Diagrama de Sequência para Validação de Axioma.................................. 32
Figura 3.3: Metamodelo dos Construtos do Componente Conceitual........................... 34
Figura 3.4: Visão Lógica das *K-Annotations*. 34
Figura 3.5: Visão Lógica da Biblioteca de Tratamento de *K-Annotations*.................... 38
Figura 3.6: Visão Lógica dos *K-Aspects* para o Componente Conceitual..................... 41
Figura 4.1: Estrutura de inferência que descreve o método de classificação 43
Figura 4.2: Visão Lógica dos *K-Aspects* para os Componentes do Modelo de Conhecimento.. 45
Figura 4.3: Exemplo de Implementação do Componente Conceitual........................... 47
Figura 4.4: *Pointcut* para validação de valor máximo.. 48
Figura 4.5: *Advice* para validação de valor máximo. 48
Figura 5.1: Fluxo de Trabalho do Sistema Petroledge 50
Figura 5.2: Tela para Entrada de Dados de Microscopia 51
Figura 5.3: Conceitos do Sistema Petroledge... 56
Figura 5.4: Metamodelo de Grafo de Conhecimento para Interpretação 57
Figura 5.5: Exemplo de Grafo de Conhecimento... 57
Figura 5.6: Modelo da Tarefa de Interpretação de Ambiente Diagenético 59
Figura 5.7: Modelo de Inferência do PSM de Interpretação de Ambientes Diagenéticos 60
Figura 5.8: Fragmento da Implementação Atual do Conceito ThinIdentification. 63
Figura 5.9: Fragmento da Implementação Atual do Conceito Microscopic. 64
Figura 5.10: Diagrama de Sequência da Implementação do PSM de Interpretação. 65
Figura 5.11: Aspectos identificados na implementação de três conceitos. 67
Figura 5.12: Fragmento da Implementação com K-Aspects do Conceito ThinIdentification. 69
Figura 5.13: Fragmento da Implementação com K-Aspects do Conceito Microscopic. 70

LISTA DE TABELAS

Tabela 3.1: *K-Annotations* para a Definição de *Facets*. 37
Tabela 4.1: Exemplo de Inferências 42
Tabela 4.2: Exemplo de Componente Conceitual. 46
Tabela 5.1: Modelo de Conceitos do Sistema Petroledge 53
Tabela 5.2: Quantidade de Funcionalidades Transversais Tratadas via Aspectos. 66

RESUMO

Esse trabalho define K-Aspects (*Knowledge Aspects*), uma abordagem para a implementação de Sistemas de Conhecimento (SC) em linguagens orientadas a objetos usando o paradigma orientado a aspectos (OA) e anotações de metadados. Essa abordagem define uma forte correspondência entre o Modelo de Conhecimento (MC) e sua implementação no paradigma da orientação a objetos (OO). K-Aspects fornece um conjunto de anotações documentacionais para facilitar a leitura da associação entre o modelo de implementação e o modelo conceitual do conhecimento; um conjunto de anotações para facilitar a separação de interesses na implementação do SC usando OA; um conjunto de bibliotecas para realizar a interpretação das anotações e sua execução em aspectos; e uma ferramenta para geração de documentação do MC a partir das anotações no código. A abordagem busca atender tanto os engenheiros de conhecimento quanto os engenheiros de desenvolvimento em projetos de SC. Os engenheiros de conhecimento tem ao seu alcance um modo adequado para elaborar a especificação do MC que resulta em uma especificação em uma linguagem orientada a objetos, permitindo aos engenheiros de desenvolvimento implementarem o sistema preservando a estrutura do modelo conceitual e mantendo clara distinção entre os requisitos associados ao MC dos demais requisitos.

K-Annotations adicionam recursos de OA ao modelo conceitual do conhecimento OO, oferecendo facilidades de tratamento separado de diversas funcionalidades transversais de um SC, através do particionamento do sistema em aspectos que implementam funcionalidades específicas, ativadas através das anotações inseridas no componente do modelo conceitual. Anotações distinguem, clara e visualmente, no código, os elementos do MC em relação ao restante do código do programa, facilitando a leitura do código pelos engenheiros de conhecimento. A função principal das anotações é prover as informações necessárias para a interpretação dos elementos de conhecimento durante a execução do programa. Anotações identificam as funcionalidades transversais relativas aos construtos do modelo e são gerenciadas pela biblioteca de aspectos.

A abordagem foi validada re-escrevendo com o uso de K-Annotations um sistema de conhecimento no domínio da análise da qualidade de reservatórios de petróleo. O modelo desse sistema representa uma ontologia de domínio sobre o qual métodos de solução de problemas para interpretação e classificação de rochas são aplicados. A análise dos resultados identificou um conjunto de vantagens no uso de K-Aspects, como distinção clara entre a implementação do MC e a implementação dos demais requisitos, suporte nativo aos construtos providos no modelo de conhecimento e alto grau de rastreabilidade entre o modelo e sua implementação. Além disso, provê redução de tarefas repetitivas de implementação e redução da dispersão de código a partir da geração automática de código. As bibliotecas de aspectos garantem o encapsulamento de inferências e tarefas. As bibliotecas tratam os construtos do modelo para garantir a reutilização em diferentes projetos de SCs.

Palavras-Chave: engenharia de software, engenharia de conhecimento, orientação a aspectos, anotações de metadados, projeto de sistemas baseados em conhecimento

K-Aspects: an approach for building knowledge systems using aspects

ABSTRACT

This work defines K-Aspects (Knowledge Aspects), an approach for implementing Knowledge Systems (KS) with object-oriented languages using the aspect-oriented paradigm (AO) and metadata annotations. This approach defines a strong link between the knowledge model (KM) and its implementation in the object-oriented paradigm (OO). K-Aspects provides a set of documentational annotations to make the association between the implementation model and the knowledge conceptual model easier to read; a set of annotations to enable the separation of concerns, using aspect orientation, of the implementation of the different requirements of the knowledge system; a set of libraries to perform the interpretation and execution as aspects of annotations; and a tool for documentation generation of the KM extracted from the annotations on the code. The approach aims to support both knowledge engineers and development engineers in KS projects, by providing to the knowledge engineers a well-defined way to elaborate the KS specification, which results in a specification presented in an object oriented language, making it easily understandable and extensible by development engineers that can thus implement the knowledge system preserving its conceptual knowledge structure and keeping a clear distinction of the requirements associated to the KM from the other requirements.

K-Annotations add aspect oriented resources to the OO conceptual knowledge model, providing features to manage separately the multiple crosscutting concerns of a KS, partitioning the system in aspects that implement specific features, activated by annotations inserted in the knowledge conceptual model. Annotations distinguish, clearly and visually, within the code, KM elements from the rest of the code, making easier the code reading by the knowledge engineers. The most important function of the annotations is to provide information necessary for interpreting knowledge elements during runtime. Annotations identify several crosscutting concerns related to the model constructs. Annotations are managed and executed by the aspect libraries.

This approach was validated by re-coding, using K-Annotations, a complex commercial KS on the domain of oil reservoir quality analysis. The model of this system represents a domain ontology on which problem-resolving methods for rock interpretation and classification are performed. The analysis of the results identified several advantages of using K-Aspects as: a clear distinction of the KM implementation among other requirements; native support for knowledge model constructs; and high traceability between the knowledge conceptual model and its implementation. Moreover, the use of K-Aspects reduces repetitive implementation tasks and code dispersion because of the automatic code generation. The provided aspect libraries enable the encapsulation of inferences and the execution of several tasks. The libraries manage the constructs of the model thus providing reusability among multiple KS projects.

Keywords: software engineering, knowledge engineering, aspect oriented paradigm, metadata annotations, project of knowledge systems.

1 INTRODUÇÃO

A evolução do desenvolvimento de sistemas de conhecimento (SC) pode ser dividida em diferentes etapas. As primeiras gerações desses sistemas foram implementadas através de processos *ad-hoc* de desenvolvimento, sem utilização de uma metodologia que definisse um processo de engenharia, controlado e repetível. A falta de uma metodologia de desenvolvimento contribui para altos custos de construção de sistemas e a baixa qualidade nos projetos. As primeiras gerações de SC produzidas no ambiente acadêmico alcançaram bons resultados científicos; entretanto, quando eram aplicadas no ambiente comercial, falhavam, pois não conseguiam atender as características essenciais aos sistemas comerciais, dentre as quais destacam-se confiabilidade, manutenabilidade e adaptabilidade à evolução do conhecimento e capacidade de tratar de um grande volume de informação (STUDER, BENJAMINS e FENSEL, 1998).

A dificuldade em transferir a tecnologia de SC do meio acadêmico para o meio comercial pode ser comparada à crise do final dos anos sessenta relativa ao desenvolvimento de sistemas de software tradicionais. Naquela época, o modo de desenvolvimento de pequenos protótipos não era suficiente para atender as necessidades de projeto e a manutenabilidade de sistemas comerciais. Desse modo, houve o desenvolvimento da Engenharia de Software para tratar essa deficiência. O mesmo pode ser identificado na área de Engenharia de Conhecimento: essa área se desenvolveu voltada para o estabelecimento de uma disciplina de engenharia no desenvolvimento de SC (STUDER, BENJAMINS e FENSEL, 1998). A Engenharia de Conhecimento tem como foco a aquisição do conhecimento que dá suporte a solução de problemas pelo sistema e sua representação em modelos processáveis por computador. Já a Engenharia de Software lida com todos os aspectos ligados ao tratamento desses modelos para que o processo de implementação seja viável e o resultado final seja um software que implemente completamente esses modelos e os algoritmos de raciocínio, atendendo aos requisitos da aplicação e dos ambientes para os quais foram concebidos.

O estabelecimento da Engenharia de Conhecimento, posterior ao da Engenharia de Software está relacionado a características específicas desses sistemas que devem ser tratadas no seu desenvolvimento. SC distinguem-se de sistemas de informação tradicionais devido às seguintes características:

- Representam o conhecimento de especialistas em algum domínio específico de conhecimento (BROMBY, MACMILLAN e MCKELLAR, 2003);
- Provêem meios para o compartilhamento de informações valiosas que frequentemente estão centralizadas por um especialista (SCHREIBER et al., 2000);
- Resolvem problemas que não são tratáveis através de soluções algorítmicas que necessitem de uma sequência finita e não ambígua de instruções. SCs propõem soluções em que algoritmos tradicionais não são capazes de resolver o problema de forma eficiente em termos de utilização dos recursos de máquina, ou cuja solução torna-se excessivamente complexa quando comparada à utilização de métodos de busca simbólica associados a heurísticas sobre o problema (BRACHMAN e LEVESQUE, 2004);

- São capazes de gerenciar o conhecimento e realizar inferências sobre os mesmos. A sequência de passos de raciocínio desenvolvida no processo de solução de problemas deve ser verificável pelo usuário (SCHREIBER et al., 2000);
- Suportam a expansão e atualização do conhecimento tratado pelo próprio sistema (SCHREIBER et al., 2000);
- Os requisitos do sistema mudam frequentemente devido à complexidade, imprecisão e constante evolução do conhecimento (CASTRO, VICTORETI, FIORINI, ABEL e PRICE, 2008).

A principal fraqueza de metodologias puramente oriundas da Engenharia de Conhecimento, segundo Knublauch (2002), é a tentativa de tratar isoladamente a modelagem de conhecimento do restante da aplicação. Esse isolamento torna bastante difícil a transição de representações de conhecimento baseadas em modelos conceituais para módulos executáveis. SCs se caracterizam por uma contínua evolução da base de conhecimento, refletindo o estado da área ao qual se aplica, portanto a arquitetura do sistema e a documentação dos modelos de conhecimento e dados devem ser concebidas de forma a absorver e facilitar a modificação frequente dos sistemas sem rupturas em sua utilização. Mais modernamente, cresce a necessidade de integração dos SCs com os demais sistemas de informação corporativos das organizações e a demanda por uma maior capacidade de processamento de informações por esses sistemas, que não se constituíam em requisitos importantes nos primeiros sistemas de conhecimento comerciais.

Essas dificuldades são ampliadas pelo fato de que projetos de SC possuírem mais agentes envolvidos no desenvolvimento dos sistemas. Alem dos problemas de comunicação e documentação entre engenheiros de desenvolvimento, existem também os problemas de comunicação destes com o engenheiro de conhecimento, responsável pela definição dos modelos e formalização da evolução dos requisitos. As linguagens de modelagem utilizadas nessas metodologias são de amplo conhecimento dos engenheiros de conhecimento, mas não são amplamente utilizadas pelos engenheiros de desenvolvimento, pois muitas não oferecem compiladores ou apenas oferecem compiladores experimentais, inviáveis para uso em sistemas comerciais.

Devido às características definidas acima, principalmente devido à dinamicidade dos requisitos de um SC, conforme Knublauch (2002), toda metodologia de desenvolvimento de SCs deve oferecer linguagens e ferramentas de modelagem de conhecimento que suportem eficientemente evolução, retroalimentação e comunicação. Durante a evolução da Engenharia de Conhecimento, uma série de metodologias para a construção de SCs foram propostas, entre elas:

- VITAL (MESEGUER e PREECE, 1995);
- MIKE (ANGELE et al., 1998);
- CommonKADS (Common Knowledge Acquisition and Design System) (SCHREIBER et al., 2000);
- Protégé (GENNARI et al., 2003);
- XP.K (KNUBLAUCH, 2002);
- RapidOWL (AUER, 2006);

- KM-IRIS (CHALMETA e GRANGEL, 2008);

CommonKADS, diferencia-se das demais por ser uma metodologia alinhada a metodologias *heavy-weight* da Engenharia de Software como RUP, pois aborda todos os aspectos de um projeto de SC: gerência de projeto, workflow de desenvolvimento, integração, aquisição de conhecimento e desenvolvimento de software. Protégé, apesar de não ser uma metodologia completa, destaca-se por ser um ambiente para o desenvolvimento de SCs que oferece uma ferramenta com suporte a ontologias e frames para modelagem de conhecimento; essa ferramenta tem evoluído continuamente e tem sido usada com sucesso em projetos de SC. A metodologia KM-IRIS é voltada para SCs que envolvem toda a empresa alvo. Ela busca definir um conjunto de processos necessários para a realização de todas as etapas de desenvolvimento de um sistema de conhecimento (CHALMETA e GRANGEL, 2008), mas não detalha os modelos envolvidos, nem trata detalhadamente a complexidade para integrar todos os sistemas de informação de uma empresa que servem de entrada para o sistema de conhecimento. Mais recentemente, XP.K e RapidOWL buscaram adaptar os princípios das metodologias ágeis ao desenvolvimento de SCs. XP.K destaca-se por ter organizado os princípios e valores de metodologias ágeis aliados a orientação a objetos para serem aplicados no desenvolvimento de SCs.

1.1 Motivação

As metodologias citadas acima tem como enfoque a fase de modelagem do projeto, especialmente, na elaboração do modelo de conhecimento. Segundo Schreiber et al. (2000), esse modelo é composto por três componentes (o termo componente é utilizado, no caso do modelo de conhecimento, para caracterizar os elementos que compõe esse modelo):

- Componente Conceitual: descreve as informações estáticas/estruturas do conhecimento (conceitos, atributos, relações, regras e axiomas) relacionadas ao domínio da aplicação (por exemplo: doenças crônicas, petrografia sedimentar);
- Componente de Tarefa: descreve as estratégias usadas pelo sistema (no componente conceitual) para resolver problemas, orientado a sistema. A tarefa, dependente do domínio da aplicação, define a relação entre um método genérico de solução de problema (PSM) e os conceitos particulares do domínio da aplicação descrito no componente conceitual (ABEL, 2001);
    - Cada método de solução de problemas (PSM) é um modelo abstrato de inferência reusável para um mesmo tipo de tarefa (ex.: interpretação, classificação) em diferentes domínios de aplicação (ex.: medicina, conserto de carro). Por exemplo, a tarefa de diagnóstico é necessária para solução de problemas em diferentes domínios (diagnosticar a doença de um paciente, interpretar um problema de carro). Nesse caso, um PSM voltado para essa tarefa deve ser reusável em ambos domínios. A metodologia CommonKADS (SCHREIBER et al., 2000) define uma biblioteca extensa de estruturas de inferência que correspondem a PSMs no âmbito da metodologia;
- Componente Inferencial: define os passos básicos de raciocínio usados para realizar tarefas usando os elementos do componente conceitual;

Essas metodologias usam diferentes linguagens para modelagem de conhecimento como Frames (FIKES e KEHLER, 1985), OCML (TANASESCU, DOMINGUE e CABRAL, 2004), CML (SCHREIBER et al., 2000), RDF (BRICKLEY e GUHA, 2000) e OWL (ANTONIOU e HARMELEN, 2004). A maioria dessas linguagens define construtos que não são nativamente suportados na OO. Desse modo, modelos de conhecimento definidos nessas linguagens não podem ser facilmente convertidos para o paradigma da OO.

A ausência de um solução padrão para a conversão de modelos de conhecimento em componentes de software traz um problema relevante para a fase de desenvolvimento de projetos de SC: muitos SC complexos são desenvolvidos usando linguagens OO, mas a implementação do modelo de conhecimento é feita utilizando uma solução *ad-hoc* que não permite a preservação das estruturas definidas no modelo de conhecimento. Desse modo, mesmo que o modelo de conhecimento possa ser modelado usando diferentes paradigmas, é importante prover uma forma de representação padrão no paradigma OO, pois muitos projetos usam Java e C# como linguagens para desenvolvimento de SCs.

Apesar de todas essas propostas de metodologias, nenhuma delas oferece uma abordagem composta por padrões, ferramentas e bibliotecas de código para o mapeamento do modelo de conhecimento para uma linguagem OO. Somente XP.K busca definir uma abordagem padrão para implementação de SCs na OO, mas apresenta uma série de desvantagens que impedem sua utilização de fato. XP.K é revisada no capítulo 2. Atualmente, cada projeto de SC utiliza uma solução *ad-hoc* para a implementação desse modelo. Por exemplo, uma solução *ad-hoc* em uso pelo sistema Petroledge (ABEL, 2001), SC complexo para a avaliação da qualidade de reservatórios de petróleo, utiliza os seguintes mapeamentos para a implementação do modelo de conhecimento usando o paradigma da OO:

- Componente conceitual é implementado pelo seguinte mapeamento (SILVA, 2001):
    - Conceitos são mapeados para classes;
    - Atributos dos conceitos são mapeados para propriedades de classes;
    - Facets axiomas e regras são mapeados para classes externas (seguindo a *design pattern validator* (FOWLER, 1997)) ou para métodos das classes que implementam conceitos ou para métodos presentes nas próprias interfaces gráficas. *Facets* são usados para definição de restrições, por exemplo, propriedade não pode ter valor *null*. Axiomas são usados para a definição de relações matemáticas que devem ser sempre verdadeiras, por exemplo a soma de duas propriedades deve ser sempre maior que zero. E regras são usadas para definir validações e implicações entre diferentes propriedades, por exemplo, se uma pessoa não tem esposa, ela não pode ser casada. O mapeamento dessas três estruturas para métodos reduz a manutenabilidade do sistema, pois esse mapeamento torna difícil distingui-las na implementação e também acaba ocasionando uma grande dispersão de chamadas à biblioteca de validações da facets, axiomas e regras que fazem o tratamento das mesmas;
- Componente de tarefa, apesar de ser modelado como um PSM, é implementado como um algoritmo dependente do domínio da aplicação e possui dependência de banco de dados para funcionamento. O principal fator de dependência do

domínio é a falta de uma distinção clara dos construtos do modelo conceitual. A dependência em relação ao domínio da aplicação impede o reuso dos artefatos de software em outros SCs. A dependência em relação ao banco de dados reduz o reuso quando a persistência é realizada em diferentes tipos de bancos de dados. Para cada tipo de SGBD (relacional, XML, OO), uma implementação diferente precisa ser realizada;

- Componente inferencial é implementado por funções que são dependentes do domínio da aplicação devido a falta de uma distinção clara dos elementos originados do modelo conceitual em relação aos demais elementos da aplicação. Entretanto, essas funções deveriam ser independentes do domínio para que sejam reusadas em diferentes projetos.

Em consequência da falta de uma solução previamente validada e uso de soluções *ad-hoc,* que não apresentam validações prévias em diferentes SCs, não apenas o projeto citado acima, mas diversos projetos de SC enfrentam os seguintes problemas (CASTRO, ABEL e PRICE, 2009):

- Na maioria dos casos, soluções *ad-hoc* geram componentes que não preservam as estruturas do modelo de conhecimento, por exemplo, conceito e relações tornam-se classes; *facets*, axiomas e regras são todos transformados em métodos. Essa falta de preservação aumenta significativamente os custos de manutenção dos SCs, pois a rastreabilidade entre modelo e implementação fica comprometida;
- A solução pode atender a demanda atual do projeto, mas rapidamente deixar de atender completamente os requisitos, lembrando que os requisitos de SC tendem a evoluir rapidamente;
- O custo de evolução e manutenção de uma solução *ad-hoc* pode tornar-se bastante elevado ao longo do projeto, pois cada modificação pode afetar todos os componentes implementados. Isso muitas vezes ocorre devido ao alto volume de relações que os elementos do modelo de conhecimento apresentam;
- A solução é conhecida somente pelos envolvidos no projeto; entretanto, ao longo do ciclo de vida de um projeto, a maioria dos participantes acaba migrando para outros projetos. Desse modo, o risco de problemas para manutenção do sistema torna-se bastante elevado;
- É difícil manter coerente a documentação do modelo de conhecimento com sua implementação.

Os problemas apontados acima motivaram a busca de uma abordagem que pudesse eliminá-los ou reduzi-los.

1.2 Objetivos

A abordagem proposta nesse trabalho utiliza como base o paradigma orientado a aspectos (OA) associado a anotações de metadados para organizar uma solução que ofereça as ferramentas e bibliotecas necessárias para a construção de componentes de software de SCs que mantenham as estruturas definidas no modelo de conhecimento. A preservação das estruturas pode ser alcançada através do suporte aos construtos das linguagens de modelagem nas linguagens OO utilizadas para implementação de componentes de software que implementam o modelo. Ao suportar os construtos de

modelagem diretamente nos componentes de software, pode-se distinguir claramente cada elemento do modelo de conhecimento garantindo a rastreabilidade entre modelo e componente de software. A rastreabilidade entre modelo e componentes reduz os custos de manutenção de SCs, pois a consistência entre modelo e implementação pode ser mais facilmente mantida.

Conforme Kiczales et al. (1997), um aspecto é "uma entidade de software que captura uma funcionalidade transversal à uma aplicação". Aspectos permitem o tratamento isolado de diferentes funcionalidades do sistema, por exemplo, é possível isolar o tratamento de auditoria de todos os outros módulos do sistema, sendo que esses módulos através de aspectos serão auditados. Em suma, um aspecto permite o tratamento de funcionalidades sem que o código para isso tenha que ser explicitamente adicionado a todos os módulos que fazem uso delas. Aspectos têm sido aplicados com sucesso em diferentes áreas como sistemas operacionais (CUNHA, SOBRAL e MONTEIRO, 2006), segurança (ZHU e ZULKERNINE, 2009) e sistemas distribuídos (KOURAI, HIBINO e CHIVA, 2007).

Anotações são elementos tipados (*type-safe*) que permitem a adição de metadados a classes, métodos, variáveis, parâmetros e pacotes (ANNOTATIONS, 2004). Anotações tem sido utilizadas em diversas áreas como bancos de dados (JSR 220B, 2005), sistemas operacionais (CUNHA, SOBRAL e MONTEIRO, 2006) e sistemas distribuídos (JSR 220A, 2005) para fins documentacionais e de marcação para geração automática de código ou tratamento por aspectos. Por exemplo, uma anotação associada a uma variável pode indicar que a variável representa um campo a ser armazenado em um banco de dados.

Anotações podem marcar pontos no código e fornecer dados necessários para a implementação por aspectos de certas funcionalidades, por exemplo, auditoria. As seções 2.1 e 2.3 revisam esses dois assuntos para melhor compreensão do trabalho.

Em *K-Aspects*, anotações (*K-Anntations*) são utilizadas para identificar claramente os construtos do modelo conceitual do conhecimento utilizados na modelagem de um SC (ex.: conceito, atributo, relação, *facet*, axioma e etc) no código-fonte do sistema, facilitando a identificação da implementação desses construtos e também a revisão dos mesmos. O K nos termos *K-Aspects* e *K-Annotations* refere-se a *Knowledge* (conhecimento em Inglês), pois estão relacionados ao tratamento dos elementos do modelo de conhecimento. Esses termos foram criados nesse trabalho.

O uso de anotações permite que a documentação do modelo seja obtida diretamente do código-fonte e aumenta a rastreabilidade entre modelo e os componentes de software gerados. O mapeamento direto entre construtos do modelo e anotações também favorece o desenvolvimento orientado a modelos, já que ferramentas de geração de código podem ler os modelos e gerar as respectivas anotações.

Anotações servem para marcar classes e seus membros (ex.: métodos, propriedades) para que os aspectos possam implementar a semântica de cada anotação através de interpretação. Por exemplo, uma classe pode ser anotada com *@Concept* para identificá-la como um conceito, uma propriedade chamada *idade* pode ser anotada com *@Attribute* para que seja reconhecida como um atributo. Essa propriedade ainda pode ser anotada com *@FacetMinValue(value=0)* para indicar que seu limite inferior é 0. Essas anotações identificam aos aspectos os pontos em que a semântica de cada anotação deve ser tratada e também fornecem dados necessários para esse tratamento (ex.: o valor mínimo para uma propriedade). O uso de anotações e aspectos, em

substituição a chamadas explícitas a funções definidas em uma biblioteca de componentes, automatiza o tratamento da semântica dos construtos, evitando que o engenheiro de desenvolvimento tenha que manualmente invocar em todos os pontos de tratamento desses construtos a biblioteca. A automatização reduz o risco de bugs relacionados a omissão de chamadas à biblioteca em pontos necessários e reduz os custos de manutenção, pois caso um problema esteja relacionado à invocação da biblioteca, não será necessário alterar todos os pontos de chamada, apenas o aspecto que realiza a chamada precisa ser corrigido.

O uso de anotações visa preservar as estruturas definidas no modelo de conhecimento, especificamente os construtos, evitando o uso de convenções de código para esse fim. Convenções de código devem ser evitadas, pois, enquanto anotações são validadas pelos interpretadores de linguagens, convenções não o são, podendo ser facilmente desrespeitadas. Além disso, convenções de código não são tipadas, suportando apenas *strings*, o que aumenta significativamente a inserção de erros relacioados a tipos.

O tratamento por aspectos da semântica associada a cada construto do modelo de conhecimento evita a dispersão de uma série de métodos e propriedades auxiliares necessários para o tratamento da semântica. Por exemplo, para determinar que uma propriedade não pode ter valor *null*, basta utilizar uma anotação que representa a *facet* com essa função não é necessário para cada ponto da aplicação em que a atribuição de valor a essa propriedade, basta utilizar uma anotação. A própria ferramenta para tratamento de aspectos passa a ser responsável por interpretar a anotação e gerar automaticamente o código necessário para prover a semântica de cada anotação. O código gerado realiza a invocação de componentes organizados em uma biblioteca para tratamento das mesmas. A biblioteca para tratamento das anotações é componente desse trabalho e visa prover um conjunto de funcionalidades básicas necessárias a diferentes projetos de SC.

1.3 Organização dos Capítulos

O capítulo 2 revisa conceitos e trabalhos relacionados à abordagem definida nesse projeto. Os conceitos de orientação a aspectos (OA) e anotações de metadados, assim como suas aplicações, são apresentados para permitir uma melhor compreensão das propostas definidas nos capítulos 3 e 4. Esse capítulo também analisa e avalia XP.K, que define uma alternativa para implementação de SCs em OO. Parte das propostas organizadas em XP.K são extendidas e melhoradas para eliminação de uma série de desvantagens identificadas associadas à proposta citada.

O capítulo 3 apresenta a proposta de utilização de aspectos para a implementação do componente conceitual de modelo de conhecimento. Essa proposta busca apresentar uma solução padrão para a implementação desse modelo, evitando os problemas de soluções *ad-hoc* e da solução proposta em XP.K.

O capítulo 4 define a proposta de utilização de aspectos para a implementação dos outros dois componentes do modelo de conhecimento: componentes de tarefa e inferencial. Essa proposta busca apresentar uma solução padrão para a implementação desse modelo, evitando os problemas de soluções *ad-hoc* caracterizados na introdução.

O capítulo 5 apresenta o sistema utilizado para validação da proposta *K-Aspects*. Na primeira parte, é apresentada a implementação atual do sistema, sem utilização de *K-*

Aspects. A segunda parte apresenta o resultado da implementação utilizando *K-Aspects* e descreve vantagens e limitações dessa abordagem.

O capítulo 6 apresenta as conclusões do trabalho, destacando os resultados alcançados com esse trabalho e propondo possíveis trabalhos futuros.

2 CONCEITOS E TRABALHOS RELACIONADOS

Entre as metodologias citadas na introdução, XP.K destaca-se para o desenvolvimento de SCs por apresentar a OO como ponto de partida e propor uma abordagem padrão para a implementação de SCs. XP.K propõe uma abordagem chamada de KBeans para a implementação do componente conceitual, mas não detalha o desenvolvimento dos componentes de tarefa e inferencial. Apenas oferece uma ferramenta para aquisição de conhecimento. A seção 2.4, utilizando como referência principal Knublauch (2002), faz uma breve introdução dessa metodologia e identifica os principais problemas dessa proposta.

2.1 Orientação a Aspectos

A orientação a aspectos (OA) foi desenvolvida pela Xerox e é considerada uma complementação da programação orientada a objetos (KICZALES et al., 1997). Esse novo paradigma define uma abordagem para o tratamento de funcionalidades transversais às funcionalidades essenciais de um software. Exemplos de funcionalidades transversais frequentemente encontradas em aplicações (CACHO et al., 2006) são segurança, persistência, injeção de recursos, tratamento de exceções, concorrência, sincronização e auditoria. Por exemplo, em uma aplicação bancária, uma funcionalidade essencial seria a realização de uma transferência de dinheiro entre duas contas. Quando essa funcionalidade é implementada, funcionalidades transversais, como log da operação e validações de segurança também precisam ser tratadas. Essas funcionalidades transversais existem em diversos módulos da aplicação e desse modo, não podem ser encapsuladas em módulos unitários como proposto na OO. Utilizando somente a OO, essas funcionalidades acabam dispersas ao longo do código, dificultando a manutenção do sistema e aumentando o custo de manutenção.

A Figura 2.1 exemplifica a dispersão ocasionada pela implementação da funcionalidade de log (transversal ao objetivo de cada componente) em três componentes, as reticências correspondem a funcionalidade do módulo, já a chamada ao *Log* identifica a dispersão de código referente à auditoria. A dispersão de código ocorre porque é necessário manter um registro de métodos invocados, nesse caso, o engenheiro de desenvolvimento para cada registro de operação precisa acrescentar um trecho de código para registrar a operação. Na plataforma Java, existem diversos pacotes que podem oferecer esse recurso, entretanto, ao ser utilizado um determinado pacote, a chamada aos métodos que permitem a realização do *log* fica dispersa ao longo de todo o código da aplicação. Essa dispersão aumenta o custo de manutenção do código, pois, caso fosse necessário mudar de pacote para o suporte de *log*, seria necessário modificar a chamada aos métodos de *log* existentes em diversas classes de diversos módulos da aplicação. Além disso, o engenheiro de desenvolvimento poderia facilmente esquecer

de fazer a chamada para o *log* em um determinado método, não atendendo a um requisito transversal do sistema. Também é difícil verificar se os logs estão sendo feitos nas diversas situações que o exigem.

```
String transforma{
     ....
     Log.grava("transforma");
     ....
}
```

```
void adiciona{
   ....
   Log.grava("adiciona");
   ....
}
```

```
void envia{
   ....
   Log.grava("envia");
   ....
}
```

Figura 2.2: Exemplo de Dispersão devido a Funcionalidades Transversais.

Para tratar adequadamente essas funcionalidades transversais, permitindo encapsulá-las adequadamente, a OA introduz o conceito chamado de aspecto. Conforme Kiczales et al. (1997), um aspecto é "uma entidade de software que captura uma funcionalidade transversal a uma aplicação". Esse conceito permite a especificação de uma entidade que fornece uma funcionalidade utilizada ao longo de toda a aplicação, evitando que, ao ser necessário uma modificação nessa funcionalidade, toda a aplicação deva ser modificada. Um aspecto é complementado por três outros conceitos básicos de OA (KICZALES et al., 1997):

- *Pointcut*: identifica os pontos da aplicação em que uma funcionalidade transversal deve ser implementada. Por exemplo, na Figura 2.2, a definição do *pointcut call(...)* define que qualquer invocação de método deve ser registrada em um log;
- *Advice*: código adicional necessário para a realização de uma funcionalidade transversal. Por exemplo, na Figura 2.2, o termo *before* identifica que o log deve ser realizado antes da efetiva invocação do método, o *advice* realiza o log através da chamada *Log.grav*a(mensagem);
- Ponto de junção: ponto na execução de um programa no qual, em torno dele, um ou mais aspectos podem ser adicionados (KICZALES et al., 1997). Os principais pontos de junção suportados pela OA em uma aplicação são chamados de construtores, conclusão de execução de construtores, chamadas de métodos, conclusão de chamadas de métodos, atribuição de valor a uma propriedade, pré-inicialização de classe, inicialização de classe, tratamento de execução e, também, execução de um *advice*.

```
pointcut invocacaoMetodo() : call(*.*(..));

before() : invocacaoMetodo() {
    Log.grava('mensagem');
}
```

Figura 2.3: Exemplo de Aspecto para Log de Invocação de Método.

O uso de aspectos faz uso do processo chamado de costura (*weaving*). Esse processo, realizado pelo costurador (*weaver*), conecta os módulos da aplicação com os aspectos para que as funcionalidades transversais sejam disponibilizadas nos pontos da aplicação especificados nos *pointcuts*. Esse processo pode ocorrer em tempo de compilação ou em tempo de execução, conforme a implementação do costurador.

Apesar das vantagens de utilização de aspectos, deve-se destacar alguns pontos de limitação em relação ao uso da OO:

- **Depuração de Programas:** a depuração de programas que fazem uso desse paradigma é mais complexa que a de programas puramente OO. Isso ocorre porque o tratamento dos aspectos envolve a geração de código adicional, não conhecido pelo engenheiro de desenvolvimento e realizado pelo costurador. Caso o código gerado apresente algum problema ou provoque alguma modificação não prevista pelo engenheiro de desenvolvimento, a depuração do programa não irá apresentar o código fonte desse código, já que ele foi gerado pelo costurador;
- **Ambiguidade de Aplicação:** na proposta inicial de OA (KICZALES et al., 2001), os *pointcuts* são especificados através da definição de uma consulta sobre a aplicação. Por exemplo, "pointcut log() : execution(* *.set*(..))" identifica que o *advice* "log" deve ser invocado em todos métodos que iniciem com set. Contudo, essas declarações podem gerar ambiguidades, por exemplo, um programador pode definir um método iniciado com *set*, mas que não deveria ser aplicado o *log*. Para evitar esse problema e permitir que os engenheiros de desenvolvimento definam explicitamente onde os aspectos devem ser aplicados, OA pode fazer o uso do conceito de anotações de metadados disponível nas principais linguagens OO (Java utiliza o nome anotações, C# utiliza o nome atributo). Além disso, as anotações são necessárias para fornecer dados essenciais para a realização de um aspecto, por exemplo, para a persistência de uma determinada classe, a anotação pode fornecer o nome da tabela em que essa classe deve ser salva. A próxima seção detalha a contribuição de anotações para o uso da OA;

O suporte a OA em linguagens OO como Java e C# é disponibilizado através de extensões da linguagem providas por bibliotecas de código. Entre as extensões para Java destaca-se AspectJ (KICZALES et al., 2001), pois ela oferece um conjunto de ferramentas que auxiliam o uso de aspectos nessa linguagem. A validação desse trabalho é realizada usando AspectJ, pois o SC apresentado é desenvolvido em Java. Extensões semelhantes estão disponíveis para C#.

2.2 Orientação a Aspectos e Sistemas de Conhecimento

O uso de aspectos para a implementação de regras de negócios (CIBRÁN, D'HONDT e JONCKERS, 2003) e (CIBRÁN, D'HONDT e SUVÉE, 2005) busca isolar a implementação das regras de negócios do restante da aplicação. Objetivo é permitir facilitar a manutenção e rastreamento dessas regras, frequentemente complexas. O isolamento também é realizado para permitir que as regras possam ser adequadamente externalizadas em caso de necessidade. O isolamento das regras de negócios dos demais requisitos pode ser visto como análogo a necessidade de isolar, em SCs, a implementação do modelo de conhecimento em relação a implementação dos demais requisitos, para melhor manutenção e rastreabilidade.

Em D'Hondt, Meuter e Wuyts (1999), a utilização de aspectos para definição de conhecimento de domínio é proposta através do paradigma lógico. Entretanto, apenas experimentos bastante reduzidos foram realizados utilizando aspectos e lógica e a distância entre o paradigma lógico e orientado a objetos dificulta a utilização dessa proposta, já que exigiria o mapeamento entre os diferentes paradigmas.

Em Filman (2000) é apresentada uma proposta para organizar os sistemas em componentes integrados via aspectos, para melhor isolamento entre cada componente e

suas funcionalidades. O objetivo do isolamento é melhorar o processo de síntese de novos sistemas, a partir da combinação de uma série de componentes.

Aspectos também têm sido utilizados para estender ferramentas para modelagem de conhecimento. Em Eriksson (2004), é proposta uma abordagem para extensão da ferramenta Protégé usando aspectos. Inicialmente essa ferramenta é estendida usando OO, entretanto, quando uma nova versão é lançada, novamente a extensão deve ser conectada ao código-fonte original da ferramenta, tornando o processo lento e repetitivo. Usando aspectos, esse processo pode ser automatizado, já que a costura entre a extensão e a nova versão passa a ser realizada diretamente por ferramentas da OA.

2.3 Anotações de Metadados

Anotações são elementos tipados (*type-safe*) que permitem a adição de metadados a classes, métodos, variáveis, parâmetros e pacotes (ANNOTATIONS, 2004). Esse mecanismo permite que partes de um código-fonte sejam *decoradas* para que, no caso de OA, um interpretador de aspectos identifique pontos em que funcionalidades transversais precisam ser tratadas e também tenha os dados necessários para esse tratamento. Por exemplo, na Figura 2.2, uma anotação @Log poderia ser declarada junto ao método para fornecer uma mensagem adicional ao log e não somente o nome do método. Esses dados são fornecidos através dos atributos das anotações.

Anotações nas linguagens Java e C# têm sido usadas para diversos fins relacionados a aspectos. Por exemplo, para tratamento de funcionalidades transversais de persistência (JSR 220B, 2005), documentação (ANNOTATIONS, 2004) e injeção de recursos (JSR 220A, 2005):

- Persistência: utiliza um conjunto de anotações para permitir, que nas próprias classes que devem ser persistidas, seja definido o modo como isso deve ser feito. Por exemplo, o uso da anotação *@Table(Name="Pessoa")* em uma declaração de classe, identifica que essa classe deve ser persistida na tabela *Pessoa*. A anotação *@Field(Name="idade")* sobre uma propriedade chamada de *age* identifica que essa propriedade deve ser persistida em um campo chamado *idade*. Essas anotações ao serem identificadas por um interpretador de anotações (por exemplo, implementado por aspectos) fornecem as informações necessárias para que o costurador dos aspectos saiba, respectivamente, em qual tabela persistir os dados de pessoa e qual campo armazenar a idade;
- Documentação: uma anotação *@RequestForEnhancement(id=2868724, synopsi ="Melhorar identação")* pode indicar que determinado código precisa ter sua identação melhorada;
- Injeção de recursos: utilizada para que recursos sejam disponibilizados em tempo de execução, sem que o engenheiro de desenvolvimento tenha que repetidamente implementar manualmente essa chamada ao recurso. Por exemplo, *@Resource* permite que uma instância de um determinado componente seja automaticamente disponibilizada em uma variável (JSR 220A, 2005). Para que isso ocorra, primeiro anota-se uma variável com a anotação citada, após, o interpretador dessa anotação gera o código necessário para que esse recurso seja instanciado e disponibilizado na variável anotada. A declaração

@Resource(type="javax.mail.Session") sobre uma variável identifica que uma sessão de serviço de envio de e-mails deve ser disponibilizada nessa variável.

Para melhor compreensão da estrutura das anotações, a Figura 2.4 apresenta o código-fonte da anotação para realização de *log*. O código-fonte mostra outras anotações, fornecidas pela linguagem Java (ou C#), que são utilizadas para a definição da anotação *Log*. A anotação *Retention* serve para definir a política de retenção de anotações declaradas no código-fonte, por exemplo, a política *RUNTIME* identifica que a anotação *Log* será disponibilizada junto ao código binário das classes em que ela for declarada, caso tivesse sido utilizada a política *SOURCE*, essa anotação estaria disponível somente no código-fonte. A anotação *Target* identifica em quais elementos da linguagem a anotação pode ser aplicada, no caso de *Log*, a definição do tipo do elemento como *TYPE* define que ela pode ser usada somente na declaração de métodos, o que corresponde a classes e interfaces na linguagem Java. Se fosse utilizado *FIELD*, essa anotação só poderia ser utilizada em propriedades. Se um engenheiro de desenvolvimento tentar utilizar a anotação *Log* em uma propriedade, o próprio compilador da linguagem informará o erro, sem necessidade de extensão. As propriedades declaradas no corpo da anotação são disponibilizadas para preenchimento pelo engenheiro de desenvolvimento, no caso de *Log*, a propriedade *message* permite que seja definida a mensagem que será gravada, quando o método anotado for chamado.

```
package com.example;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Retention(RetentionPolicy.RUNTIME)
@Target(value = {ElementType.METHOD})
public @interface Log {
	String message();
}
```

Figura 2.4: Código-fonte da Anotação *Log*.

As anotações de metadados, após definidas, são acessíveis em tempo de compilação e de execução. O acesso em tempo de compilação é necessário para que as ferramentas de tratamento dos aspectos identifiquem pontos em que as funcionalidades transversais precisam ser gerenciadas e tenham as informações necessárias para o tratamento. O acesso aos metadados em tempo de execução, pode ser feito, tanto em Java, quanto C# através de reflexão estrutural. Reflexão é definida como o processo que permite um programa observar e modificar sua própria estrutura e comportamento (MAES, 1987). Esse tipo de reflexão permite que, em tempo de execução, o próprio software consiga acessar informações sobre os tipos abstratos de dados que o compõem (informações de classes, métodos) e, desse modo, pode alterar o seu comportamento (DEMERS e MALENFANT, 1995). A Figura 2.5 ilustra o uso de reflexão para obtenção de todas as anotações que existem em uma determinada classe.

```
Pessoa p = new Pessoa("Eduardo");
Annotation[] annotations = p.getClass().getAnnotations();
```

Figura 2.5: Código-fonte da obtenção de anotações de uma classe via reflexão.

2.4 Orientação a Objetos e Sistemas de Conhecimento

O paradigma OO tem sido amplamente utilizado na construção de sistemas; entretanto, devido às características de SCs, para ser utilizado nesse desenvolvimento desse tipo sistema, extensões ou conversões são necessárias. Essa seção detalha trabalhos relacionados a construção de SCs que fazem uso da OO.

2.4.1 Sistemas Orientados a Objetos para Processamento de Regras

Em Meditskos e Bassiliades (2006) e Bassiliades e Vlahavas (2006), O-DEVICE e R-DEVICE são respectivamente propostos. Esses sistemas orientados a objetos são voltados para a realização de raciocínio sobre OWL e RDF, linguagens tradicionais de modelagem de conhecmento. Ambos os sistemas utlizam a linguagem C Language Integrated Production System (CLIPS) e convertem modelos definidos em OWL e RDF para objetos da linguagem CLIPS, chamados de COOL. A conversão para OO realiza o mapeamento de classes OWL/RDF para classes OO e propriedades de classes OWL/RDF para atributos de classes OO. *Facets,* axiomas precisam também ser expressos através de regras, a obrigação da definição de *facets* e axiomas por regras é uma desvantagem, pois o processo de conversão não peserva a estrutura original do modelo, por exemplo, um *facet* passa a ser uma regra, o que dificulta a rastreabilidade do modelo e consequentemente a sua manutenção. Essa conversão para OO busca otimizar o processo de raciocínio, já que encapsula todos os elementos associados a uma classe na própria classe, diferentemente do uso de triplas que não o fazem. O uso de triplas exigiria junção de diversas triplas para a constituição de uma única classe.

A principal desvantagem dessa abordagem é que CLIPS foi proposta quando se acreditava que SCs poderiam ser construídos apenas com uso de regras. Entretanto, sabe-se atualmente que é inviável a construção de SCs complexos apenas com regras, já que o número tende a ser exponencial e torna inviável a manutenção dos sistemas. CLIPS não oferece um conjunto completo de bibliotecas para construção de interface gráficas, comunicação com outros sistemas e portabilidade para diferentes sistemas operacionais. Essa linguagem parece adequada apenas para SCs de pequeno porte, que não apresentam um grande número de regras.

2.4.2 XP.K

XP.K busca resolver a limitação das metodologias focadas na construção de modelos de conhecimento de forma isolada dos aspectos de implementação. SC devem ser integrados ao ambiente de TI pré-existente, desse modo, os componentes de conhecimento precisam interagir com componentes escritos em linguagens amplamente usadas comercialmente, por exemplo, Java e C#. Esse isolamento e a necessidade de acessar recursos de linguagens usuais provocam a criação de sistemas híbridos que utilizam múltiplos paradigmas, linguagens e ferramentas. Sistemas híbridos são de difícil manutenção, pois geram redundâncias de implementação, por exemplo, um modelo é implementado com classes da orientação a objetos e também em uma linguagem funcional para aplicação de técnicas de IA. Sistemas híbridos também acabam muitas vezes sofrendo com os problemas de baixo desempenho encontrados em linguagens de inteligência artificial (PREE, BECKENKAMP e VIADEMONTE, 1997).

XP.K aborda a possibilidade de utilização da orientação a objetos como ponto de partida para o desenvolvimento de SCs, integrando idéias da Engenharia de Conhecimento e de metodologias ágeis. O desenvolvimento de SC usando XP.K é

baseado no uso do paradigma da OO nas fases de modelagem e implementação. O uso da OO visa atender o requisito de suporte à evolução através de artefatos de modelagem facilmente manuteníveis, reduzindo o custo de mudança e facilitando o uso da prototipação. O suporte à retroalimentação necessita de ferramentas eficientes e reduzido tempo de transição entre modelos conceituais e código fonte. O suporte à colaboração requer uma linguagem de modelagem facilmente comunicável entre especialistas do domínio e os demais participantes de um projeto. Para alcançar isso, XP.K utiliza a abordagem de centralizar o processo de desenvolvimento ao redor de um artefato, o componente conceitual orientado a objetos (ontologia). Objetos estão relativamente próximos do modo de pensamento de um especialista e também não estão distante de código executável, diferentemente de linguagens de modelagem tradicionais (RDF, OWL, entre outras). Entretanto, conforme Abel (2001), o desenvolvimento de um SC normalmente requer um modelo de conhecimento composto por um conjunto de componentes para que o conhecimento tratado seja explicitado o suficiente para ser implementado em um programa de computador. Múltiplos componentes são necessários devido à complexidade do conhecimento e cada componente possui construtos diferentes; entretanto em XP.K isso não é abordado.

Ontologias e linguagens orientadas a objetos possuem características semelhantes como alta coerência interna, baixo acoplamento, extensibilidade, uso de categorias naturais e nomes (ex.: classes, propriedades). Ontologias e linguagens orientadas a objetos utilizam abordagens similares para a declaração de estruturas estáticas como classes (conceitos), hierarquia de classes (usando herança), atributos, relações e instâncias. Apesar das semelhanças citadas, existem diferenças significativas na representação semântica. Ontologias permitem definir a semântica declarativamente através de restrições, axiomas e regras. Na orientação a objetos, a semântica é implementada imperativamente através de métodos (sequência imperativa de comandos), apesar de UML representar a semântica da aplicação por casos de uso, diagramas de atividades, diagramas de estado, além de diagramas de objetos com seus atributos e métodos.

Mesmo considerando a expressividade de ambas as abordagens, a abordagem declarativa é mais adequada para a modelagem conceitual. Por exemplo, em um modelo conceitual de relações familiares, diversos pressupostos semânticos precisam ser representados [E.g.: número máximo de filhos (restrição/*facet*), idades dos filhos deve ser menor que a dos pais (axioma), a definição de um valor para esposa implica que o estado civil deixe de ser casado (regra)]. Esses pressupostos são importantes para definir o domínio válido das instâncias e permitir a sua reutilização. Contudo, em uma linguagem imperativa, por exemplo, Java, os construtos usados no componente do modelo conceitual não são suportados e a implementação não preserva as estruturas definidas no modelo conceitual. Por exemplo, a restrição sobre idade de uma pessoa, pode ser implementada por uma cláusula **if**; entretanto, no modelo conceitual ela é expressa somente por um *facet*, que não declara a forma como isso é feito. Desse modo, as estruturas originais são implementadas de modo implícito, principalmente para que todos possíveis caminhos de execução respeitem as definições do modelo conceitual. Essa representação implícita apresenta os seguintes problemas:

- Complexidade dos caminhos de execução torna a aplicação muito suscetível a erros. Qualquer alteração no código pode gerar novos caminhos que deixam de atender os pressupostos semânticos;

- A manutenção da aplicação torna-se bastante complexa e torna-se praticamente impossível realizar a engenharia reversa do código para modelos conceituais, pois as estruturas não são preservadas entre o modelo e sua respectiva implementação (DEMEYER, DUCASSE e TICHELAAR, 1999);

A origem dos problemas está na implementação não transparente desses pressupostos semânticos. Para evitar isso, XP.K propõe utilizar a orientação a objetos para permitir a modelagem conceitual e sua implementação com transparência semântica.

Transparência semântica significa que humanos e máquinas avaliam os modelos sem ambiguidade (tendo uma precisa interpretação de significado) e que satisfazem simultaneamente um conjunto de restrições (COVER, 1998). No caso de linguagens declarativas como UML, pressupostos semânticos podem ser adicionados em comentários ou em *Object Constraint Language* (OCL); entretanto, UML não é executável e OCL não é amplamente utilizada na Engenharia do Conhecimento. Para tratar esse problema e permitir que o componente do modelo conceitual seja desenvolvido no paradigma OO, XP.K propõem KBeans, uma proposta para a construção de modelos conceituais utilizando a orientação a objetos. A próxima seção detalha KBeans para posterior análise.

2.4.2.1 KBeans

KBeans é uma proposta para adição de transparência semântica a estruturas de classes da orientação a objetos através da extensão de JavaBeans (KNUBLAUCH, 2002). Para máquinas, a transparência semântica pode ser reduzida à informação formal sobre elementos de um modelo de objeto e suas relações. Desse modo, o elemento chave para o desenvolvimento de objetos transparentes semanticamente é prover essa informação sobre os objetos. Uma forma de prover estes dados, os metadados, em linguagens como Java e C#, é por reflexão (MAES, 1987), sendo assim possível adicionar transparência semântica a classes dessas linguagens. Essa proposta, implementada em Java, utiliza um conjunto extenso de convenções de código para representar os metadados diretamente no código fonte e uma API baseada em reflexão para extrair esses metadados. Cada KBeans pode ser visto como um componente de conhecimento reusável, por exemplo, podemos criar um KBeans para representar uma pessoa e suas relações familiares. Além de permitir a construção de componentes de conhecimento, essa proposta oferece um catálogo pré-definido de tipos de *facets* de KBeans (Exemplo: *minValue, maxValue, maxCardinality*). *Facet*, na modelagem de conhecimento por frames, corresponde a uma relação ternária para expressar restrições em valores de *slots* (semelhante a uma propriedade de uma classe), por exemplo, a *facet minValue* define o valor mínimo que uma propriedade pode assumir.

Para utilizar uma *facet*, por exemplo, *minValue*, é necessário seguir estritamente a convenção de código *<property>MinValue*. No caso, para a propriedade *idade*, a restrição do menor valor possível é feita através de *idadeMinValue = 0*.

Essa proposta suporta a transparência semântica de classe de linguagens orientadas a objetos como Java e C# através da declaração e avaliação de *facets* (metadados). KBeans provê essa transparência no tempo de construção das classes através de convenções que permitem aos *parsers* extrair declarações de ontologias das correspondentes classes Java e, então, permitir a conversão de ontologias UML para Java e vice-versa. Os *parsers* necessitam analisar métodos e propriedades que estejam

de acordo com as convenções de facets. Essas convenções, em tempo de execução, podem ser analisadas através do mecanismo de reflexão, permitindo que as restrições semânticas sejam identificadas e analisadas para cada instância de KBeans.

2.5 Análise Crítica dos Trabalhos Relacionados

Entre os trabalhos citados acima para implementação do modelo de conhecimento na OO. Destacam-se O-DEVICE, R-DEVICE e a abordagem KBeans, entretanto, todos apresentam o problema da ambiguidade do mapeamento dos construtos das linguagens de modelagem para a implementação em OO.

Nos sistemas O-DEVICE e R-DEVICE, o problema de ambiguidade pode ser identificado devido o mapeamento de diversos construtos para regras. Por exemplo, todos os *facets*, axiomas e regras são mapeados para regras. Isso ocorre porque esses sistemas foram baseados em CLIPS, uma linguagem desenvolvida quando se acreditava que SCs poderiam ser construídos apenas com o uso de regras.

A abordagem KBeans apresenta uma proposta para o tratamento dos construtos das linguagens de modelagem em OO, entretanto, os seguintes problemas podem ser apontados, conforme Castro, Abel e Price (2009) – artigo do autor dessa dissertação:

- O uso de convenções de código para representar conhecimento na própria classe cria ambiguidade sobre a função dos diferentes elementos: uma classe pode representar um conceito ou uma relação; um método pode ser a implementação de uma regra ou de um axioma; uma propriedade de uma classe pode representar um atirbuto de um conceito ou uma facet. Por exemplo, uma propriedade chamada *salarioMinValue* pode ser lida como o menor valor permitido para o salário de uma pessoa (nesse caso é um facet para o atributo) ou como o menor valor de salário que uma pessoa recebe mensalmente (nesse caso é apenas um atributo). Apesar de KBeans tentar solucionar o problema da ambiguidade entre a implementação do componente do modelo conceitual e o restante da implementação, o uso de padrões acaba mantendo o problema (diferentes construtos são mapeados para um mesmo elemento, por exemplo, atributos e facets são implementados como propriedades). Essa ambiguidade impede que o modelo abstrato seja completamente obtido pela leitura da implementação;
- A biblioteca exige que todos os métodos e propriedades de restrições sejam declarados como públicos, dificultando a leitura da classe pelo engenheiro de desenvolvimento e podendo gerar problemas de modificações indevidas a propriedades;
- Ocasiona a proliferação de uma grande quantidade de métodos auxiliares nas classes que implementam o componente do modelo conceitual do SC. Por exemplo, para cada propriedade que tem um facet associado, uma segunda propriedade precisa ser definida. Isso aumenta a complexidade do código, dificultando a manutenção do sistema e a leitura de diagramas UML dessas classes;
- Exige o domínio de uma ampla convenção de código. Isso exige um forte treinamento da equipe de desenvolvimento, longas sessões de revisão de código e torna o código produzido bastante suscetível a erros, pois se a convenção não é respeitada, o código não é interpretado corretamente pela biblioteca de software do KBeans. Por exemplo, a declaração de uma facet como *idadeMinimumValue*

é inválida, o correto seria *idadeMinValue*; entretanto, o interpretador da linguagem não é capaz de identificar esse erro;

- A convenção de código obrigatória para a correta interpretação pela biblioteca de software do KBeans aumenta significativamente a chance de inserção de erros quando refatoração é utilizada. Refatoração, conforme Fowler (1999), é definida como o processo de modificação de um sistema no qual o comportamento externo do sistema não é alterado, mas sua estrutura interna é melhorada. Por exemplo, a refatoração de uma propriedade chamada *idade* para *tempo*, que possui uma facet *idadeMinValue*, geraria um problema na implementação, pois a facet *idadeMinValue* não seria automaticamente renomeada para *tempoMinValue*. O mecanismo de refatoração não identifica dependências entre propriedades e facets;
- Obriga a extensão de classe padrão (AbstractKBeans) impedindo o uso de herança, mesmo que simples, entre conceitos do modelo conceitual;
- Destaca-se também que KBeans não utiliza nenhuma linguagem para definição de axiomas, o que exige a implementação diretamente em código-fonte da linguagem OO.

KBeans foi proposto quando as linguagens Java e C# ainda não proviam mecanismos para inserção de metadados em classes e seus elementos (seção 2.3), nem a OA (seção 2.1) oferecia ferramentas para utilização nessas linguagens. Posteriormente, essas linguagens passaram a suportar o conceito de anotações de metadados (ANNOTATIONS, 2004), um mecanismo para inserção de metadados tipados em classes e seus elementos e ferramentas como AspectJ (KICZALES et al., 2001) foram desenvolvidas para o suporte a aspectos nas linguagens mencionadas anteriormente.

Anotações e aspectos são utilizados como base para a definição de *K-Aspects*, pois podem reduzir/eliminar os problemas identificados acima. Conforme detalhado no capítulo 3, as anotações permitem a implementação dos construtos das linguagens de modelagem sem ambigüidades. A possibilidade de acesso a essas anotações em tempo de execução, capítulo 4, possibilita que as mesmas sirvam de base para a implementação dos componentes de tarefa e inferencial independentes do domínio da aplicação, por exemplo, uma tarefa para realização de interpretação na área da geologia ou medicina (Capítulo 5).

3 K-ASPECTS: ASPECTOS PARA IMPLEMENTAÇÃO DO COMPONENTE CONCEITUAL

A existência de um série de metodologias para desenvolvimento de SCs, conforme apresentado no capítulo 1, ainda não atende adequadamente a fase de desenvolvimento de sistemas que necessitem modelar conhecimento e inferência e evoluir requistos com agilidade. A fase de desenvolvimento precisa tratar adequadamente a implementação dos componentes do modelo de conhecimento e os problemas de comunicação entre os diferentes agentes de condução do projeto: analistas, engenheiros de desenvolvimento e engenheiros de conhecimento. Esse capítulo define uma abordagem para tratar do mapeamento do componente conceitual de sistemas de conhecimento para uma implementação. As soluções propostas neste capítulo foram inicialmente analisadas pelo autor no artigo (CASTRO, ABEL e PRICE, 2009).

A abordagem proposta permite a implementação do componente conceitual do SC usando o paradigma OO estabelecendo um caminho padrão para a implementação desse modelo. O principal objetivo dessa abordagem é prover uma forma para que o componente conceitual e o código relacionado ao modelo não sejam distintos a tal ponto que não seja possível identificar claramente as estruturas definidas no modelo na sua respectiva implementação. Uma abordagem padrão permite que detalhes de implementação sejam mais facilmente identificados e compreendidos tanto por programadores recém-chegados quanto programadores sazonais desse tipo de sistema. Isso pode reduzir o custo de manutenção e reduzir o risco de inserção de defeitos por problemas de compreensão entre modelo e implementação.

A proposta de implementação do componente conceitual definida nesse capítulo é composta por um conjunto de aspectos (*K-Aspects*) e um conjunto de anotações de metadados (*K-Annotations*). As anotações são utilizadas para distinguir claramente os elementos do componente conceitual em relação aos demais elementos que compõem um SC. Por exemplo, uma anotação @*Concept* identifica, sem ambiguidades, que uma determinada classe representa um conceito. As anotações para implementação do componente conceitual, definidas na seção 3.2, permitem que o engenheiro de conhecimento identifique o seu modelo no código implementado e, também, identificam os pontos em que o interpretador dessas anotações precisa atuar para a realização do processamento de *facets*, axiomas e chamada às tarefas.

A possibilidade do engenheiro de conhecimento identificar facilmente os construtos do componente conceitual diretamente na implementação do mesmo, ou em documentação gerada automaticamente, aumenta significativamente a rastreabilidade entre o modelo e sua correspondente implementação. Atualmente, a ausência de elementos de marcação (*anotações* nesse trabalho) dificulta imensamente o processo de verificação da consistência entre o modelo de conhecimento e sua correspondente

implementação, pois a implementação faz uso de uma série de convenções que mapeiam os construtos para métodos e propriedades auxiliares (quebra do princípio da preservação de estruturas (SCHREIBER et al., 2000)). A falta de preservação de estruturas acaba dificultando a comunicação entre o engenheiro de conhecimento e os engenheiros de desenvolvimento, pois enquanto o engenheiro de conhecimento identifica claramente as estruturas no modelo, os engenheiro de desenvolvimento precisam conhecer detalhadamente uma série de convenções e não utilizam os construtos definidos no modelo.

Além de servirem para preservar as estruturas definidas no modelo de conhecimento e estabelecer um protocolo de comunicação entre engenheiro de conhecimento e de desenvolvimento, as anotações marcam claramente o código-fonte associado à implementação dos componentes do modelo de conhecimento. Essa marcação auxilia na identificação de pontos da aplicação (*pointcuts*) em que uma funcionalidade transversal deve ser implementada. No caso da implementação do modelo de conhecimento, pode-se citar validações de valores aceitos para um determinado atributo. As anotações também fornecem os dados necessários para que os aspectos realizem o tratamento adequado da semântica associada a cada construtor.

Além disso, as anotações são usadas para distinguir claramente os elementos do componente conceitual do restante do código. A marcação clara desses elementos é essencial para que seja possível identificar todos os pontos da aplicação em que aspectos do componente conceitual estão presentes. As anotações fornecem os dados necessários para que os aspectos sejam adequadamente tratados. Os aspectos identificados para a modelagem conceitual visam evitar a dispersão de código gerada pelo tratamento da semântica de facets, regras e axiomas e padronizar o tratamento das mesmas. Reduzindo custo de manutenção e aumentando a reusabilidade dos componentes gerados.

Na seção 3.1 é definido o processo padrão de utilização da abordagem proposta. Na seção 3.2 é definida a biblioteca de anotações utilizadas para a marcação dos elementos do componente conceitual e implementação de aspectos relacionados a esse componente. Na seção 3.3 é apresentada a biblioteca que realiza o tratamento da semântica das anotações definidas na seção 3.2. Na seção 3.4 é apresentada a biblioteca de aspectos identificados para o tratamento da implementação do componente conceitual.

3.1 Processo de Uso de K-Annotations

O processo de uso de *k-annotations* define os passos necessários para que a implementação OO do componente do modelo conceitual seja gerada com sucesso. Esse processo é definido para ser reutilizado em diferentes projetos de SCs, independente do domínio de aplicação.

O processo, Figura 3.1, inicia quando o engenheiro de desenvolvimento recebe uma especificação do modelo conceitual. É necessária a implementação da especificação usando OO acrescido de *k-annotations*, definidas na seção 3.2. Após a implementação do componente do modelo conceitual, o interpretador de aspectos recebe como entrada a implementação elaborada pelo engenheiro de desenvolvimento, a biblioteca de anotações, a biblioteca de aspectos e a biblioteca para tratamento de *facets*, axiomas e regras (as bibliotecas são reusáveis em diferentes projetos). Se o código não apresentar nenhum problema, o interpretador gera o *byte-code* (código intermediário usado por

linguagens interpretadas), esse código intermediário da implementação do modelo e dos aspectos é então costurado pelo *weaver* (fornecido junto com o interpretador de aspectos) para a implementação da semântica associada às anotações via aspectos. O resultado da interpretação é a implementação executável do componente conceitual em uma máquina virtual (Java ou C#). Além da implementação do componente conceitual, o compilador também invoca a ferramenta *KA-DocGen*, que gera a documentação da implementação do componente conceitual, para que mais facilmente seja possível realizar revisões entre a especificação e a implementação. Na Figura 3.1, os retângulos sombreados identificam elementos desenvolvidos nesse trabalho.

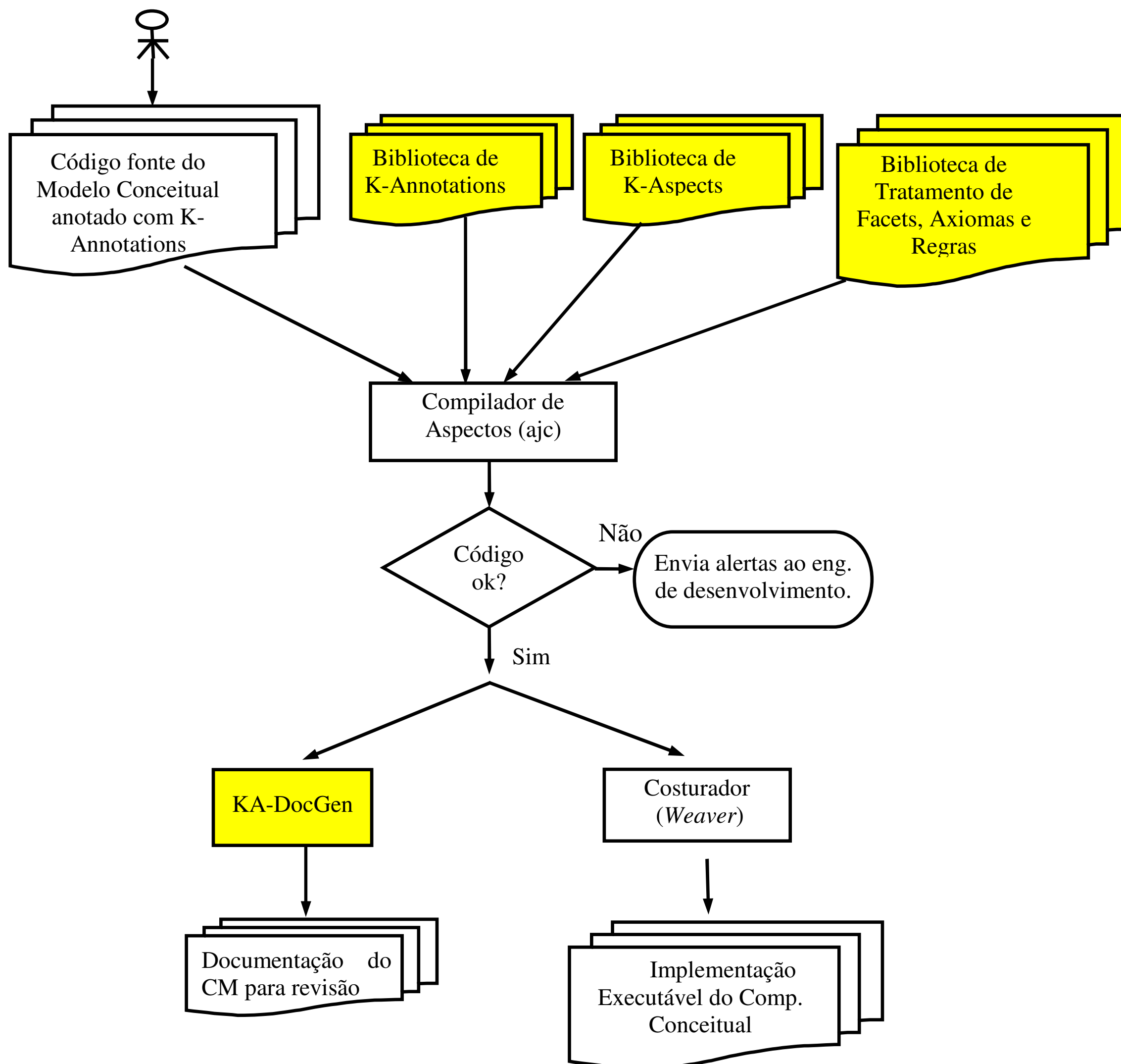

Figura 3.1: Processo de Uso de *K-Annotations*.

A sequência de passos realizados no tratamento de anotações é exemplificada na Figura 3.2. O primeiro passo decorre da interface gráfica que invoca um método para modificação do valor de um atributo do conceito Pessoa. Antes da atribuição do valor ao atributo, o *advice evalAxiom* do aspecto de validação de axiomas é acionado; esse *advice* invoca o método *manage* da biblioteca de tratamento de axiomas (*AxiomManager*). Esse método utiliza a biblioteca JEP (JAVA, 2009) para processar a especificação do axioma e validá-lo. A biblioteca *Java Expression Parser* (JEP), definida em (JAVA, 2009), possibilita o processamento de axiomas em Java e bibliotecas semelhantes estão disponíveis para C#. Essa biblioteca pode ser utilizada já

que sua gramática é compatível com a gramática de CML para definição de axiomas, evitando que o programador tenha que converter expressões definidas no modelo de conhecimento.

Na ilustração, a validação identificou que o axioma foi atendido, então a sequência de retorno indica o valor *VERDADE* e, por fim, o valor é atribuído ao atributo *Idade* (propriedade *pIdade*). Caso a validação tivesse indicado o não-atendimento ao axioma, o valor não seria atribuído ao atributo e o tratamento de exceção seria lançado.

No processo apresentado, caso o engenheiro de desenvolvimento não necessite estender os aspectos fornecidos como biblioteca, não há necessidade de utilização do compilador de aspectos, apenas é necessário utilizar o costurador para a implementação da semântica associada às anotações via aspectos.

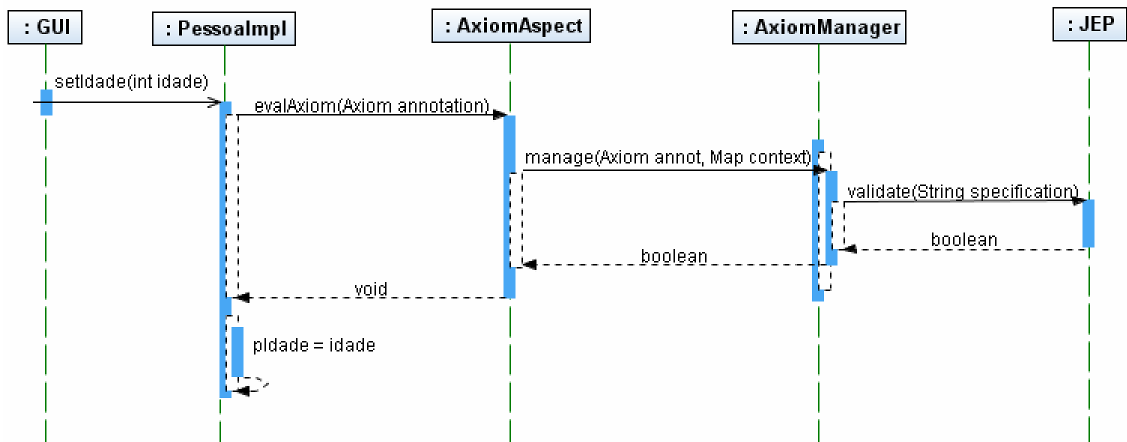

Figura 3.2: Diagrama de Sequência para Validação de Axioma.

3.2 Biblioteca de K-Annotations

Uma abordagem padrão para o desenvolvimento de SCs usando OO com apoio da OA precisa suportar os construtos existentes em linguagens de modelagem de SCs. Destaca-se que esses construtos não estão disponíveis nativamente na OO e muitas vezes são tratados por soluções *ad-hoc* que não preservam as estruturas dos mesmos, dificultando severamente a manutenção desses sistemas e gerando problemas, conforme definido no capítulo 1.

Os desvantagens apontadas na seção 2.5. podem ser reduzidas usando a abordagem de *k-annotations*. Anotações podem reduzir ou eliminar o uso de convenções de código, pois cada anotação tem um papel claramente definido e facilita a identificação dos aspectos que precisam ser tratados, permitindo a geração automática de código. Anotações são verificadas pelo interpretador de linguagem, ao passo que convenções de código não o são (PIVETA et al., 2007). O problema de ambiguidade entre os papéis de propriedades e métodos é evitado com eliminação/redução do uso de convenção de código. Os construtos do componente conceitual são claramente identificados.

A proliferação de propriedades e métodos auxiliares é reduzida usando anotações. Essas anotações podem ser facilmente interpretadas pelo interpretador de aspectos responsável pela geração de código, e também pela ferramenta responsável por gerar documentação.

A equipe de desenvolvimento não precisa mais utilizar um amplo conjunto de convenções de código não verificado pelo interpretador. Anotações, sendo validadas pelo interpretador, evitam erros cometidos pelos engenheiros de desenvolvimento.

O problema da refatoração de código é reduzido. Por exemplo, uma facet que restringe um valor mínimo de uma propriedade é definida usando *@FacetMinValue*. Se a propriedade é renomeada, isso não afeta a facet.

O suporte a esses construtos, não nativamente suportados pela OO, pode ser atingido a partir da definição de um conjunto de anotações que representam claramente esses construtos no domínio da OO entre os demais elementos que atendem outros requisitos (ex.: persistência, auditoria). A implementação do componente conceitual precisa tratar a semântica desses construtos. O correto tratamento desses construtos é necessário para que o SC tenha o comportamento esperado, conforme o modelo de conhecimento. Os construtos tratados nesse trabalho foram extraídos de duas linguagens amplamente utilizadas na modelagem de conhecimento: Frames (FIKES e KEHLER, 1985) e CML (SCHREIBER et al., 2000), visando permitir a utilização dessa abordagem na grande parte dos projetos de SCs.

A seguir são apresentados os construtos suportados e suas relações (a Figura 3.3 apresenta o metamodelo dos construtos e suas relações):

- **Conceitos:** representam *classes* de objetos no domínio de aplicação. Semelhante ao termo *entidade* da modelagem E-R e *classe* da orientação a objetos. Um conceito obrigatoriamente deve ter um nome único e suporta herança simples (esta restrição é devida ao uso na implementação de linguagens de OO como Java e C#, que suportam apenas herança simples). Um conceito também pode ser parte de outro conceito e pode possuir atributos;
- **Relações:** representam a ligação entre diferentes conceitos e devem possuir um nome. Conceitos são argumentos de relações. Por exemplo, a relação ATENDIDO-POR pode realizar a ligação entre médico e paciente;
- **Atributos/Slots:** representam propriedades de um conceito ou relação. Um atributo possui um nome e um tipo associado. Um atributo pode pertencer a um conceito ou a uma relação. Por exemplo, a idade de uma pessoa. O termo *slot* é utilizado em frames;
- **Facets:** definem restrições (*constraints*) para os valores válidos para atributos. Por exemplo, valor mínimo para idade é 0. As *facets* suportadas nesse trabalho foram extraídas de (KNUBLAUCH, 2002) e correspondem ao principal conjunto de *facets* suportadas em linguagens de modelagem conceitual;
- **Axiomas:** especificam relações matemáticas que precisam ser sempre verdadeiras. São aplicados a atributos. Por exemplo, filho.idade < pai.idade;
- **Regras:** definem regras de implicações aplicadas a atributos ou relações. Por exemplo, podemos definir a seguinte regra relacionada à idade:
    - SE pessoa.idade > 18 ENTÃO pessoa.maioridade = VERDADEIRA

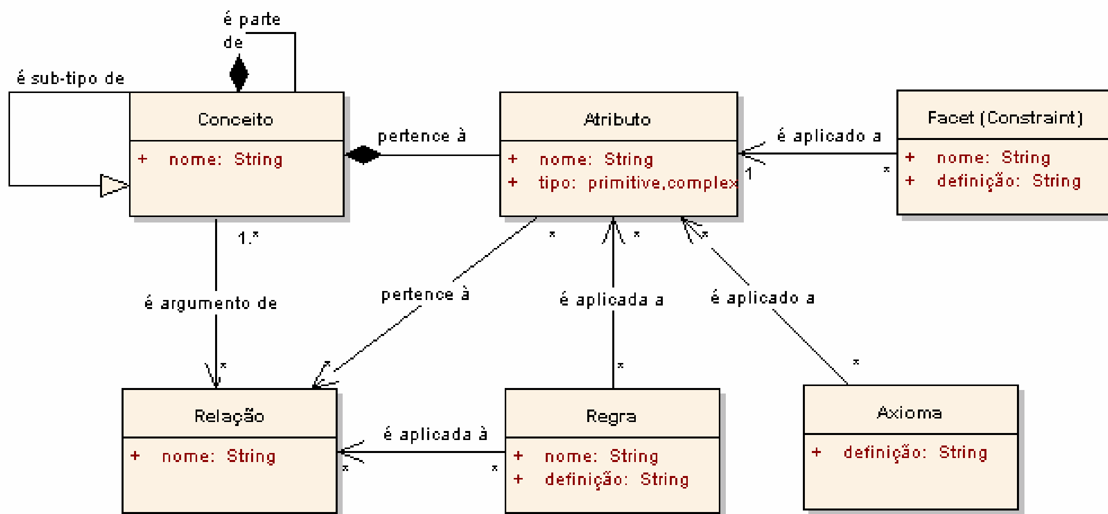

Figura 3.3: Metamodelo dos Construtos do Componente Conceitual.

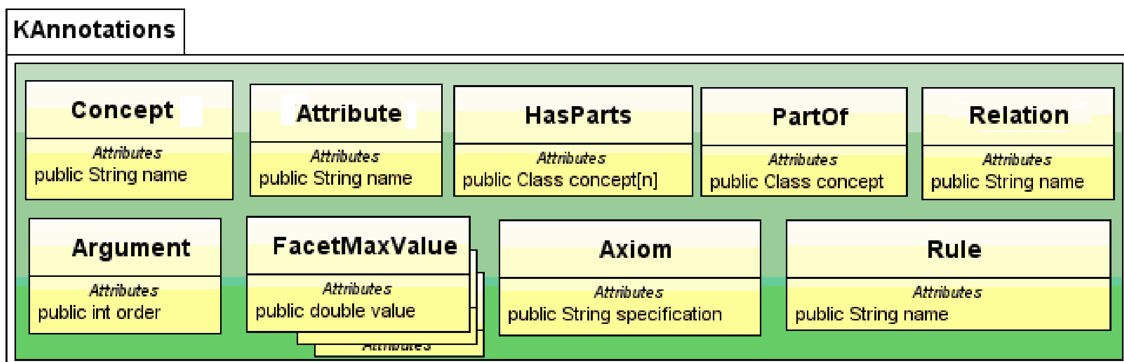

Figura 3.4: Visão Lógica das *K-Annotations*.

Os construtos definidos acima correspondem ao seguinte conjunto de *k-annotations* - Figura 3.4 (a validação e o estudo de caso são apresentados no capítulo 5). Recomenda-se que qualquer classe ou propriedade que esteja implementando um elemento do modelo de conhecimento deve apresentar exatamente o mesmo nome definido no modelo, por exemplo, um conceito *Pessoa* deverá implementado por uma classe chamada de *Pessoa*. A exigência do mesmo nome busca garantir a consistência e o rastreamento entre modelo e implementação. Caso não seja possível manter o mesmo nome, por exemplo, quando K-Annotations são aplicadas em um sistema legado, o atributo *name* deve ser preenchido com o nome definido no modelo. As seguintes K-Annotations são providas:

- **@Concept:** usada junto à declaração de classe para defini-la como um conceito. A interpretação dessa classe gera código necessário para monitorar que uma instância foi criada/destruída. O registro das instâncias válidas é necessário para a realização de inferências sobre as mesmas. Caso contrário, não seria possível tratar com diferentes conceitos em uma mesma inferência. Essa anotação também é processada pela KA-DocGen para gerar a documentação de conceito;

- **@Attribute:** usada junto à declaração de propriedade de uma classe (conceito ou relação) para defini-la como atributo. A interpretação dessa anotação gera código necessário para monitoração de troca de valores do atributo, pois os

aspectos monitoraram mudanças nos valores, acionando a validação de *facets*, regras e axiomas;

- **@HasParts:** define as partes de um conceito. Neste modelo, HasParts é uma relação não-reflexiva e transitiva entre classes que descreve uma partonomia não completa. Cada parte corresponde a uma classe. Optou-se por exigir que as classes sejam informadas e não somente o nome do conceito, para evitar problemas de grafia com nomes de conceitos. Por exemplo, se fosse permitido *@HasParts(parts={'Pessoa'})*, o engenheiro de desenvolvimento poderia digitar *pesoa* ao invés de Pessoa , inserindo um erro na anotação e quebrando a especificação do modelo. Como é obrigatório informar a classe do conceito, *@HasParts(parts={Pessoa.class})*, caso haja um erro de grafia, o interpretador da linguagem OO irá informar o erro, pois o interpretador não encontrará a classe especificada. Caso ocorra uma refatoração no nome da classe de um conceito, o interpretador automaticamente irá renomear todas as referências à classe renomeada, por exemplo, na anotação *@HasParts*;
- **@PartOf:** define que um conceito é parte de outro conceito. É uma relação não-reflexiva e transitiva entre classes e relação inversa de HasParts. Deve ser usada junto à declaração de classe. Por exemplo, quando a anotação *@PartOf(partOf=A.class)* é declarada em uma classe B, o interpretador de aspectos irá verificar se a classe B possui um construtor cujo único argumento válido é uma classe do tipo A. Se isso não for detectado, um erro é emitido ao engenheiro de desenvolvimento e o processo é encerrado. Essa verificação é necessária porque o relacionamento *part-of* define que uma classe parte de outra, só pode existir se a possuidora existir;
- **@Relation:** usada junto à declaração de uma classe para definir uma relação. A definição da aridade da relação é realizada através do uso da anotação @Argument. Por exemplo:
    - Para uma relação binária, 2 argumentos são necessários, nesse caso, a classe da relação binária deve possuir 2 propriedades anotadas com *@Argument(order=1)* e *@Argument(order=2)*;
    - Para uma relação ternária, 3 argumentos são necessários, nesse caso, a classe da relação ternária deve possuir 3 propriedades anotadas respectivamente com *@Argument(order=1)*, *@Argument(order=2)* e *@Argument(order=3)*;
- **@Argument(*order*):** o parâmetro *order*, obrigatório, identifica a ordem do atributo na relação. Essa anotação identifica o argumento de uma relação, pois é usada junto à declaração de uma propriedade de uma relação. Por exemplo:
    - Em uma relação binária 'atendido-por', teríamos como primeiro argumento um paciente e como segundo argumento um médico;
    - Em uma relação do tipo *n:m*, onde um paciente é atendido por vários médicos, a cardinalidade de cada argumento é definida pelas anotações *@FacetMinCardinality*/*@FacetMaxCardinality*;
- **@Facet<Função>:** todas anotações que iniciam por *Facet* são definições de restrições sobre os valores válidos para atributos/argumentos. Os parâmetros dessas *facets* variam conforme sua função. Essas anotações indicam ao

interpretador de aspectos a presença do aspecto de tratamento de *facets*. A interpretação dessas anotações gera o código necessário para tratar a semântica correspondente, evitando que o desenvolver tenha que manualmente realizar esse tratamento, evitando problemas de implementação devido a falhas do engenheiro de desenvolvimento e reduzindo custos de manutenção. Chamadas a bibliotecas que tratam a semântica dessas anotações são geradas pelo próprio interpretador de aspectos, evitando dispersão destas por toda a implementação do componente conceitual. A Tabela 3.1 sumariza o conjunto de *facets* suportadas, conjunto extraído do catálogo de KBeans (KNUBLAUCH, 2002);

- **@Axiom(*specification*):** definida junto a declaração de conceito para definir expressões matemáticas que devem ser sempre válidas. Essa anotação tem como parâmetro obrigatório a especificação do axioma. A especificação é realizada usando a gramática do JEP, definida em (JAVA, 2009) – ANEXO. Quando o interpretador de aspectos detecta essa anotação, ele identifica o aspecto para tratamento de axiomas. Nesse caso, o *advice* realiza a invocação da biblioteca de tratamento de axiomas toda vez que os valores dos atributos envolvidos no axioma são modificados;
- **@Rule(*value*):** definida junto à declaração de conceito para especificação de regras que são aplicadas sobre os atributos dos conceitos. Exige o preenchimento do parâmetro *value* que identifica a classe que implementa uma ou mais regras. Essa classe deve obrigatoriamente ter um método chamado de *runRules*, que contém as regras. Quando o interpretador de aspectos detecta essa anotação, ele identifica o aspecto para tratamento de regras. O *advice* desse aspecto é acionado para cada modificação de valores de atributos participantes da regra. Nesse caso, o *advice* invoca a classe de tratamento de regras. Essa invocação é necessária para verificar os valores e possíveis alterações devido à execução da regra. As regras são especificadas usando a sintaxe da linguagem OO usada para o desenvolvimento do SC (ex.: Java, C#). Exemplos dessas regras são apresentados na seção 5.3.1;

Um importante ponto a ser ressaltado é a gerência de exceções. A fase de modelagem não define como são tratadas as violações de conhecimento, por exemplo, durante a execução da aplicação, qual deveria ser o comportamento do sistema quando a *facet* que impede valores *null* é violada. Por exemplo, o sistema poderia bloquear todas as operações correntes e exibir um alerta ao usuário para que ele corrigisse a inconsistência. Para permitir o tratamento de exceções de violação de conhecimento, todas as anotações *@Facet<Função>*, *@Axiom* e *@Rule* possuem um parâmetro chamado de *exceptionManager*. Esse parâmetro recebe uma classe como argumento. Essa classe define como são gerenciadas exceções em tempo de execução. Uma classe de tratamento de exceções deve ter obrigatoriamente um construtor sem argumentos, para ser invocado por reflexão, e um método com a assinatura *manage(Map context)*. Esse método é invocado quando uma exceção ocorre e o contexto fornecido indica qual facet foi violado, o atributo alvo da *facet* e a instância do conceito que isso ocorreu. Essas informações são necessárias para que o tratamento seja adequadamente executado.

Herança simples é suportada para conceitos, pois é diretamente suportada pela OO. Não é necessário utilizar uma anotação específica. Herança múltipla não é suportada

pela OO, nem por essa abordagem. Para verificar que um conceito é supertipo de outro, Java e C# oferecem mecanismos de reflexão estrutural que fornecem essa informação.

O detalhamento da biblioteca que implementa a semântica de cada anotação é apresentado na seção 3.3.

Tabela 3.1: *K-Annotations* para a Definição de *Facets*.

Anotação	Papel
@FacetNotNull	Define que um atributo não pode ter valor null.
@FacetDefaultValue (value = x)	Define que o valor padrão de um atributo é *x*.
@FacetMinInclusive @FacetMaxInclusive (value = x)	Define que um valor de atributo precisa ser maior/menor ou igual a *x*.
@FacetMinExclusive @FacetMaxExclusive (value = x)	Define que um valor de atributo precisa ser maior/menor que *x*.
@FacetMinLength @FacetMaxLength (value = x)	Define o valor mínimo/máximo de caracteres de uma string.
@FacetFractionDigits (value = x)	Define o número máximo de dígitos que um valor pode possuir.
@FacetMaxCardinality @FacetMinCardinality (value = x)	Define a cardinalidade mínima/máxima que uma coleção pode possuir.
@FacetValidClasses @FacetInvalidClasses	Define as classes que uma coleção suporta (não-suporta). Parâmetros omitidos.
@FacetPatterns (patterns = {x, y, …})	Define que um valor de atributo precisa respeitar o padrão *x* ou *y*. Os padrões são definidos como expressões regulares.
@FacetOrdered	Define que os valores de uma coleção devem ser ordenados.
@FacetDuplicateFree	Define que uma coleção não pode ter valores duplicados.
@FacetValidValues @FacetInvalidValues	Define os valores válidos/inválidos para um atributo. Essa anotação oferece o parâmetro *values* caso o engenheiro de desenvolvimento queira fornecer a lista de valores diretamente na anotação (prática não recomendada, pois um novo valor válido iria exigir a recompilação do modelo). O parâmetro *source* recebe um classe que deve obrigatoriamente ter um método com a assinatura *Objects[] getValues()*; esse método fornece todos os valores válidos. Isso permite, por exemplo, que os valores sejam recuperados de um banco de dados relacional.

3.3 Biblioteca de Tratamento de *Facets*, Axiomas e Regras

A biblioteca de tratamento de *facets*, axiomas e regras fornece o código necessário para implementar a semântica das anotações referentes a esses elementos. Por exemplo, para prover a semântica da anotação *@FacetNotNull*, é necessário que o *advice* do aspecto que trata essa anotação contenha um teste que verifique se o valor que será atribuído à propriedade não é *null*. A Figura 3.5 apresenta a visão lógica dessa biblioteca.

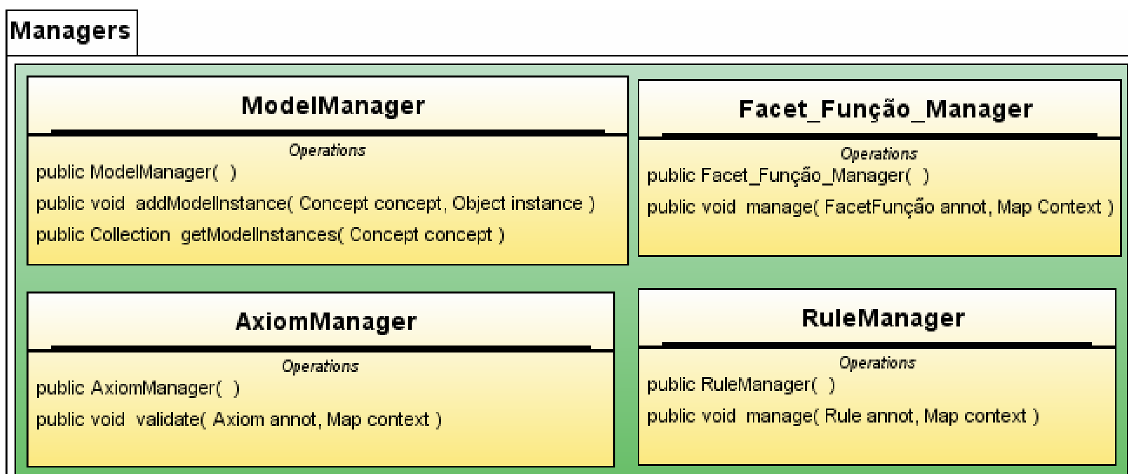

Figura 3.5: Visão Lógica da Biblioteca de Tratamento de *K-Annotations*.

Para evitar que *advice* contenha em seu corpo o próprio teste de valor e evitar que a customização desse método requeira a modificação da biblioteca de aspectos (conhecimento em OA), optou-se por chamar a classe da biblioteca de tratamento de anotações que realiza essa operação. Desse modo, caso o engenheiro de desenvolvimento queira customizar o tratamento de cada anotação, ele não precisa ter conhecimento em OA, mas somente em OO, pois a biblioteca de tratamento somente utiliza OO. Essa biblioteca, também, por exemplo, poderia invocar serviços web para realizar o tratamento. A estrutura de classes dessa biblioteca é a seguinte:

- **Facet<Função>:** para cada *facet* existe uma classe chamada *Facet<Função>Manager*. Por exemplo, para a anotação *@FacetValidValues* existe uma classe que implementa a função de validação cujo nome é *FacetValidValuesManager*. Essa classe oferece a função que compara o valor que será atribuído a uma propriedade com os valores válidos. Caso o valor não seja válido, a atribuição não será concluída e o tratamento de exceção de violação de conhecimento é acionado. Toda classe de tratamento de *facet* possui um método com a assinatura *manage(Facet<Função> annotation, Map context)* que realiza o tratamento da *facet*.

- **Axiom:** para a anotação *@Axiom*, a classe *AxiomManager* recebe a definição do axioma como String (definição fornecida na própria anotação) e então invoca a biblioteca que trata de axiomas, no caso desse trabalho, a biblioteca do JEP (JAVA, 2009). Caso o engenheiro de desenvolvimento opte por utilizar outra biblioteca de processamento de axiomas, basta modificar o código da classe *AxiomManager*;

- **Rule:** para a anotação *@Rule*, a classe *RuleManager* é responsável por instanciar a classe (fornecida como parâmetro dessa anotação) que implementa

a(s) regra(s). Após a instanciação da classe, o método *runRules* é invocado para execução das regras.

Classes adicionais podem ser adicionadas a essa biblioteca conforme as necessidades de cada projeto. Essas classes servem para estender as funcionalidades dessa biblioteca, por exemplo, se uma nova linguagem para definição de axiomas fosse disponibilizada, uma nova classe para o tratamento dessa linguagem poderia ser acrescentada. Se um novo projeto exigisse a utilização de uma linguagem para processamento de regras, uma classe poderia ser acrescentada para que o tratamento fosse possível. Se novos tipos de *facets* fossem propostos, bastaria criar as anotações correspondentes e as classes necessárias para o respectivo tratamento. A extensão das bibliotecas contidas em *K-Aspects* é semelhante ao proposto em KBeans (KNUBLAUCH, 2002), no qual novas funcionalidades são providas através da inclusão de novas classes e métodos na biblioteca. No caso de *K-Aspects* novos aspectos também podem ser necessários, quando novas funcionalidades transversais forem identificadas. Destaca-se que essas classes não devem ter dependência com o domínio da aplicação, para que sejam facilmente reutilizadas em diferentes projetos.

Essa biblioteca foi organizada para prover os construtos definidos em CML e Frames. CML foi escolhida por apresentar os principais construtos providos pelas linguagens de modelagem de conhecimento e ser a linguagem de modelagem de CommonKADS, metodologia extensivamente utilizada para desenvolvimento de SCs. Frames também é suportado inicialmente por essa biblioteca por ser a linguagem de modelagem utilizada como base para a elaboração do SC apresentado na validação dessa proposta.

3.4 Biblioteca de K-Aspects do Componente Conceitual

Aspectos, conforme a seção 2.1, têm sido utilizados com sucesso no tratamento de diversas funcionalidades transversais encontradas em sistemas, por exemplo, persistência e auditoria. No caso de SCs, aspectos podem ser usados para o tratamento da implementação do componente do modelo conceitual. Aspectos evitam que a implementação do componente do modelo conceitual acabe tendo que fazer chamadas explícitas a bibliotecas que tratam a semântica de *facets*, axiomas e regras. O engenheiro de desenvolvimento do componente do modelo conceitual precisa apenas identificar claramente os elementos desse modelo usando as anotações. As anotações são essenciais para identificar os pontos em que o tratamento por aspectos deve ocorrer e fornecer as informações necessárias para que esse tratamento seja realizado com sucesso.

O uso de aspectos também evita a dispersão de chamadas a bibliotecas que tratam *facets,* axiomas e regras. Facilitando a leitura do código e evitando erros do engenheiro de desenvolvimento, por exemplo, a falta da invocação do código que trata a validação de valores de acordo com uma *facet*. Ao evitar dispersão e tarefas repetitivas manuais, reduz-se o custo de desenvolvimento e manutenção do SC.

Os aspectos identificados para o tratamento de funcionalidades necessárias à implementação da modelagem conceitual são os seguintes (Figura 3.6 apresenta a visão lógica desses aspectos):

- **ConceptAspect:** aspecto que monitora a criação/destruição de qualquer instância de conceito do componente conceitual. Esse aspecto evita que o

engenheiro de desenvolvimento tenha que manualmente registrar a criação/destruição de instâncias. O *pointcut* desse aspecto corresponde ao instante seguinte à instanciação de um conceito. O *advice* desse aspecto invoca uma classe que armazena as instâncias atuais dos conceitos. Armazenar as instâncias correntes é importante para realização de tarefas, pois essas operam em cima das instâncias dos conceitos (detalhado no capítulo 4);

- **AttributeAspect:** aspecto que monitora a modificação de qualquer valor de atributo de conceito do componente conceitual. Esse aspecto evita que o engenheiro de desenvolvimento tenha que manualmente registrar a modificação de valores. O *pointcut* desse aspecto corresponde ao instante seguinte à atualização do valor de qualquer atributo. Destaca-se que os aspectos das *facets*, axiomas e regras só permitem a modificação do valor do atributo se ele estiver de acordo com as definições dos mesmos. Desse modo, não ocorre o caso de ser necessário reverter o valor do atributo e as operações que levaram a essa modificação. O *advice* desse aspecto invoca todas as classes que realizam inferências em cima do atributo cujo valor foi modificado. A realização de inferências é a base para realização de tarefas (detalhado no capítulo 4);
- **RelationAspect:** aspecto que monitora a criação/destruição de qualquer relação do componente conceitual. Esse aspecto evita que o engenheiro de desenvolvimento tenha que manualmente registrar a criação/destruição de instâncias. O *pointcut* desse aspecto corresponde ao instante seguinte à instanciação de uma relação. O *advice* desse aspecto invoca uma classe que armazena as instâncias atuais das relações. Armazenar as instâncias correntes é importante para realização de tarefas, pois essas operam em cima das instâncias de relações (detalhado no capítulo 4);
- **Facet<*Função*>Aspect:** para cada *facet* um aspecto foi identificado. Cada aspecto trata a semântica de uma *facet*. Esses aspectos evitam que o engenheiro de desenvolvimento tenha que manualmente realizar chamadas ao tratamento de *facets* em cada troca de valor de atributo. O *pointcut* desses aspectos corresponde ao instante anterior à troca de um valor de atributo. Antes de permitir a troca de um valor, todas *facets* associadas àquele atributo precisam ser validadas. Os *advices* desses aspectos invocam a biblioteca que trata a semântica de cada *facet*. Essa biblioteca de tratamento é extensível para que diferentes SCs, que apresentam diferentes formas de tratamento de *facets,* possam usar a biblioteca de aspectos definidos nessa proposta. Caso o valor viole alguma *facet*, esse valor não é atribuído ao atributo e o gerenciamento de exceção é acionado;
- **AxiomAspect:** aspecto que trata os axiomas definidos no modelo conceitual. Esse aspecto evita que o engenheiro de desenvolvimento tenha que manualmente realizar chamadas ao tratamento de axiomas em cada troca de valor de atributo. O *pointcut* desse aspecto corresponde ao instante anterior à troca de um valor de atributo. Antes de permitir a troca de um valor, todos os axiomas associados àquele atributo precisam ser validados. O *advice* desse aspecto utiliza a biblioteca JEP para validar os axiomas. Caso algum axioma seja violado, o valor não é atribuído ao atributo e o gerenciamento de exceção é acionado;
- **RuleAspect:** aspecto que trata as regras definidas no componente conceitual. Esse aspecto evita que o engenheiro de desenvolvimento tenha que manualmente realizar chamadas ao tratamento de regras em cada troca de valor de atributo. O

pointcut desse aspecto corresponde ao instante posterior à troca de um valor de atributo. Após a troca de um valor, todos as regras associadas àquele atributo precisam ser executadas. O *advice* desse aspecto invoca as classes que implementa as regras e estas classes avaliam os valores dos atributos e, conforme, necessário alteram outros atributos;

Esses aspectos são fornecidos em uma biblioteca resultante desse trabalho. O objetivo dessa biblioteca é permitir o reuso desses aspectos em diferentes projetos de SCs, fornecendo uma solução padrão, já validada, evitando os problemas de soluções *ad-hoc*. Conforme citado anteriormente, KBeans (KNUBLAUCH, 2002) buscou organizar uma biblioteca para implementação de SCs na OO, entretanto, os problemas identificados (seção 2.4.2.1) acabam dificultando sua utilização de fato. A validação da biblioteca de aspectos é apresentada no capítulo 5.

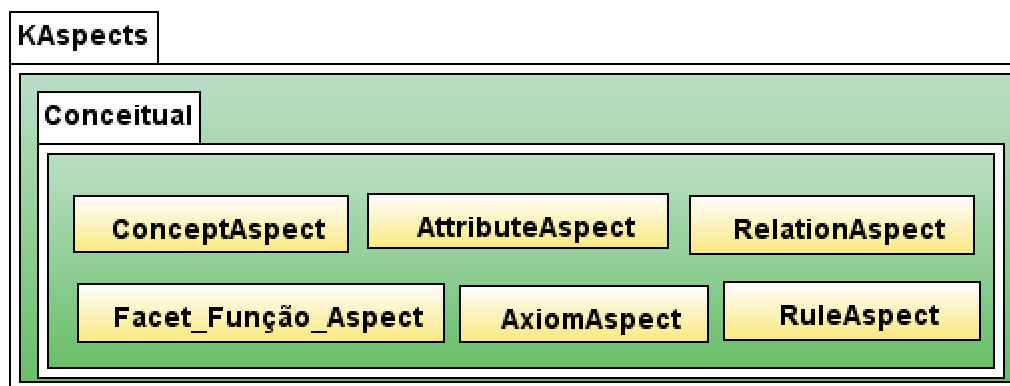

Figura 3.6: Visão Lógica dos *K-Aspects* para o Componente Conceitual.

3.5 Sumário do Capítulo 3

O uso de anotações para representação dos construtos em linguagens OO permite a definição de uma função bijetora entre modelo e implementação. Cada construto da linguagem de modelagem apresenta uma representação única na implementação garantindo a rastreabilidade entre modelo e implementação. A preservação das estruturas também facilita a comunicação entre engenheiros de conhecimento e desenvolvimento, pois as anotações padronizam a comunicação sobre as estruturas.

O uso de aspectos permitiu encapsular todo o código necessário para o tratamento da semântica de cada construto evitando o desenvolvimento manual de uma série de chamadas manuais a bibliotecas que fazem o tratamento desses construtos. A redução das tarefas manuais reduz o risco de bugs nos componentes de software que implementam o componente conceitual, pois o tratamento passa a ser responsabilidade do interpretador de aspectos, que gera o código automaticamente necessário para o tratamento.

A definição de bibliotecas reusáveis para diferentes projetos de SC garante que o conhecimento adquirido pelos engenheiros de desenvolvimento seja válido para diferentes projetos, reduzindo o custo de desenvolvimento desse tipo de sistema e também de manutenção.

4 K-ASPECTS: ASPECTOS PARA IMPLEMENTAÇÃO DOS COMPONENTES DE TAREFA E INFERENCIAL

Os componentes de tarefa e inferencial pertencem ao modelo de conhecimento. O componente de tarefa descreve as estratégias utilizadas para que o sistema resolva tarefas intensivas em conhecimento. CommonKADS (SCHREIBER et al., 2000) define um conjunto de tarefas intensivas em conhecimento e propõem um modelo abstrato de algoritmo para solução de cada tarefa. Por exemplo, tarefas de classificação e interpretação. Exemplos concretos de tarefas são o diagnóstico de qual doença um paciente possui ou a interpretação de ambientes (ex.: seco, chuvoso) a que uma rocha foi submetida. Destaca-se que as tarefas buscam simular o raciocínio realizado pelo especialista humano no domínio da aplicação (ABEL, 2001).

Para a resolução de tarefas, passos básicos de raciocínio precisam ser utilizados, esses passos são definidos no componente inferencial. A Tabela 4.1 apresenta um conjunto de inferências e seu significado. Essas inferências foram propostas em CommonKADS (SCHREIBER et al., 2000) e são utilizadas nos PSMs de classificação e interpretação:

Tabela 4.1: Exemplo de Inferências (Extraído de ABEL, 2001).

Inferências de CommonKADS	
Inferência	**Significado**
Abstrai *Abstract*	A entrada é um dado e a saída é um modelo abstrato desse dado. O modelo deve ser fornecido para que a abstração seja realizada. Nesse projeto, as informações do modelo abstrato são expressas nas anotações definidas na seção 3.2.
Avalia *Evaluate*	A entrada é um dado e uma norma e a saída é um valor-verdade indicando se o dado se adequa a norma.
Compara *Match*	A entrada são dois valores e a saída é indica se o primeiro é maior, igual ou menor do que o segundo.
Decompõe *Decompose*	A entrada é um conceito único e a saída é o conjunto das partes que o compõe. A decomposição exige o fornecimento do modelo.
Especifica *Specify*	A entrada é um objeto e a saída é um novo objeto associado de alguma maneira com o objeto da entrada. A decomposição exige o fornecimento do modelo.
Seleciona *Select*	A entrada é um conjunto de dados e a saída é um elemento ou subconjunto desses dados.

A biblioteca de PSMs (modelo abstratos para solução de tarefas) fornecida em CommonKADS é apresentada somente no nível abstrato, não havendo nenhuma

definição padrão para implementação. A falta de proposta padrão obriga cada projeto de SC utilizar uma solução *ad-hoc*, que, frequentemente, acaba sendo dependente da aplicação e pouco reusável em diferentes projetos. Para tratar desses problemas, aspectos e anotações podem ser utilizados, como apresentado na próxima seção.

4.1 Biblioteca de K-Aspects para Componentes de Tarefa e Inferencial

Um PSM é um modelo abstrato de inferência reusável (exemplo na Figura 4.1) para um mesmo tipo de tarefa (ex.: classificação, interpretação) em diferentes domínios de aplicação (ex.: medicina, geologia). No caso da tarefa de interpretação, diferentes domínios precisam realizá-la (interpretar o ambiente diagenético a que uma rocha foi submetida, interpretar os processos a que um paciente foi submetido). Nesse caso, um PSM voltado para essa tarefa deve ser reusável em ambos os domínios. Pode-se dizer que a realização de uma tarefa específica a um domínio, por exemplo, a tarefa interpretação de ambiente diagenético instancia o PSM para interpretação.

A implementação usual de PSMs na OO tem utilizado soluções *ad-hoc* e tem sido orientada pela definição de uma sequência de passos/inferências que devem ser realizados. Por exemplo, a Figura 4.1 define a sequência de inferências realizadas pelo método de classificação. Observa-se que as implementações de PSMs não têm explorado a característica atômica de cada inferência e a independência entre algumas dessas inferências.

Por exemplo, a abstração e comparação poderiam ser realizadas ao longo do próprio preenchimento das instâncias do componente conceitual; não seria necessário aguardar a finalização do preenchimento pelo usuário para que elas fossem realizadas. Já a solução deve aguardar um certo número de dados para ser atualizada, para evitar sobrecarga de processamento do sistema, já que SCs podem trabalhar com um grande volume de dados. Por exemplo, em alguns SCs, tarefas intensivas em conhecimento podem utilizar mais de dez mil regras.

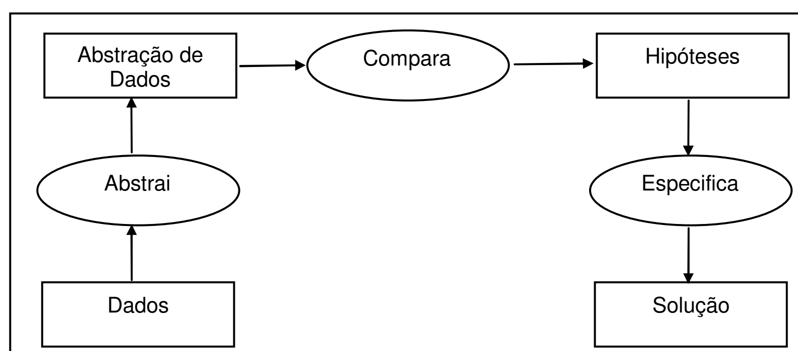

Figura 4.1: Estrutura de inferência que descreve o método de classificação (Adaptado de ABEL, 2001).

Aspectos e anotações podem ser utilizados em conjunto para que uma solução padrão para a implementação de PSMs seja alcançada. Por exemplo, pode-se visualizar uma inferência como um aspecto (funcionalidade transversal) que deve ser tratado ao longo do preenchimento da instância dos componentes do componente conceitual. Destaca-se que os PSMs são usados para a modelagem das tarefas que SCs realizam.

Além disso, como PSMs buscam simular a forma de raciocínio humano, eles deveriam explorar a característica de concorrência que o cérebro humano possui. Por

exemplo, quando é necessário classificar um animal a partir da leitura de um texto, em paralelo à realização da leitura do texto, o leitor identifica uma série de evidências (características) que comprovam/refutam hipóteses de classificação para esse animal. Ao final da leitura ou até mesmo antes (em alguns casos), a classificação já estará determinada, sem necessidade de avaliação de todas as evidências e hipóteses, pois grande parte delas já terá sido avaliada em paralelo à leitura.

Simular a concorrência do raciocínio humano em PSMs pode melhorar o tempo de resolução de tarefas de SCs, já que ao longo do preenchimento de dados pelo usuário, pode-se realizar uma série de inferências. Evitando que todas as inferências sejam realizadas somente ao final do preenchimento dos dados pelo usuário, exigindo que o usuário aguarde a realização de todas as inferências. Atualmente, isso não tem sido adotado. A realização de tarefas é sequencial e é realizada somente quando o usuário, após ter finalizado o preenchimento das instâncias do componente conceitual, solicita a realização da tarefa. No entanto, etapas intermediárias de inferências podem ser executadas ao longo do preenchimento das instâncias do componente conceitual, explorando o potencial da concorrência.

Para exemplificar a proposta, a Figura 4.1 apresenta a estrutura de inferência para a tarefa de classificação. As inferências (elementos ovais) podem ser tratadas como aspectos, que são acionados conforme o usuário manipula as instâncias do componente conceitual, por exemplo, enquanto ele preenche dados que devem ser classificados. A inferência *abstrai* pode ser identificada com um aspecto cujo *pointcut* corresponde ao instante seguinte à modificação de um valor de atributo. O encerramento da execução do *advice* do aspecto *abstrair* também identifica outro aspecto, o aspecto de *compara* que realiza a comparação. O *advice* desse aspecto realiza comparação com as hipóteses presentes na base de conhecimento, usando triplas do tipo <conceito,atributo,valor>. Observa-se que uma tripla é válida para qualquer domínio de aplicação, sendo esse algoritmo genérico para qualquer tipo de comparação. Cada hipótese validada é armazenada, até que um conjunto mínimo que valide uma determinada classificação (solução) seja atingido. Quando o conjunto mínimo é atingido, a classificação é exibida como resultado, usando a inferência *especializa* que identifica o objeto alvo como sendo da classe validada.

Sem o uso de aspectos, a implementação da solução acima, teria que invocar os métodos que iniciam as inferências em todos os pontos da aplicação em que valores de atributos são modificados, pois cada modificação de valor pode influenciar os resultados das inferências. Usando aspectos, essas chamadas deixam de ser necessárias, pois os pontos de ação das inferências (*pointcuts*) são determinados diretamente nos aspectos, sem necessidade de intervenção do engenheiro de desenvolvimento. Cada aspecto trata uma inferência específica, destaca-se que esses aspectos são modulares para que a função de inferência possa ser determinada pelo engenheiro de desenvolvimento da aplicação. Por exemplo, a comparação de um determinado atributo pode exigir operações complexas que podem variar conforme a estratégia de implementação de projeto para projeto. O objetivo dessa modularidade é oferecer certa flexibilidade aos engenheiros de desenvolvimento que adotem *k-aspects* em diferentes projetos. Por exemplo, a modularidade permite que diversos projetos usem uma mesma biblioteca de comparações.

Os aspectos fornecidos como biblioteca resultante desse trabalho oferecem uma solução padrão para as inferências que podem ser realizadas concorrentemente ao preenchimento de instâncias do modelo pelo usuário. O conjunto fornecido permite a

implementação de diferentes PSMs usando aspectos, Figura 4.2 (validação apresentada no capítulo 6):

- **AbstractAspect:** esse aspecto trata a inferência *abstrai*. O *pointcut* desse aspecto corresponde ao instante seguinte à modificação do valor de um atributo. O *advice* identifica o atributo cujo valor foi modificado e abstrai gerando uma tripla <conceito,atributo,valor>. Essa abstração é possível porque o componente conceitual construído usando *k-annotations* fornece em tempo de execução as anotações (metadados) que identificam o atributo e seu respectivo conceito;
- **EvaluateAspect:** esse aspecto trata a inferência *avalia*. O *pointcut* desse aspecto corresponde ao instante seguinte à finalização do *advice* do aspecto *AbstractAspect*. O *advice* verifica se a tripla fornecida está adequada a uma determinada norma (o engenheiro de desenvolvimento deve fornecer a classe que obtém as normas). Caso esteja adequada, o resultado é verdadeiro, caso contrário é falso. Esse aspecto é configurável de modo que seja possível o engenheiro de desenvolvimento fornecer o método que realiza a avaliação;
- **MatchAspect:** esse aspecto trata a inferência *compara*. O *pointcut* desse aspecto corresponde ao instante seguinte a finalização do *advice* do aspecto *EvaluateAspect* quando tiver sido verificada como verdade a adequação do valor à norma. Esse *advice* verifica se a soma dos valores atribuídos às normas é superior ao limiar estabelecido para uma conclusão, caso o seja, ele aciona a inferência *especifica* que fornece ao usuário o resultado da tarefa do PSM (ex.: uma interpretação);

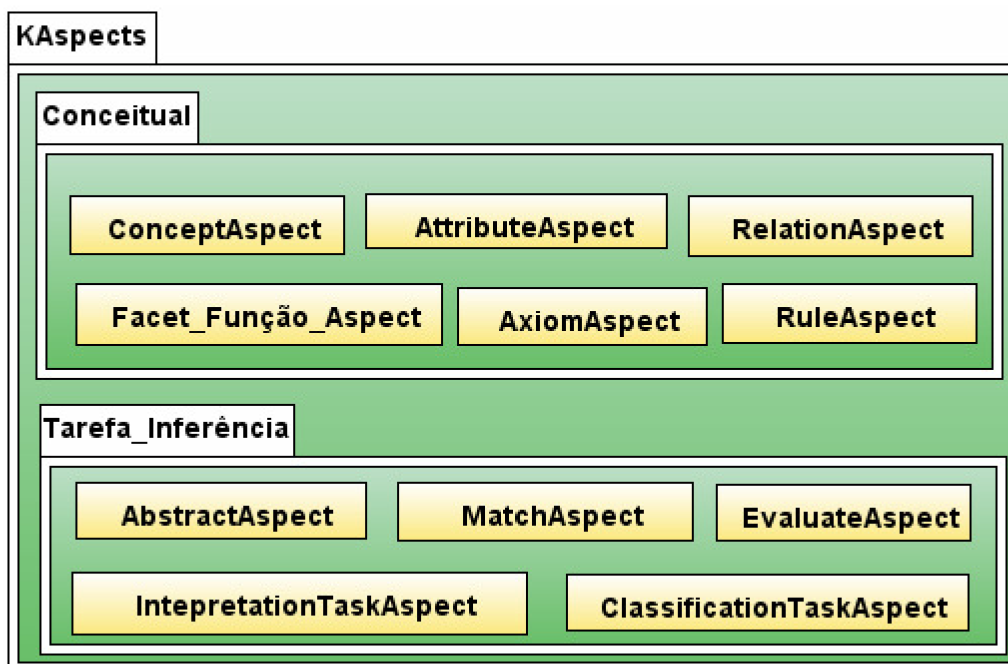

Figura 4.2: Visão Lógica dos *K-Aspects* para os Componentes do Modelo de Conhecimento.

As demais inferências não foram implementadas por aspectos, pois são utilizadas a partir das três inferências implementadas como aspectos. Por exemplo, a inferência *seleciona* é utilizada para selecionar uma *norma* de um conjunto de normas. Essa inferência é invocada diretamente no *advice* do aspecto *EvaluateAspect*, pois é na

avaliação que as normas são utilizadas. O mesmo ocorre para a inferência *decompõe*, a decomposição de um conceito em suas partes é utilizada quando é realizada uma abstração do conceito, no caso, na inferência *abstrai*.

Além das inferências, foram criados aspectos para as diferentes tarefas. Esses aspectos são responsáveis por realizar validações finais das tarefas e então fornecer os resultados para o usuário. Na validação apresentada no capítulo seguinte, uma interface gráfica chamada de *Knowledge Advisor*, exibe os resultados das tarefas ao longo do próprio preenchimento da instância de amostra de rocha pelo usuário, sem necessidade que o usuário explicitamente solicite a realização dessa tarefa.

A validação da proposta de uma solução padrão para a implementação dos modelos citados nesse capítulo usando PSMs é apresentada no capítulo seguinte. A seção 4.2 exemplifica didaticamente o uso da abordagem apresentada para facilitar a compreensão da validação.

4.2 Exemplo Didático

O exemplo didático dessa seção foi definido usando como base o exemplo organizado em (SCHREIBER et al., 2000). O exemplo trata da tarefa de autorizar/negar o aluguel de uma residência por um determinado solicitante (pessoa que deseja alugar o imóvel).

O componente conceitual desse exemplo é composto por dois conceitos e uma relação conforme a Tabela 4.2. Cada conceito apresenta um conjunto de atributos. Cada atributo apresenta um domínio válido de valores.

Tabela 4.2: Exemplo de Componente Conceitual.

Conceito Residência	
Codigo	integer, [0, 1000000]
Tipo	string(50) one-of [apartamento, casa]
ValorAluguel	real, range [0.0 - 10000]
NumeroHabitantes	integer, [1, 10] – indica o número máximo de habitants para a residência
Conceito Solicitante	
Nome	string(200)
DataNascimento	date, [DD/MM/YYYY]
Idade	integer, range [0 – 150]
Salario	real, range [0.0 - 30000]
Axiomas: • Solicitante.Idade = Hoje – Solicitante.DataNascimento	
Relação SolicitaçãoAluguel	
Argumento-1	one of Solicitante
Argumento-2	one of Residência
Atributo Autorizada	boolean **//identifica se a solicitação foi autorizada ou negada.**

Cada conceito corresponderá a uma classe. Essas classes deverão ser anotadas com *@Concept* para identificá-las que são conceitos oriundos de um componente conceitual.O nome de uma classe anotada é igual nome do conceito. Cada classe também terá anotações que identificam atributos e suas respectivas *facets*. Por exemplo, para *Residência*, o atributo *Codigo* será implementado como uma propriedade chamada *Codigo* do tipo inteiro anotada com *@Attribute*. Além da anotação *@Attribute*, duas anotações adicionais *@FacetMinValue(0)* e *@FacetMaxValue(1000000)* serão utilizadas para, respectivamente, restringir o valor mínimo e máximo permitido para essa propriedade. A Figura 4.3 apresenta a implementação dos dois conceitos.

```
@Concept
public class Residencia{
   @Attribute
   @FacetMaxInclusive(100000)
   @FacetMinInclusive(0)
   Integer Codigo;

   @Attribute
   @FacetValidValues(values={"apartamento, casa"})
   String Tipo;

   @Attribute
   @FacetMaxInclusive(10000f)
   @FacetMinInclusive(0f)
   Float Aluguel;

   @Attribute
   @FacetMaxInclusive(10)
   @FacetMinInclusive(0)
   Integer NumeroHabitantes;

   //métodos de getters/setters omitidos
}

@Relation
public class SolicitacaoAluguel{
   @Argument(order=1)
   Solicitante solicitante;

   @Argument(order=2)
   Residencia residencia;

   @Attribute
   Boolean Autorizada;

   //métodos de getters/setters omitidos
}
```

```
@Concept
@Axiom(expression="Solicitante.Idade = Hoje –
Solicitante.DataNascimento ")
)
public class Solicitante{
   @Attribute
   @FacetMaxLength(length=200)
   Integer Codigo;

   @Attribute
   @FacetPattern(pattern="[0-9][0-9]/[0-9][0-9]/[0-9][0-
9][0-9][0-9]")
   String Date = "";

   @Attribute
   @FacetMaxInclusive(150)
   @FacetMinInclusive(0)
   Integer Idade;

   @Attribute
   @FacetMaxInclusive(30000f)
   @FacetMinInclusive(0f)
   Float Salario;

   //métodos de getters/setters omitidos
}
```

Figura 4.3: Exemplo de Implementação do Componente Conceitual.

Essa implementação do componente conceitual será então tratada no processo de costura dos aspectos. Esse processo inicialmente verifica nas classes os *pointcuts* existentes de acordo com os aspectos fornecidos na biblioteca de aspectos. Por exemplo, a existência da anotação *@FacetMaxInclusive(10)* sobre a propriedade *intHabitantes* e a existência de um método (*setHabitantes*), que modifica o valor da propriedade, corresponde ao *pointcut evalMaxRangeField* (Figura 4.4) definido no aspecto *FacetValueRangeAspect*.

Durante o processo de costura, quando esse ponto é identificado, um código adicional (*advice* - Figura 4.5) é inserido para ser invocado antes da efetiva atribuição do valor a propriedade. Esse código invoca o componente da biblioteca de validação de intervalos numéricos, especificamente, o método *validaMaxRange(...)*. Esse método recebe como argumento o conceito, o atributo, a *facet* e o valor que se deseja atribuir a propriedade. A *facet* corresponde a anotação *@FacetMaxInclusive(10)*, essa anotação fornece o valor *10* para indicar qual o valor máximo permitido para esse atributo. Esse processo termina quando todos os códigos adicionais foram inseridos para todos os *pointcuts* identificados na implementação fornecida. O resultado desse processo é a implementação executável do componente conceitual.

```
pointcut evalMaxRangeField (Object instance) :   (set (@FacetMaxExclusive * *) ||
                                                   set (@FacetMaxInclusive * *)) && args(instance);
```

Figura 4.4: *Pointcut* para validação de valor máximo.

```
//Advice para validação de valor máximo permitido
before(Object instance) : evalMaxRangeField(instance) {
   //Obtém a lista de anotações existentes sobre a classe em que o advice está agindo.
   Annotation[] annotations = getField(thisJoinPoint.getTarget(),
                  thisJoinPointStaticPart.getSignature().getName()).getAnnotations();
   //Obtém o conceito.
   Concept concept = thisJoinPoint.getTarget().getClass().getAnnotation(Concept.class);
   //Obtém o atributo.
   Attribute attrib = (Attribute)getAnnotation(annotations, Attribute.class);
   //Obtém a anotação de valor máximo.
   Annotation facet = getAnnotation(annotations, FacetMaxInclusive.class);
   //Invoca o componente RangeValidator, responsável por validar o valor.
   //Instance é o valor que se deseja atributir.
   RangeValidator.validateMaxRange(concept, attrib, facet, instance);
}
```

Figura 4.5: *Advice* para validação de valor máximo.

Quando uma instância do conceito *Solicitante* é criada e invoca-se o método *setHabitantes* (*20*), antes de ocorrer a atribuição do valor 20 a propriedade *intHabitante*, conforme definido no aspecto acima, o *advice* de validação é acionado. Esse *advice*, fazendo uso da biblioteca de validação, identificará que o valor passado como argumento é superior ao máximo permitido. Desse modo, esse *advice* não permitirá que esse valor seja atributo à variável *intHabitantes* (o valor máximo é 10) e enviará uma mensagem de exceção ao usuário através do *KnowledgeAdvisor*. Nesse exemplo, a exceção é exibida no *KnowledgeAdvisor* porque a implementação não fez uso da propriedade *exceptionManager* presente em *@FacetMaxInclusive.* Por padrão, a biblioteca utiliza o *KnowledgeAdvisor* para exibir mensagens. Caso tivesse sido fornecido um outro gerenciador de exceções para envio de um sinal sonoro, esse sinal seria emitido quando invocado *setHabitantes* (*20*). Para todos os demais conceitos, atributos, *facets,* axiomas e regras o tratamento é realizado conforme o exemplificado.

O objetivo da tarefa de autorização de aluguel é analisar as características de um solicitante com as características da *residência* para determinar se ele pode alugar a residência solicitada. Na implementação desse exemplo, sem *k-aspects*, essa tarefa seria realizada somente quando um conjunto de instâncias da relação *SolicitacãoAluguel* tivesse sido criado. Supondo-se que esse conjunto tivesse mais de 100 mil solicitações, a realização dessa tarefa iria exigir diversos minutos para ser executada, pois é realizada sequencialmente. Na implementação com *k-aspects*, faz-se uso dos aspectos de inferências e também de um aspecto adicional específico para a tarefa de autorização.

O aspecto *AutorizaAluguelTask* define o *pointcut validaAutorizacao* acionado a cada cinco modificações em valores atributos de *Solicitante, Residência* e *SolicitaçãoAluguel*. Esse valor de cinco modificações é apenas um exemplo. Recomenda-se que um limiar de modificações seja definido para evitar que a realização da tarefa ocorra a cada modificação, o que poderia provocar problemas de desempenho. O *AutorizaAluguelTask* faz uso dos resultados das inferências *abstrai, valia e compara*, também implementadas como aspectos, conforme definido na seção anterior. A inferência *abstrai*, aspecto *AbstractAspect*, identifica o conceito, o atributo e o valor modificado e abstrai para uma tripla. Essa tripla é então processada pelo aspecto *EvaluateAspect*; esse aspecto, a partir do conceito e atributo, obtém todas as normas

associadas ao atributo. As normas (determinam as regras para autorizar/negar o aluguel) são validadas quando o valor da tripla é compatível com o valor especificado na norma. Cada norma tem um peso associado a decisão, desse modo, quando validada, ela é marcada como válida para posterior avaliação na inferência *compara*. Após a avaliação das normas, o aspecto *MatchAspect* é acionado para verificar se o limiar mínimo para autorização foi atingido; para isso, o somatório dos pesos das normas consideradas válidas é comparado com o limiar. Se o somatório for maior ou igual ao limiar, o próprio *advice* modifica a propriedade *Autorizada* da relação *SolictaçãoAluguel* para valor *verdade*. Além de modificar o valor, quando uma solicitação é autorizada, *advice* informa ao usuário da autorização através do *KnowledgeAdvisor*.

Esse exemplo demonstrou didaticamente a utilização da proposta. O próximo capítulo apresenta a validação dessa proposta em um SC de grande porte atualmente em uso.

4.3 Sumário do Capítulo 4

O uso de aspectos para a realização dos passos de raciocínio, inferências, necessários para a realização das tarefas atribuídas a um SC permitiu explorar a atomicidade das inferências para compartilhamento dos resultados entre diferentes tarefas. Ao tratar cada inferência individualmente, as tarefas passam a ser realizadas através da composição de uma série de inferências. Os resultados das inferências são armazenados em estruturas como pilhas e conjuntos para acesso posterior pelas demais inferências.

O processamento das mesmas em paralelo ao preenchimento das instâncias do modelo conceitual, sem necessidade do engenheiro de desenvolvimento utilizar recursos específicos de processamento paralelo para obter resultados similares, evita a dispersão de métodos associados ao processamento paralelo. Resultados obtidos pela realização das tarefas podem ser exibidos ao longo do próprio preenchimento das instâncias, sem necessidade que o usuário aguarde todo o preenchimento para poder obter os resultados das tarefas.

Além disso, a utilização de inferências fornecidas pela biblioteca desenvolvida nesse trabalho, evita a implementação manual das mesmas e permitindo o reuso. O compartilhamento dessa biblioteca padrão de inferência permite que diferentes projetos a utilizem. Ressalta-se que o conhecimento adquirido pelos engenheiros de desenvolvimento permanece válido para diferentes projetos.

5 VALIDAÇÃO DA PROPOSTA

A validação das propostas apresentadas nos capítulos 3 e 4 foi realizada através da comparação entre a solução atual, *ad-hoc*, adotada em um SC de grande porte, construído a partir de uma ontologia de domínio sobe a qual são executados métodos de solução de problemas, e a respectiva solução com *k-aspects*. Essa seção apresenta as duas soluções e, por fim, avalia a proposta de *k-aspects* para implementação de SCs.

5.1 Estudo de Caso: Sistema Petroledge

O sistema de conhecimento Petroledge® é voltado para a avaliação da qualidade de reservatórios de petróleo usando como base descrições de rochas sedimentares e interpretações automatizadas pelo sistema. Esse sistema é resultado dos trabalhos de (ABEL, 2001; SILVA, 2001; MASTELLA, 2005 e VICTORETI, 2007) no âmbito do Grupo de Bancos de Dados Inteligentes da UFRGS e atualmente está sendo usado por diversas empresas do setor do petróleo.

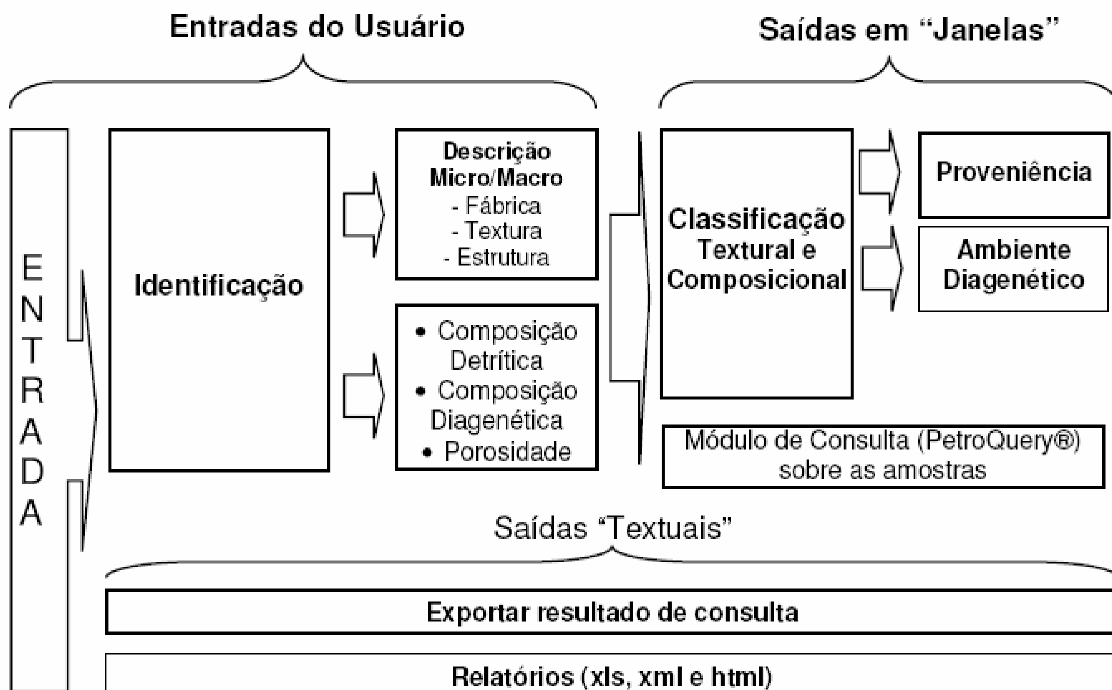

Figura 5.1: Fluxo de Trabalho do Sistema Petroledge (GUIA, 2009).

O fluxo de uso desse sistema inicia quando o usuário (petrógrafo) recebe uma amostra de rocha. Essa amostra é então descrita no sistema Petroledge a partir da observação de uma lâmina de rocha realizada usando um microscópio. O fluxo de

trabalho nesse sistema é apresentado na Figura 5.1. Inicialmente o usuário insere um conjunto de dados de descrição em diferentes interfaces gráficas (Figura 5.2 - entrada de dados de microscopia). Após completar a inserção dos dados, o usuário pode salvar os dados (dados são persistidos em banco de dados relacional) e solicitar que o sistema realize algumas tarefas sobre os dados: tarefa de interpretação e classificação. Essas tarefas são realizadas através de algoritmos que implementam os PSMs de interpretação e classificação, respectivamente. Atualmente, essas tarefas podem levar mais de 5 minutos, pois todas as inferências necessárias são realizadas somente após a solicitação da realização da tarefa e essas utilizam centenas de consultas a um banco de dados relacional, sendo assim diretamente influenciadas pela latência de acesso ao banco e tempo de resposta.

Figura 5.2: Tela para Entrada de Dados de Microscopia (GUIA, 2009).

A descrição realizada pelo geólogo instancia os conceitos definidos no componente conceitual. Esses conceitos apresentam atributos com domínio pré-definidos em conceitos que compõem a nomenclatura do sistema. A pré-definição de valores é necessária para padronização da nomenclatura e automatização de tarefas. Por fim, o usuário solicita ao sistema que realize as tarefas de interpretação. Essas tarefas, conforme a descrição realizada pelo usuário, instanciam automaticamente conceitos que descrevem as interpretações obtidas. Os modelos que compõem o sistema são descritos nas seções 5.1.1 e 5.1.2.

5.1.1 Componente Conceitual

O componente conceitual do Petroledge, Figura 5.3, é composto por três grupos de conceitos:

- **Conceitos para Descrição de Rocha (Figura 5.3a):** conjunto de conceitos que são instanciados pelo usuários (petrógrafo) durante a tarefa de descrição de rochas;
- **Conceitos de Definição de Nomenclatura (Figura 5.3b):** conjunto de conceitos que compõem a nomenclatura do sistema. Certos atributos de conceitos, descritos pelo usuário, apresentam uma nomenclatura geológica pré-definida. Por exemplo, uma descrição de rocha pode ser realizada para diferentes fins. Esses fins são pré-definidos pelo especialista que forneceu o conhecimento disponibilizado no sistema;
- **Conceitos para Realização de Tarefas (Figura 5.3c):** conjunto de conceitos que são instanciados automaticamente pelas tarefas que o sistema é capaz de realizar. Por exemplo, a tarefa que realiza interpretações sobre os ambientes (ex: clima seco, chuvoso, etc) a qual a rocha foi submetida, instancia o conceito de interpretação de ambiente diagenético.

Cada conceito é formado por um conjunto de atributos, os domínios de valores para esses atributos são definidos via *facets*. Axiomas e regras também são aplicados a esses conceitos. A Tabela 5.1 apresenta a modelagem em *concepts* dos conceitos de descrição de rocha e conceitos de definição de nomenclatura utilizados na validação desse trabalho. A descrição completa desse componente conceitual pode ser encontrada em ABEL, 2001.

A modelagem dos conceitos para realização da tarefa de interpretação de ambiente diagenético, especificamente do conceito “Ambiente Diagenético”, é realizada através de grafos de conhecimento. Grafos de conhecimento são árvores que representam graficamente as associações entre os conceitos de descrição de rocha e respectivas interpretações diagenéticas (ABEL, 2001). O metamodelo de grafo de conhecimento é definido na Figura 5.4. Um grafo corresponde a uma interpretação de ambiente (por exemplo: ambiente de clima seco) e é composto por um conjunto de pacotes (Figura 5.5a). Cada pacote representa um conjunto de feições visuais (pré-definidas pelo especialista em geologia) e possui um peso. Esse peso é utilizado para determinar se uma determinada interpretação foi atingida, pois cada interpretação apresenta um limiar. Cada feição visual (Figura 5.5b) é identificada a partir de um conjunto de evidências. Cada evidência (Figura 5.5c) corresponde a uma tripla <conceito,atributo,valor> que deve ser encontrada na instância preenchida pelo petrógrafo. A Figura 5.5 apresenta um exemplo de grafo de conhecimento para interpretação de ambiente diagenético. Segundo Abel (2001, p. 158), os grafos representam:

> No modelo, o papel de regras, que orientam as inferências possíveis do sistema, porém representam as informações de forma mais estruturada. A semântica de um único grafo, se fosse representado através de regras exigiria mais de 20 regras para ser representada. Isso tornaria bastante complicada a manutenção dessas regras.

Os grafos de conhecimento são utilizados nos modelos de tarefa e inferencial para realização da tarefa de interpretação. O detalhamento de como eles são utilizados é realizado na seção 5.1.2.

Tabela 5.1: Modelo de Conceitos do Sistema Petroledge (Adaptado de ABEL, 2001).

Concept Sample	
Is-a	Object
Sample-ID	string(20)
Concept ThinIdentification	
Is-a	Object
Part-of	Concept Sample
Thin-ID	string(20)
Unit	string(60) – Pattern [a-zA-Z]*-[0-9]
Basin	string(40)
Field	string(20)
Well	string(20)
TopDepth	real, range [0.0 - 9999.99]
BaseDepth	real, range [0.0 - 9999.99]
Place	string(40)
Use	string(80), list-of [Depositional, Diagenetic, Ecologic, Paleogeographic/paleogeologic, Provenance, Reservoir, Stratigraphic, Other use], MAX [3 ocorrences]
Date	date, [DD/MM/YYYY]
Petrographer	string(20)
Axiomas: • Identification.BaseDepth >= Identification.TopDepth || Identification.BaseDepth = 0	
Concept Microscopic	
Is-a	Object
Part-of	Concept Sample
GrainSize	string(20), one-of [gravel, very coarse sand, coarse sand, medium sand, fine sand, very fine sand, silt, clay]
NumericGrainSize	string(15), one-of [<number and metric unit(mm)> or <number - number and metric unit(mm)>], range [0 - 100], [ex.: 23 mm, 12 - 21 mm]
ModalGrainSizes	string(80), list-of [gravel, very coarse sand, coarse sand, medium sand, fine sand, very fine sand, silt, clay], MAX [4 ocorrences]
NumericModalGrainSize	string(15), list-of [<number and metric unit(mm)> or <number - number and metric unit(mm)>], range [0 - 100], MAX [4 ocorrences], [ex.: 23 mm, 12 - 21 mm]
Structures	string(90), list-of [parallel bedding, cross bedding, normal-graded bedding, reverse-graded bedding, massive bedding, parallel lamination, climbing lamination, linser lamination, flaser lamination, massive, bioturbation, fluidized, load, folded, fractured, stylolites, nodules, crystalline, crusts, roots, spotted, vesicular, amigdaloidal, flow, bioconstruction, bioacumulation], MAX [3 ocorrences]
Gravel	real, range [0.0 - 100.00]
Sand	real, range [0.0 - 100.00]
Mud	real, range [0.0 - 100.00]
Sorting	string(20), one-of [very well sorted, well sorted, moderately sorted, poorly sorted, very poorly sorted]
NumericSorting	real, range [0.0 - 100.00]
Roundness	string(20), one-of [well rounded, rounded, sub-rounded, sub-angular, angular]
RoundnessModifiers	string(75), list-of [soft intraclasts, intraclasts, pressure dissolution, corrosion, deformation, replacement, overgrowths], MAX [3 ocorrences].
Sphericity	string(10), one-of [high, medium, low]
Orientation	string(20), one-of [parallel, imbricated, homogeneous, heterogeneous, oriented, chaotic]
Packing	string(15), one-of[loose, normal, tight]

PackingIndex	[real, range [0.0 - 55.00]]
PointContacts	string(10), one-of [abundant, common, rare, trace] or [real, range [0.0 - 100.00]]
SuturedContacts	string(10), one-of [abundant, common, rare, trace] or [real, range [0.0 - 100.00]]
PointContacts	string(10), one-of [abundant, common, rare, trace] or [real, range [0.0 - 100.00]]
ConcaveConvexContacts	string(10), one-of [abundant, common, rare, trace] or [real, range [0.0 - 100.00]]
Support	string(30), one-of [grain-supported, grain to matrix-supported, grain to cement-supported, matrix-supported, matrix to cement-supported, cement-supported]

Axiomas:
Gravel + Sand + Mud = 0 || Gravel + Sand + Mud = 100
PointContacts + LongContacts + ConcavoConvexContacts + SuturedContacts = 0 || PointContacts + LongContacts + ConcavoConvexContacts + SuturedContacts = 0

Regras:
//Regras para Packing
- IF PackingIndex <= 40 THEN Packing = loose
- IF PackingIndex > 40 AND PackingIndex <= 55 THEN Packing = normal
- IF PackingIndex > 55 THEN Packing = tigtht

//Regras para Sorting
- IF NumericSorting <= 0.35 THEN Sorting = very well sorted
- IF NumericSorting > 0.35 AND NumericSorting <= 0.5 THEN Sorting = well sorted
- IF NumericSorting > 0.5 AND NumericSorting <= 0.7 THEN Sorting = moderately sorted
- IF NumericSorting > 0.7 AND NumericSorting <= 2 THEN Sorting = poorly sorted
- IF NumericSorting > 2 THEN Sorting = very poorly sorted

Concept Composition-Description

Is-a	Object
Part-of	Concept Sample
Observations	string(100)

Concept Primary-Description

Is-a	Object
Part-of	Concept Composition-Description
PrimaryComposition	list-of [instance-of <Concept Primary-Composition>]

Concept Primary-Composition-Item

Is-a	Object
Part-of	Concept Primary-Description
MineralName	string(80), one-of [Primary-Constituent].
ConstituentSet	string(40), one-of [Detrital quartz, Detrital feldspar, Plutonic rock fragments, Volcanic rock fragments, Sedimentary rock fragments, Metamorphic rock fragments, Micas/chlorite, Heavy minerals, Intrabasinal grains, Detrital matrix, Other detrital constituents]
Location	string(40), one-of [in metamorphic rock fragment, in plutonic rock fragment, in sedimentary rock fragment, in volcanic rock fragment, in intrabasinal fragment, as monomineralic grain]
Amount	real, range [0.0 - 100.00]
Description	string(256)

Concept Primary-Constituents

Is-a	Object
MineralName	string(100), one-of [Acritarch, Aglutinant benthic foraminifer bioclast, Algae bioclast undifferentiated, Alterite fragment, Amphibole, Amphibolite rock fragment, Andalusite, Annelid (worm) bioclast, Apatite, Argillaceous mud intraclast, Arthropod bioclast undifferentiated, Asteroid bioclast, Basic/ultrabasic plutonic rock fragment, Benthonic foraminifer bioclast, Blastoid bioclast, Bone undifferentiated, Brachiopod bioclast, Bryozoan

	bioclast, Carbonate pisolith, Carbonate sand intraclast, Cephalopod bioclast, Conodont bioclast, Crustacean bioclast, Detrital quartz, Detrital quartz polycrystalline, Detrital quartz polycrystalline stretched, Fish scale, Gypsum monocrystalline detrital, Hornfelse rock fragment, Marl fragment, Meta-sandstone rock fragment, Mudstone fragment, Muscovite, Phosphate grain, Phyllite rock fragment]
Concept Diagenetic-Description	
Is-a	Object
Part-of	Concept Composition-Description
DiageneticComposition	list-of [instance-of <Concept Diagenetic-Composition>]
Concept Diagenetic-Composition	
Is-a	Object
Part-of	Concept Diagenetic-Description
MineralName	string(80), one-of [Diagenetic-Constituent]
Habit	string(20), one-of [Blocky, Booklet, Botryoid, Bridge, Coarsely-crystalline, Coarse mosaic, Coating, Discrete crystal, Fibro-radiated, Fibrous, Fine mosaic, Framboid, Ingrowth, Internal sediment, Lamella, Large rhomb, Massive, Meniscus, Microcrystalline, Ooid, Outgrowth, Overgrowth, Parallel-prismatic, Pelletoid, Peloid, Pigment, Poikilotopic, Prismatic, Prismatic-radiated, Radiated, Rim, Rosette, Sheaf, Small rhomb, Spherulite, Vermicule]
Amount	real, range [0.0 - 100.00]
Location	string(40), one-of [intergranular continous pore-lining, intergranular discontinous pore-lining, intergranular pore-filling, intergranular discrete, intergranular displacive, intragranular replacive, intragranular pore-lining, intragranular pore-filling, intragranular discrete crystals, intragranular displacive, moldic pore-lining, moldic pore-filling, oversized pore-lining, oversized pore-filling, grain fracture-filling, grain fracture-lining, rock fracture-filling, rock fracture-lining, concretions/nodules, massive beds/lenses]
Modifier	string(40), one-of [dissolved, zoned, fractured, recrystallized]
Description	string(256)
Concept Diagenetic-Nomenclature	
Is-a	Object
MineralName	list-of [instance-of < Concept Diagenetic-Constituents >]
Concept Diagenetic-Constituents	
Is-a	Object
MineralName	string(80), one of [Albite, Analcime, Anhydrite, Aragonite, Attapulgite/sepiolite, Authigenic clay mineral undifferentiated, Baryte, Berthierine/chamosite, Bitumen, Bornite, Calcite, Carbonate pseudomatrix, Carbonate undifferentiated, Carnalite, Carnotite, Celadonite, Celestite, Chabasite, Chalcedony, Chalcocite, Chalcopyrite, Chlorite, Chlorite/smectite, Clay pseudomatrix, Collophane, Diagenetic amphibole, Diagenetic anatase, Diagenetic apatite, Diagenetic brookite, Diagenetic carbonate, Diagenetic carbonate undifferentiated, Diagenetic clay undifferentiated, Diagenetic epidote]

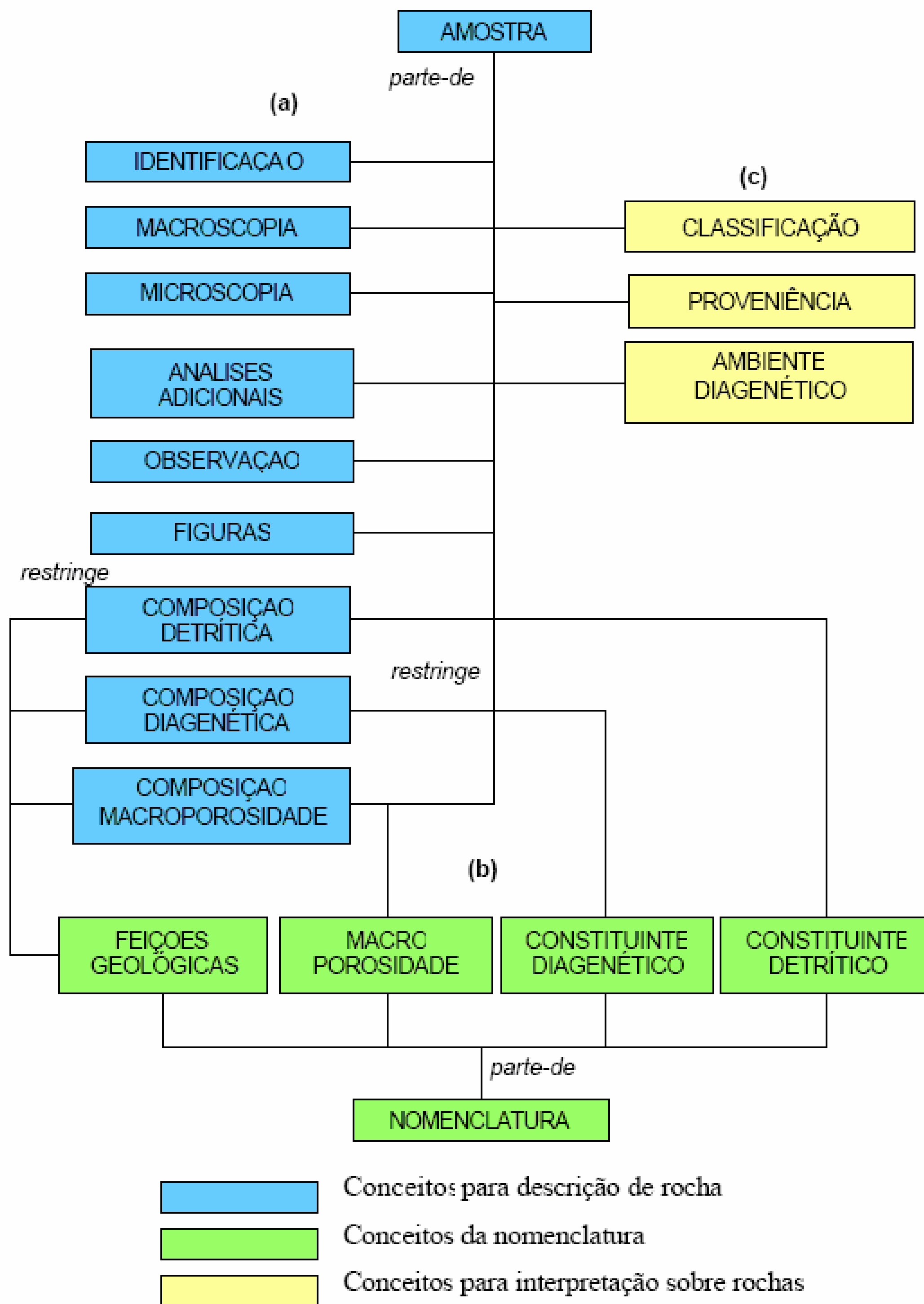

Figura 5.3: Conceitos do Sistema Petroledge (Adaptado de ABEL, 2001).

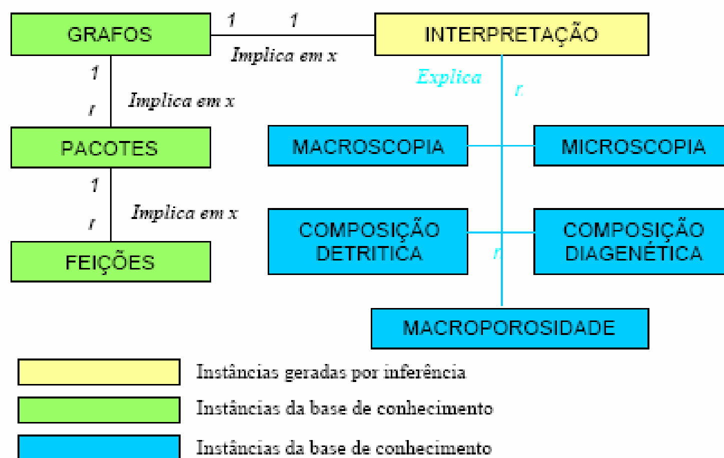

Figura 5.4: Metamodelo de Grafo de Conhecimento para Interpretação (ABEL, 2001).

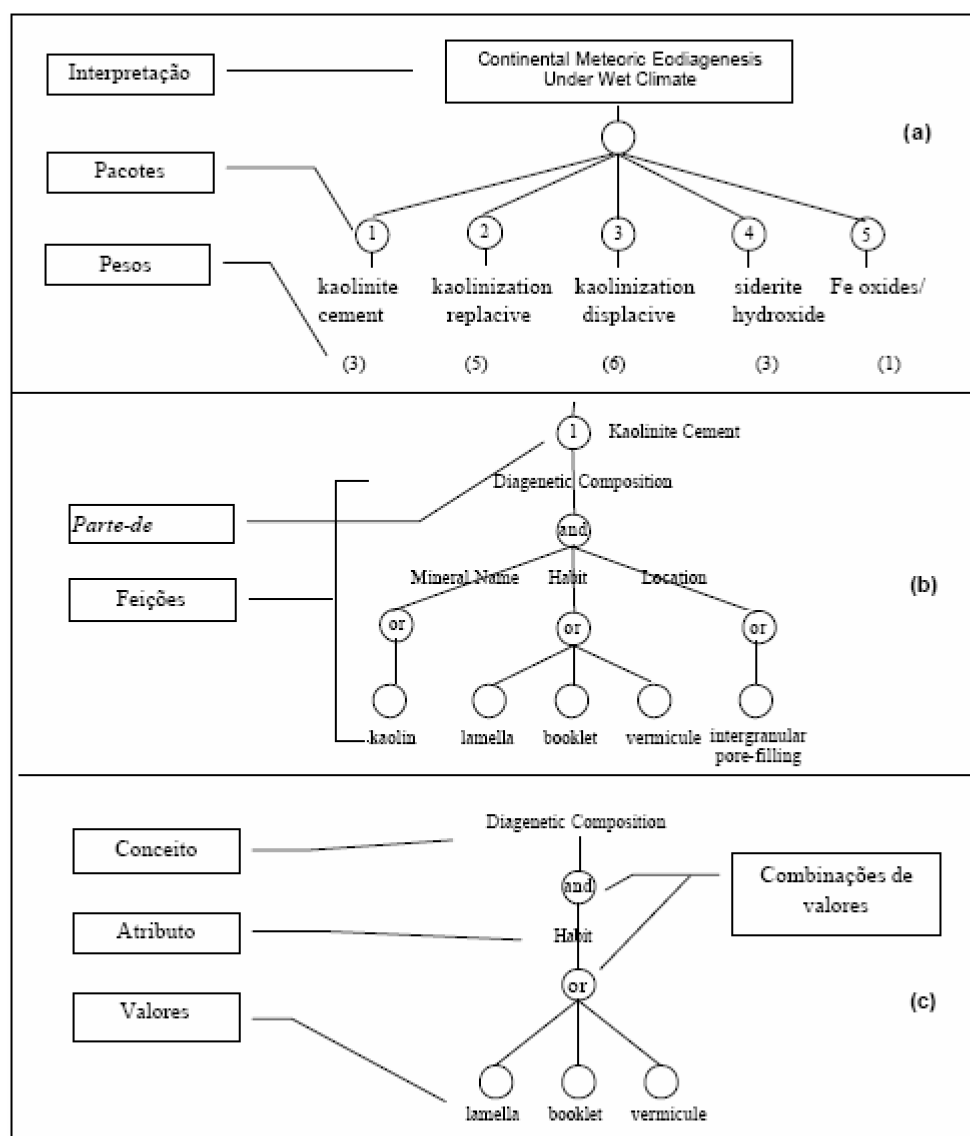

Figura 5.5: Exemplo de Grafo de Conhecimento (ABEL, 2001).

5.1.2 Componentes de Tarefa e Inferencial

O sistema Petroledge é capaz de resolver automaticamente tarefas de interpretação e classificação. Nessa validação, somente a tarefa de interpretação é avaliada, pois ela apresenta maior quantidade de inferências, permitindo melhor comparação entre a implementação atual e a implementação por *k-aspects*.

A tarefa de interpretação por grafos, descrita na Figura 5.6, conforme Abel (2001, p. 170) ocorre da seguinte forma:

> Os dados da amostra são comparados sucessivamente a cada um dos grafos e, em cada um dos grafos, a cada um dos pacotes. O processo é não determinístico. Dependendo das feições que o usuário decidiu descrever, o sistema vai comparar sucessivamente com as respectivas feições descritas nos pacotes dos grafos, cortando a busca sempre que for possível descartar um pacote. A conclusão final é definida pelo conjunto das interpretações associadas aos grafos validados.

```
TAREFA  interpretação de ambiente diagenético  por geração e teste usando grafos de conhecimento;
        PAPEIS:
                ENTRADA: caso: "Amostra a ser interpretada"; ;
                           conjunto de normas: "Grafos de conhecimento"
                SAIDA:     interpretação: "Lista de ambientes diagenéticos";
        FIM TAREFA  avaliação;

MÉTODO DA TAREFA interpretação por encadeamento progressivo;
        REALIZA:        interpretação;
        DECOMPOSIÇÃO:
                INFERENCIAS : abstrai, seleciona, avalia, decompõe, especifica, compara;
        PAPEIS:
                INTERMEDIÁRIOS:
                caso-abstraído: "Dados da amostra selecionados pela interface"
                conceito-atributo-valor: ""Feições geológicas"
                conjunto de normas:  "Conjunto dos grafos de conhecimento
                norma: "Grafo selecionado para avaliação"
                expressões de domínio: "Pacotes visuais"
                expressão de domínio: "Pacote selecionado para comparação"
                pesos da expressão de domínio: "Peso do pacote no grafo"
                valor do conceito-atributo-valor: "Feição encontrada ou não encontrada"
                valor da expressão de domínio:  "Pacote encontrado ou não encontrado"
                valor da norma:  "Grafo validado ou não validado"
        ESTRUTURA DE CONTROLE:
                abstrai (caso -> caso abstraído)
        ENQUANTO  TEM-GRAFOS
        DO
                seleciona ( conjunto normas -> norma);
                decompõe (norma -> expressões de domínio);
                ENQUANTO TEM-PACOTES
                DO
                seleciona (expressões de domínio -> expressão de domínio);
                decompõe (expressão de domínio -> conjunto de conceito-atributo-valor)
                ENQUANTO  valor do conceito-atributo-valor <> falso
                DO
                        seleciona (conjunto de conceito-atributo-valor -> conceito-atributo-valor)
                        seleciona  (caso abstraído -> conceito-atributo-valor)
                        se conceito-atributo-valor <> nulo;     "Usuário descreveu a feição
                                                                  geológica"
                        então decompõe (conceito-atributo-valor -> valores)
                                valor do conceito atributo valor := compara (valor = valores ) ;
                valor da expressão de domínio := valor do conceito atributo valor ;
```

```
                    calcula (valor norma := soma (pesos das expressões de domínio) );
                    se valor norma > limiar
                      então especifica (interpretação := conclusão da norma)
                    FIM ENQUANTO
                    FIM ENQUANTO
        FIM ENQUANTO
FIM METODO DA TAREFA interpretação por encadeamento progressivo.
```

Figura 5.6: Modelo da Tarefa de Interpretação de Ambiente Diagenético (Adaptado de ABEL, 2001).

Essa tarefa de interpretação foi modelada usando o PSM de interpretação por grafos de conhecimento. Cada PSM pode ser visto como uma solução padrão para determinada tarefa que não deve apresentar dependências específicas com a interpretação realizada. Por exemplo, um PSM de interpretação poderia ser aplicado para interpretar um ambiente diagenético, mas também para interpretar quem é o criminoso de uma história policial.

O modelo de inferências usado nesse PSM (Figura 5.7) descreve como as interpretações são realizadas por uma sequência de inferências:

- A entrada para interpretação desse modelo é chamada de caso e corresponde uma amostra descrita pelo usuário do sistema (Figura 5.7a);
- As feições geológicas descritas (ex.: microscopia, composição) são selecionadas de acordo com os grafos de interpretação disponíveis (Figura 5.7b);
- Cada grafo assume o papel de norma na inferência. Cada um deles é então selecionado (Figura 5.7c) para avaliação;
- A avaliação é feita pela decomposição de cada grafo em pacotes (Figura 5.7d). Cada pacote corresponde a uma premissa a ser validada;
- Os pacotes são selecionados um a um para comparação (Figura 5.7e);
- Ao serem selecionados, os pacotes são decompostos (Figura 5.7f) em feições geológicas para que seja possível a comparação com as informações descritas pelo usuário (ex.: microscopia, composição). A decomposição gera triplas com a assinatura <conceito, atributo, valor>;
- As feições oriundas dos pacotes e as feições fornecidas pelo usuário são então comparadas (Figura 5.7g). Por exemplo, no grafo da Figura 5.5, uma feição do pacote *kaolinite cement* é <*diagenetic composition, habit, lamella*>. Essa feição durante a inferência é comparada com as existentes no caso fornecido pelo usuário;
- Caso o conjunto de feições validadas seja superior ao limiar necessário para validação do grafo (Figura 5.7h), a interpretação é confirmada (Figura 5.7i).

Os modelos apresentados atualmente foram implementados no Petroledge usando uma solução *ad-hoc,* pois as metodologias para construção de SCs não fornecem uma solução padrão para a implementação desses modelos na OO. A seção seguinte, 5.2, apresenta a implementação atual dos modelos descritos no sistema Petroledge para posterior avaliação.

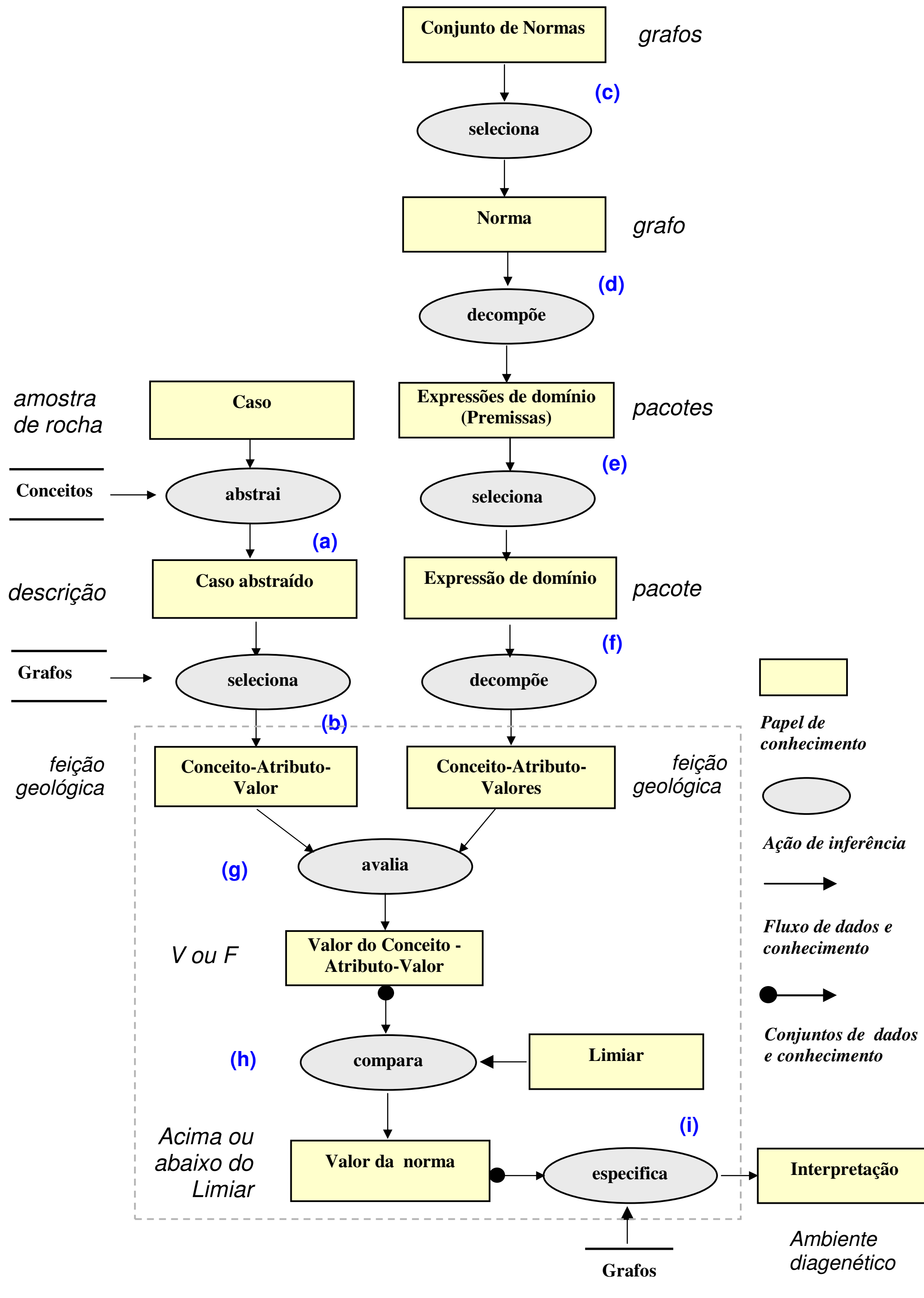

Figura 5.7: Modelo de Inferência do PSM de Interpretação de Ambientes Diagenéticos (ABEL, 2001).

5.2 Solução Atual

O sistema Petroledge é desenvolvido no paradigma da OO. A solução *ad-hoc* atualmente utilizada para implementação dos componentes do modelo conhecimento foi desenvolvida ao longo do projeto do sistema. Os principais problemas identificados nessa solução são os seguintes:

- **As estruturas definidas no modelo de conhecimento não são facilmente identificáveis no código-fonte da aplicação**: não é possível identificar facilmente o código-fonte que implementa os conceitos, atributos, *facets*, axiomas e regras. Por exemplo, no conjunto de duas mil classes dessa aplicação, não há marcação que identifique quais classes implementam conceitos e quais classes implementam regras. A falta de marcação torna praticamente impossível realizar a engenharia reversa do código de modo que um Engenheiro de Conhecimento possa verificar se o código-fonte atende exatamente a especificação do modelo de conhecimento
- **Dispersão de código:** o tratamento de *facets,* axiomas e regras precisa ser implementado manualmente pelo engenheiro de desenvolvimento do sistema. Por exemplo, se um *facet* que limita o valor máximo é especificado para um atributo chamado *idade*, todos os pontos no software, em que a valor desse atributo é modificado precisam invocar o método que valida se o valor é inferior ao valor máximo. Facilmente um engenheiro de desenvolvimento pode esquecer de definir essa invocação e um ponto em que o valor é modificado e gerar uma inconsistência na implementação do modelo. Por exemplo, se em um método que modifica o valor do atributo *idade*, por falha do engenheiro de desenvolvimento, não valida o valor, uma idade abaixo de zero poderia ser atribuída;
- **Implementação de PSMs dependem de mapeamentos entre modelo e respectiva implementação:** as implementações dos PSMs, devido a ausência de marcação no componente conceitual precisam fazer uso de mapeamentos elaborados manualmente pelos engenheiros de desenvolvimento. Por exemplo, como não é possível extrair em tempo de execução, o modelo abstrato (conceito, atributo, valor) das próprias instâncias do modelo, é necessário que o engenheiro de desenvolvimento forneça uma tabela de mapeamento que identifique qual classe implementa tal conceito e qual propriedade implementa tal atributo. Esse mapeamento é bastante suscetível a falhas. Se a implementação do modelo for alterada (atributo renomeado) e o mapeamento não for atualizado, as tarefas podem ser realizadas incorretamente. Um atributo pode não ser encontrado e, caso ele seja essencial para uma determinada interpretação ou classificação, a tarefa poderá indicar um resultado incorreto;
- **Implementação de PSM somente segue passos sequenciais:** a implementação do PSM de interpretação por grafos é constituída por uma sequência de inferências que são realizadas somente após a finalização da descrição. Parte dessas inferências (ex.: *avalia*), podem ser realizadas ao longo do preenchimento das instâncias do componente conceitual (ex.: entrada de dados de microscopia, de composição), evitando que o usuário tenha que, ao final, aguardar a realização de todas as inferências. Por exemplo, atualmente, a realização de todas inferências leva aproximadamente dois minutos para ser completada. A implementação do PSM de classificação sofre o mesmo problema, não explora a

atomicidade de certas inferências, que podem ser realizadas antes da finalização do preenchimento das instâncias do componente conceitual.

Para melhor caracterizar os problemas, o detalhamento da implementação atual é mostrado a seguir.

5.2.1 Implementação do Componente do Modelo Conceitual sem K-Aspects

A solução atual utilizada para implementação do componente do modelo conceitual segue o seguinte mapeamento:

- Conceitos são mapeados para classes. Entretanto, não há marcação nessas classes para que haja distinção do seu papel em relação a demais classes do SC;
- Atributos são mapeados para propriedades de classes. Entretanto, não há marcação que faça distinção entre propriedades que representam atributos e propriedades usadas para atender outros requisitos, por exemplo, auditoria e segurança;
- *Facets* são implementados por métodos de classes externas a própria classe do atributo que tem seu domínio definido pelas *facets*. Utiliza-se a *pattern validator* para realização de validações. Entretanto, o engenheiro de desenvolvimento deve obrigatoriamente invocar as validações em todos os pontos do *software* que podem modificar o valor de um atributo. A invocação em todos os pontos gera dispersão de código e é bastante suscetível a falhas, pois basta o engenheiro de desenvolvimento esquecer uma validação para o modelo ficar inconsistente;
- Axiomas são implementados por métodos localizados na própria classe que representam conceitos ou em classes externas (dificultando o rastreamento dos axiomas já implementados). A especificação das expressões dos axiomas definidas no modelo precisa ser manualmente traduzida para código-fonte pelo engenheiro de desenvolvimento. Essa conversão é suscetível a falhas e não preserva o construto axioma, pois ele é transformado para um método. Métodos dos axiomas não são distinguíveis de métodos que implementam outras funções, não há marcação para essa função;
- Regras são implementadas por métodos localizados nas próprias classes que representam conceitos. Os métodos das regras não são distinguíveis de métodos que implementam outras funções, não há marcação para essa função.

A Figura 5.8 e a Figura 5.9 mostram o código-fonte parcial de duas classes que implementam conceitos. Os comentários '//Conceito e //Atributo' apenas identificam que a classe e suas propriedades representam o conceito e sues atributos; entretanto, são apenas comentários textuais que facilmente podem ser esquecidos e também não são tipados, trazendo problemas para interpretação desse modelo pelo computador. Já os comentários '//PartOf, //Facet, //Axioma e //Regra' identificam pontos em que houve quebra na preservação de estrutura, pois o construto do componente conceitual foi transformado em validações ou métodos. Observa-se uma grande quantidade de chamadas de métodos e validações que são implementadas manualmente, essas chamadas e validações implementam a semântica dos *facets*, axiomas e regras definidas no modelo. Observando essas classes pode-se identificar claramente que todos os construtos para *facets,* axiomas e regras definidos no componente conceitual acabam sendo transformados manualmente para validações e métodos, tornando praticamente impossível a engenharia reversa dessa implementação. Além disso, a necessidade de

implementação manual aumenta consideravelmente o risco de o engenheiro de desenvolvimento cometer um erro e não adicionar uma validação especificada no modelo.

```
//Conceito
public class ThinIdentificationImpl{
   private SampleImpl thinParent = null;  //PartOf
   private String strUnit = null;  //Atributo
   private String strBasin = null; //Atributo
   private Float fltTop = null; //Atributo
   private Float fltBase = null; //Atributo
   private List<String> colUses  = new ArrayList();
   //Demais atributos omitidos
   public ThinIdentificationImpl(Thin parent) {
      //PartOf
      if (parent != null){
         thinParent = (Thin) parent;
         thinParent.setThinIdentification(this);
      } else {
         throw new IllegalArgumentException();
      }
   }
   //Unit
   public void setUnit(String unit) {
      if (!validateUnit(unit)){
           throw new KnowledgeException() ;
      }
      strUnit = unit;
   }
   public String getUnit(){..}
   //Basin
   public void setBasin(String basin) {
      if (!validateBasin(basin)){
           throw new KnowledgeException() ;
      }
      strBasin = basin;
   }
   public String getBasin(){..}
   //TopDepth
   public void setTop(Float depth) {
      if (!validateTop(depth)){
           throw new KnowledgeException() ;
      }
      fltTop = depth;
   }
   public Float getTop(){..}    }
   //Uses
   public void addUses(String use) {
      if (!validateUse(use)){
           throw new KnowledgeException() ;
      }
      colUses.add(use);
   }
   //Parent
   public Parent getParent() {..}
   //Demais métodos set/get omitidos.
}
```

```
//Validator para o conceito ThinIdentification.
public class ThinIdentificationValidator{
   boolean validateUnit(String value) {
       boolean isValid = true ;
       isValid &= (value == null ? false : true); //Facet
       isValid &= (value.length < 60); //Facet
       isValid &= value.match("[a-zA-Z]*-[0-9]");
//Facet
        return isValid ;
   };
   boolean validateBasin(String value) {
       boolean isValid = true ;
       isValid &= (value == null ? false : true); //Facet
       isValid &= (value.length < 40); //Facet
       return isValid;
   };
   boolean validateTop(Float value) {
       boolean isValid = true ;
       isValid &= (value == null ? false : true); //Facet
       isValid &= (value < 0 ? false : true; //Facet
       isValid &= (value > 9999.99 ? false : true; //Facet
       isValid &= (value > fltBase?false:true); //Axioma
       return isValid ;
   };
   boolean validateBase(Float value) {
       boolean isValid = true ;
       isValid &= (value == null ? false : true ; //Facet
       isValid &= (value < 0 ? false : true ; //Facet
       isValid &= (value > 9999.99?false : true ; //Facet
       isValid &= (value < fltTop ?false:true;) //Axioma
       return isValid;
   };
   boolean validateUses(String uses) {
       Collection validValues = ThinDAO.getUses();
       if (colUses.size() == 3){
           throw new          KnowledgeException('Máximo
atingido');
       }
       if ( !validUses.contains(uses)){return false ;}
   };
   boolean validateField(String value) {
       boolean isValid = true ;
       isValid &= (value == null ? false : true); //Facet
       isValid &= (value.length < 20); //Facet
       return isValid;
   };
   boolean validateOrigin(String value) {..};
   boolean validatePlace(String value) {..};
   boolean validateStage(String value) {..};
   boolean validateCountry(String value) {..};
   boolean validateCoreNumber(String value) {..};
   boolean validateBoxNumber(String value) {..};
   boolean validateSampleType(String value) {..};
}
```

Figura 5.8: Fragmento da Implementação Atual do Conceito ThinIdentification.

```
//Conceito
public class MicroscopicImpl {

   private SampleImpl pdParent; //PartOf
   private OntologyIntervalarValue<String>
oivGrainsize = null;
   private OntologyIntervalarMetricValue<Float>
oimNumericGrainsize = null;
   private TreeMap<Integer,
OntologyIntervalarMetricValue<Integer>> mapScales =
new TreeMap();
   private TreeMap<Integer, String> mapStructures =
new TreeMap();
   private TreeMap<Integer, String>
mapModalGrainSizes = new TreeMap();
   private TreeMap<Integer, String>
mapRoundnessModifiers = new TreeMap();
   private TreeMap<Integer, Float>
mapNumericModalGrainSizes = new TreeMap();
   private Float fltGravel = null;
   private Float fltMud = null;
   private Float fltSand = null;
   private String odvSorting = null;
   private Float fltSortingNumeric = null;
   private String odvSupport = null;
   private String odvPacking = null;
   private Integer intPackingIndex = null;
   private String odvPointContacts = null;
   private Float fltPointContacts = null;

   /** Creates a new instance of MicroscopicImpl */
   public MicroscopicImpl(SampleImpl parent) {
      setParent(parent);
   }
   public void setGrainSize(
        OntologyIntervalarValue<String>
           grainsize) {
      oivGrainsize = grainsize;
   }
   public void setScale(Integer importance,
         OntologyIntervalarMetricValue<Integer>
         scale) {
     //Facets e axiomas no validator
      if (mvValidator.validateScale(importance)){
         mapScales.put(importance, scale);
      } else {
         throw new IllegalArgumentException();
      }
   }
   public void setStructure(Integer importance,
       String  structure) {
     //Facets e axiomas no validator
      if (mvValidator.validateStructure(importance)){
         mapStructures.put(importance, structure);
      } else {
         throw new IllegalArgumentException();
      }
   }
public void setNumericModalGrainSize(Integer
   importance, Float grainsize) {
   if (mvValidator.
        validateNumericModalGrainSize(importance)){
       mapNumericModalGrainSizes.put(importance,
       grainsize) } else {
      throw new IllegalArgumentException();
   }
}
public void setRoundnessModifier(Integer
  importance, String roundnessmodifier) {
  //Facets e axiomas no validator
   if (mvValidator.
        validateRoundnessModifier(importance)){
           mapRoundnessModifiers.put(importance,
            oundnessmodifier);
   } else {
      throw new IllegalArgumentException();
   }
}

public void setPackingIndex (Integer index) {
   intPackingIndex = index;
   //Regra
    if (intPackingIndex <= 40){
      instance.setPacking(MicroscopicD2D().
            getPackingByName("loose"));
   }else if(intPackingIndex > 40
                && intPackingIndex <= 55){
      instance.setPacking(MicroscopicD2D().
            getPackingByName("normal"));
   }else if(intPackingIndex > 55){
        setPacking(
          MicroscopicD2D().
               getPackingByName("tight"));
    }
}
public void setPointContacts(String contact) {
   odvPointContacts = contact;
   //Axioma
   if (fltPointContacts + fltSuturedContacts +
       fltConcaveConvexContacts + fltLongContact !=
       100){
           throw new RuntimeException(‘Knowledge
          Violation’ ) ;
}
 }
public void setPointContactsPercentage(Float
  percentage) {
   fltPointContacts = percentage;
}
public void setSupport(String support) {
   odvSupport = support;
}
public Parent getParent() {
   return pdParent;
}
  //Demais métodos omitidos
```

Figura 5.9: Fragmento da Implementação Atual do Conceito Microscopic.

5.2.2 Implementação dos Componentes de Tarefa e Inferencial sem K-Aspects

A implementação do componente de tarefa atualmente também engloba o componente inferencial. Esse acoplamento entre os dois componentes ocorre porque a implementação atual realiza todas as inferências somente ao final do preenchimento das instâncias do componente conceitual e não explora a possibilidade de realizar parte delas ao longo do próprio preenchimento. As inferências são vistas como sub-elementos do componente de tarefa, algo que reduz a reusabilidade das mesmas. Esse alto acoplamento ocorre porque elas acabam sendo implementadas por métodos que fazem parte da própria implementação do componente de tarefa.

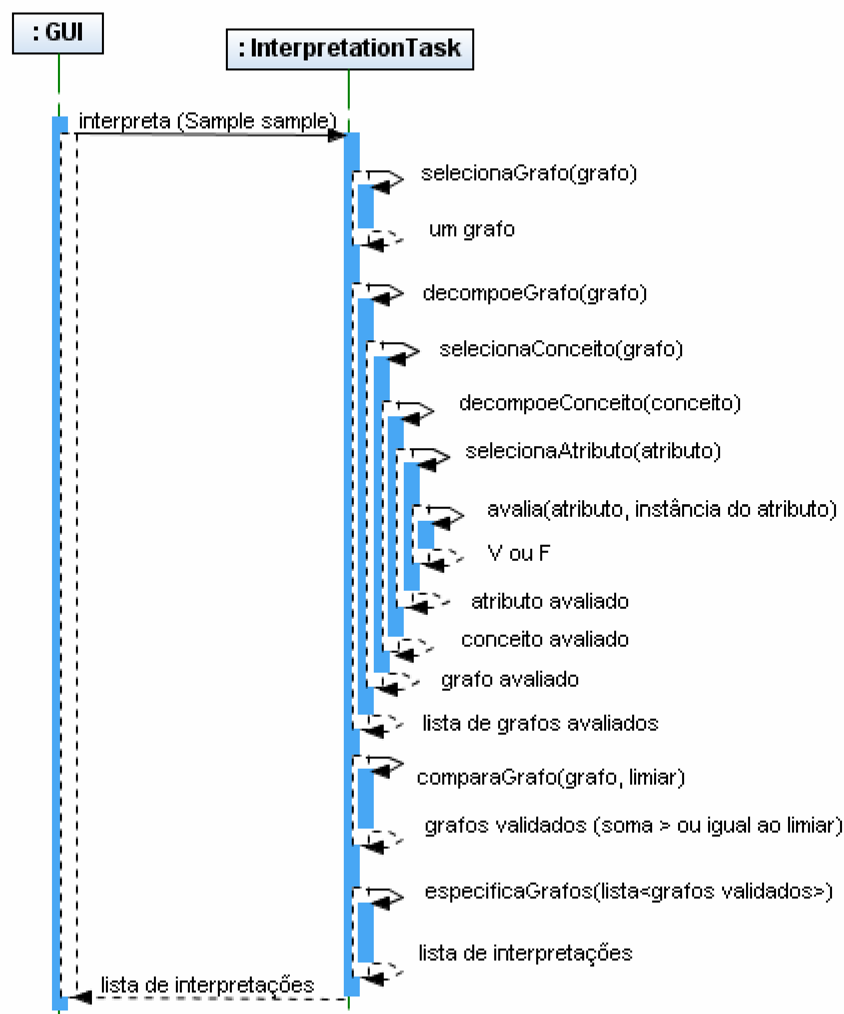

Figura 5.10: Diagrama de Sequência da Implementação do PSM de Interpretação.

Para facilitar a compreensão da implementação do PSM para interpretação por grafos e devido à extensão do código-fonte desse PSM, optou-se por apresentar o digrama de sequência (Figura 5.10) atual dessa tarefa para posterior comparação com a abordagem por *k-aspects*.

A realização da tarefa somente inicia quando o usuário solicita a interpretação de uma determinada amostra. A interpretação inicialmente obtém todos os modelos de interpretação (os grafos de interpretação existentes) e seleciona sequencialmente cada um dos grafos. Para cada grafo, um conceito é selecionado e decomposto em atributo, esses atributos são decompostos até uma lista de valores. Para cada atributo, pelo menos

um valor precisa ser encontrado na amostra para que ele seja validado (os valores correspondem a evidências de interpretações). A inferência *avalia* é responsável por verificar se o valor existe na amostra. Para avaliar e validar o valor, o mapeamento entre implementação e modelo abstrato, elaborado manualmente pelo engenheiro de desenvolvimento, é utilizado. Esse mapeamento, na implementação atual, indica em qual tabela e campo do banco de dados está localizado o valor procurado, para então ser procurado. Esse mapeamento aponta para o banco de dados para utilizar os mecanismos de consultas disponíveis nos bancos de dados e evitar que o engenheiro de desenvolvimento tenha que utilizar uma série de chamadas por reflexão estrutural à instância que está sendo validada.

Quando todos os atributos são avaliados como *V*, o conceito é considerado validado. Se a soma dos pesos dos conceitos for superior ao limiar determinado para a interpretação (*inferência compara*), a interpretação é considerada válida. Caso contrário, a interpretação não é validada e não é apresentada para o usuário. Todas interpretações validadas são apresentadas ao usuário no final do processo. Atualmente, em um ambiente de produção em que o sistema opera, esse processo leva em média dois minutos para ser realizado, devido à latência do banco de dados e ao grande número de consultas realizadas para a validação de cada interpretação (grafo).

5.3 Solução com K-Aspects

Os problemas definidos na seção 5.2 podem ser reduzidos/eliminados com o uso da abordagem *k-aspect*. O uso de anotações para marcação do código-fonte referente a implementação do modelo de conhecimento permite a distinção clara dos elementos oriundos desse modelo. Além disso, o uso de aspectos para tratamento da semântica desses construtos elimina a dispersão de código relacionada ao tratamento das funcionalidades transversais a própria implementação do modelo.

Tabela 5.2: Quantidade de Funcionalidades Transversais Tratadas via Aspectos.

Conceito	Tratamentos via Aspectos
ThinIdentification	120
Microscopic	263
Macroscopic	84
DiageneticComposition-Item	47
PrimaryComposition-Item	33
PoreComposition-Item	39
Total	93
AdditionalAnalysis	9
PoreSystem	13

A Figura 5.11, obtida a partir da biblioteca *AspectJ*, mostra a implementação de três conceitos do componente conceitual. Para cada classe, as linhas que apresentam diferentes tons de cinza mostram pontos em que aspectos estão agindo e eliminando a necessidade de implementação manual do engenheiro de desenvolvimento. Linhas em que múltiplos tons são apresentados indicam que mais de um aspecto está atuando nessa linha. Os aspectos detectados são apresentados na legenda da figura, cada um com uma cor diferente. As regiões sombreadas são pontos do código em que funcionalidades transversais foram encapsuladas via aspectos, eliminando a necessidade de

implementação manual pelo engenheiro de desenvolvimento. As porções em branco são pontos em que funcionalidades transversais não foram detectadas, nessa implementação, correspondem principalmente aos métodos *getters* (métodos que retornam valores de propriedades), nos quais não é feita nenhuma validação, nem axiomas e regras são aplicados. Para demonstrar o impacto positivo do uso dessa abordagem, a Tabela 5.2 apresenta para implementações de conceitos, o número de pontos em que funcionalidades transversais (validações de *facets*, axiomas, aplicações de regras, realização de inferências) foram tratadas por aspectos, evitando que o engenheiro de desenvolvimento tivesse que implementar manualmente chamada a métodos ou validações.

Considera-se o tratamento das *facets,* axiomas e regras como funcionalidades transversais a implementação do modelo. A implementação do modelo deve apenas ser responsável por refletir os construtos do próprio modelo, a semântica deve ser tratada via aspectos, evitando tarefas manuais e repetitivas de implementação.

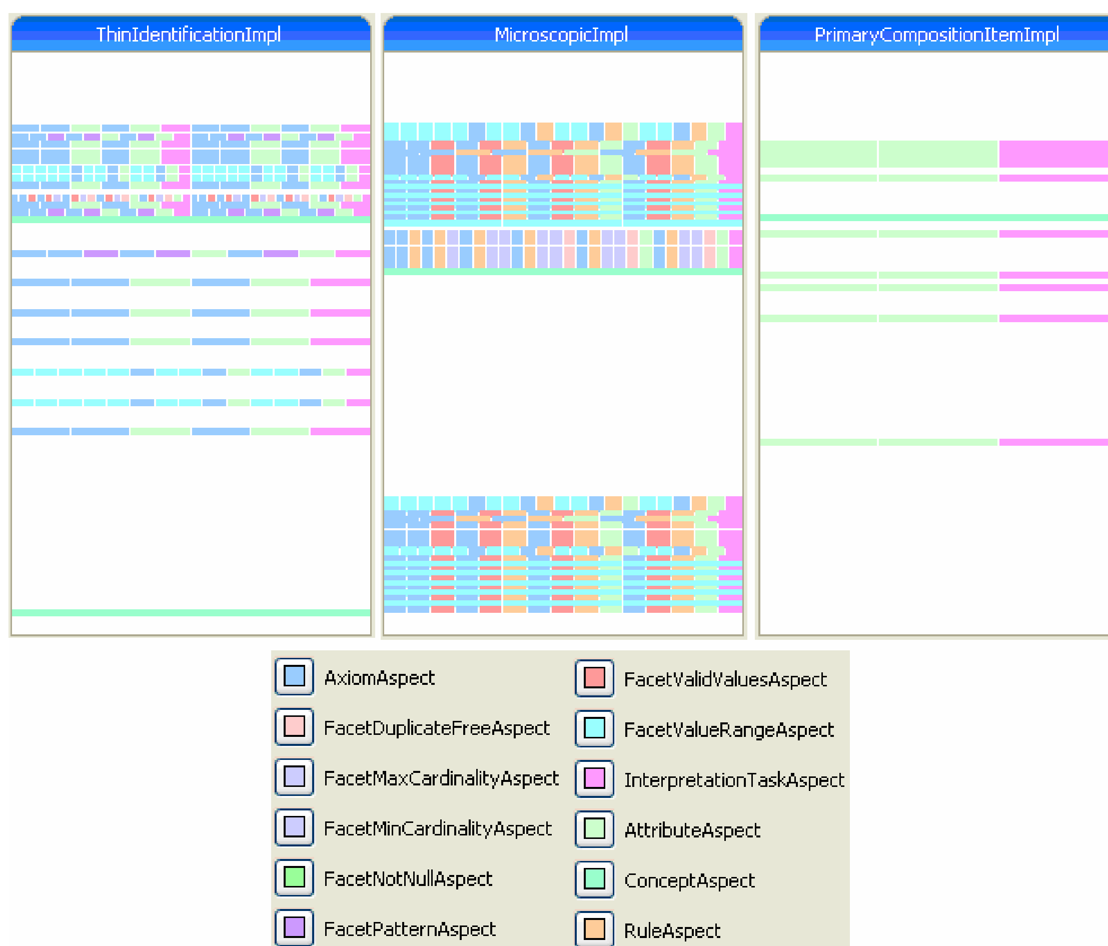

Figura 5.11: Aspectos identificados na implementação de três conceitos.

5.3.1 Implementação do Componente do Modelo Conceitual com K-Aspects

A implementação do componente do modelo conceitual usando *k-aspects* reduz significativamente o código-fonte de implementação do modelo, já que o engenheiro de desenvolvimento não mais precisa implementar manualmente as validações e chamadas a métodos para tratar regras, axiomas e etc. O tratamento todo é feito via aspectos, já previamente fornecidos como biblioteca.

As classes de validações, construídas pelo engenheiro de desenvolvimento, para cada conceito são eliminadas, já que as validações passam a ser realizadas pela biblioteca de tratamento de *facets*, axiomas e regras. As classes oriundas da implementação do componente conceitual são claramente identificáveis a partir das anotações utilizadas e podem ser recuperadas em tempo de execução pela própria aplicação. A obtenção dessas anotações em tempo de execução é necessária para o tratamento adequado das anotações. Por exemplo, quando a anotação *@FacetValidValues* apresenta a propriedade *source* preenchida, em tempo de execução o sistema precisa recuperar o valor de *source*, para então obter todos os valores válidos para um determinado atributo.

A Figura 5.12 e a Figura 5.13 apresentam implementações de dois conceitos usando *k-aspects* e *k-annotations*. Pode-se observar que todas as chamadas a *validators* foram eliminadas. Além disso, as validações, execução de regras e execução de axiomas que eram implementadas diretamente nos métodos *set* (modificam valores de atributo) foram todas suprimidas. Essas supressões são possíveis pelo uso de aspectos, já que são esses que passam a tratar esses pontos, evitando que o engenheiro de desenvolvimento tenha que fazê-los manualmente.

Na Figura 5.13 pode-se identificar que uma classe adicional (*MicroscopicPackingRules*) foi criada para implementar as regras aplicadas sobre o atributo *Packing*. Recomenda-se que regras aplicadas sobre um mesmo atributo sejam implementadas em uma mesma classe e o nome dessa classe identifique o atributo a qual a regra aplica-se. Essa prática facilita a identificação das regras aplicadas sobre cada conceito/relação/atributo. Deve-se também evitar a criação de uma única classe que implemente todas as regras para um determinado conceito/relação, pois a centralização em uma única classe pode dificultar o rastreamento das regras implementadas.

```
@Concept
@PartOf(SampleImpl.class)
@Axiom(specification="(ThinIdentification.TopDepth <=
ThinIdentification.BaseDepth ) || ( ThinIdentification.BaseDepth == 0 )")
public class ThinIdentification {
    @FacetNotNull
    SampleImpl thinParent = null;

    @Attribute
    @FacetMaxLength(length=20)
    String thinID = "";

    @Attribute
    @FacetPattern(pattern="[a-zA-Z]*-[0-9]")
    @FacetMaxLength(length=60)
    String unit = "";

    @Attribute
    @FacetMaxLength(length=40)
    String basin = "";

    @Attribute
    @FacetMaxLength(length=20)
    String field = "";

    @Attribute
    @FacetMaxLength(length=20)
    String well = "";
```

```
/** Creates a new instance */
public ThinIdentification(
  SampleImpl parent){
    setParent(parent);
}

public String getUnit() {
    return unit;
}
public void setUnit(String u) {
    unit = u;
}

public String getBasin() {
    return basin;
}
public void setBasin(String b)
{
    basin = b;
}

public String getField() {
    return field;
}
public void setField(String f) {
   field = f;
}
```

```java
    @Attribute
    @FacetMaxExclusive(9999.99f)
    @FacetMinExclusive(0f)
    Float topDepth = 0f;

    @Attribute
    @FacetMaxInclusive(9999.99f)
    @FacetMinInclusive(0f)
    Float baseDepth = 0f;

    @Attribute
    @FacetMaxLength(length=40)
    String place = "";

    @Attribute
    @FacetMaxCardinality(cardinality = 3)
    @FacetDuplicateFree
    @FacetValidValues(values={"Depositional", Diagenetic",
  "Ecologic","Provenance", "Reservoir", "Stratigraphic", "Other use"})
    List<String> use = new ArrayList<String>();

    @Attribute
    @FacetMaxLength(length=20)
    String petrographer = "";

    @Attribute
    @FacetPattern(pattern="[0-9][0-9]/[0-9][0-9]/[0-9][0-9][0-9][0-9]")
    String date = "";
    public String getWell() {
      return well;
    }
    public void setOrigin(String
      o)
    {
      well = o;
    }

    public Float getBase() {
      return base;
    }
    public void setBase(Float depth) {
      base = depth;
    }

    public Float getTop() {
      return top;
    }
    public void setTop(Float depth) {
      top = depth;
    }
  }
```

Figura 5.12: Fragmento da Implementação com K-Aspects do Conceito ThinIdentification.

```java
@Concept
@PartOf(SampleImpl.class)
@Axioms( axioms = {
    @Axiom(expression="( Microscopic.Gravel +
    Microscopic.Sand + Microscopic.Mud == 100 ) ||
    (Microscopic.Gravel + Microscopic.Sand +
    Microscopic.Mud == 0 )"),
    @Axiom(expression="
        ( ( ( ThinIdentification.BaseDepth > 4000 ) &&
            ( Microscopic.Gravel > 10 ) ) ||
        ( ( ThinIdentification.BaseDepth <= 4000 ) &&
          ( Microscopic.Gravel <= 10 ) ))")
})
@Rules({@Rule(MicroscopicPackingRules.class),
        @Rule(MicroscopicSortingRules.class)}
)
public class Microscopic{
   @Attribute
   @FacetMaxInclusive(100f)
   @FacetMinInclusive(0f)
   Float gravel = 0f;

   @Attribute
   @FacetMaxInclusive(100f)
   @FacetMinInclusive(0f)
   Float mud = 0f;

   @Attribute
   @FacetMaxInclusive(100f)
   @Attribute
   @FacetValidValues(source=
      MicroscopicValueProvider.class)
   String longContacts = null;
   @FacetMaxInclusive(100f)
   @FacetMinInclusive(0f)
   Float nLongContacts = 0f;

   @Attribute
   @FacetMinCardinality(cardinality=0)
   @FacetMaxCardinality(cardinality=3)
   @FacetDuplicateFree
   TreeMap<Integer,
      OntologyIntervalarMetricValue<Integer>> scales
 = new TreeMap();

   @Attribute
   @FacetMinCardinality(cardinality=0)
   @FacetMaxCardinality(cardinality=3)
   @FacetDuplicateFree
   TreeMap<Integer, String> structures = new
 TreeMap();
   @Attribute
   @FacetMinCardinality(cardinality=0)
   @FacetMaxCardinality(cardinality=3)
   @FacetDuplicateFree
   TreeMap<Integer, String> roundnessModifiers = new
 TreeMap();
```

```
   @FacetMinInclusive(0f)
   Float sand = 0f;

   @Attribute
   @FacetValidValues(source=
     MicroscopicSupportValueProvider.class)
   String support = null;

   @Attribute
   @FacetValidValues(source=
      MicroscopicValueProvider.class)
   String sorting = null;

   @Attribute
   Float numericSorting = null;

   @Attribute
   @FacetValidValues(source=
      MicroscopicValueProvider.class)
   String roundness = null;

   @Attribute
   @FacetValidValues(source=
      MicroscopicValueProvider.class)
   String orientation = null;
   @Attribute
   @FacetValidValues(source=
      MicroscopicValueProvider.class)
   String packing = null;

   @Attribute
   @FacetMaxInclusive(100)
   @FacetMinInclusive(0)
   Integer packingIndex = null;

   @Attribute
   @FacetValidValues(source=
      MicroscopicValueProvider.class)
   String pointContacts = null;
   @FacetMaxInclusive(100f)
   @FacetMinInclusive(0f)
   Float nPointContacts = 0f;
```

```
   @Attribute
   @FacetMinCardinality(cardinality=0)
   @FacetMaxCardinality(cardinality=4)
   @FacetDuplicateFree
   TreeMap<Integer, String> modalGrainSizes = new
TreeMap();

   @Attribute
   @FacetMinCardinality(cardinality=0)
   @FacetMaxCardinality(cardinality=3)
   @FacetDuplicateFree
   TreeMap<Integer, Float> numericModalGrainSizes =
new TreeMap();

   //Getters/setters omitidos, pois são apenas
atribuições
}

//Regras
public class MicroscopicPackingRules {
   public MicroscopicPackingRules(){
   }
   public void runRules(Map context) throws
Exception{
       MicroscopicImpl instance =
          context.get("instance");
       Integer packingIndex =
          instance.getPackingIndex();
       if (packingIndex <= 40){
         instance.setPacking(MetaModelDAOFactory.
            getMicroscopicD2D().
              getPackingByName("loose"));
       }else if(packingIndex > 40
              && packingIndex <= 55){
         instance.setPacking(MetaModelDAOFactory.
            getMicroscopicD2D().
              getPackingByName("normal"));
       }else if(packingIndex > 55){
           instance.setPacking(
             MetaModelDAOFactory.
               getMicroscopicD2D().
                 getPackingByName("tight"));
        }
   }
}
```

Figura 5.13: Fragmento da Implementação com K-Aspects do Conceito Microscopic.

5.3.2 Implementação dos Componentes de Tarefa e Inferencial com K-Aspects

A implementação dos componentes de tarefa e inferencial buscou explorar a possibilidade de realizar inferências em paralelo ao preenchimento da instância de amostra de rocha pelo usuário e também aumentar a reusabilidade das inferências implementadas no projeto de validação dessa proposta.

Para tratar a realização das inferências em paralelo ao preenchimento da instância da amostra de rocha, optou-se por utilizar aspectos. Os aspectos evitam a dispersão de código que realizaria a chamada a classes e respectivos métodos que implementam as inferências necessárias para a interpretação. Essas chamadas precisariam ser inseridas em todos os pontos do software em que valores da instância da amostra corrente sofrem

modificações. Por exemplo, para a realização da inferência de *avaliação*, que valida o valor esperado para o atributo (definido no modelo de cada interpretação) com o valor existente na instância, todo ponto do software que modifica valor de atributo precisaria chamar as inferências de *avaliação* para verificar o novo valor encontrado.

Além disso, além dos aspectos que realizam inferências (ex.: *MatchAspect* e *EvaluateAspect*), foi criado um aspecto para cada tipo de tarefa (ex.: *InterpretationTaskAspect, ClassificationTaskAspect*). Esses aspectos são necessários para realizar a conclusão das tarefas e organizar a exibição dos resultados ao usuário. Por exemplo, o aspecto referente à tarefa de interpretação é responsável por verificar todos os gráficos validados e prepará-los para exibição ao usuário, como essa validação é custosa, pois requer a verificação de uma lista de conceitos e somatório de pesos, esse aspecto pode ser configurado para ser acionado somente a cada dez ou *x* modificações de valores de atributos (evitando verificações repetitivas a cada modificação de valor). Recomenda-se a criação de um aspecto para cada tarefa e deve-se evitar a realização de inferências nesses aspectos, eles devem utilizar os resultados das inferências realizadas pelos aspectos de cada inferência.

A realização da tarefa de interpretação usando essa abordagem, evitando mapeamentos manuais (o próprio modelo abstrato é fornecido através das anotações e acessível em tempo de execução), evitando acessos ao banco de dados para realização de inferências e evitando que as inferências sejam realizadas somente ao final do preenchimento da instância de amostra rocha, permitiu uma redução significativa no tempo de realização dessas tarefas. O tempo foi reduzido para a média de 20 segundos (implementação atual tem média de 2 minutos). A avaliação de desempenho detalhada não é o foco desse trabalho.

Essa organização em múltiplos aspectos para uma determinada tarefa visa explorar melhor a concorrência possível nessas tarefas e melhorar a reusabilidade de cada inferência. Os aspectos que implementam tarefas e inferências fazem parte da biblioteca desse trabalho, mas podem ser estendidos conforme as necessidades de cada projeto.

A seção 5.3.3 apresenta os benefícios trazidos pela implementação do Petroledge usando *K-Aspects* em comparação com a solução original, *ad-hoc*. A seção 5.4 define os benefícios e limitações da abordagem *K-Aspects* em termos gerais, quando aplicada a diferentes projetos de SCs

5.3.3 Benefícios da Implementação do Petroledge usando K-Aspects

A implementação do Petroledge usando *K-Aspects* em comparação a solução original, *ad-hoc* trouxe os seguintes benefícios:

- **Redução de dispersão e linhas de código:** o uso de aspectos para o tratamento da semântica associada aos construtos dos componentes do conhecimento impactaram diretamente na dispersão de código relacionado ao tratamento desses construtos. Conforme apresentado na Tabela 5.2, 701 pontos para tratamento de funcionalidades transversais foram identificados. A implementação na solução *ad-hoc* apresentava 4437 linhas, já a implementação usando *K-Aspects* reduziu o número de linhas para 2718, uma redução de 38%. Essa significativa redução é devido à eliminação da necessidade do engenheiro de desenvolvimento manualmente implementar métodos e classes relacionadas ao tratamento de *facets*, axiomas, regras e inferências. Todos esses métodos e grande parte das classes foram encapsulados em bibliotecas para reuso em outros

projetos. A invocação dessas bibliotecas passa a ser inserida automaticamente no código interpretado através da biblioteca de aspectos, também fornecida com essa proposta. A redução de linhas de código e centralização do código de tratamento em uma única biblioteca pode reduzir o custo de manutenção do sistema, pois, caso um problema seja identificado em um tratamento, basta apenas corrigi-lo na biblioteca. Na solução *ad-hoc*, devido a pontos de duplicação, um problema poderia se repetir em diversos pontos e exigir múltiplas correções. Destaca-se que a porcentagem de redução pode ser ainda maior quando o modelo possui um grande número de *facets*, axiomas e regras associados, pois cada elemento adicional implicaria em novas linhas de código implementadas manualmente na solução *ad-hoc*, o que não acontece quando utilizado *K-Aspects*;

- **Aumento da rastreabilidade entre modelo e implementação:** a utilização de anotações para marcar as implementações dos componentes de conhecimento do Petroledge permite que facilmente seja possível rastrear a implementação resultante a partir de um elemento do modelo. Por exemplo, para rastrear a implementação de um axioma em um determinado conceito, basta procurar a própria especificação do axioma no código-fonte, pois não há transformação entre a especificação e sua implementação. Para rastrear uma relação, basta localizar a anotação @*Relation* que tenha como nome a relação que se deseja encontrar. Aumentando a rastreabilidade, mais facilmente é possível identificar inconformidade entre o modelo e o código, pois basta localizar as estruturas usando as estruturas como referência;

- **Geração automática de documentação:** na solução *ad-hoc* era praticamente impossível extrair documentação específica à implementação do modelo de conhecimento. A documentação gerada era compreensível somente aos engenheiros de desenvolvimento, pois os construtos definidos no modelo não eram mantidos na implementação e acabam sendo transformados em classes, propriedades e métodos. Com o uso da ferramenta *KA-DocGen*, a documentação do modelo de conhecimento pode ser gerada automaticamente, essa documentação é compreensível também ao engenheiro de conhecimento, pois os construtos definidos no modelo e as estruturas desse modelo são mantidos e apresentados na documentação gerada pela ferramenta;

- **Aumento da comunicação entre engenheiros de conhecimento e de desenvolvimento:** a definição de um conjunto de anotações que utiliza a mesma nomenclatura utilizada no modelo de conhecimento facilita a comunicação entre os engenheiros de conhecimento e de desenvolvimento. Esses últimos passam a poder utilizar no código-fonte os mesmos construtos conhecidos pelos engenheiros de conhecimento. Desse modo, o mesmo vocabulário passa a ser utilizado por esses dois grupos de participantes;

- **Segmentação de realização das tarefas:** a organização das inferências em aspectos passou a permitir que, ao longo do próprio preenchimento das instâncias do modelo, possa-se acionar automaticamente a realização de algumas inferências. Esse acionamento permite segmentar a realização das tarefas, pois boa parte da mesma é realizada ao longo do preenchimento da instância, reduzindo o processamento necessário ao final do preenchimento para que a tarefa seja realizada e o resultado exibido ao usuário. O sistema passou a ter a

capacidade de exibir resultados de tarefas ao longo do próprio preenchimento. Essa segmentação através de aspectos não exigiu a implementação de chamadas explícitas para a realização de inferência. Por exemplo, o uso de aspectos evita que para toda modificação de valor de atributo, seja necessário explicitamente invocar a inferência *abstrai*;

- **Reuso de inferências:** a organização de parte das inferências em uma biblioteca evita que elas sejam reimplementadas em diferentes pontos do sistema, causando duplicação de código. O reuso permite que uma vez testada e validada uma inferência, ela possa ser usada em diferentes pontos, sem necessidade de novos testes. Essa redução na quantidade de testes e validações colabora para reduzir o custo de desenvolvimento e manutenção do sistema.

Os benefícios citados mostram que a implementação usando *K-Aspects* colabora para a redução de tarefas manuais na etapa de desenvolvimento e garante maior rastreabilidade entre modelos e implementações. A manutenção da coerência entre modelo e implementação é um ponto fundamental de um projeto de SC, pois os requisitos tendem a mudar frequentemente e os engenheiros de desenvolvimento devem ser capazes de refletir essas mudanças no código, sem demandar um longo período de desenvolvimento ou revisão. A geração automática de documentação facilita o processo de revisão da implementação e apóia a comunicação entre engenheiros de conhecimento e de desenvolvimento. A otimização do processo de implementação colabora para a redução de custos de projeto de SC.

5.4 Sumário do Capítulo 5

A comparação das abordagens apresentadas nas seções anteriores permite identificar os seguintes benefícios relacionados à abordagem *k-aspects*:

- **Suporte aos construtos do modelo conhecimento e preservação das estruturas:** o conjunto de *k-annotations* desenvolvido nessa proposta permite que os construtos do componente conceitual sejam preservados na implementação OO. As anotações identificam as estruturas definidas nos componentes do modelo de conhecimento claramente no código-fonte da aplicação. Por exemplo, a anotação @*Concept* permite que a implementação de um conceito seja reconhecida imediatamente pelo engenheiro de desenvolvimento e também permite que um programa faça esse reconhecimento. Já que essa anotação está disponível em tempo de execução e pode ser recuperada através de reflexão estrutural;
- **Eliminação da dispersão de código relacionado aos construtos do modelo de conhecimento:** a utilização de aspectos para tratar as funcionalidades transversais exigidas para implementar a semântica dos construtos de *facets,* axiomas e regras elimina o problema de dispersão de código existente na solução atual. Todo o código que era implementado manualmente pelo engenheiro de desenvolvimento para o tratamento dos construtos citados, passa a ser gerado automaticamente pelo costurador (ferramenta que conecta os módulos da aplicação com os aspectos para que as funcionalidades transversais sejam disponibilizadas nos pontos da aplicação especificados nos *pointcuts*);
- **Tratamento padronizado de *Facets*, Axiomas e Regras:** essa abordagem fornece uma biblioteca padrão para o tratamento de *facets,* axiomas e regras.

Essa biblioteca pode ser reutilizada em diversos projetos, evitando duplicação de código e diminuindo o risco de problemas relacionados a ela, já que ela foi previamente validada e não foi desenvolvida ao longo de um único projeto de SC;

- **Reusabilidade de Bibliotecas:** as bibliotecas de anotações, tratamento de anotações e aspectos foram concebidas para serem reutilizadas em diferentes projetos de SC, pois tratam construtos do modelo de conhecimento e não são dependentes do domínio de aplicação do sistema;
- **Geração automática de documentação:** a marcação clara dos elementos do modelo de conhecimento no código-fonte e a possibilidade de recuperação desses dados através de reflexão estrutural permitem que uma ferramenta de geração de documentação específica ao modelo de conhecimento seja disponibilizada. Desse modo, o engenheiro de conhecimento pode mais facilmente revisar o modelo e suas respectiva implementação, reduzindo os riscos de problemas de conformidade;
- **Exploração da concorrência para inferências evitando dispersão de código:** o uso de aspectos permite que inferências sejam realizadas ao longo do preenchimento das instâncias do modelo pelo usuário, evitando que todas sejam realizadas somente ao final do preenchimento. Isso visa aperfeiçoar a realização das tarefas sem exigir que uma série de chamadas manuais sejam inseridas no sistema, gerando uma grande dispersão de código, muitas vezes duplicado em diferentes pontos. Além disso, a realização das inferências sobre a própria instância do modelo no paradigma OO evita que mapeamentos tenham que ser feitos para outros paradigmas (ex.: relacional) gerando uma carga adicional de manutenção desses mapeamentos durante a etapa de manutenção/evolução de um SC;
- **Encapsulamento da implementação do modelo de inferências:** a implementação de cada inferência como aspecto permite que elas sejam reusadas por diferentes projetos e diferentes tarefas. De fato, elas são realizadas, previamente a tarefa, desse modo, a tarefa que precisa dos resultados dessas inferências já os terá disponíveis quando for acionada, pois os aspectos das inferências são acionados antes das mesmas. Isso ocorre porque as tarefas podem ser configuradas para serem executadas somente após um certo número de modificações nas instâncias do modelo, permitindo que inferências sejam realizadas previamente;
- **Simplificação de Testes de SCs:** as vantagens citadas acima impactam positivamente na definição de testes para os componentes do modelo de conhecimento. A identificação clara dos componentes do modelo e a redução de dispersão de código facilita a criação de testes unitários para os mesmos. A própria especificação do modelo identifica uma série de características que podem ser testadas unitariamente. Por exemplo, se na especificação o modelo define que um atributo não pode ter valores acima de 100, pode-se criar um teste unitário que tenta preencher o atributo com um valor maior que 100. Esse teste deve esperar que uma exceção seja lançada nesse caso. Caso não ocorra exceção, identifica-se uma falha na implementação e, nessa abordagem, essa falha indica a ausência da anotação *@FacetMaxInclusive(100).* Já que o tratamento da anotação é parte de uma biblioteca previamente validada. A verificação entre o

modelo e sua implementação também é facilitada pela geração automática de código, que reduz a necessidade de revisão diretamente em código. Testes em relação a tarefas continuam a demandar casos reais de problemas fornecidos pelo especialista do domínio do SC, para que o sistema aponte a solução e então esta seja comparada com a resposta previamente fornecida pelo especialista;

- **Simplificação da Manutenção de SCs:** as vantagens citadas acima também impactam positivamente no processo de manutenção de SCs. A identificação clara, o encapsulamento dos componentes do modelo e a geração automática de documentação facilitam a manutenção dos mesmos, pois quando inconformidades são detectadas nesses componentes, são mais rapidamente rastreados tanto pelos engenheiros de desenvolvimento quanto pelos engenheiros de conhecimento. Os engenheiros de desenvolvimento passam a compartilhar o vocabulário dos engenheiros de conhecimento (por exemplo, conceito, atributo e não mais classe e propriedade), melhorando a comunicação para resolução de quaisquer dúvidas associadas à manutenção desses componentes. A redução da dispersão de código e geração de código também evita que um problema detectado esteja replicado em diversos outros componentes, pois parte de tarefas repetitivas manuais foram substituídas pela geração automática;

Apesar dos diversos benefícios existentes no uso dessa abordagem, deve-se destacar algumas limitações que devem ser levadas em consideração por projetos que optem por utilizá-la:

- **Dificuldade de depuração:** o processo de costura insere uma quantidade significativa de código referente aos diversos aspectos. Esse código não é visualizável claramente pelos engenheiros de desenvolvimento, pois ele é gerado somente após a compilação do código inicial. Desse modo, em certos casos, pode haver certa dificuldade em realizar a depuração do código. Essas limitações são inerentes ao paradigma OA e existem em todos os projetos que o utilizam. Todo novo paradigma exige um exercício mental dos engenheiros de desenvolvimento, para que ele seja corretamente assimilado e seu potencial seja corretamente utilizado. Espera-se, que a evolução desse paradigma ofereça melhores ferramentas para o tratamento da depuração;

- **Evolução/manutenção das bibliotecas:** a evolução/manutenção das bibliotecas desenvolvidas nessa proposta pode impactar nos projetos que fazem uso de versões anteriores dessas bibliotecas. Desse modo, assim como qualquer outra biblioteca, a evolução deve manter ao máximo as funções que as versões prévias ofereciam, sem exigir manutenção no código que faz uso delas. Por exemplo, as bibliotecas que compõem a linguagem Java, ao evoluírem de versão, mantêm todas as funções que existiam anteriormente, mesmo que essas funções sejam estendidas, as funcionalidades pré-existentes continuam apresentando o mesmo comportamento.

6 CONCLUSÃO

Projetos de SCs envolvem altos custos e riscos. Mesmo com a existência de uma série de metodologias para a realização desses projetos, identificou-se uma lacuna na etapa de desenvolvimento desse tipo de sistema: diferentes linguagens e metodologias são propostas para a construção do modelo de conhecimento, mas nenhuma delas oferece uma solução padrão para a implementação desse modelo no paradigma mais utilizado na área de desenvolvimento de software, o paradigma OO.

Esse trabalho definiu uma proposta padrão, chamada de *k-aspects* para a implementação do modelo de conhecimento no paradigma OO utilizando os recursos providos pela OA. O uso de OA visa melhor tratar as funcionalidades transversais referentes ao tratamento da implementação do modelo de conhecimento, evitando dispersão e duplicação de código, antes manualmente inserido pelos engenheiros de desenvolvimento. Esses tratamentos se referem à semântica de cada construtor utilizado nesse tipo de modelo e também ao tratamento de tarefas que SCs precisam realizar.

Tarefas necessitam de uma série de inferências para serem realizadas e essas inferências podem ser utilizadas por diferentes tarefas. Desse modo, permitir que elas sejam reutilizadas evita custos relacionados à manutenção de código duplicado e aumenta a reusabilidade dos componentes em diferentes projetos.

O uso de anotações visa fornecer o próprio modelo abstrato da implementação junto à sua implementação, para que inferências que dependem desse modelo possam ser realizadas no próprio paradigma OO, sem necessidade de mapeamentos. Além disso, as anotações permitem a definição de construtos não suportados nativamente pela OO nesse paradigma. O suporte a esses construtos é essencial para que as estruturas definidas no modelo de conhecimento sejam facilmente identificáveis na implementação resultante, facilitando o processo de manutenção de coerência entre modelos e respectivas implementações.

As bibliotecas construídas nesse trabalho organizam um conjunto de anotações, aspectos e componentes que tratam elementos comuns a diferentes projetos de SC, por exemplo, tratamento de *facets,* axiomas e inferências. As bibliotecas dessa proposta, independentes de um projeto específico de SC, oferecem uma base de elementos para a construção de SCs, evitando múltiplas implementações para o tratamento de uma mesma funcionalidade. O uso dessas bibliotecas, validadas anteriormente em projetos de SC, reduz os riscos associados a soluções *ad-hocs* e também reduz os riscos associados a uma nova implementação a cada projeto, por exemplo, existência *bugs*. Espera-se que a utilização dessa biblioteca em diferentes projetos facilite a comunicação entre as equipes de desenvolvimento e os engenheiros de conhecimento, já que os termos utilizados buscam aumentar a rastreabilidade entre os modelos e suas implementações.

Espera-se que *k-aspects* e o conjunto de bibliotecas elaboradas nesse trabalho sejam utilizados como solução de base para diferentes SCs que utilizem o paradigma OO, evitando-se os riscos e problemas associados a soluções *ad-hoc.*

6.1 Trabalhos Futuros

A proposta de *K-Aspects* foi desenvolvida tendo como base diferentes linguagens para modelagem de conhecimento. Apesar de uma série de construtos serem suportados por essa proposta, a análise detalhada das principais linguagens disponíveis para modelagem de conhecimento pode identificar a necessidade de novas *k-annotations* e extensões das bibliotecas de tratamento dos construtos.

Ferramentas para construção de modelos de conhecimento, como *Protégé* poderiam ser estendidas para suportar a geração de código em diferentes linguagens OO. A geração automatizada de código pode reduzir a tarefa manual de implementação dos componentes do modelo na linguagem OO.

A validação desse trabalho foi realizada em um SC de grande porte que faz amplo uso dos construtos do modelo de conhecimento, mas novas características eventualmente poderiam ser identificadas em outros projetos. Desse modo, a biblioteca de tratamento de *facets*, axiomas e regras pode requerer novas funcionalidades identificáveis pela validação dessa proposta em outros projetos de SCs.

Especificamente a implementação das regras definidas no componente conceitual ainda é convertida manualmente para a linguagem OO usada no desenvolvimento do SC. Uma linguagem para definição de regras e uma biblioteca para o *parser* e avaliação dessas regras evitaria a tarefa manual de conversão, muito suscetível a erros e de difícil engenharia reversa para comparação com a regra especificada no modelo.

REFERÊNCIAS

ABEL, M. **Estudo da Perícia em Petrografia Sedimentar e sua Importância para a Engenharia de Conhecimento**. Porto Alegre: UFRGS, 2001. 239 f. Tese (Doutorado) – Programa de Pós-Graduação em Computação, Universidade Federal do Rio Grande do Sul, Porto Alegre, 2001.

ANGELE, J. et al. Developing Knowledge-Based Systems with MIKE. **Automated Software Engg**. v. 5, n. 2, p. 389-418, out. 1998.

ANNOTATIONS. [S.l.]: Sun Microsystems, 2004. Disponível em: <http://java.sun.com/j2se/1.5.0/docs/guide/language/annotations.html>. Acesso em: 20 fev. 2006.

ANTONIOU, G.; HARMELEN, F.V. Web ontology language: OWL. **Handbook on ontologies**. Berlin: Springer-Verlag, 2004. p. 67-92.

AUER, S. **Towards Agile Knowledge Engineering: Methodology, Concepts and Applications.** Leipzig: Universidade de Leipzig, 2006. 156 f. Tese (Doutorado), Universidade de Leipzig, Leipzig, 2006.

BASSILIADES, N.; VLAHAVAS, I. R-DEVICE: An Object-Oriented Knowledge Base System for RDF Metadata. **International Journal on Semantic Web and Information Systems**, v. 2, n. 2, p. 24-90, 2006.

BRACHMAN, R.; LEVESQUE, H. **Knowledge Representation and Reasoning**. Morgan Kaufmann Publishers Inc, 2004.

BRICKLEY, D.; GUHA, R.V. Resource description framework (RDF) schema specification 1.0. In W3C, 2000. Disponível em: <http://www.w3.org/TR/2000/CR-rdf-schema-20000327/>. Acesso em: 15 fev. 2009.

BROMBY, M.; MACMILLAN, M.; MCKELLAR, P. A CommonKADS Representation for a Knowledge-based System to Evaluate Eyewitness Identification. **International Review of Law, Computers & Technology**, vol. 17, n. 1, p. 99-108, mar. 2003.

CACHO, N. et al.. Improving modularity of reflective middleware with aspect oriented programming. In: INTERNATIONAL WORKSHOP ON SOFTWARE ENGINEERING AND MIDDLEWARE, 6., SEM'06, 2006, Portland, **Proceedings…** New York: ACM, 2006, p. 31-38.

CASTRO, E.S.E.; VICTORETI, F.; FIORINI, S.; ABEL, M; PRICE, R.T. Um Caso de Integração de Gerenciamento Ágil de Projetos à Metodologia CommonKADS. In:

WORKSHOP DE GERÊNCIA DE PROJETO DE SOFTWARE, 1., WGPS'08, 2008, Florianópolis, **Proceedings...** Florianópolis: [s.n], 2008, p. 1-4.

CASTRO, E.S.E.; ABEL, M; PRICE, R.T. K-Annotations: An Approach for Conceptual Knowledge Implementation using Metadata Annotations. In: INTERNATIONAL CONFERENCE ON ENTERPRISE INFORMATION SYSTEMS, 11., ICEIS'09, 2009, Milão, **Proceedings…** [S.l.:s.n], 2009 (TO BE PUBLISHED).

CHALMETA, R. and GRANGEL, R. Methodology for the implementation of knowledge management systems. **Journal of American Society for Information Science and Technology,** vol. 59, n. 5, p. 742-755, mar. 2008.

CIBRÁN, M.A.; D'HONDT, M.; JONCKERS, V.. Aspect-oriented programming for connecting business rules. In: INTERNATIONAL CONFERENCE ON BUSINESS INFORMATION SYSTEMS, 6., BIS'03, 2003, Colorado Springs, USA, **Proceedings...** [S.l.:s.n], 2003, p. 1-6.

CIBRÁN, M.A.; D'HONDT, M.; SUVÉE, D. Linking business rules to object-oriented software using JAsCo. **Journal of Computational Methods in Sciences and Engineering,** vol. 5, n. 1, p. 13-27, IOS, 2005.

COVER, R. XML and semantic transparency. [S.l.:s.n], 1998. Disponível em <http://www.oasis-open.org/cover/xmlAndSemantics.html>. Acesso em: 10 abr. 2008.

CHUNA, C. A.; SOBRAL, J. L.; MONTEIRO, M. P. Reusable aspect-oriented implementations of concurrency patterns and mechanisms. In: INTERNATIONAL CONFERENCE ON ASPECT-ORIENTED SOFTWARE DEVELOPMENT, 5., AOSD'06, 2006, Bonn, Germany, **Proceedings...** New York: ACM, 2006, p. 134-145.

DEMERS, F. N.; MALENFANT, J. Reflection in logic, functional and object-oriented programming: a short comparative study. In: WORKSHOP ON REFLECTION AND METALEVEL AND THEIR APPLICATIONS in ARTIFICIAL INTELLIGENCE, 1., IJCAI '95, 1995, Quebec, **Proceedings...** [S.l.:s.n], 1995, p. 29-38.

DEMEYER, S.; DUCASSE, S.; TICHELAAR, S. Why unified is not universal: UML shortcomings for coping with round-trip engineering. In: INTERNATIONAL CONFERENCE ON THE UNIFIED MODELING LANGUAGE, 2., UML'99, 1999, Fort Collins, USA, **Proceedings...** [S.l.]: Springer-Verlag, 1999, p. 630-645.

D'HONDT, M.; MEUTER, W.D.; WUYTS, R. Using Reflective Logic Programming to Describe Domain Knowledge as an Aspect. In: INTERNATIONAL SYMPOSIUM ON GENERATIVE AND COMPONENT-BASED SOFTWARE ENGINEERING, 1., GCSE'99, 1999, Erfurt, Germany, **Proceedings...** London: K. Czarnecki and U. W. Eisenecker, Eds. Lecture Notes In Computer Science, vol. 1799, Springer-Verlag, 2000, p. 16-23.

ERIKSSON, H. Using aspect-oriented programming to extend protégé**.** In: International Protégé Conference, 7., 2004, Maryland. **Proceedings...** Maryland: [s.n], 2004.

FIKES, R.; KEHLER, T. The role of frame-based representation in reasoning. **Communications of the ACM,** [S.I.], v. 238, n. 9, p. 904-920, 1985.

FILMAN, R. E. **Applying Aspect-Oriented Programming to Intelligent Synthesis**. Technical Report. UMI Order Number: 00000047., RIACS, 2000.

FOWLER, M. **Analysis Patterns: Reusable Object Models**. Addison-Wesley, 1997.

FOWLER, M. **Refactoring. Improving the Design of Existing Code**. Addison-Wesley, 1999.

GENNARI, J.H. et al. The evolution of Protégé: an environment for knowledge-based systems development. **International Journal of Human-Computer Studies**, vol. 58, n. 1, p. 89-123, 2003.

GUIA do Usuário Petroledge. [S.l.]: ENDEEPER, 2009. Disponível em <http://www.endeeper.com/customer_area/Guia_Usuario_PETROLEDGE.pdf>. Acesso em: 20 abr. 2009.

JAVA Expression Grammar. Singular Systems, 2009. Disponível em < http://www.singularsys.com/jep >. Acesso em: 20 abr. 2009.

JSR220A: Enterprise JavaBeans Specification, v3.0, EJB Core Contracts and Requeriments, v3.0 – Proposed Final. [S.l.]: Sun Microsystems, 2005. Disponível em: <http://jcp.org/aboutJava/communityprocess/pr/jsr220/>. Acesso em: 2 mar. 2006.

JSR220B: Enterprise JavaBeans Specification, v3.0, Java Persistence API – Proposed Final. [S.l.]: Sun Microsystems, 2005. Disponível em: <http://jcp.org/aboutJava/communityprocess/pr/jsr220/>. Acesso em: 2 mar. 2006.

KICZALES, G. et al.. Aspect-Oriented Programming, In: EUROPEAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING, 11., ECOOS'97, 1997, Finland, **Proceedings...** [S.l.]: Springer-Verlag, 1997, p. 220-242.

KICZALES, G. et al.. An overview of AspectJ. In: EUROPEAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING, 15., ECOOS'01, 2001, Budapest, Hungary, **Proceedings...** London: Springer-Verlag, 2001, p. 327-353.

KNUBLAUCH, H. **An Agile Development Methodology for Knowledge-Based Systems.** Ulm: Universidade de Ulm, 2002. 216 f. Tese (Doutorado) – Universidade de Ulm, Berlin, 2002.

KOURAI, K.; HIBINO, H.; CHIBA, S. Aspect-oriented application-level scheduling for J2EE servers. In: International Conference on Aspect-Oriented Software Development, 6., AOSD'07, 2007, Vancouver, Canada, **Proceedings...** New York: ACM, 2007, p. 1-13.

MAES, P. Concepts and experiments in computational reflection. **SIGPLAN Not. 22**, vol. 12, p. 147-155, dec. 1987.

MASTELLA, L.. **Um Modelo de Conhecimento Baseado em Eventos para Aquisição e Representação de Seqüências Temporais.** Porto Alegre: UFRGS, 2005. 162 f. Dissertação (Mestrado) – Programa de Pós-Graduação em Computação, Universidade Federal do Rio Grande do Sul, Porto Alegre, 2005.

MEDITSKOS, G.; BASSILIADES, N. O-DEVICE: An Object-Oriented Knowledge Base System for OWL Ontologies. In: HELLENIC CONFERENCE ON ARTIFICIAL INTELLIGENCE, 4., SETN'06, 2006, **Proceedings...** Crete: Springer-Verlag: ANTONIOU G. et al., LNAI 3955, p. 256-266, 2006.

MESEGUER, P.; PREECE, A. D. Verification and Validation of Knowledge-Based Systems with Formal Specifications. **The Knowledge Engineering Review**, v. 10, n. 4, 1995.

PIVETA, E, et al.. Avoiding Bad Smells in Aspect-Oriented Software**.** In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING AND KNOWLEDGE ENGINEERING, 19., SEKE'07, 2007, Boston, USA, **Proceedings...** Boston: Knowledge Systems Institute Graduate School, 2007, p. 81-84.

PREE, W.; BECKENKAMP, F.; VIADEMONTE, S. OO design and implementation of a flexible software architecture for decision support systems. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING AND KNOWLEDGE ENGINEERING, 9., SEKE'97, 1997, Madrid, Spain, **Proceedings...** Spain: [s.n], 1997.

SCHREIBER, G. et al. **Knowledge Engineering and Management: The CommonKADS Methodology**. Cambridge: MIT Press, 2000.

SILVA, L.A.L. **Aplicando métodos de solução de problemas em tarefas de interpretação de rochas**. Porto Alegre: UFRGS, 2001. 160 f. Dissertação (Mestrado) – Programa de Pós-Graduação em Computação, Universidade Federal do Rio Grande do Sul, Porto Alegre, 2001.

STUDER, R.; BENJAMINS, V. R.; FENSEL, D. Knowledge Engineering: Principles and Methods. **Data & Knowledge Engineering**, v. 25, n. 1/2, p. 161-197, 1998.

TANASESCU, V.; DOMINGUE, J.; CABRAL, L. OCML Ontologies to XML Schema Lowering, **In: First AKT Workshop on Semantic Web Services** (AKT-SWS04), KMi, The Open University, Milton Keynes, UK, 2004.

VICTORETI, F.I. **Mapeamento e Documentação de Feições Visuais Diagnósticas para Interpretação em Sistema Baseado em Conhecimento no Domínio da Petrografia**. Porto Alegre: UFRGS, 2007. 87 f. Dissertação (Mestrado) – Programa de Pós-Graduação em Computação, Universidade Federal do Rio Grande do Sul, Porto Alegre, 2007.

ZHU, Z. J.; ZULKERNINE, M. A model-based aspect-oriented framework for building intrusion-aware software systems. **Information and Softwre Technology**, v. 51, n. 5, p. 865-875, 2009.

ANEXO GRAMÁTICA DO JEP (AXIOMAS)

A gramática suportada pelo Java Expression Parser é apresentada a seguir. Extraída de <http://www.singularsys.com/jep/doc/html/grammar.html#prod3>. Operadores com menor precedência estão no topo, operadores com maior precedência estão na base.

Start	::= (Expression (<EOF> | <SEMI>) | (<EOF> | <SEMI>))
Expression	::= AssignExpression
	| RightExpression
AssignExpression	::= (LValue <ASSIGN> Expression)
RightExpression	::= OrExpression
OrExpression	::= AndExpression ((<OR> AndExpression))*
AndExpression	::= EqualExpression((<AND>EqualExpression))*
EqualExpression	::= RelationalExpression ((<NE> RelationalExpression) | (<EQ> RelationalExpression))*
RelationalExpression	::= AdditiveExpression ((<LT> AdditiveExpression) | (<GT> AdditiveExpression) | (<LE> AdditiveExpression) | (<GE> AdditiveExpression))*
AdditiveExpression	::= MultiplicativeExpression ((<PLUS> MultiplicativeExpression) | (<MINUS> MultiplicativeExpression))*
MultiplicativeExpression	::= UnaryExpression((PowerExpression)|(<MUL> UnaryExpression) | (<DOT> UnaryExpression) | (<CROSS> UnaryExpression) | (<DIV> UnaryExpression) | (<MOD> UnaryExpression))*
UnaryExpression	::= (<PLUS> UnaryExpression)
	| (<MINUS> UnaryExpression)
	| (<NOT> UnaryExpression)
	| PowerExpression
PowerExpression	::= UnaryExpressionNotPlusMinus((<POWER>UnaryExpression))?
UnaryExpressionNotPlusMinus	::= AnyConstant
	| ArrayAccess
	| Function
	| Variable

```
                         | <LRND> Expression <RRND>
                         | ListExpression
ListExpression           ::= <LSQ> ( Expression ( <COMMA> Expression )* )? <RSQ>
LValue                   ::= ArrayAccess
                         | Variable
ArrayAccess              ::= Variable ListExpression
Variable                 ::= ( Identifier )
Function                 ::= ( Identifier <LRND> ArgumentList <RRND> )
ArgumentList             ::= ( Expression ( <COMMA> Expression )* )?
Identifier               ::= ( <INDENTIFIER1> | <INDENTIFIER2> )
AnyConstant              ::= ( <STRING_LITERAL> | RealConstant )
RealConstant             ::= (<INTEGER_LITERAL>| <FLOATING_POINT_LITERAL> )
```