UNIVERSIDADE FEDERAL DO ESPÍRITO SANTO
CENTRO DE CIÊNCIAS AGRÁRIAS
DEPARTAMENTO DE COMPUTAÇÃO

ANDRÉ TEIXEIRA LOPES

AVALIAÇÃO DA EVOLUÇÃO DA ARQUITETURA DE SOFTWARE COM ONTOLOGIAS NA ABORDAGEM MDA

ALEGRE
2013

ANDRÉ TEIXEIRA LOPES

AVALIAÇÃO DA EVOLUÇÃO DA ARQUITETURA DE SOFTWARE COM ONTOLOGIAS NA ABORDAGEM MDA

Trabalho de Conclusão de Curso apresentado ao Departamento de Computação do Centro de Ciências Agrárias da Universidade Federal do Espírito Santo, como requisito parcial para obtenção da aprovação na disciplina de Trabalho de Conclusão de Curso em Ciência da Computação II.

Orientador: Prof. MSc. Clayton Vieira Fraga Filho

ALEGRE

2013

ANDRÉ TEIXEIRA LOPES

AVALIAÇÃO DA EVOLUÇÃO DA ARQUITETURA DE SOFTWARE COM ONTOLOGIAS NA ABORDAGEM MDA

Trabalho de Conclusão de Curso apresentado ao Departamento de Computação do Centro de Ciências Agrárias da Universidade Federal do Espírito Santo, como requisito parcial para obtenção da aprovação na disciplina de Trabalho de Conclusão de Curso em Ciência da Computação II.

Aprovado em 04/09/2013

COMISSÃO EXAMINADORA

Prof. MSc. Clayton Vieira Fraga Filho
Universidade Federal do Espírito Santo
Orientador

Prof. MSc. Antonio Almeida de Barros Junior
Universidade Federal do Espírito Santo

Prof. MSc. Rodrigo Freitas Silva
Universidade Federal do Espírito Santo

AGRADECIMENTOS

A Deus, por guiar e permitir encerrar mais essa etapa de minha vida.

À minha família, por sua capacidade de acreditar em mim e investir em mim. Mãe, seu cuidado e dedicação foi que deram, em alguns momentos, a esperança e força para seguir. Pai, sua presença significou segurança e certeza de que não estou sozinho nessa caminhada.

A todos os professores do DCOMP, pelos ensinamentos transmitidos.

Ao professor Clayton Vieira Fraga Filho, pela orientação neste trabalho e especialmente por todos os conhecimentos transmitidos.

À Gislayni, minha amada. Obrigado pelo carinho, a paciência e por sua capacidade de me trazer paz na correria de cada semestre.

A todas as amizades construídas na faculdade.

A todos, muito obrigado.

RESUMO

Durante o processo de desenvolvimento de sistemas é comum que a arquitetura definida inicialmente evolua. A grande deficiência dessa evolução é que, em muitos casos, a documentação do sistema não é atualizada. Esse fator contribui para baixa qualidade de sistemas, visto que a documentação da arquitetura é base para processos de tomadas de decisões de implementação. Dessa forma, os arquitetos de software precisam ter uma forma de avaliar a evolução da arquitetura conceitual, para poder diminuir, ou até evitar, a evolução descontrolada da arquitetura emergente. No presente trabalho, foi proposta uma abordagem AESCOM (Avaliação da Evolução de Software Com Ontologias na abordagem MDA), que tem por objetivo apoiar desenvolvedores e arquitetos de software no processo de percepção e compreensão da evolução arquitetural de sistemas, baseada em ontologias e na *Model Driven Architecture* (MDA). As ontologias são utilizadas para apoiar a definição do modelo de arquitetura conceitual do software e a abordagem MDA utilizada na transformação do código fonte da aplicação para sua representação na UML – *Unified Modelling Language*. Uma ferramenta para avaliação da evolução de arquitetura de software, apoiada nas abordagens ontológicas propostas, foi desenvolvida, com suporte a visualização temporal da evolução utilizando a técnica de Detalhamento. A ferramenta também possibilita o cálculo das métricas de Precisão e Cobertura, adaptadas no presente trabalho para utilizar os conceitos da ontologia proposta, para verificação da aderência entre a arquitetura conceitual e emergente. Ao final do trabalho foi conduzido um estudo de caso, utilizando a ferramenta desenvolvida, em um software acadêmico para apresentação da abordagem proposta.

Palavras Chave: Ontologias, MDA, Arquitetura de Software, Evolução, Visualização de Software.

SUMÁRIO

1 Introdução 12
1.1 O Problema e sua importância 14
2 Objetivos 17
2.1 Objetivos Gerais 17
2.2 Objetivos Específicos 17
3 Revisão da Literatura 18
3.1 Arquitetura de Software 18
3.2 Evolução da Arquitetura de Software 21
3.3 Avaliação da Evolução da Arquitetura de Software 23
3.3 Abordagem MDA 28
3.4 Ontologias 31
3.4 Trabalhos Relacionados 33
4 Metodologia 36
4.1 Introdução 36
4.2 Definição das Ontologias 37
4.3 Métricas 39
4.4 Visualização da Evolução 41
4.5 MDA 42
4.6 Implementação 43
4.6.1 Arquitetura da Ferramenta 44
4.6.2 Anotações para caracterização das ontologias 46
4.6.3 Obtenção de dados de repositórios de código fonte 48
4.6.4 Representação do modelo conceitual e emergente 49
4.6.5 Comparação e Visualização de Modelos 52
4.6.6 Cálculo das métricas de Precisão e Cobertura 54
5 Resultados Obtidos 55
5.1 Ontologias 56
5.1.1 Aplicação da Ontologia Desenvolvida no Processo de Desenvolvimento de Software 58

5.2 Visualização da Evolução 60
5.3 MDA 62
5.4 Ferramenta desenvolvida 63
5.5 Estudo de Caso 65
5.5 Discussão 75
5.6 Análise Comparativa 77
6 Conclusão 79
6.1 Considerações Gerais 79
6.2 Contribuições 80
6.3 Trabalhos Futuros 81
7 Referências 82
Apêndice A 86
Apêndice B 87
Apêndice C 88

LISTA DE FIGURAS

Figura 1 - Arquitetura de Sistemas. Fonte: Vogel et al (2009) 18
Figura 2 - Estilo arquitetural em camadas. Fonte: Vogel et al (2009).....20
Figura 3 - Estilo arquitetural cliente/servidor. Fonte: Vogel et al (2009). 21
Figura 4 - Estilo arquitetural Pipes-and-Filters. Fonte: Vogel et al (2009). 21
Figura 5 - Técnica de Zoom. Fonte: Oliveira (2011).....24
Figura 6 - Técnica de Filtragem. Fonte: Oliveira (2011) 25
Figura 7 - Técnica de Agrupamento. Fonte: O Autor.....25
Figura 8 – Representação em Hierarquia. Fonte: O Autor 25
Figura 9 - Técnica de Drill-Down. Fonte: Oliveira (2011) 26
Figura 10 - Exemplo da Técnica de Foco + Contexto. Fonte: Oliveira (2011).....26
Figura 11 - Exemplo da Técnica Visão Geral + Detalhes. Fonte: Oliveira (2011) 27
Figura 12 - Evolution Matrix. Fonte: Lanza e Ducasse (2002) 28
Figura 13 - Níveis da MDA e suas transformações. Fonte: O Autor.....29
Figura 14 - Transformação do PIM em PSM. Fonte: O Autor 30
Figura 15 - Ontologia no sentido filosófico. Fonte: Almeida et al(2003) 31
Figura 16 - Pirâmide de ontologia desenvolvida por Würsh et. al., 2012. Fonte: Würsh et. al., 2012 35
Figura 17 - Processo para construção de ontologias. Fonte: O Autor.....37
Figura 18 - Relação entre elementos das métricas. Fonte: O Autor.....40
Figura 19 - Transformação de código fonte em diagrama UML. Fonte: O Autor.....42
Figura 20 - Diagrama de Arquitetura da ferramenta desenvolvida. Fonte: O Autor...45
Figura 21 - Diagrama com as classes anotadas. Fonte: O Autor 47
Figura 22 - Implementação da anotação ClasseModel. Fonte: O Autor 47
Figura 23 - Diagrama de Classe do mecanismo de obtenção de dados de repositório. Fonte: O Autor.....48
Figura 24 – Meta-modelo de classe para representação da UML. Fonte: O Autor ...51
Figura 25 - Padrão Adapter para leitura do modelo de arquitetura conceitual. Fonte: O Autor.....52
Figura 26 - Modelo de arquitetura conceitual da Ontologia Proposta. Fonte: O Autor 56
Figura 27 - Diagrama de Classe com Ontologias. Fonte: O Autor 59
Figura 28 - Código Java anotado com as ontologias. Fonte: O Autor 60
Figura 29 - Exemplo de visualização da evolução. Fonte: O Autor 61
Figura 30 - Diagrama de caso de uso da ferramenta desenvolvida. Fonte: O Autor.63
Figura 31 - Diagrama de atividades representando o fluxo de execução da ferramenta desenvolvida. Fonte: O Autor.....64
Figura 32 - Diagrama gerado ao final da etapa de design. Fonte: O Autor 66

LISTA DE PSEUDOCÓDIGOS

Pseudocódigo 1 - Verificação de diferenças entre modelos 48
Pseudocódigo 2 - Criação do diagrama UML a partir do modelo emergente 58
Pseudocódigo 3 - Cálculo da métrica de Precisão 81
Pseudocódigo 4 - Cálculo da métrica de Cobertura 82

LISTA DE SIGLAS

CIM – *Computation Independent Model*

MDA – *Model Driven Architecture*

MVC – *Model View Controller*

OMG – *Object Management Group*

PSM – *Platform Specific Model*

PIM – *Platform Independent Model*

UML – *Unified Modeling Language*

XML – *Extensible Markup Language*

XMI – *XML Metadata Interchange*

1 Introdução

O software tem hoje uma larga penetração em todos os ramos da sociedade, sendo parte integrante e essencial de processos de negócio, e apesar disso a indústria de software é confrontada com dois problemas, cada vez mais agravantes (VOINEA, 2007). O primeiro refere-se à complexidade do software, enquanto sistemas de médio porte há vinte anos eram formados por uma média de dezenas de milhares de linhas de código, hoje em dia aplicações do mesmo porte têm dezenas de milhões. O segundo problema diz respeito à contínua evolução e modificação nas quais o software passa durante seu ciclo de vida. Uma pesquisa realizada no setor de TI, organizada por Grady Booch (2005) estima que o número total de linhas de código em manutenção seja em torno de 800 bilhões. Destes, 30 bilhões de linhas de código são novas ou têm que ser modificada todos os anos por cerca de 15 milhões engenheiros de software. Um estudo realizado por Erlikh (2000) estima que o custo de manutenção gire em torno de 80-90% do custo total do projeto, outro estudo realizado por Sebesta (2011) afirma que esse valor pode chegar a 400% do valor inicial do projeto. E, portanto, é necessário prover ferramentas e técnicas que forneçam mecanismos para percepção da aderência entre o que está sendo implementado com relação ao que foi planejado (OLIVEIRA, 2011).

A evolução pode ser definida como crescimento, transformação gradual e progressiva ou até mesmo aperfeiçoamento de determinado produto ou ideia. No contexto de desenvolvimento de software, o termo pode ser compreendido da mesma forma, visto que a evolução de software é dada, dentre outros fatores (tais como correções e migrações de plataformas de desenvolvimento), principalmente pela demanda por novas funcionalidades (requisitos, funcionais e não funcionais). A evolução do projeto do software como um todo, tal qual de sua arquitetura, pode ocorrer tanto durante o processo de desenvolvimento, quanto como resultado da adição de novos requisitos após o software já estar em ambiente de produção, como consequência dessa evolução, sistema podem aumentar ou diminuir sua complexidade (VOINEA, 2007). Em se tratando de desenvolvimento de software, é amplamente aceito que a complexidade dos sistemas só aumenta à medida que evoluem no tempo – segunda lei da evolução de software de Lehman’s (LEHMAN, 1996).

Sendo a arquitetura a organização fundamental de um software incorporada em seus componentes e seus relacionamentos com o ambiente, essa é comumente modificada no decorrer do tempo devido a mudanças e evolução do software (OLIVEIRA, 2011), com isso é necessário defini-la em dois momentos: conceitual, na etapa de planejamento do *design*, como um conjunto limitado de conceitos (padrões e estilos arquiteturais) e os relacionamentos entre eles e a emergente, durante e após a implementação, ou seja, a programação desses conceitos no desenvolvimento de uma aplicação em particular (VÖLTER, 2007).

Como resultado da evolução, a arquitetura do software tem uma crescente divergência entre o modelo de arquitetura conceitual e o emergente, que ocorre devido às modificações inseridas no projeto do software serem realizadas diretamente no seu código-fonte, sem passar antes pela remodelagem do projeto arquitetural (OLIVEIRA, 2011). Surge então a necessidade de termos mecanismos para verificar tanto a evolução da arquitetura do software, quanto à adequação do modelo emergente com o conceitual. Espera-se que com esse tipo de mecanismo, os arquitetos e responsáveis pelo desenvolvimento do software possam identificar discrepâncias o mais cedo possível e corrigi-las. Tais discrepâncias são principalmente caracterizadas por: elementos e relações presentes no modelo de arquitetura conceitual e inexistentes no modelo emergente e elementos e relações presentes no modelo emergente e inexistentes no modelo de arquitetura conceitual.

Várias são as técnicas possíveis de se realizar a avaliação da evolução da arquitetura, dentre as principais destaca-se a visualização e o uso de métricas. A métrica tem por objetivo demonstrar numericamente os impactos de alterações durante a evolução do software, com a implementação de operações matemáticas realizadas a partir de medidas coletadas no código-fonte. Apesar de sua implementação consumir pouco esforço, podem se tornar de difícil interpretação em casos de grande volume de informação. Outra técnica é a de visualização, que têm por objetivo aprimorar a visualização das métricas por meio de gráficos e imagens, que podem mostrar para o usuário as diferenças entre a arquitetura conceitual e o código-fonte do software.

1.1 O Problema e sua importância

O processo de definição da arquitetura conceitual do software representa um grande esforço da etapa de projeto. É comum que durante a implementação do sistema a arquitetura conceitual seja deixada de lado e o foco principal do desenvolvedor fique na arquitetura emergente, segundo Taylor et al (2009) isso ocorre, dentre outros fatores, devido a: negligencia por parte dos desenvolvedores, prazos curtos que impedem o planejamento e a execução da especificação, falta da documentação da arquitetura conceitual e necessidade ou vontade de efetuar refatorações no código. Além da menor qualidade do software que foi idealizado, o trabalho de elaboração da arquitetura é ignorado, gerando um prejuízo para a empresa nos dois momentos – trabalho inutilizado e software entregue com baixa qualidade. A baixa qualidade é justificada devido ao menor controle que a equipe tem sobre o software após as sucessivas modificações sem atualização da arquitetura, visto que a especificação arquitetural desempenha papel crucial na compreensão do software passível de manutenção, uma vez que ela possui uma descrição em alto nível do sistema implementado, servindo como mapa para guiar toda a equipe (OLIVEIRA, 2011). Dessa forma o processo de teste é, em partes, falho, pois o planejamento desses torna-se inviável de ser feito com base na arquitetura conceitual desatualizada; a estimativa do valor ou esforço para determinada tarefa também fica comprometida, uma vez que a especificação do sistema é o ponto de partida para realizar tal atividade.

A divergência entre as arquiteturas aumenta o custo para a análise de impacto de mudanças e evoluções na arquitetura, bem como a integração com outros subsistemas é dificultada, pois o desenvolvedor passa a ter que avaliar a arquitetura emergente e o código-fonte ao invés do modelo de arquitetura conceitual. Estudos organizados por Standish (1984) e por Corbi (1989) mostram que a atividade de entendimento dos sistemas consome mais da metade do esforço aplicado ao desenvolvimento de software. A responsabilidade pela discrepância entre a arquitetura conceitual e emergente, em parte, pode ser atribuída à falta de técnicas e ferramentas especializadas que podem permitir aos arquitetos de software realizar um acompanhamento da evolução da arquitetura com o passar do tempo,

verificando discrepâncias e tomando medidas corretivas – sejam essas na arquitetura conceitual ou emergente.

Em alguns trabalhos é realizada a análise da evolução da arquitetura e identificadas discrepâncias entre a arquitetura lógica (modelo de arquitetura conceitual) e a emergente, por meio de versionamento e comparação visual de modelos arquiteturais. Uma abordagem para comparação de documentos que descrevem a arquitetura do sistema, representados em UML, é utilizada por Ohst et al, (2003). Knodel et al (2006) apresenta uma abordagem onde os dados são obtidos por meio de engenharia reversa do código-fonte para comparação da arquitetura conceitual e emergente. O trabalho de Gîrba et al (2005) consiste na utilização um meta-modelo baseado na *Model Driven Architecture* (MDA), chamado de HISMO, para caracterizar a evolução temporal do código-fonte do software. Uma abordagem mais recente, proposta por Oliveira (2011), além de permitir a visualização da evolução temporal da arquitetura propõe métricas para avaliação da evolução da arquitetura emergente.

O trabalho de Gîrba et al (2005) utiliza a abordagem MDA para transformação do código-fonte da aplicação em sucessivos meta-modelos para gerar uma visualização da evolução do software. No trabalho o software é dividido em modelos conceituais, baseados em níveis de abstração, entre cada nível existem regras de transformação que permitem que um nível inferior seja gerado com base em seu nível imediatamente superior. No entanto, não realiza uma verificação semântica dos elementos envolvidos, sendo baseada apenas na análise sintática dos elementos para determinar sua evolução. Segundo Vasconcelos (2007) tal técnica, não permite um controle adequado à granularidade dos elementos que compõe um modelo arquitetural.

Würsch et al (2012) propõe em seu trabalho uma pirâmide de ontologias para evolução de software. O uso de ontologias permite criar relações explicitas de relacionamento entre os artefatos, caracterização (criação de uma identidade única) e classificação dos mesmos, possibilitando a análise da evolução de forma semântica, sendo uma proposta de solução do problema levantado por Vasconcelos (2007), que identifica uma falha nos processos de avaliação da evolução baseado somente nas propriedades sintáticas dos elementos.

Em pesquisas realizadas na literatura não foi identificada uma abordagem que trata a análise da evolução semântica da arquitetura conceitual para emergente apoiado nos modelos e meta-modelos da MDA. Considerando o exposto, o uso de ontologias para representação de modelos conceituais e emergentes de arquitetura, a transformação do código fonte para diagramas UML, apoiada na abordagem MDA pode ser empregado na avaliação da evolução da arquitetura de software?

Dando continuidade ao presente trabalho, o capítulo 2 apresenta os objetivos gerais e específicos do trabalho. O capítulo 3 realiza um revisão da literatura sobre arquitetura de software, evolução de software, visualização, ontologias e apresenta os trabalhos relacionados ao atual. O capítulo 4 apresenta a metodologia utilizada no presente trabalho e como foram alcançados os objetivos apresentados no capítulo 2. O capítulo 5 apresenta os resultados obtidos no trabalho e uma discussão entre a abordagem proposta e os trabalhos relacionados. Por fim, o capítulo 6 apresenta uma síntese conclusiva sobre a abordagem proposta, as contribuições e trabalhos futuros.

2 Objetivos

2.1 Objetivos Gerais

Desenvolver uma abordagem que permita a avaliação da evolução arquitetural de sistemas apoiada por conceitos arquiteturais definidos ontologicamente, utilizando a abordagem MDA para transformação entre a arquitetura lógica (conceitual) e a emergente.

2.2 Objetivos Específicos

São objetivos específicos do trabalho:

a) Analisar estratégias de avaliação da evolução da arquitetura de software para software orientados a objeto e sua arquitetura baseada na abordagem MVC;
b) Definir modelos e meta-modelos da MDA que serão utilizados na abordagem
c) Definir uma abordagem baseada em ontologias que sirva como mapeamento entre as várias camadas da MDA;
d) Implementar uma ferramenta de avaliação da evolução do software apoiada nos resultados obtidos em a, b e c;
e) Realizar um estudo de caso, com um software acadêmico, para demonstrar o uso das técnicas proposta nos itens a, b e c, utilizando a ferramenta desenvolvida em d.

3 Revisão da Literatura

3.1 Arquitetura de Software

O estudo em arquitetura de software iniciou-se no final de 1980, emergiu como o estudo de princípios estruturais e componentes de software em macro nível, em larga escala. Teve seu início baseado em observações empíricas na organização de sistemas bem sucedidos (CLEMENTS e SHAW, 2009) e hoje é uma área cientifica de primordial importância dentro da Engenharia de Software.

A arquitetura de um software se refere à descrição dos seus principais elementos, suas formas de interação (comunicação), os padrões que possibilitaram sua posterior composição com outros sistemas e suas restrições (SHAW e GARLAN, 1996). Da mesma forma, a definição dada pela norma 1471 do *Institute of Electrical and Electronics Engineers (IEEE*), arquitetura de software é “a organização fundamental de um sistema incorporada em seus componentes, seus relacionamentos com o ambiente, e os princípios que conduzem seu design e evolução” (IEEE, 2007). Segundo Vogel et al (2009) a arquitetura de sistemas é compostas por dois componentes principais, a arquitetura do software e a arquitetura da plataforma na qual o software será desenvolvido. O esquema da organização dessa arquitetura é apresentado na Figura 1.

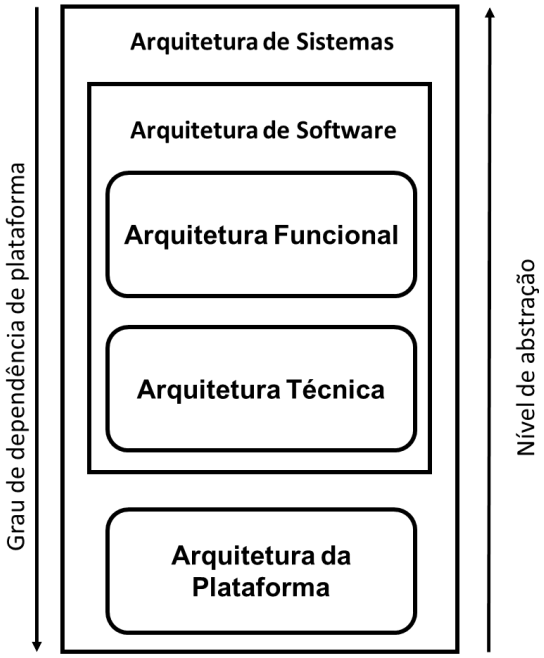

Figura 1 - Arquitetura de Sistemas. Fonte: Vogel et al (2009)

A arquitetura do software é composta por duas arquiteturas, a funcional e a técnica. A arquitetura funcional está diretamente ligada ao modelo de domínio do sistema, ela consiste na subdivisão em blocos funcionais (conjunto de requisitos funcionais relacionados). Por exemplo, em um sistema de processamento de pedidos no qual os seguintes blocos funcionais são identificados: entrada de pedidos, gerenciamento de pedidos e gestão de clientes. O grau de abstração da arquitetura funcional é elevado (está mais relacionado aos conceitos utilizados pelo cliente) e possui um baixo grau de dependência com a plataforma – devido a utilizar somente conceitos do domínio da aplicação. A arquitetura técnica tem por objetivo atender aos requisitos não funcionais do software, ela define blocos não funcionais de construção do sistema, tais como: registro de transações, auditoria, acesso a dados e gerenciamento de transações. A arquitetura de plataforma se refere ao componentes da plataforma que o sistema utilizará, esta plataforma pode ser entendida como *framework* e linguagem de programação utilizados para desenvolvimento – *Java*, *.Net Framework*, dentre outros.

Em síntese, Vogel et al (2009) propõem que a arquitetura do software está diretamente relacionada com a forma de funcionamento do sistema, seus módulos, comunicação e relacionamento entre esses módulos, hierarquia do sistema (camadas, componentes, cliente-servidor, *Model-View-Controller (*MVC), orientado a objetos, dentre outras) e sua capacidade de composição com outros sistemas – os principais estilos arquiteturais serão discutidos na seção 3.1.1.

Nos processos de desenvolvimento de software que tratam do *design*, a definição da arquitetura do software e manutenção desse modelo arquitetural, os benefícios são muitos, dentre os quais, relatados por Eeles (2006), pode-se citar:

- Apoio ao processo de planejamento de mudanças;
- Preservação da integridade dos módulos do sistema;
- Auxílio à gerência de complexidades;
- Base para reutilização;
- Redução dos custos de manutenção; e
- Apoio à análise de impacto.

Um projeto de desenvolvimento de software conta basicamente com duas arquiteturas, a arquitetura lógica (conceitual) e a emergente. A arquitetura lógica refere-se aos conceitos que foram planejados para o sistema (padrões e estilos arquiteturais) e a comunicação entre eles, já a arquitetura emergente se refere à instanciação desse conjunto de ideias para desenvolver o sistema, ou seja, o código-fonte em si (VÖLTER ,2007).

3.1.1 Principais Estilos Arquiteturais

Sistemas baseado no estilo arquitetura de camadas contêm blocos de elementos com alto índice de coesão entre si e baixo índice de acoplamento com outros blocos (VOGEL et al, 2009). Cada camada do sistema possui um propósito bem definido, de acordo com os autores, cada camada deve somente acessar elemento de sua camada imediatamente inferior. A Figura 2 apresenta o modelo arquitetural de camadas.

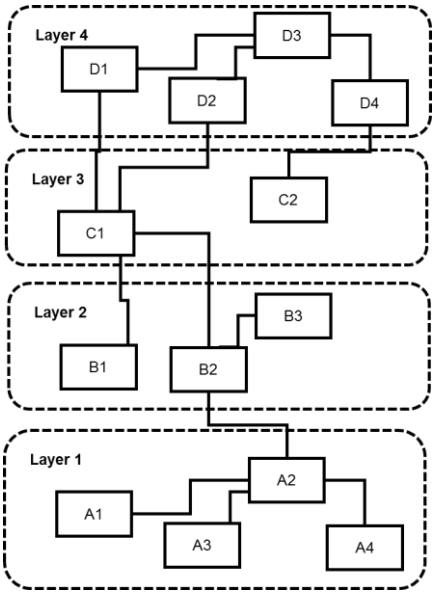

Figura 2 - Estilo arquitetural em camadas. Fonte: Vogel et al (2009).

No modelo arquitetural Cliente/Servidor o software é dividido em dois componentes principais, cliente e servidor (VOGEL et al, 2009). Ainda segundo os autores, o cliente é o software que é operado pelo usuário, em seu computador e o servidor é um software que opera, sem a intervenção de um usuário, atuando como provedor

de recursos para os clientes. Essa arquitetura é baseada, como pode ser observado na Figura 3, em um modelo de requisição e resposta, onde as requisições são feitas pelo clientes e processadas e respondidas pelo servidor.

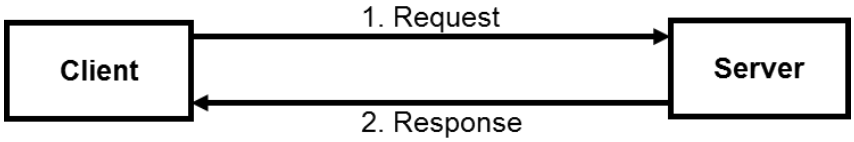

Figura 3 - Estilo arquitetural cliente/servidor. Fonte: Vogel et al (2009).

O estilo arquitetural *Pipe-and-Filters* descreve uma arquitetura flexível para o processamento sequencial de fluxos de dado (VOGEL et al, 2009). Segundo os autores o estilo tem dois componentes principais:

- Filtros (*filters*) – transformam o fluxo de dados de entrada em fluxos de dados de saída de forma incremental; e
- Tubos (*pipes)* – movem os dados da saída de um filtro para a entrada de outro.

Dessa forma, um filtro pode proporcionar dados de entrada a vários outros filtros por meio de vários tubos, o processo de transmissão continua até que não exista nenhum tubo conectado ao último filtro, e, assim, o processamento termina. A Figura 4 apresenta o estilo arquitetural *Pipe-and-Filters*.

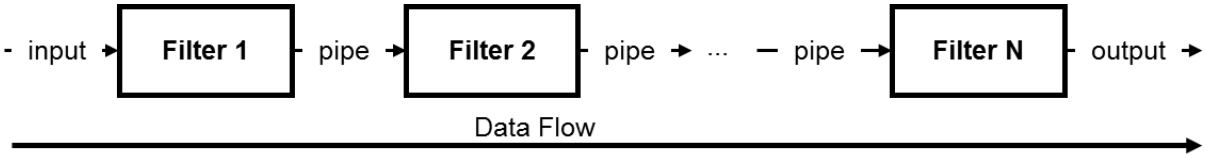

Figura 4 - Estilo arquitetural *Pipes-and-Filters*. Fonte: Vogel et al (2009).

3.2 Evolução da Arquitetura de Software

A evolução do software é uma parte inevitável do desenvolvimento de software (LEHMAN E FERNÁNDEZ-RAMIL, 2006). Com o passar do tempo ocorrem

mudanças no ambiente para o qual o software foi projetado para operar e junto com essas mudanças surgem novas demandas de funcionalidades e requisitos que o software deve atender, eventualmente até revisões nos requisitos já implementados se tornam necessárias, mudanças que por sua vez acarretam modificações no código-fonte e até mesmo nas decisões arquiteturais do sistema (KÜMMEL, 2007).

A inevitabilidade da evolução do software implica que sistemas devem sofrer continuas modificações para minimizar a distância entre o sistema e as constantes modificações ocorridas no ambiente do domínio da aplicação (LEHMAN, 1996). Ainda tratando do trabalho de Lehman (1996) ("As Leis da evolução de software"), uma lei diz que a qualidade do software irá diminuir a menos que a atividade regular de manutenção atinja também os documentos estruturais do sistema (arquitetura conceitual) e outra lei salienta que a partir dessas evoluções a complexidade do sistema irá aumentar ao longo do tempo.

O cenário ideal para evolução de sistemas é alcançado realizando as modificações primeiramente sobre a arquitetura conceitual e somente após isso sua implementação no código-fonte da aplicação (TAYLOR et al, 2009). Desta forma, a qualquer momento todos os elementos estruturais do software de determinada versão da arquitetura contém uma implementação em determinada versão de código-fonte respectivo (OLIVEIRA, 2011). No entanto, esse cenário ideal é utópico, o que acontece é que a versão implementada do sistema tem, muita das vezes, uma arquitetura própria, consideravelmente diferente do que foi especificado.
Para Taylor et al (2009) tal discrepância acontece, dentre outros fatores, devido a:

- Negligência por parte dos desenvolvedores;
- Prazos curtos que impedem o planejamento e execução da especificação;
- Falta de documentação da arquitetura conceitual; e
- Técnicas e ferramentas de apoio inadequadas.

Vários são os fatores que levam a discrepância entre o projeto e a implementação do sistema, porém sabe-se que é importante manter a consistência entre o modelo de arquitetura conceitual e emergente (VÖLTER, 2007). Oliveira (2011) destaca a importância de termos um mecanismo para verificação da aderência entre as arquiteturas, para que dessa forma seja possível evitar posteriores impactos negativos e auxiliar o processo de tomada de decisão em nível de projeto e

implementação. A arquitetura conceitual é de vital importância para os dois processos citados, devido ao fato de conter a descrição em alto nível dos componentes do sistema, podendo então ser utilizada para entendimento dos fluxos de dados no momento de decisões para modificações no sistema.

3.3 Avaliação da Evolução da Arquitetura de Software

A avaliação da evolução da arquitetura de software consiste no uso de ferramentas que comparem diferentes versões do sistema com a arquitetura conceitual, e tem por objetivo verificar divergências entre o que está sendo implementado com relação ao que foi projetado. Essa divergência pode representar, por exemplo, elementos da arquitetura conceitual que ainda não foram implementados, ou mesmo elementos implementados – presentes na arquitetura emergente – que não foram descritos na arquitetura conceitual.

O grande problema nessa área consiste em lidar com a grande quantidade de informações (LANZA e DUCASSE, 2002) – várias versões de um mesmo software, vários artefatos por versão e várias versões do mesmo artefato.

Uma técnica de avaliação que permite diminuir a complexidade, causada pela grande quantidade de informação, é a visualização de software. Ela tem por objetivo promover maior entendimento e a percepção do software, e consiste na utilização de técnicas de visualização da informação (CHEN, 2006) aplicadas ao software. Lanza e Ducasse (2002) mostram que a visualização permite utilizar técnicas de apresentação visual para estudo de problemas complexos em paralelo. Já Gallagher *et al* (2008) define que a visualização de software consiste basicamente em mapear as entidades do sistema para representações gráficas visando prover suporte à compreensão e ao desenvolvimento.

CASERTA & ZENDRA (2010) consideram que a visualização da evolução da arquitetura do software é um dos tópicos mais importantes no domínio da visualização da evolução,

segundo os autores é importante ter uma visão global da evolução do sistema para que seja possível explicar e documentar o estado atual do projeto de software.

Segundo Diehl (2007), para a visualização de software três etapas são necessárias:

- Aquisição de dados: Extração de dados (de alguma fonte de dados, por exemplo, repositórios) de gerência de configuração, tais como código-fonte, arquivos de documentação, diagramas UML, dentre outros;
- Análise: Tratamento dos dados relevantes, filtragem dos dados obtidos, focando somente no que é relevante para a operação atual; e
- Visualização: Mapeamento dos dados analisado em modelos visuais, que são convertidos em uma representação visual a apresentados para o usuário.

Várias técnicas de visualização são discutidas em Oliveira (2011): *Zoom*, Filtragem, Agrupamento, Hierarquias, Detalhamento, Foco + Contexto e Visão Geral + Detalhe.

A técnica de Zoom permite que o tamanho do elemento possa ser ampliado ou reduzido dinamicamente, conforme interação do usuário (DIEHL, 2007). Esta técnica é muito utilizada quando o objeto visualizado está em escala muito pequena, dificultando a exploração de detalhes. A Figura 5 exemplifica a técnica de Zoom.

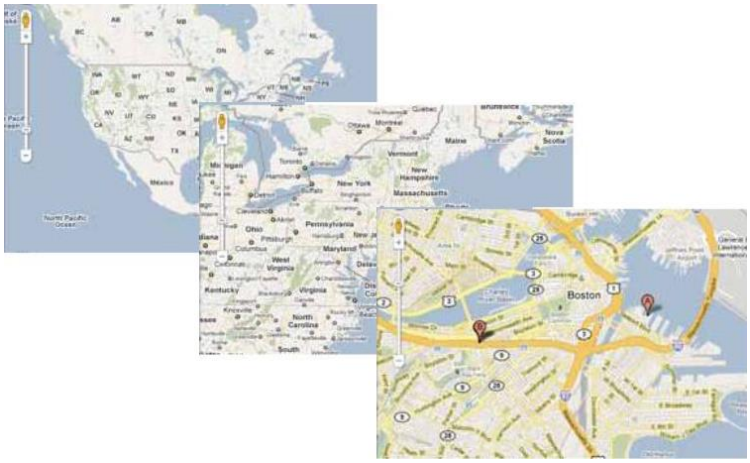

Figura 5 - Técnica de Zoom. Fonte: Oliveira (2011)

A técnica de Filtragem consiste em realçar/incluir os itens mais relevantes ou suavizar/remover/ocultar os menos relevantes em um dado contexto. A Figura 6 mostra um exemplo da aplicação desta técnica em uma visão de dependências entre classes; a parte superior apresenta os nós e as arestas, enquanto na parte inferior as arestas foram filtradas, exibindo apenas os nós.

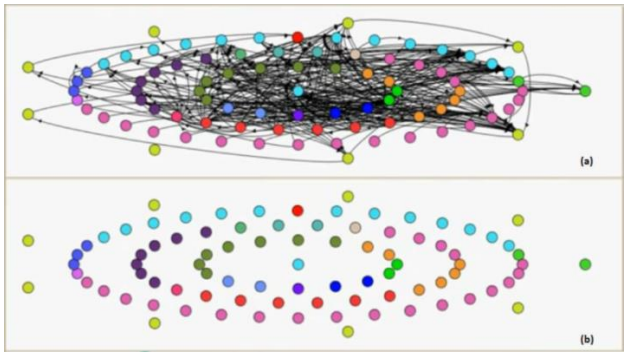

Figura 6 - Técnica de Filtragem. Fonte: Oliveira (2011)

A técnica de Agrupamento, ou *Clustering*, consiste em dividir grandes conjuntos de dados em um número de subconjuntos, de acordo com determinadas medidas de similaridade (CHEN, 2006). A Figura 7 demonstra a técnica de Agrupamento.

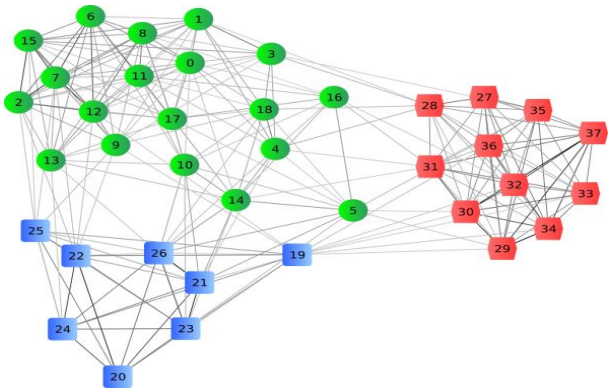

Figura 7 - Técnica de Agrupamento. Fonte: O Autor

A técnica de Hierarquia, por sua vez, permite a visualização de estruturas mais complexas, em forma de árvores (CHEN, 2006). Pode-se citar, como exemplos de hierarquia: estrutura de um sistema de arquivos, a estrutura de um sistema de classificação e até mesmo a representação em camadas de um software. A Figura 8 exemplifica a técnica de Hierarquia.

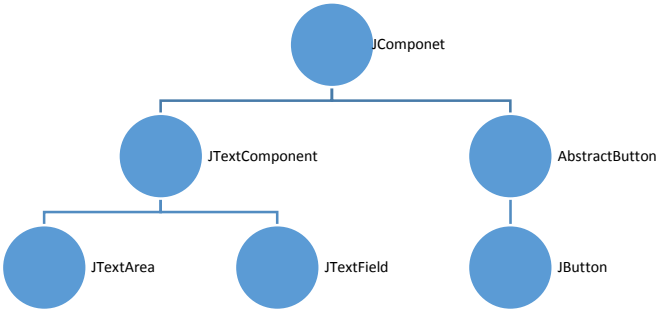

Figura 8 – Representação em Hierarquia. Fonte: O Autor

A técnica de Detalhamento, ou *Drill-Down,* consiste em revelar mais detalhes do elemento visual conforme a necessidade cognitiva do usuário. Essa técnica resulta na redução do custo de obtenção de informações, visto que essas somente serão obtidas quando forem realmente necessárias. A Figura 9 exemplifica a técnica de *Drill-Down*.

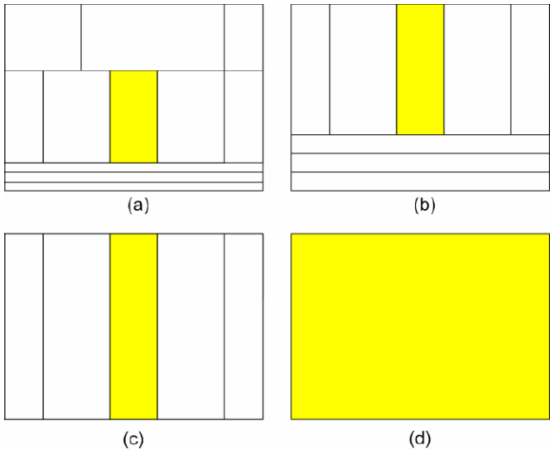

Figura 9 - Técnica de Drill-Down. Fonte: Oliveira (2011)

A técnica de “foco + contexto” busca eliminar a separação temporal e espacial, por meio da exibição do foco dentro do contexto, em uma única visão contínua (COCKBURN et al, 2008). Consistem na redução seletiva da informação apresentada, tais como filtragem de informações irrelevantes, agregação seletiva de informações relacionadas, micro e macroleitura, realce (*highlighting*) e distorção (CARD et al,1999).

A Figura 10 exemplifica a técnica de Foco + Contexto. Ao selecionar a aplicação “*iPhoto”* seu tamanho aumenta, e o tamanho dos ícones das demais aplicações vai diminuindo gradativamente, de forma que os itens mais relacionados ao aplicativo selecionado tenham um tamanho maior do que os outros.

Figura 10 - Exemplo da Técnica de Foco + Contexto. Fonte: Oliveira (2011)

A técnica de Visão Geral + Detalhe que é caracterizada pela exibição simultânea de uma visão geral e uma visão detalhada de um mesmo espaço de informação, em espaços distintos de apresentação.

A Figura 11 exemplifica a técnica de Visão Geral + Detalhe. Como pode-se observar existe uma visão geral do diagrama (parte superior do lado esquerdo), quando determinada área desse diagrama é selecionada, essa é ampliada na área do centro da Figura, com seus detalhes. Essa técnica é também chamada de *MiniMap*.

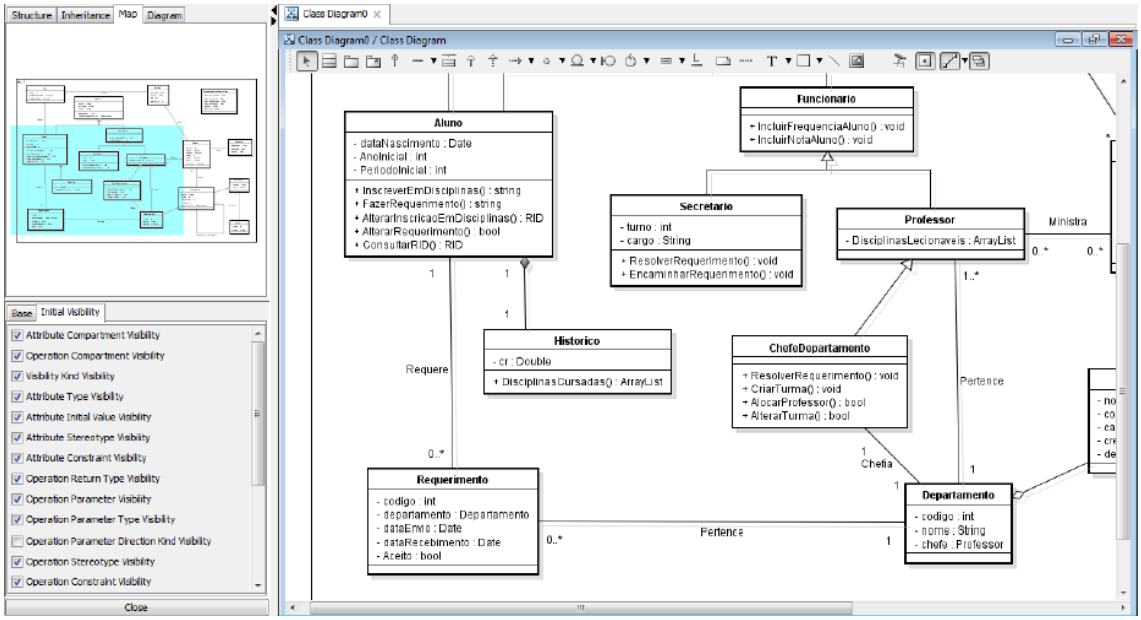

Figura 11 - Exemplo da Técnica Visão Geral + Detalhes. Fonte: Oliveira (2011)

Outra forma de análise de evolução de software é o uso de métricas. As métricas podem ajudar a avaliar a complexidade do software e descobrir artefatos de software com medidas incomuns (LANZA e DUCASSE, 2002), além de possibilitar mecanismos para controle da qualidade de software. Algumas métricas para análise da evolução do software são propostas em Lanza e Ducasse (2002), além de ser definido um método misto, chamado de *Evolution Matrix* que combina as duas técnicas – visualização e métricas.

A abordagem *Evolution Matrix*, combina as duas técnicas de avaliação, visualização e métricas. As métricas são representadas junto com a visualização da evolução do software. A visualização da evolução do software é feita por meio de uma matriz, na qual as colunas representam uma versão do software e cada linha representa as

várias versões de uma mesma classe. As classes são representadas utilizando caixas bidimensionais e as medidas de largura e altura dessas são utilizadas para representar, respectivamente, as métricas: número de métodos e número de variáveis de instância. A Figura 12, demonstra a matriz.

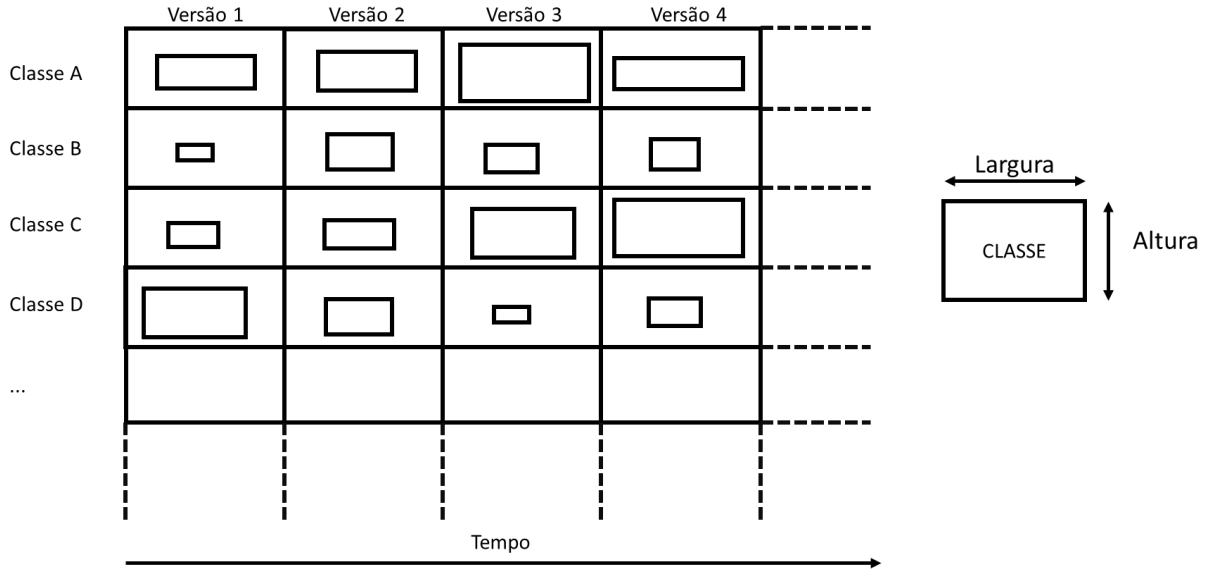

Figura 12 - Evolution Matrix. Fonte: Lanza e Ducasse (2002)

3.3 Abordagem MDA

A *Object Management Group* (OMG) – um consórcio criado em 1989 para definir padrões para sistemas orientado a objetos, criadora da *Unified Modeling Language* (UML) e da *Business Process Modeling Notation* (BPMN) – definiu um conjunto de padronizações que possibilita a definição de abordagens de desenvolvimento orientado por modelos independentes de plataforma. Este conjunto de padronizações é conhecido como Arquitetura Orientada por Modelos – *Model Driven Architecture* (MDA) – e tem como foco principal a separação entre a especificação das funcionalidades e a especificação da implementação em uma plataforma específica (OMG, 2003). As abordagens baseadas na MDA permitem o desenvolvimento de software por meio da modelagem e aplicação de mapeamentos automáticos entre os modelos e suas respectivas implementações (MAIA, 2006).

Com o advento da MDA, os desenvolvedores podem se concentrar mais nos requisitos de negócio da aplicação sem se preocupar com a implementação desses.

A MDA pode ser vista como um *framework* para desenvolvimento de aplicações, pois trata o software em uma divisão de camadas conceituais, baseadas em níveis de abstração. São definidas, basicamente, três camadas – porém a abordagem prevê o uso de tantas camadas quanto forem necessárias – a *Computation Platform Independent Model* (CIM), *Platform Independent Model (*PIM) e *Platform Specific Model (*PSM), e regras de transformação entre essas camadas (transformação de CIM para PIM e de PIM para PSM). A Figura 13 mostra as relações entre os vários níveis da MDA, seus meta-modelos e os processos de transformação entre os níveis.

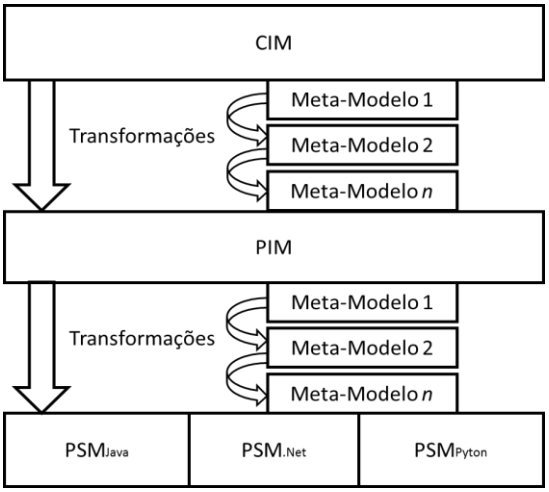

Figura 13 - Níveis da MDA e suas transformações. Fonte: O Autor

A primeira camada da MDA, chamada de CIM (*Computation Independent Model*) é utilizada no nível de definição de requisitos da aplicação, é o ponto de vista independente de computação, Fouad (2011) especifica um processo para a definição de requisitos na linguagem do *stakeholders* utilizando a camada CIM da MDA. Esta camada facilita a comunicação entre o cliente e o engenheiro de software (MAIA, 2006).

A segunda camada, chamada de PIM (*Platform Independent Model*) descreve o sistema independente de tecnologia (linguagem de programação, *framework*, sistema operacional, dentre outros), dentro do contexto do trabalho, é nessa camada em que será feito o modelo de arquitetura conceitual do sistema. Essa camada é

utilizada para modelagem em alto nível do software e seus processos, geralmente utilizando a UML.

A terceira e última camada, é a PSM (*Platform Indenpendent Model*) que pode ser visualizada como sendo a implementação do sistema, em alguma linguagem de programação. A MDA prevê também que existam meta-modelos além das camadas especificadas anteriormente, esses meta-modelos servem para diminuir os degraus entre os três modelos padrões.

Entre todos esses modelos, existem os chamados ‘mapeamentos’, que são regras que transformam um modelo em outro. Por exemplo, o PIM é gerado com base em um CIM e um conjunto de regras que mapeiam os requisitos em diagramas UML, e o PSM, por sua vez, é gerado com base no PIM e um conjunto de regras que transformam os diagramas UML em código-fonte.

Na Figura 14 é possível ver transformação do PIM no PSM. Um modelo (1) no nível do PIM é feito utilizando uma linguagem especificada por um meta-modelo (2). Uma plataforma em particular é escolhida (4) (5). Uma especificação da transformação para essa plataforma está disponível (3). Essa especificação da transformação é feita em termos dos mapeamentos entre os meta-modelos. O mapeamento orienta a transformação do PIM (1) no PSM (5).

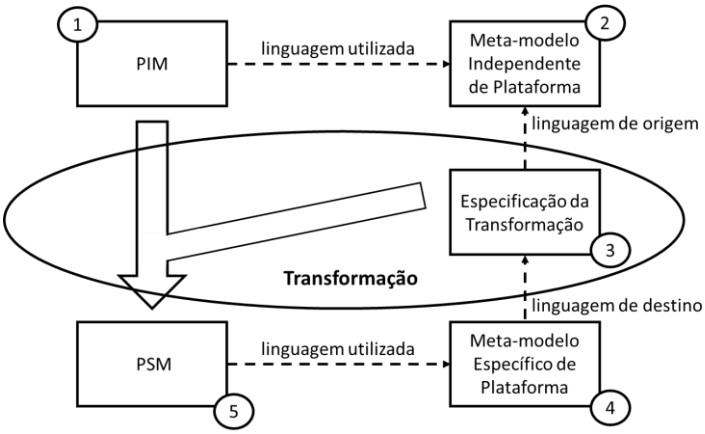

Figura 14 - Transformação do PIM em PSM. Fonte: O Autor

3.4 Ontologias
3.4.1 Sentido Filosófico

Em seu sentido filosófico, ontologia possui diversas definições. De acordo com Almeida et al (2003), é o ramo da metafísica que estuda os tipos de coisas que existem no mundo. A palavra é derivada do grego *ontos*, ser, e *logos*, palavra. Entretanto, seu termo de origem é a palavra aristotélica *categoria*, termo este utilizado no sentido de classificação. Neste sentido, Aristóteles[1] apresenta categorias que servem de base para classificar entidades, e introduz o termo *differentia* (Figura 15) para propriedades diferentes na classificação de espécies de mesmo gênero (categoria).

Figura 15 - Ontologia no sentido filosófico. Fonte: Almeida et al(2003)

Entretanto, apesar dos filósofos utilizarem ontologias durante anos para descrever domínios naturais do mundo, ainda não há um consenso, principalmente na área de Ciência da Computação, sobre a real semântica do termo ontologia (GUIZZARDI, 200).

[1] Filósofo grego nascido em Estagira, colônia grega da Trácia, no litoral setentrional do mar Egeu, em 384 a.C, foi um dos maiores pensadores de todos os tempos e considerado o criador do pensamento lógico

3.4.2 Ontologias e Ciência da Computação

O estudo de ontologias tem sofrido muitas pesquisas nas últimas décadas (GUARINO E WELTY, 2002). Isso contribuiu para o avanço de muitas áreas da ciência da computação, dentre as quais pode-se citar inteligência artificial (IA), gestão de conhecimento e engenharia de software (PATEL *et al*, 2007). De acordo com Gruber (1993), “uma ontologia é uma especificação explicita de um conceito”. A ontologia é um modelo de dados que representa um conjunto de conceitos dentro de um domínio e os relacionamentos entre eles. Uma ontologia é utilizada para realizar inferências sobre os objetos do domínio, são utilizadas como uma forma de representação do conhecimento sobre o mundo ou alguma parte dele.

Uma definição mais específica é feita por Borst (1997). Neste caso, o autor define uma ontologia como sendo a especificação formal e explícita de uma conceitualização compartilhada, onde especificação formal quer dizer algo que é legível para os computadores, explícita são os conceitos, propriedades, relações, funções, restrições e axiomas explicitamente definidos, conceitualização representa um modelo abstrato de algum fenômeno do mundo real e compartilhada significa conhecimento consensual.

Na prática, uma ontologia define uma “linguagem”, conjunto de termos, que será utilizada para formular consultas (ALMEIDA, 2003). A ontologia define as regras de combinação entre os termos e seus relacionamentos, estes relacionamentos são criados por especialistas, e os usuários formulam consultas usando os conceitos especificados.

Uma abordagem baseada em ontologias dentro do contexto da MDA pode prover o aprofundamento necessário no desenvolvimento do modelo de domínio (PATEL et al, 2007). Ainda segundo Patel et al (2007), o modelo de domínio não deve ser prejudicado por dependências computacionais, e sim expressar com clareza o processo de negócio da aplicação. Dessa forma, a abordagem ontológica contribuirá para o desenvolvimento de um modelo de domínio localizado na CIM.

3.4 Trabalhos Relacionados

A abordagem proposta por Oliveira (2011) apresenta uma técnica para análise da evolução da arquitetura de software baseada em visualização de software e métricas. A abordagem trabalha com a extração de dados de repositórios de gerência de configuração e permite a comparação entre: duas versões do modelo de arquitetura conceitual, duas versões do modelo emergente (gerado a partir da engenharia reversa do código-fonte) e de uma versão do modelo de arquitetura conceitual e do modelo emergente. Porém o trabalho não tem relação nenhum com o projeto dos sistemas, tanto a visualização como o cálculo de métricas são aplicados a todo o código fonte do software, não existe uma diferença clara entre elementos pertencentes a arquitetura do software e elementos específicos de implementação.

O trabalho de Ohst et al (2003) consistiu no desenvolvimento de um visualizador de diferenças em diagramas UML. A abordagem apresenta em um único diagrama elementos comuns entre as versões e elementos específicos de cada versão. O trabalho não trata da comparação entre arquiteturas (emergente e conceitual) e não permite a visualização da sequência temporal das versões e suas diferenças.

A ferramenta SAVE, proposta por Knodel et al (2006), permite a avaliação e visualização de arquiteturas, a partir da comparação entre a arquitetura conceitual e a emergente. Um aspecto não tratado pela ferramenta é a captura e o armazenamento dos modelos obtidos em diferentes instantes do tempo, para que seja feita a análise temporal da evolução, sendo esse um requisito fundamental para ferramentas de visualização da evolução de softwares, como relatado por Oliveira (2011).

Uma abordagem baseada em ontologias para a definição do Modelo Independente de Computação da MDA é proposta por Patel et al (2007). A ontologia desenvolvida tem como propósito somente a definição do modelo de domínio da aplicação e seus relacionamentos.

Maia (2006) propõe uma abordagem para a definição de transformações de modelos da MDA. Estas transformações são definidas por meio de uma sintaxe *XML* e um conjunto de tratadores implementados em linguagem *Java*, e podem ser posteriormente executadas de maneira bidirecional, por meio da máquina de transformações implementada para apoiar o uso da abordagem proposta. É apresentado também, um exemplo de transformação de modelos para a plataforma *Enterprise JavaBeans*.

Um meta-modelo para obtenção dos dados históricos dos arquivos de código-fonte, chamado de HISMO, é proposto por Gîrba et al (2005). O meta-modelo é baseado nas noções explicitas de histórico e versionamento. Como esses conceitos (histórico e versionamento) são genéricos eles foram aplicados a entidades específicas como pacotes, classes e métodos. O meta-modelo utiliza as várias versões de um determinado artefato (arquivo de código-fonte) obtidas por meio de uma ferramenta de controle de versão, para gerar a história do artefato. O histórico do projeto como um todo é então gerado a partir da junção do histórico de todos os artefatos. Após o meta-modelo ser preenchido com todos os dados de histórico do projeto é gerado um modelo da evolução do software.

Um conjunto de ontologias para evolução de software é proposto por Würsch et al (2012) em seu trabalho, titulado de SEON. As ontologias propostas por ele têm por objetivo mapear toda a pirâmide da evolução do software, desde os conceitos abstratos do domínio até o código-fonte da aplicação. A estrutura ontológica proposta por ele permite analisar o impacto de determinada modificação nos conceitos abstratos do sistema (requisitos) no código-fonte da aplicação. Como exemplo, ele propõe três propostas para aplicação da metodologia: um serviço de análise de software, uma interface de consulta de linguagem natural para desenvolvedores e um visualizador de software em larga escala. A Figura 16 apresenta a pirâmide de ontologias proposta por Würsch et al (2012).

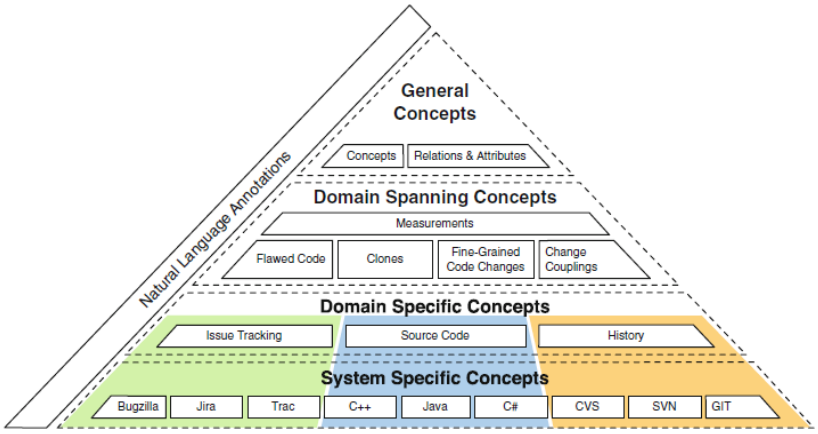

Figura 16 - Pirâmide de ontologia desenvolvida por Würsh et. al., 2012. Fonte: Würsh et. al., 2012

4 Metodologia

4.1 Introdução

A partir da revisão bibliográfica realizada neste trabalho, especialmente nos trabalhos de Oliveira (2011), Ohst et al (2003) e Knodel et al (2006) ficou claro a necessidade e os ganhos obtidos com o uso de técnicas de visualização na análise da evolução arquitetural do software. Além desses, os trabalhos de Würsch et al (2012) e Patel et al (2007) mostraram também importância e os ganhos obtidos com o uso de ontologias para definir conceitos de software.

Neste sentido, a abordagem proposta neste trabalho, denominada AESCOM (Avaliação da Evolução de Software Com Ontologias na abordagem MDA), tem por objetivo apoiar desenvolvedores e arquitetos de software no processo de percepção e compreensão da evolução arquitetural de sistemas, utilizando uma abordagem ontológica para caracterização do modelo de arquitetura conceitual do sistema e métricas e técnicas de visualização para avaliação da evolução arquitetural.

Além dessa seção introdutória, este capítulo está organizado da seguinte forma: a seção 4.2 introduz o processo de desenvolvimento de uma ontologia para caracterização da arquitetura do software. A seção 4.3 detalha as métricas de Precisão e Cobertura, propostas por Oliveira (2011) e adaptadas para o cálculo com base nas ontologias propostas, que foram também implementadas na ferramenta desenvolvida. O processo de visualização da evolução da arquitetura do software, utilizado na abordagem proposta, é detalhado na seção 4.4. Na seção 4.5 é apresentado uma adaptação dos mecanismos da MDA para a transformação do modelo emergente em conceitual. E por fim, a seção 4.6 apresenta detalhes de implementação de uma ferramenta para avaliação da evolução de software baseada nos mecanismos propostos pela abordagem AESCOM.

4.2 Definição das Ontologias

Como qualquer artefato de engenharia de software, ontologias têm de ser construídas seguindo métodos e técnicas adequadas (NADIR e FALBO, 2006). Para desenvolver a ontologia de avaliação da evolução de software foi adotado o método SABiO (Systematic Approach for Building Ontologies), desenvolvido por Falbo (2004). SABiO define um processo para construção de ontologias, cujas principais atividades estão representadas pela Figura 17.

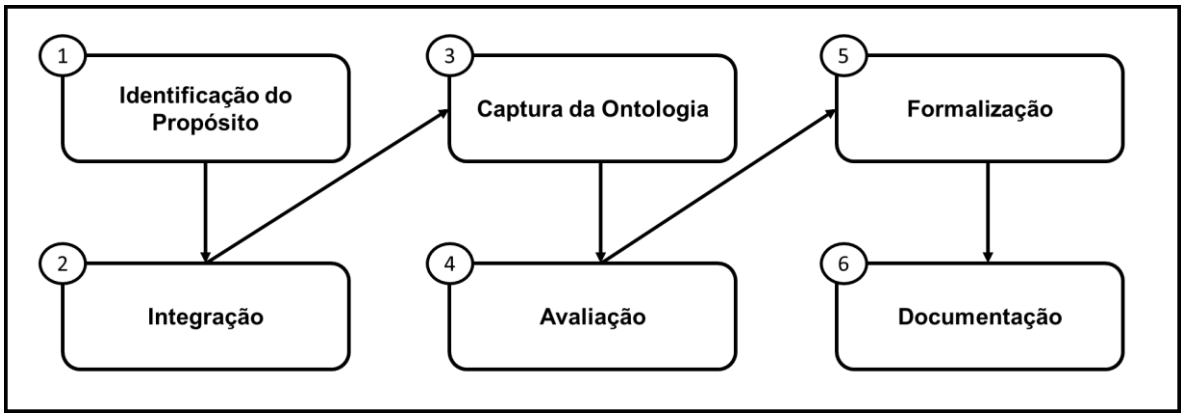

Figura 17 - Processo para construção de ontologias. Fonte: O Autor

1. Identificação do propósito e especificação de requisitos, que visa identificar questões que a ontologia deve ser capaz de responder;
2. Integração com ontologias existentes, que visa a reutilizar conceituações existentes e integrar a ontologia em desenvolvimento a uma rede de ontologias mais ampla;
3. Captura da ontologia, que tem por objetivo capturar os conceitos, relações, propriedades e restrições relevantes sobre o domínio em questão;
4. Avaliação da ontologia, que, dentre outros, trata de avaliar se a ontologia é capaz de responder às questões de identificadas;
5. Formalização, que busca escrever os axiomas da ontologia em uma linguagem formal (neste trabalho, optou-se pela lógica de primeira ordem); e

6. Documentação da ontologia, que visa a registrar o desenvolvimento da ontologia.

Como definido pelo método SABiO (FALBO, 2004), a primeira etapa do processo de criação de uma ontologia consiste na definição de seu propósito. No presente trabalho a ontologia tem por objetivo servir de suporte para avaliação da evolução arquitetural, utilizando a abordagem MVC (KRASNER e POPE, 1988), de um determinado software. A ontologia proposta deve garantir que somente modificações nos principais componentes e suas relações sejam caracterizadas como evoluções arquiteturais – ou seja, elementos específicos de implementação sejam desconsiderados no momento da análise da evolução. Ainda segundo o método, para caracterização da ontologia, devem ser propostas as questões que serão respondidas pela mesma. Dessa forma, as seguintes questões foram propostas:

Q1. Todos os elementos especificados no modelo de arquitetura conceitual foram implementados?

Q2. Todos os elementos implementados foram especificados no modelo arquitetural?

Q3. Todas as relações, entre os elementos arquiteturais, impostas no modelo de arquitetura conceitual foram mantidas no modelo emergente?

Q4. Todas as relações, entre os elementos arquiteturais, existentes no modelo emergente foram especificadas no modelo de arquitetura conceitual?

Q5. A camada de visualização (*view* da abordagem MVC) somente faz acesso, de leitura, a camada de modelo (*model* da abordagem MVC)?

Q6. A camada de domínio é independente das demais camadas, ou seja, não faz acesso a nenhuma outra funcionalidade da camada de controle (*controller* da abordagem MVC) ou visualização (*view* da abordagem MVC)?

As questões Q1 a Q6 tem o propósito de avaliar a evolução da arquitetura (Q1, Q2, Q3 e Q4) e garantir as restrições impostas pelo modelo MVC (Q5 e Q6), obtidas de Krasner e Pope (1988).

A ontologia desenvolvida, seus conceitos, restrições e axiomas serão apresentados na seção 5.1.

4.3 Métricas

As métricas de Precisão e Cobertura, provenientes da área de recuperação de informação (BAEZA-YATES & RIBEIRO-NETO, 1999), adaptadas por Oliveira (2011) para evolução arquitetural, tem o intuito de quantificar a adequação entre as arquiteturas conceitual e emergente, possibilitando a avaliação do grau de divergência entre essas. Essa divergência pode representar, por exemplo, elementos da arquitetura conceitual que ainda não foram implementados, ou mesmo elementos implementados – presentes na arquitetura emergente – que não foram descritos na arquitetura conceitual. Oliveira (2011) destaca que a verificação da aderência entre as arquiteturas pode evitar posteriores impactos negativos e auxiliar no processo de tomada de decisões em nível de projeto e implementação.

As métricas de Precisão e Cobertura podem indicar, respectivamente, a exatidão e a completude de um determinado instante de desenvolvimento em relação ao modelo de arquitetura conceitual. Os cálculos dessas métricas são feitos utilizando as seguintes fórmulas

$$Precisão = \frac{n_{ci}}{n_i}, \text{ onde } n_i = n_{ti} - n_{ei} \ (i) \qquad Cobertura = \frac{n_{ci}}{n_c} \ (ii)$$

Nas fórmulas (i) e (ii), n_{ci} é o número de elementos conceituais implementados, n_c representa o número de elementos conceituais, e n_i o número de elementos implementados, resultante da subtração entre o número total de elementos implementados (n_{ti}) e o número de elemento implementados identificados como específicos de implementação (n_{ei}), isto é, que devem ser subtraídos de forma a não interferir no cálculo da precisão.

Os elementos conceituais são todos os elementos presentes no modelo de arquitetura conceitual e caracterizados por uma ontologia, sejam eles classes ou métodos. Os elementos específicos de implementação são aqueles que não contem uma anotação das ontologias definidas, sejam eles métodos ou classes. Ou seja, n_{ti} são todas as classes e métodos implementados, enquanto n_{ei} são as classes e métodos implementados que não estão anotados com alguma ontologia.

A Figura 18 mostra como estão relacionados os elementos, utilizados nas métricas de Precisão e Cobertura.

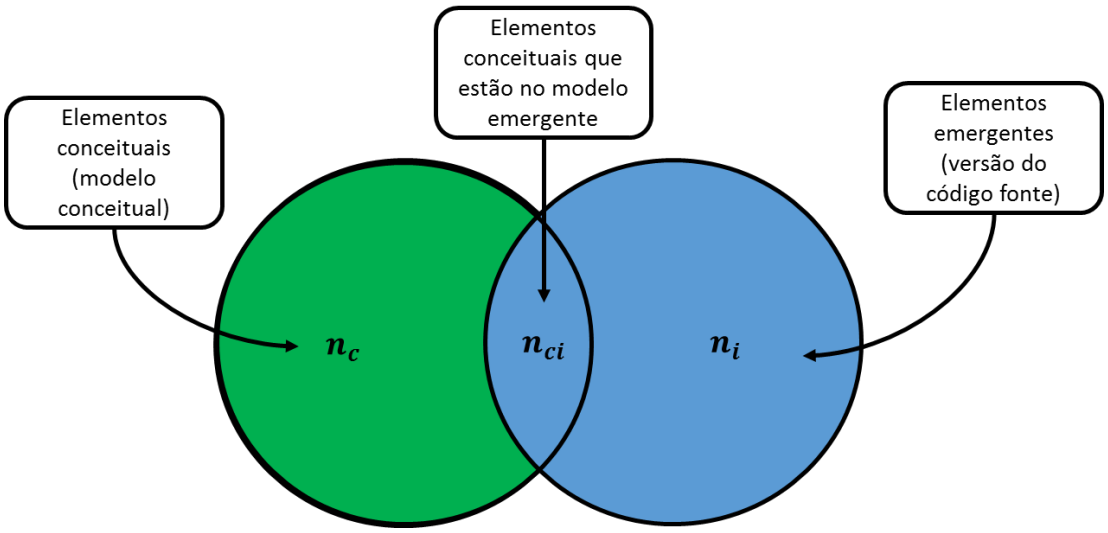

Figura 18 - Relação entre elementos das métricas. Fonte: O Autor

Os valores das métricas variam entre 0 e 1. A Tabela 1, mostra as interpretações dos valores extremos para ambas.

TABELA 1 - INTERPRETAÇÃO DAS MÉTRICAS DE PRECISÃO E COBERTURA

Métrica	Valor	Interpretação
Precisão	0	Indica que nenhum dos elementos implementados foram definidos no modelo de arquitetura conceitual
	1	Indica que todos os elementos implementados estão no modelo de arquitetura conceitual.
Cobertura	0	Indica que nenhum elemento definido no modelo de arquitetura conceitual foi implementado.
	1	Indica que todos os elementos conceituais foram implementados.

Fonte: Adaptado de Oliveira (2011).

Segundo Oliveira (2011), quanto mais distantes de 1 os valores de ambas as métricas estiverem, menor é a relação de pertinência dos elementos de cada modelo no outro e, consequentemente, maior é o grau de divergência entre os modelos conceitual e emergente. Mais interpretações para os valores das métricas serão apresentadas no capítulo 5.

4.4 Visualização da Evolução

Para visualização da evolução da arquitetura foi utilizado um mecanismo de comparação entre modelos UML. Os modelos utilizados são o modelo emergente e um modelo UML gerado por meio de engenharia reversa de uma versão do código fonte. Esta versão, pode ser obtida, atualizada, em qualquer instante do desenvolvimento do software de um processo que utilize mecanismos de versionamento de código fonte – a obtenção dessa versão para monitoramento pode ser uma tarefa da etapa de monitoramento e controle. Com base na versão, são feitas sucessivas comparações de modelos (conceitual e emergente), de forma a detectar os elementos comuns a ambas e as diferenças existentes. Realizada as comparações, é gerado um diagrama resultante que contém todos os elementos de ambos os modelos, exibidos de acordo com o resultado das comparações.

Com intuito de possibilitar a compreensão da aderência e da evolução dos modelos, conceitos e técnicas de visualização de software vistos no Capítulo 3 são aplicados na etapa de visualização do diagrama resultante.

4.5 MDA

Como discutido da seção 3.3.1 a MDA permite a separação da especificação das funcionalidades da especificação da implementação em uma plataforma específica. Além dessas especificações, a MDA prevê também o uso de regras de transformação, regras essas que geram um nível mais inferior (por exemplo, PSM) com base em um nível mais superior (por exemplo, PIM). Para a abordagem proposta, foram criadas regras de transformação entre o PIM e PSM de forma que fosse possível gerar o diagrama UML (PIM) com base no código fonte (PSM).

Para que houvesse uma forma de comparar ambos os modelos (conceitual e emergente) a representação de cada um deveria ser numa mesma plataforma, porém o modelo de arquitetura conceitual é comumente feito utilizando a UML e o modelo emergente, no trabalho atual, é feito em Java. Dessa forma foi escolhida a representação UML para comparação entre os modelos, fazendo necessária a transformação do código-fonte (modelo específico de plataforma) em um diagrama UML (modelo independente de plataforma).

A Figura 19 exemplifica o processo de transformação de código fonte em diagrama UML.

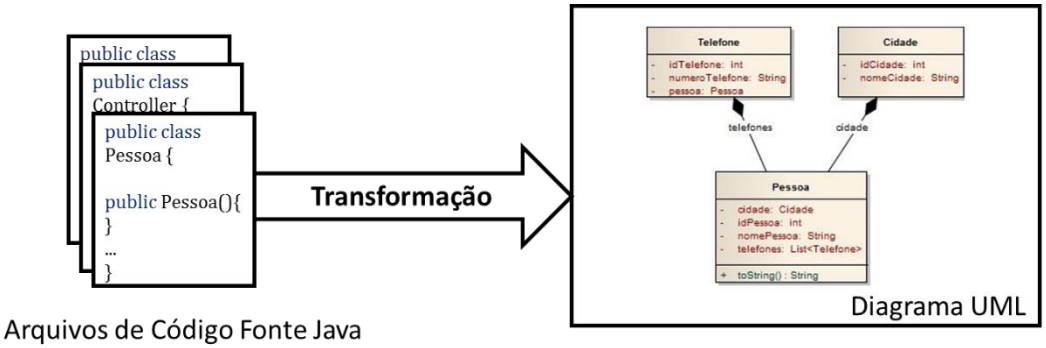

Figura 19 - Transformação de código fonte em diagrama UML. Fonte: O Autor

4.6 Implementação

Inicialmente, para a implementação de uma ferramenta de avaliação da evolução da arquitetura de software, que utilizasse os mecanismos propostos (ontologias e métricas, descritos na seção 4.2 e 4.3 respectivamente), optou-se por estender a ferramenta EvolTrack (CEPÊDA et al, 2010), um mecanismo de extração e visualização do ciclo de evolução de projetos de software implementado como um *plug-in* do ambiente de desenvolvimento Eclipse (ECLIPSE FOUNDATION, 2010). A escolha dessa ferramenta deu-se devido à similaridade com a abordagem proposta, Oliveira (2011) utilizou a mesma para implementação da abordagem PREViA, portanto EvolTrack (CEPÊDA et al, 2010) já incluía o cálculo da métricas de Precisão e Cobertura, fazendo necessário somente a adequação das ontologias propostas para análise da evolução da arquitetura de software. Porém, não foi possível obter acesso ao código fonte da aplicação, responsáveis pelo desenvolvimento da ferramenta informaram que a mesma estava passando por um processo de manutenção e não havia um prazo para disponibilização do seu código fonte. Iniciou-se então o desenvolvimento de uma nova ferramenta utilizando a linguagem Java, sem qualquer vínculo com as já existentes.

O desenvolvimento da ferramenta consistiu na implementação de mecanismos para:

a) Uso das ontologias propostas na forma de anotações da linguagem *Java*;
b) Extração dos dados de repositórios;
c) Extração dos dados de modelo de arquitetura conceitual (lógica);
d) Comparação de modelos (conceitual e emergente);
e) Visualização de diferenças entre modelos (conceitual e emergente) e
f) Cálculo das métricas de Precisão e Cobertura.

Todos os mecanismos serão discutidos nas seções seguintes.

4.6.1 Arquitetura da Ferramenta

A ferramenta baseou-se no estilo arquitetural em camadas, além da utilização de alguns padrões de projeto que serão discutidos a seguir, ainda nessa seção. A divisão em camadas foi escolhida visando os benefícios que essa traz para o software, principalmente o aumento da coesão e a diminuição do acoplamento (VOGEL et al, 2009). Cada camada do software contêm pacotes, que por sua vez agregam classes, no presente trabalho foram utilizadas 3 camadas:

- Camada de Apresentação – responsável pelo controle da interação entre usuário e sistema. Contém componentes responsáveis pela lógica de apresentação da aplicação. Dente as suas responsabilidades estão a captura de dados, a apresentação de resultados e o controle de navegação. Sua principal missão é tratar as regras de apresentação independente das regras de negócio, isolando a complexidade pertinente a cada camada. Essa camada contém dois pacotes: *View* e *Presenter*. O pacote *View* contém as janelas, somente o desenho, quem controla as interações e apresentação dos dados é o pacote *Presenter*.
- Camada de Domínio – Esta camada contém as classes do domínio da solução, os objetos de negócio. Esses objetos compõem o modelo de dados OO do sistema.
- Camada de Serviço - A camada de serviços possui componente que são utilizados por todas as outras camadas do software. Cada pacote tem uma responsabilidade específica.
    - Compilador – responsável por compilar o modelo emergente, para posterior extração de informações.
    - Métricas – responsável pela aplicação das métricas de Precisão e Cobertura.
    - Repositório – responsável por controlar a conexão com repositórios de código fonte, download de arquivos e navegação entre versões.
    - Ontologias – responsável pela interpretação das ontologias dos modelos emergente e conceitual.

- Visualização – responsável pela implementação das técnicas de visualização de evolução

O diagrama de arquitetura da ferramenta é apresentado na Figura 20.

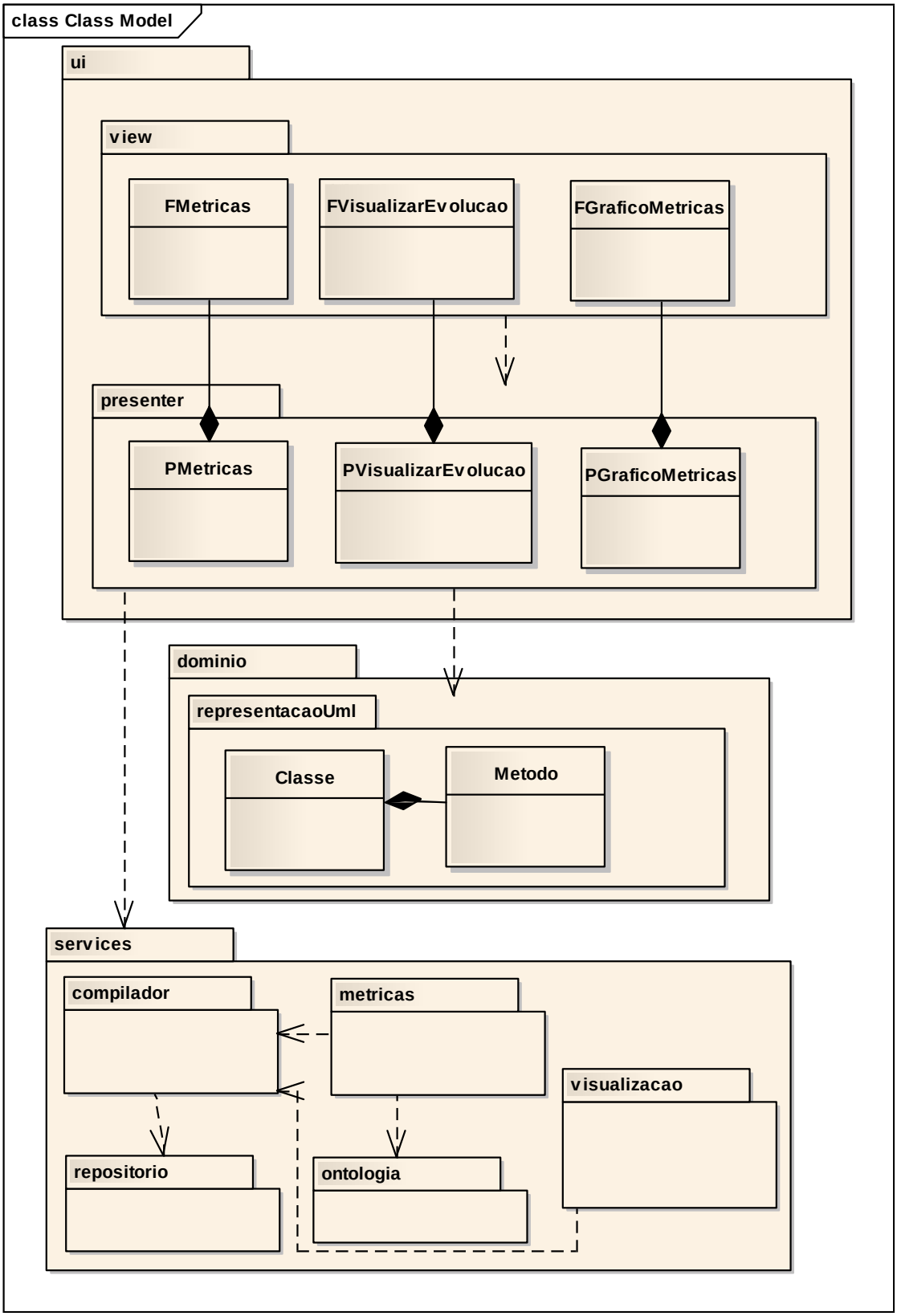

Figura 20 - Diagrama de Arquitetura da ferramenta desenvolvida. Fonte: O Autor

4.6.2 Anotações para caracterização das ontologias

A implementação das anotações para as classes e métodos, que descrevem as ontologias, foi feita em outro módulo, na forma de biblioteca. Dessa forma os projetos que forem utilizar a ferramenta proposta para análise da evolução somente devem vincular essa biblioteca na aplicação e indicar quais classes e métodos fazem parte da arquitetura. A biblioteca contém as seguintes anotações:

- *ClasseModel* – anotação utilizada para identificar uma classe que está dentro da camada de modelo, da abordagem MVC;
- *ClasseView* – anotação utilizada para identificar uma classe que é responsável pela visualização de dados, da abordagem MVC;
- *ClasseController* – anotação utilizada para identificar uma classe que está dentro da camada de controle, da abordagem MVC;
- *AtualizaView* – anotação utilizada para identificar métodos da classe *Controller* que são responsáveis pela atualização de sua respectiva classe de visualização;
- *AtualizaModel* – anotação utilizada para identificar métodos da classe *Controller* que são responsáveis pela atualização dos dados de sua classe de domínio respectiva;
- *ObtemDadosModel* – anotação utilizada para identificar métodos da classe *View* que são responsáveis por obter dados de sua classe de domínio respectiva.

Em todas as anotações, quando utilizadas, deve ser informado a qual entidade de domínio ela se refere. A Figura 21 apresenta um diagrama de um código Java anotado.

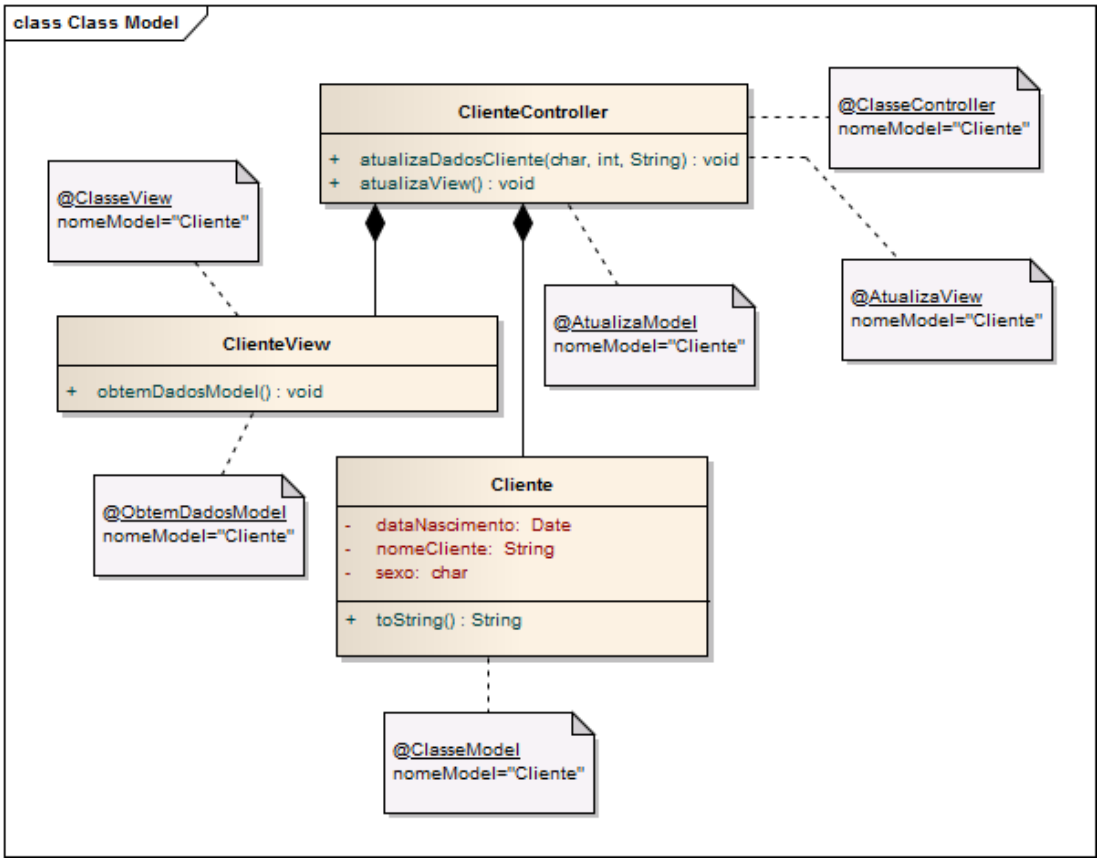

Figura 21 - Diagrama com as classes anotadas. Fonte: O Autor

A Figura 22 apresenta a implementação da anotação para a classe *model* na linguagem Java. O exemplo de uso dessa anotação será apresentado na seção 5.2.

Anotação para classes Model

```
1.  @Target(ElementType.TYPE)
2.  @Retention(RetentionPolicy.CLASS)
3.  public @interface ClasseModel {
4.        String nomeModel();
5.  }
```

Figura 22 - Implementação da anotação ClasseModel. Fonte: O Autor

4.6.3 Obtenção de dados de repositórios de código fonte

O mecanismo para obtenção dos dados de repositório foi implementado utilizando a biblioteca SvnKit, versão 1.7.8, juntamente com o padrão *Adapter* (GAMMA et al, 1994), com a utilização desse padrão de projeto a ferramenta permite que a qualquer momento essa biblioteca possa ser trocada, sem impacto em outras funções do sistema, que fazem uso do mecanismo. A Figura 23 contém o diagrama de classe do mecanismo de obtenção de dados de repositório.

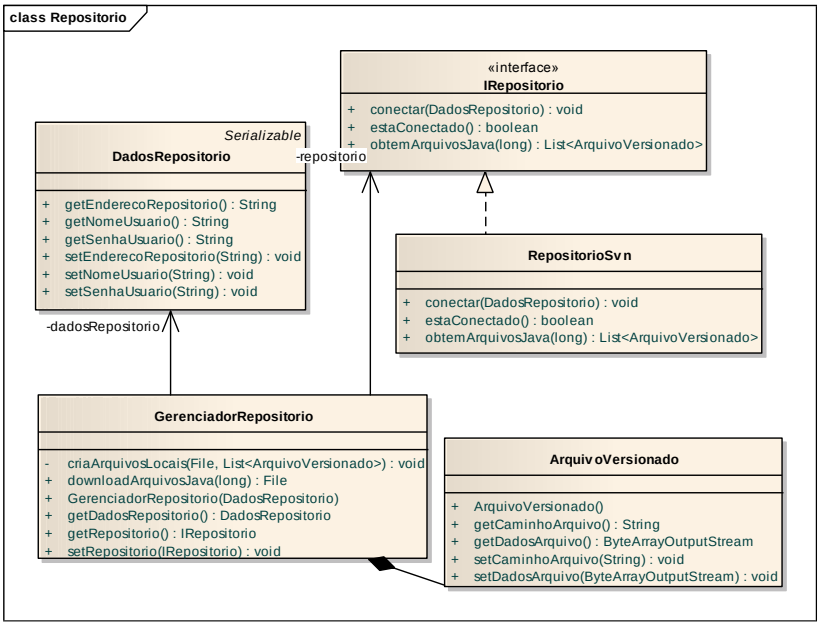

Figura 23 - Diagrama de Classe do mecanismo de obtenção de dados de repositório. Fonte: O Autor

A ferramenta implementada, que será discutida no próximo capítulo, permite a navegação por toda a estrutura de arquivos e pastas do repositório, navegação entre versões e *download* de arquivos e pastas de uma determinada versão.

4.6.4 Representação do modelo conceitual e emergente

Além da obtenção dos dados do repositório foi implementado um mecanismo para compilação e obtenção dos dados do modelo emergente. A compilação dos arquivos de código fonte é feita utilizando a biblioteca JavaParser, na sua versão 1.0.8, uma limitação dessa biblioteca é a compilação somente de projetos que utilizem o *framework* Java versão 1.5. Após a compilação dos elementos, uma estrutura interna com a representação dos elementos da versão do modelo emergente é criada. Essa estrutura de representação é semelhante a um diagrama de classe da UML e é utilizada para representar tanto o modelo emergente quanto o conceitual, foi necessária sua criação para que houvesse uma maneira de comparar os modelos. Nesse sentido, um meta-modelo foi criado, representando um subconjunto dos elementos da UML.

Com base nessa estrutura, é possível representar todo um diagrama de classes da UML, como pode ser visto na Figura 24, a estrutura contempla a representação de classes, interfaces, métodos, atributos, composições, herança e as ontologias referentes ao elemento (classe ou método).

Inicialmente, para a obtenção do modelo de arquitetura conceitual, pretendia-se utilizar o XML *Metada Exchange* (XMI) (OMG, 2013), padrão proposto pela OMG para compartilhamento de informações em XML. O uso mais comum do XMI é como um formato de troca de meta-dados entre ferramentas de modelagem UML, embora também possa ser utilizado para modelos de outras linguagens. Dessa forma, o desenvolvedor poderia utilizar a ferramenta de sua preferência para montar o modelo de arquitetura conceitual de seu sistema, e depois somente importaria essa informação para dentro da ferramenta que seria capaz de ler o XMI e obter os dados do diagrama. Porém, ao analisar ferramentas de modelagem UML, tais como *Enterprise Architect* (*Trial*) e *Star UML*, verificou-se que ambos possuíam a possibilidade de exportação dos dados via XMI, porém os documentos gerados não tinham um padrão – documentos gerados pelo *Enterprise Architect (Trial)* eram lidos incorretamente pelo Star UML e vice-versa. O apêndice C apresenta um exemplo de como a transferência de um diagrama de classe, utilizando o XMI, do *Enterprise*

Architect para o Star UML não funciona. Dessa forma, a abordagem de importação utilizando XMI foi invalidada. Para a leitura do modelo de arquitetura conceitual foi criado um mecanismo de leitura para cada plataforma de modelagem, baseado também no padrão de projeto (*design pattern*) *Adapter*, para que, caso seja necessário incluir novos padrões de ferramentas de modelagem UML nenhuma modificação estrutural na ferramenta seja necessária, apenas a implementação das classes do novo padrão. Atualmente a ferramenta possibilita a obtenção de modelos do *Enterprise Architect* (*Trial*).

O meta-modelo que representa essa estrutura é demonstrado pela Figura 23.

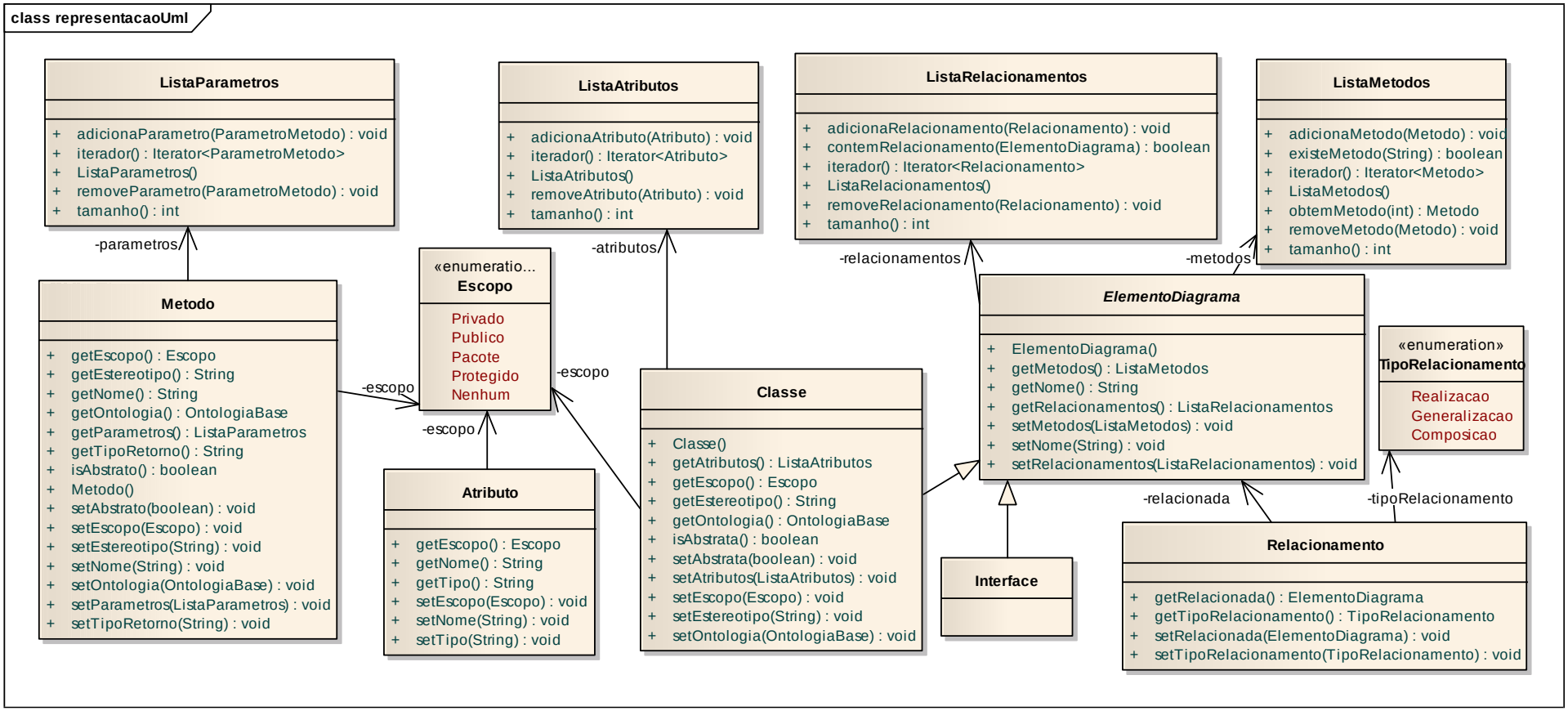

Figura 24 – Meta-modelo de classe para representação da UML. Fonte: O Autor

A Figura 25 representa o diagrama de classe do padrão *Adapter* para leitura de modelos conceituais.

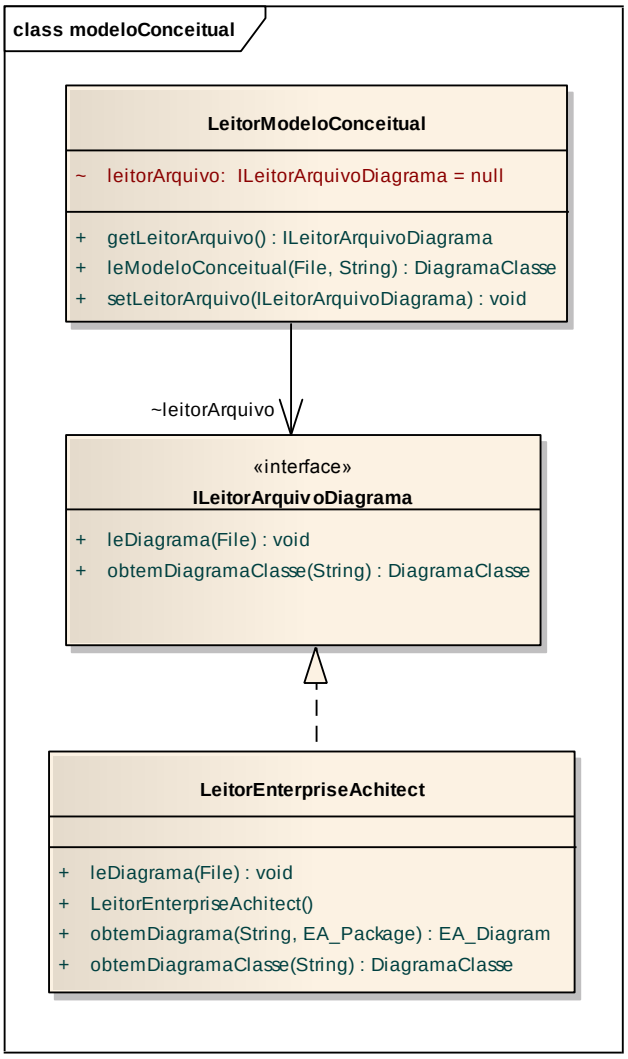

Figura 25 - Padrão Adapter para leitura do modelo de arquitetura conceitual. Fonte: O Autor

4.6.5 Comparação e Visualização de Modelos

A visualização das diferenças entre modelos foi implementada utilizando a biblioteca *JGraph*, de código fonte aberto. Dessa forma foi possível montar o diagrama de classe de forma visual (semelhante a ferramentas de modelagem), explicitando seus elementos e conexões. Para a visualização das diferenças entre os modelos, o modelo de arquitetura conceitual foi adotado como base e qualquer diferença do

conceitual para o emergente era marcada com uma cor diferente no diagrama, seguindo as regras:

- Verde – para elementos que estão tanto no modelo de arquitetura conceitual quanto emergente;
- Vermelho – para elementos que estão somente no modelo de arquitetura conceitual (ou seja, estão faltando no modelo emergente); e
- Azul – para elementos que somente no modelo emergente (ou seja, foram adicionados).

O pseudocódigo 1 demonstra como é feita a verificação de diferenças entre os modelos:

```
DiferençasDiagrama(modeloConceitual, modeloEmergente)
1.  PARA-CADA classe EM modeloConceitual FAÇA
2.    SE modeloEmergente.Contém(classe) ENTÃO
3.      marcaElemento(classe, Verde);
4.    SENÃO
5.      marcaElemento(classe, Vermelho);
6.    FIM-SE;
7.  FIM-PARA;
8.  PARA-CADA classe EM modeloEmergente FAÇA
9.    SE modeloConceitual.nãoContém(classe) ENTÃO
10.      marcaElemento(classe, Azul);
11.    FIM-SE;
12.  FIM-PARA;
13.FIM.
```

Pseudocódigo 1 – Verificação de diferenças entre modelos

O pseudocódigo 1, verifica as diferenças de classes em dois modelos, o mesmo procedimento é feito para enumerar e verificar a existência dos relacionamentos entre as classes – composição, generalização e realização.

4.6.6 Cálculo das métricas de Precisão e Cobertura

As métricas de Precisão e Cobertura, propostas por Oliveira (2011), foram calculadas com base no modelo de arquitetura conceitual e uma versão do emergente. O cálculo de métricas foi também projetado para que seja possível a adição de novas métricas, sem que haja alteração estrutural na ferramenta. O apêndice A e B apresentam, respectivamente, o algoritmo para cálculo de precisão e cobertura.

5 Resultados Obtidos

Como resultado do presente trabalho foi desenvolvida a abordagem AESCOM, com objetivo de apoiar desenvolvedores e arquitetos de software no processo de percepção e compreensão da evolução arquitetural de sistemas, sendo essas atividades essenciais nos processos de desenvolvimento de software – como discutido na seção 1.1. Parte do desenvolvimento da abordagem foi a modelagem de uma ontologia para caracterização da arquitetura de softwares, baseada na abordagem MVC, essa caracterização utilizando ontologias permitiu classificar cada elemento semanticamente – seu papel e relacionamentos com os demais. A ontologia desenvolvida permite a caracterização tanto do modelo de arquitetura conceitual como emergente, possibilitando dessa forma a comparação de ambos, em qualquer espaço de tempo, para cálculo das métricas e visualização da evolução.

Mecanismos já existentes para avaliação da evolução da arquitetura de softwares (visualização e métricas) foram adaptados para ter como base de avaliação de evolução arquitetural os conceitos e restrições da ontologia proposta. Complementado os objetivos do trabalho, foi implementada uma ferramenta para avaliação da evolução de softwares utilizando as ontologias, métricas e formas de visualização propostas.

Dando continuidade à apresentação dos resultados obtidos, a seção 5.1 apresenta as ontologias desenvolvidas e demonstra como as são utilizadas no processo de desenvolvimento de software, tanto na modelagem do modelo de arquitetura conceitual, como na implementação (codificação) do software. A seção 5.2 apresenta a técnica de visualização proposta para visualização da evolução. Na seção 5.3 é demostrado como é feito a transformação entre o modelo emergente e conceitual – apoiado na abordagem MDA. Detalhes das funcionalidades da ferramenta implementada são descritos na seção 5.4. Em estudo de caso é apresentado na seção 5.5, para demonstração da ferramenta desenvolvida e demonstração de como a mesma, apoiada nos mecanismos de avaliação propostos, pode ajudar desenvolvedores e arquitetos de software a avaliarem a evolução da arquitetura do software, para que possam diminuir, ou até evitar, a evolução

descontrolada da arquitetura emergente. Na seção 5.6 é apresentada uma discussão sobre a abordagem proposta. E, por fim, a seção 5.7 faz uma comparação da abordagem AESCOM com as demais abordagens disponíveis no contexto de avaliação da evolução de software.

5.1 Ontologias

A Figura 26 mostra o modelo de arquitetura conceitual da ontologia de avaliação da evolução de software proposta, apresentado seus conceitos e relações.

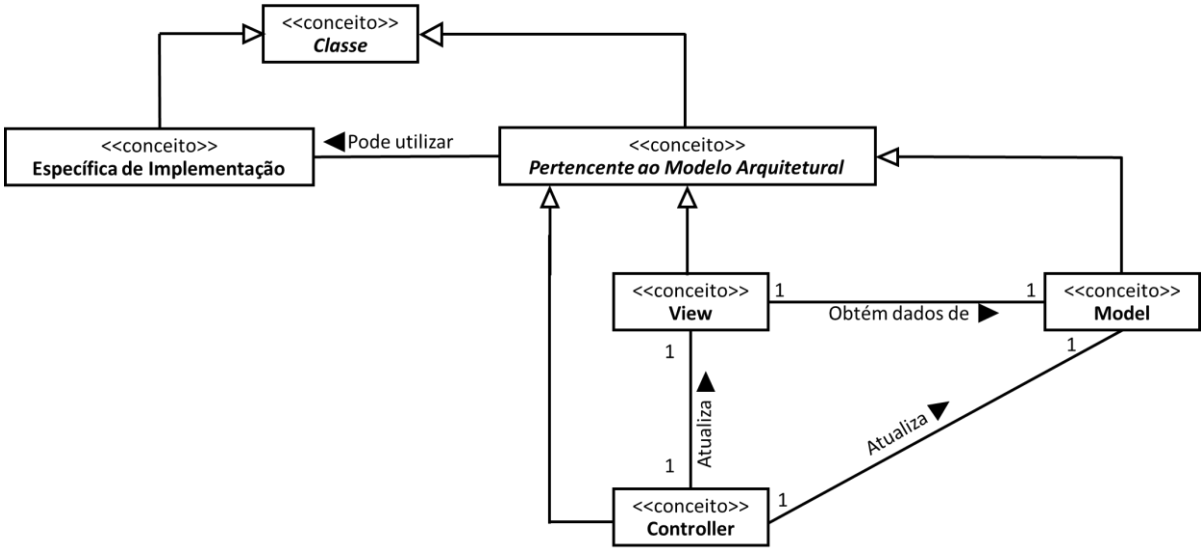

Figura 26 - Modelo de arquitetura conceitual da Ontologia Proposta. Fonte: O Autor

O tipo *Classe* é um conceito abstrato (não pode ser instanciado), elemento base de implementação do software. Desse elemento herdam dois conceitos, *Específico de Implementação* e *Pertencentes ao Modelo Arquitetural*. Elementos específicos de implementação não fazem parte da arquitetura conceitual do software e não devem ser levados em conta no momento da avaliação da evolução da arquitetura. Já o conceito *Pertencente ao Modelo Arquitetural*, também abstrato, é base para os elementos arquiteturais do modelo MVC.

Do conceito *Pertencente ao Modelo Arquitetural* é que serão extraídas as informações para análise da evolução arquitetural, dele herdam os conceitos: *View*, *Model* e *Controller*. O *View*, refere-se aos elementos responsáveis pela apresentação das informações e interação com o usuário, conforme modelo MVC. Esse elemento se relaciona com o *Model*, porém somente obtém os dados desse para apresentação, não realiza nenhuma alteração nas informações. O *Model*, que caracteriza a camada de domínio da aplicação, não faz referência a nenhum outro elemento arquitetural, se não a outros da mesma camada, sua responsabilidade conforme modelo MVC, é somente representar os dados do domínio da aplicação. O *Controller*, por sua vez, tem livre acesso tanto escrita como leitura, às demais camadas do modelo arquitetural, é quem gerencia o funcionamento da aplicação. Por fim, todos os conceitos pertencentes ao modelo arquitetural podem utilizar elementos específicos de implementação.

No modelo apresentado na Figura 26, existem algumas restrições importantes que não são explicitas e, portanto, foram escritas na forma de axiomas. Porém, antes de apresentar os axiomas, faz-se necessário conhecer os predicados utilizados nesses. O predicado $controladoraModelo(c,\ m)$, indica que a classe controladora c é responsável pelas informações do modelo m, ou seja, é ela quem gerencia seus dados. O predicado $controladoraView(c,\ v)$, indica que a classe controladora c é responsável por manipular a interface de visualização v. E o predicado $associação(v,\ m)$, indica que a classe de apresentação v é responsável pela visualização das informações do modelo m. A relação entre esse predicados é dada pela seguinte sentença:

Sentença – $(\forall m_1, v_1, c_1)\big(controladoraModel(c_1, m_1) \wedge controladoraView(c_1, v_1)\big) \rightarrow associacao(v_1, m_1)$

A Sentença 1 indica que, estando uma visualização v_1 e um modelo m_1 controlados pela mesma controladora c_1, a visualização v_1 será responsável pela apresentação do modelo m_1. Outro axioma importante é apresentado a adiante, porém, é necessário introduzir mais alguns predicados. O predicado $controladoraAtualizacao(c,\ v,\ m)$ indica que a controladora c tem a responsabilidade de atualizar a visualização v e o modelo m. A relação entre esses predicados é dada pela seguinte sentença:

Sentença 2 - $(\forall m_1, v_1, c_1)\ \exists\ controladoraAtualizacao(c_1, v_1, m_1)$

A Sentença 2 indica que, para todo modelo m_1 e visualização v_1, deve existir uma controladora que atualize (gerencie) suas informações.

5.1.1 Aplicação da Ontologia Desenvolvida no Processo de Desenvolvimento de Software

Os conceitos, restrições e axiomas gerados para a ontologia são utilizados para verificar a evolução da arquitetura do software. Nesse sentido, o presente trabalho adotou a utilização desses conceitos em dois momentos:

i. Especificação: no momento da modelagem da arquitetura do sistema, o arquiteto de software irá utilizar os conceitos (*view, model, controller, atualiza* e *obtemDados)* na forma de estereótipos nas classes UML e seus métodos respectivos.

ii. Implementação: no momento da implementação do software, o desenvolvedor irá utilizar os conceitos (*view, model, controller, atualiza* e *obtemDados)* na forma de anotações nas classes e seus métodos respectivos.

Todos os conceitos da ontologia têm o atributo *nomeModel*. Esse atributo deve conter o nome da entidade de domínio que será referenciada pelo conceito. A Figura 27 exemplifica o uso das ontologias em um diagrama UML.

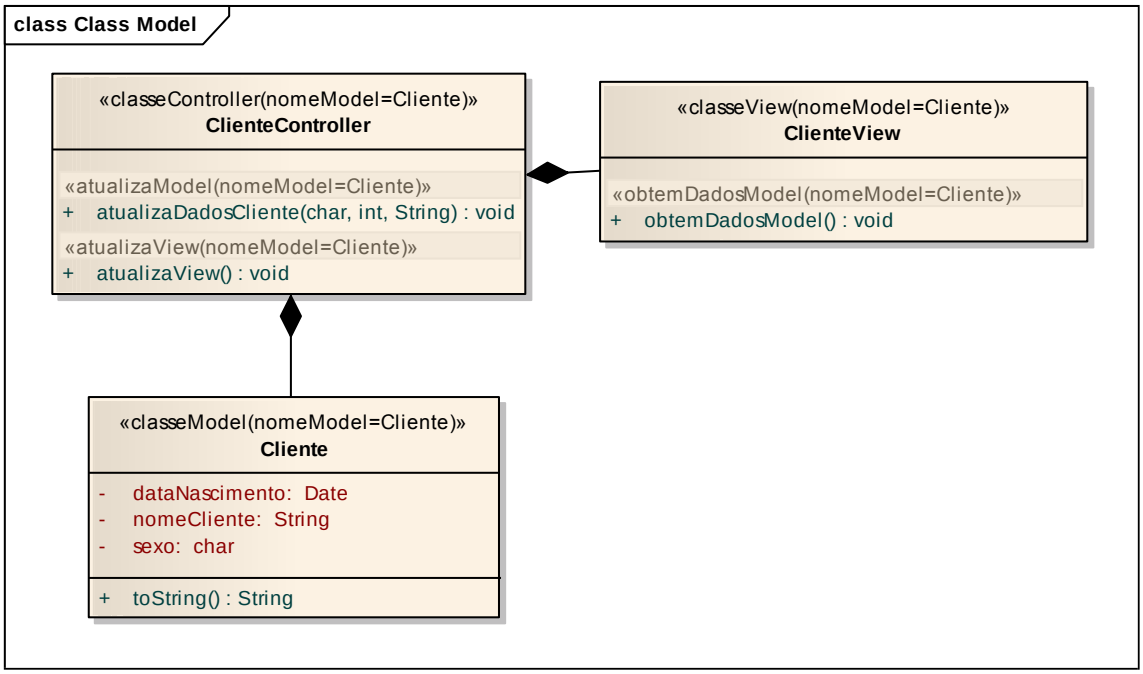

Figura 27 - Diagrama de Classe com Ontologias. Fonte: O Autor

No diagrama apresentado na Figura 27, todos os conceitos da ontologia foram aplicados, expostos na forma de estereótipos da UML. A classe *ClienteController* foi anotada com o conceito *classeController* e como parâmetro foi informado a qual *model* se refere a *controller*. Ainda na classe *ClienteController* foram criados dois métodos, um para atualizar os dados da *model* que foi anotado com a relação *atualizaModel* e outro para atualizar a *view* que foi anotado com a relação *atualizaView*. A classe *ClienteView* foi anotada com o conceito *classeView* e como parâmetro foi informado a qual *model* se refere a *view*, essa classe tem um método para obter os dados da *model* que foi anotado com a relação *obtemDadosModel*. A classe *Cliente* foi anotada com o conceito *classeModel* e passado como parâmetro qual o nome da *model*.

Paralelamente, na Figura 28, são apresentadas as ontologias especificadas na forma de anotações, na linguagem Java:

```
@ClasseController(nomeModel = "Cliente")
public class ClienteController
{

    private Cliente cliente;

    public ClienteController() {
    }

    @AtualizaModel(nomeModel = "Cliente")
    public void atualizaModel(){

    }

    @AtualizaView(nomeModel = "Cliente")
    public Cliente atualizaView(){
        return null;
    }
}
```

Figura 28 - Código Java anotado com as ontologias. Fonte: O Autor

Para ser possível realizar as anotações como exibido na Figura 28, é necessário adicionar um *JAR* (biblioteca Java), desenvolvido no presente trabalho, a qualquer projeto em Java e as anotações já estarão disponíveis para uso. Existem anotações para todos os conceitos e relações da ontologia. O exemplo mostra somente as anotações da classe *controller*, porém existem, similarmente, para as classes *model* e *view*.

5.2 Visualização da Evolução

A visualização da evolução é feita utilizando somente um diagrama, onde são apresentados elementos de ambos os modelos (conceitual e emergente) destacando suas diferenças, o foco de visualização é de aderência entre as arquiteturas, essa forma de visualização tem por objetivo transmitir a ideia de sobreposição ou preenchimento da arquitetura emergente à arquitetura conceitual (OLIVEIRA, 2011). Os elementos que são considerados diferentes ganham uma coloração diferente, de forma que tal realce seja percebido pelo usuário.

No diagrama a cor dos elementos indica os estado de dos mesmos – elementos na cor verde estão em ambos os modelos, elementos vermelhos só estão no modelo de

arquitetura conceitual (ou seja, estão faltando no modelo emergente), enquanto elementos em azul estão somente no modelo emergente (ou seja, foram adicionados). Os mesmos mecanismos de comparação para os elementos são também aplicados para as relações entre esses elementos. A visualização é exemplificada pela Figura 29.

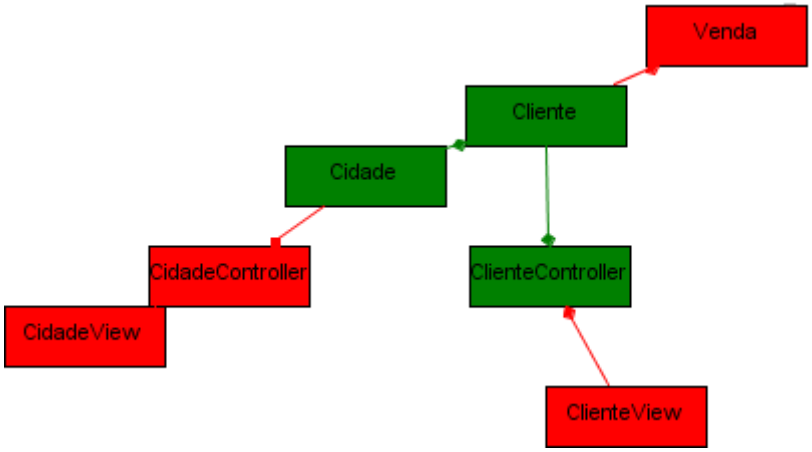

Figura 29 - Exemplo de visualização da evolução. Fonte: O Autor

O diagrama gerado contém informações tanto do modelo de arquitetura conceitual como do emergente, agrupados em uma só visualização. A Figura 29 exibe 7 elementos (*CidadeView*, *CidadeController*, *Cidade*, *Cliente*, *ClienteController*, *ClienteView* e *Venda*), desses somente 3 elementos estão implementados na versão analisada do modelo emergente (*Cidade*, *Cliente* e *ClienteController*). Da mesma forma as relações entre os elementos, conectores na cor verde indicam a igualdade entre os modelos, enquanto os conectores vermelhos indicam que não existe tal referência no modelo emergente.

Com intuito de aumentar a quantidade de informações disponíveis, ainda com um baixo custo de obtenção de informações, mantendo a qualidade de percepção visual – mostrando somente o principais componentes do software e de acordo com a necessidade ir explicitando os demais – a técnica de Detalhamento, ou *Drill-Down,* foi agregada à visualização. Dessa forma, o usuário poderá obter mais informações de um determinado elemento. Ao clicar sobre qualquer elemento do diagrama, uma nova interface de visualização é aberta, explicitando para o usuário os métodos da entidade, seguindo o mesmo padrão de cores do diagrama, métodos que existem no modelo de arquitetura conceitual e não estão presentes no emergente em vermelho,

métodos que estão implementados porém não existem no modelo de arquitetura conceitual em azul e os métodos presentes em ambos os modelos em verde.

Para visualização temporal da evolução da arquitetura, a ferramenta desenvolvida possibilita ao usuário navegar entre as versões (disponíveis no mecanismo de controle de versões) do modelo emergente, enquanto o diagrama, exemplificado pela Figura 15, é gerado dinamicamente – sendo a navegação e a geração dinâmica da visualização requisitos essenciais para uma ferramenta de visualização da evolução de software, como relatado por Oliveira (2011). Mais detalhes sobre a ferramenta serão tratados na seção 5.5.

5.3 MDA

Como explicado na seção 4.5 é necessária a transformação do modelo específico de plataforma (código fonte implementado em Java) em modelo independente de plataforma (UML). A transformação de uma versão do código-fonte, obtida da ferramenta de controle de versão, em diagrama UML é feita automaticamente. Para tanto foi criada uma abstração do diagrama de classes da UML (detalhada na próxima seção), em Java, dessa forma para cada arquivo de código fonte encontrado no repositório, seu conteúdo é verificado, e caso haja classes e métodos nesse arquivo, esses são adicionados ao diagrama UML. Por fim, caso as classes ou métodos referenciem outras classes as relações são criadas no diagrama UML. O pseudocódigo 2 mostra como o diagrama UML é gerado.

```
criaDiagrama(versaoCodigoFonte, diagramaUML)
1.  PARA-CADA classe EM versaoCodigoFonte FAÇA
2.    SE classe.contemOnotologia() ENTÃO
3.      diagramaUML.insereClasse(classe);
4.      PARA-CADA metodo EM classe.metodos() FAÇA
5.        SE metodo.contemOntologia() ENTÃO
6.          diagramaUML.insereMetodo(método);
7.          SE éClasseDoProjeto(metodo.tipo) ENTÃO
8.            diagramaUML.insereRelacionamento(classe, metodo.tipo)
9.          FIM-SE;
10.       FIM-SE;
11.     FIM-PARA;
12.    FIM-SE;
13. FIM-PARA;
14.FIM.
```

Pseudocódigo 2 – Criação do diagrama UML a partir do modelo emergente

5.4 Ferramenta desenvolvida

A ferramenta desenvolvida, apoiada nas abordagens propostas, para avaliação da evolução da arquitetura de software, conta com as seguintes funcionalidade, representadas pelo diagrama de caso de uso na Figura 30.

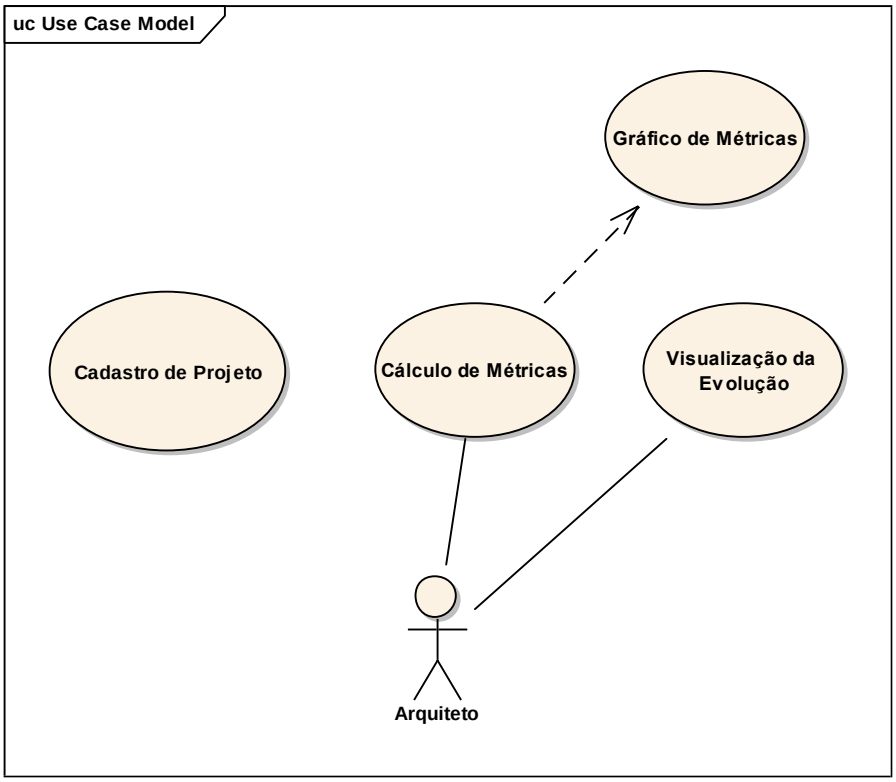

Figura 30 - Diagrama de caso de uso da ferramenta desenvolvida. Fonte: O Autor

No caso de uso “Cadastrar Projeto” o arquiteto de software insere algumas informações sobre o projeto no qual será feita a análise da evolução: nome, modelo conceitual e caminho do repositório onde está o modelo emergente. Esse caso de uso é pré-requisito para os demais. O caso de uso “Cálculo de Métricas” possibilita o ator (arquiteto) calcular as métricas de Precisão e Cobertura para qualquer intervalo de versões, uma extensão desse é o caso de uso “Gráfico de Métricas”, onde é possível visualizar um gráfico de séries (cada série uma métrica) referente ao valor das métricas em cada versão do modelo emergente. Por fim, o caso de uso “Visualização da Evolução” permite ao ator visualizar a evolução temporal da arquitetura do software, navegando entre as versões do modelo emergente. Todos os casos de uso serão abordados no estudo de caso (seção 5.5).

O diagrama de atividades, Figura 31, apresenta o fluxo de execução da ferramenta desenvolvida.

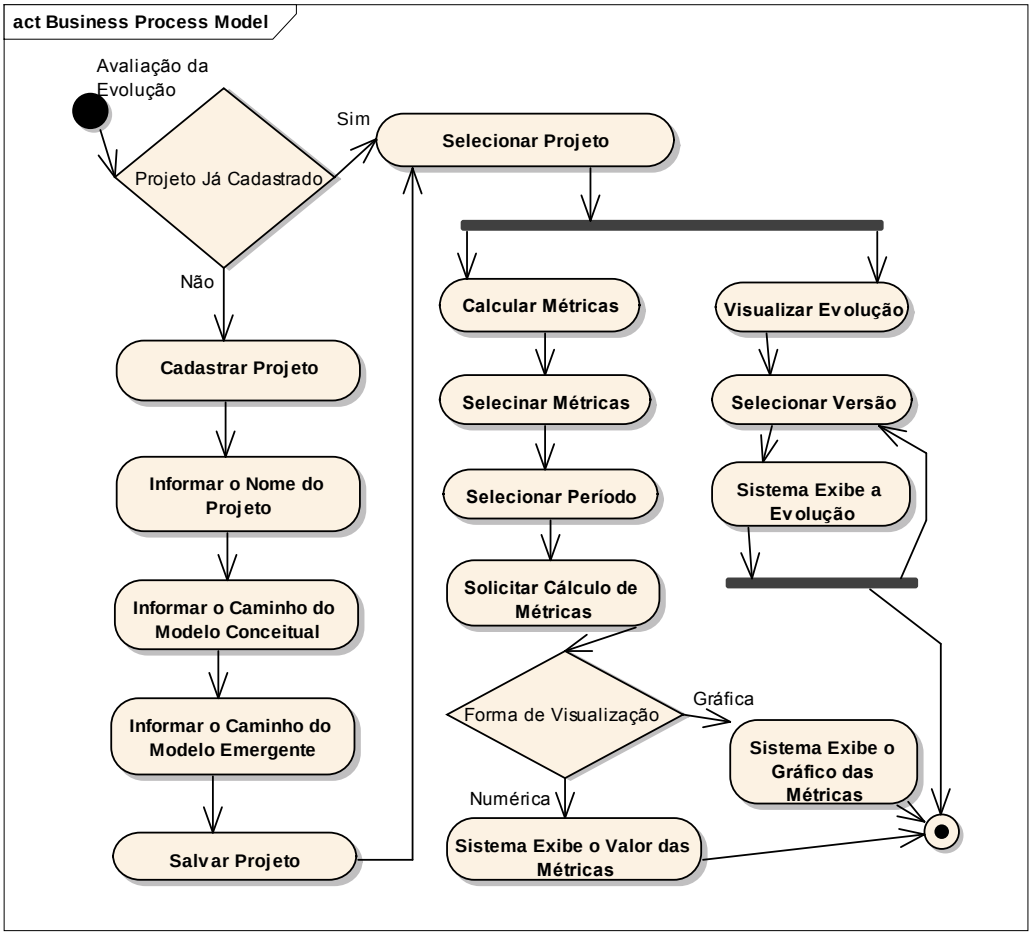

Figura 31 - Diagrama de atividades representando o fluxo de execução da ferramenta desenvolvida.
Fonte: O Autor.

5.5 Estudo de Caso

Para exemplificar a utilização da ferramenta desenvolvida, apoiada na abordagem ontológica proposta, utilizou-se neste trabalho um processo de desenvolvimento de um software hipotético. O processo utiliza o ciclo de vida em cascata e conta com as etapas:

1. Levantamento e análise de requisitos;
2. Validação dos requisitos;
3. *Design* (projeto) da solução;
4. Implementação do software;
5. Elaboração do plano de testes funcionais;
6. Realização dos testes funcionais.

Considerando que as etapas 1 e 2 já foram realizadas e ao final dessas tenham sido obtidos os seguintes requisitos:

1. O software deve possibilitar o cadastro (inclusão, edição, consulta e exclusão) de clientes com as seguintes informações: Nome, Data de Nascimento, CPF, Telefone Fixo, Telefone Celular, Sexo e Cidade.
2. O software deve possibilitar o cadastro (inclusão, edição, consulta e exclusão) de cidades com as informações: Nome e UF.
3. O software deve possibilitar o cadastro (inclusão, edição, consulta e exclusão) de produtos com as informações: Descrição, Quantidade em estoque e Preço.
4. O software deve possibilitar o registro de vendas (inclusão e consulta) de vendas com as seguintes informações: Data da Venda, Número da Venda, Cliente e Produtos Vendidos.

Seguindo o fluxo do processo apresentado, a próxima etapa é a de *design* ou projeto da solução. Nessa etapa entra em cena o arquiteto de software, que por sua vez irá modelar o sistema, na abordagem MVC e já utilizando as ontologias apresentadas, para que posteriormente possa fazer o acompanhamento do evolução da arquitetura do software.

No final da etapa de *design*, o arquiteto apresenta o seguinte diagrama, representado pela Figura 32:

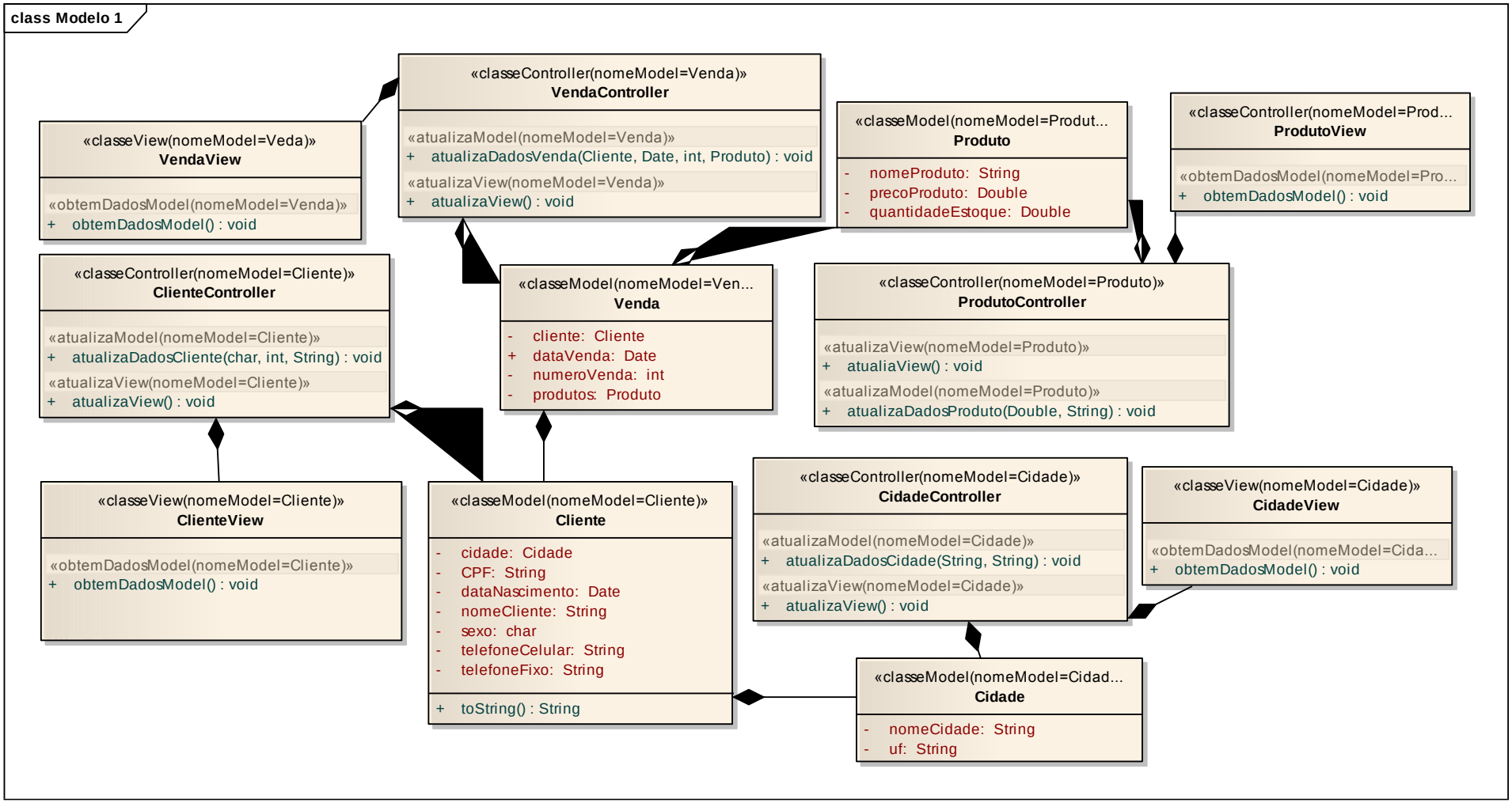

Figura 32 - Diagrama gerado ao final da etapa de design. Fonte: O Autor

Como apresentado na Figura 32, os elementos da abordagem MVC foram identificados por meio de estereótipos. As classes *ClienteController*, *CidadeController*, *ProdutoController* e *VendaController*, foram identificadas com o estereótipo *classeController* e o nome da classe de domínio a que se refere, do mesmo modo foi feito para classes do modelo de domínio e da camada de visualização. Similarmente as relações entre os conceitos da ontologia, representadas pelos métodos das classes, foram também identificadas com estereótipos.

Uma vez pronto o *design* do software, o mesmo passa para a etapa de implementação. Para tanto é criado um repositório, no qual ficará armazenado o código fonte da aplicação e demais documentos que se fizerem necessários. Paralelamente ao trabalho dos desenvolvedores na implementação do software, o arquiteto responsável pelo projeto da solução, acompanha a evolução do mesmo utilizando a ferramenta desenvolvida. Para tanto, primeiramente é necessário cadastrar o projeto, com informações como: Nome, endereço do repositório e o modelo de arquitetura conceitual do software, como pode ser visto na Figura 32.

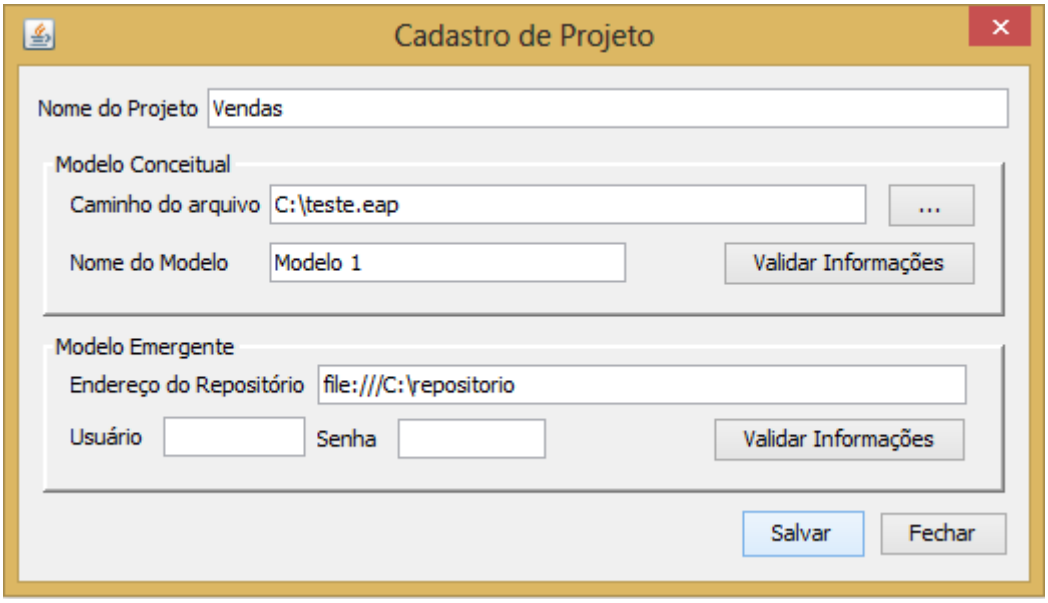

Figura 32 - Interface para cadastro de Projeto. Fonte: O Autor

Após configuradas as informações do projeto, o arquiteto poderá a qualquer momento fazer a verificação das métricas de Precisão e Cobertura ou visualizar graficamente a evolução do software. Para apresentação das métricas, existem duas

formas: numérica e visual. A interface gráfica que possibilita o cálculo e visualização das métricas é apresentada pela Figura 33.

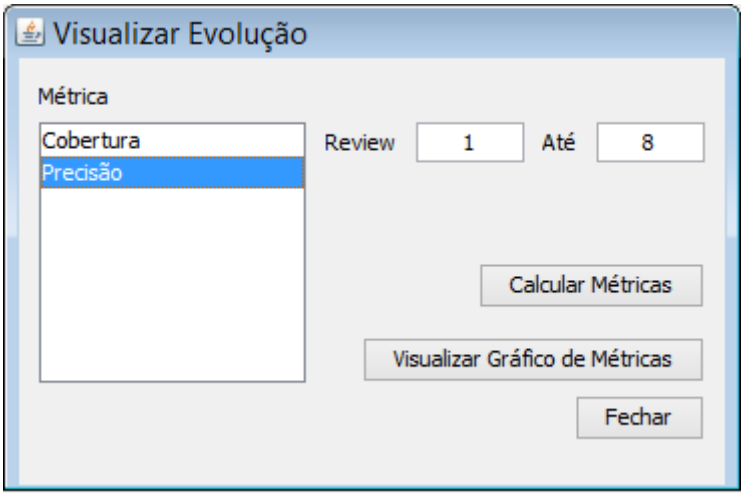

Figura 33 - Interface para cálculo das métricas. Fonte: O Autor

Como é possível observar na Figura 33, o cálculo das métricas pode ser feito separadamente ou juntas. Além disso, ao invés de selecionar somente uma versão para análise pode-se fazer utilizando um intervalo. Para exemplificar considera-se que, na versão 1 foi implementada somente a controladora do cliente (*ClienteController*) ou seja, um total de 3 elementos (a própria classe juntamente com seus métodos, como já discutido na seção de métricas). Com base nisso são calculadas as métricas de Precisão e Cobertura, o seguinte resultado é apresentado na Figura 34.

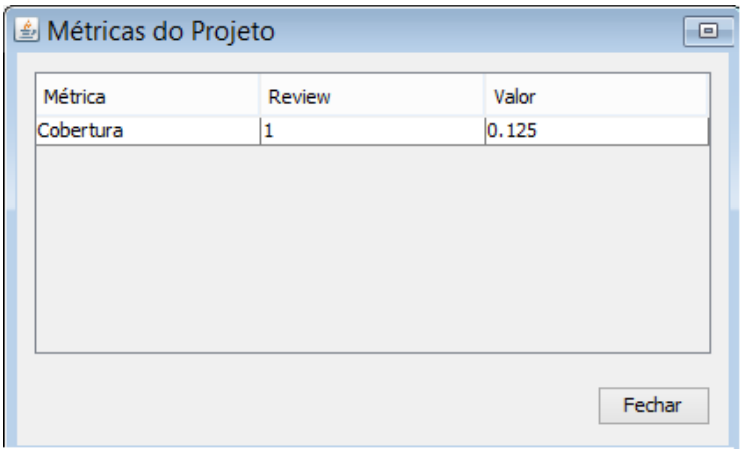

Figura 34 - Interface do resultado do cálculo das métricas. Fonte: O Autor

Para a métrica de Cobertura, tem-se o valor de 0.125, ou seja, 12,5% do projetado foi implementado na versão 1. Utilizando essa métrica é possível acompanhar o andamento do projeto, considerando que a versão 1 fosse um marco do projeto, como percentual do projeto que deveria estar pronto nesse marco, é possível tirar conclusões sobre o estado do projeto – atrasado, adiantado ou dentro do previsto. A métrica é calculada segundo a seguinte fórmula:

$$Cobertura = \frac{n_{ci}}{n_c}$$

Para o n_{ci} tem-se o valor de 3, que foi o total de elementos implementados na versão 1, e para o n_c obteve-se o valor de 24 (classes e métodos, do diagrama de classe apresentado), logo:

$$Cobertura = \frac{3}{24} = 0{,}125$$

Além da forma comum de apresentação das métricas, em forma de número apenas, foi também implementado um gráfico de séries, em que cada série representa uma métrica, o eixo X representa a versão e o Y representa o seu valor, como pode ser visto na Figura 31. Dessa forma é possível verificar a evolução de uma determinada métrica de forma visual, e compara-la com outras ao mesmo tempo, se for o caso.

Dando continuidade ao desenvolvimento, até a versão 7, os seguintes elementos foram implementados, além do que já implementado na versão 1: *CidadeController*, *Produto*, *Cidade*, e *CidadeView*. Realizando a mesma análise das métricas, porém visualizando ambas as métricas (precisão e cobertura) de forma gráfica. Semelhante à interpretação da métrica em sua forma numérica, conclusões sobre o andamento do projeto podem ser tiradas do gráfico. Com relação à métrica de Precisão, como pode-se observar na Figura 31, esta se manteve constante durante todas as versões do projeto, tal fato deve-se a natureza da métrica, que sinaliza divergências entre o que foi projetado e o que foi implementado, como somente foram implementados elementos constantes no modelo de arquitetura conceitual, o modelo emergente manteve-se 100% preciso. Valores diferentes de 1 (ou 100%) para a métrica de Precisão podem sinalizar desvios no desenvolvimento, implementação de elementos arquiteturais não definidos no modelo de arquitetura conceitual. Com a seleção das duas métricas, o período (versão 1 até 7) e solicitar à ferramenta que gere o gráfico, a seguinte interface gráfica é apresentada, representada pela Figura 35.

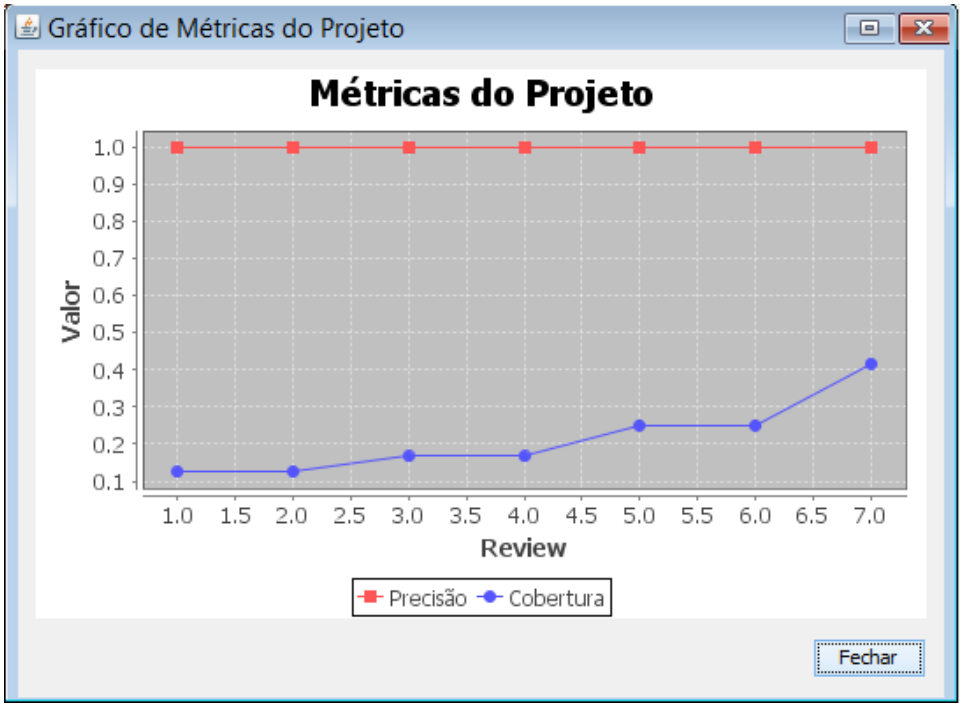

Figura 35 - Interface do gráfico de métricas. Fonte: O Autor

Como apresentado na Figura 35, a métrica de Precisão manteve-se constante durante todas as versões do projeto, tal fato deve-se a natureza da métrica, que sinaliza divergências entre o que foi projetado e o que foi implementado, como somente foram implementados elementos constantes no modelo de arquitetura conceitual, o modelo emergente manteve-se 100% preciso. Modelos emergentes que contenham elementos arquiteturais não previstos no modelo conceitual são 100% precisos – por exemplo, caso uma nova entidade de domínio seja adicionada ao modelo emergente. Nesses casos cabe ao arquiteto de software decidir qual decisão tomar:

- Adicionar as novas classes de arquitetura no modelo conceitual, ou seja, atualizar o modelo arquitetural do sistema; ou
- Solicitar ao desenvolvedor a remoção dos novos elementos do código fonte, caso as alterações efetuadas pelo mesmo são façam parte do escopo do projeto.

A visualização da evolução arquitetural pode ser realizada na ferramenta desenvolvida utilizando a interface apresentada pela Figura 36. Ao executar essa funcionalidade, com a versão zero selecionada, o modelo conceitual é exibido, com todos elementos na cor cinza. Também nessa interface, ao realizar a navegação entre versões, as métricas de Precisão e Cobertura são calculadas.

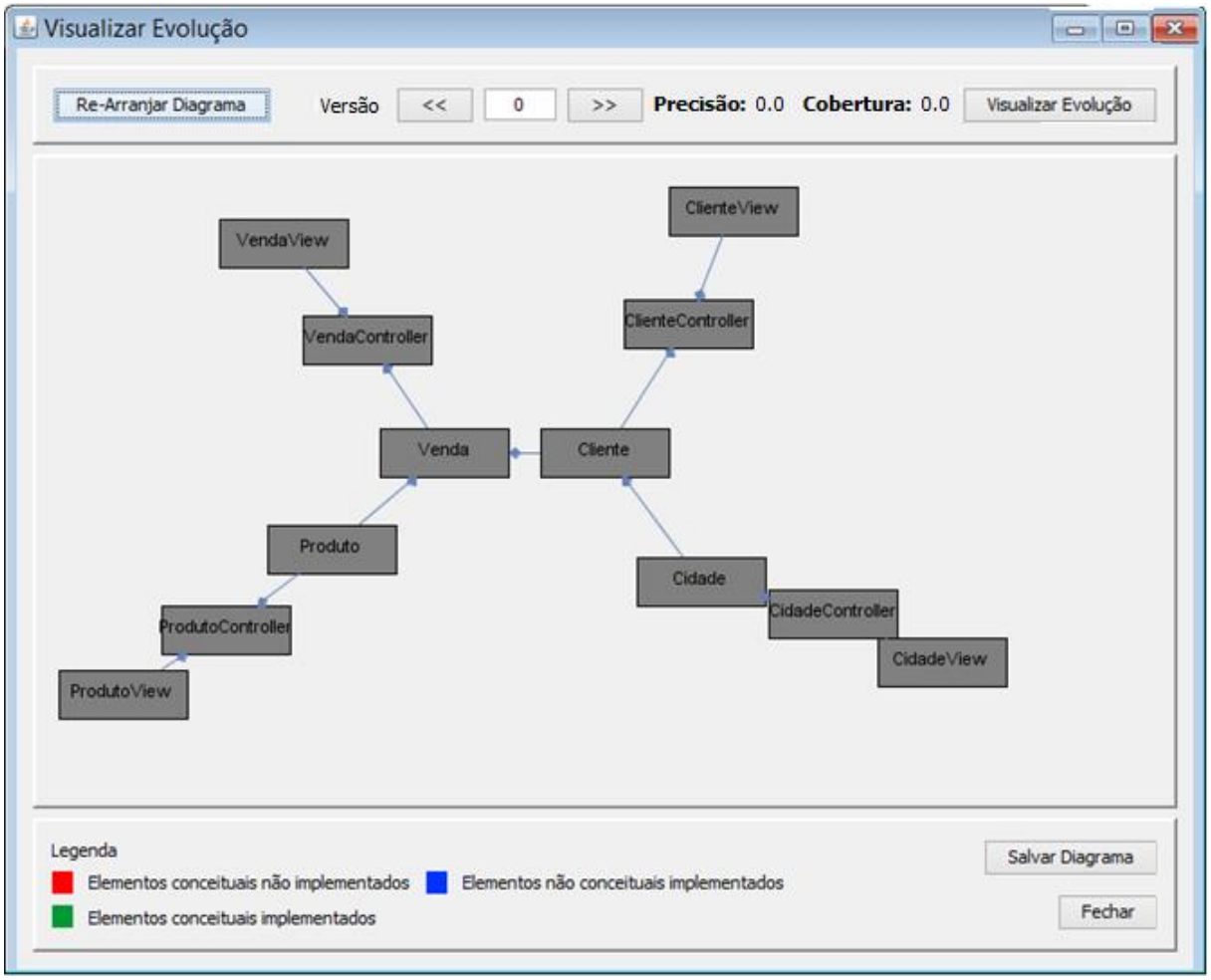

Figura 36- Interface de visualização da evolução. Fonte: O Autor

Na interface apresentada pela Figura 36, o usuário pode navegar pelas versões do software, o painel inferior contém as legendas referentes às cores que são utilizadas no diagrama. O recurso de navegação permite ao usuário visualizar a evolução temporal do modelo emergente e sua aderência ao modelo de arquitetura conceitual. Como nenhuma versão está selecionada (versão 0) o diagrama é o modelo de arquitetura conceitual, ao clicar no botão “>>” o diagrama é montado e a seguinte imagem é apresentada, representada pela Figura 37.

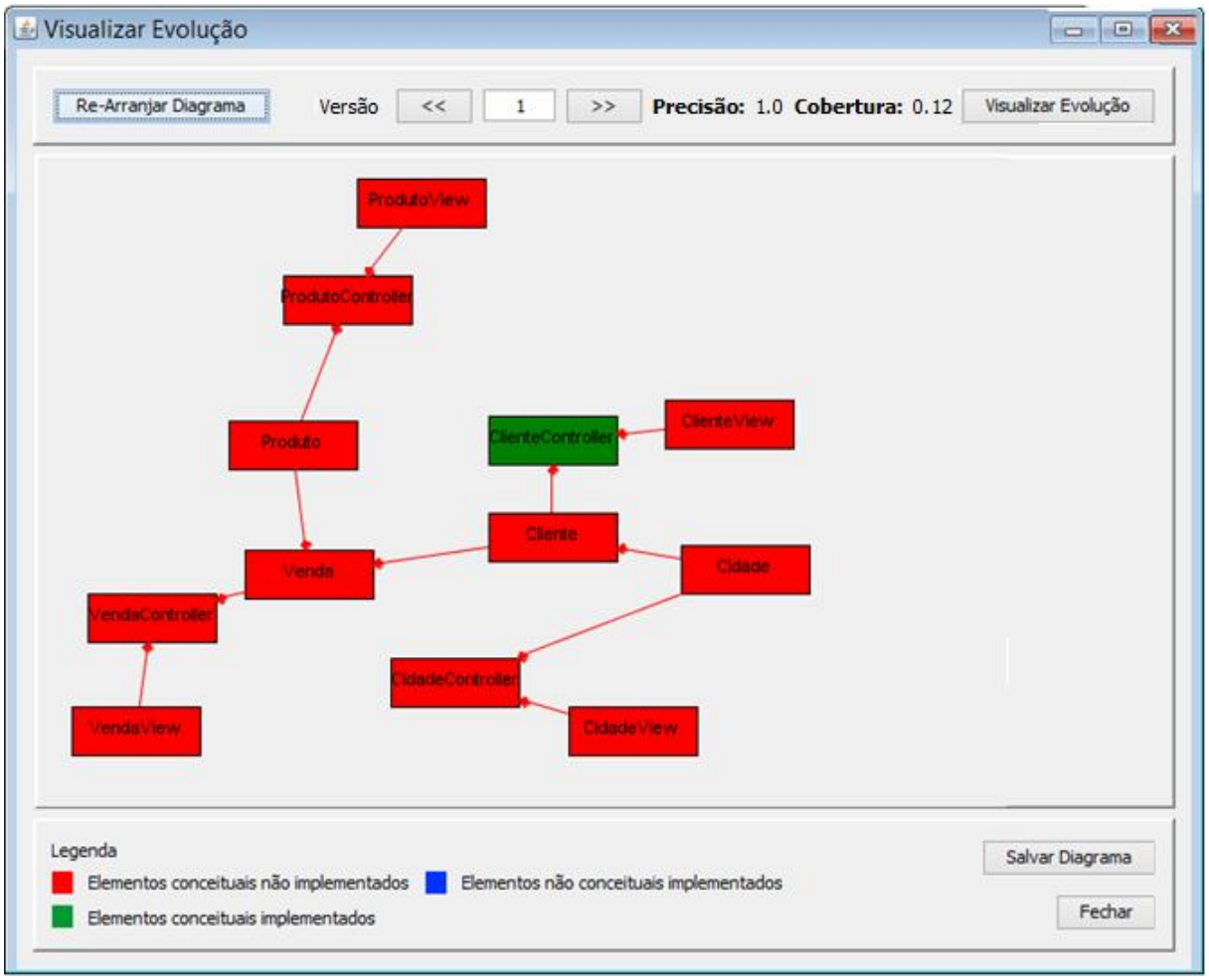

Figura 37 - Interface de visualização da evolução, versão 1. Fonte: O Autor

Como já mencionado, na versão 1, somente a classe *ClienteController* foi implementada, destacada em verde no diagrama (que representa elementos constantes tanto no modelo de arquitetura conceitual quanto no emergente), os demais elementos, como são foram implementados nessa versão, estão em vermelho.

Passando para a versão 7, tem-se o seguinte diagrama, representado pela Figura 38.

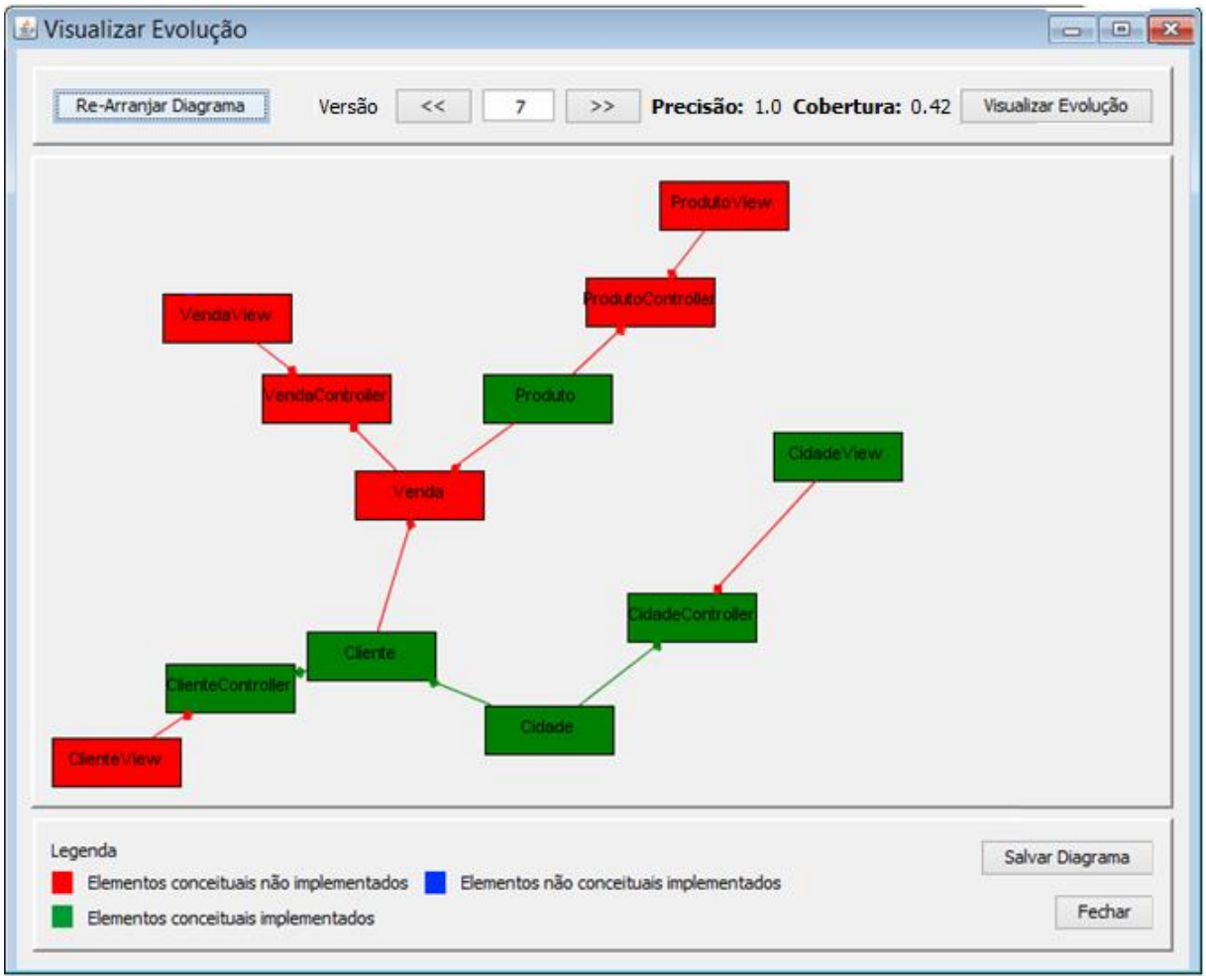

Figura 38 - Interface de visualização da evolução, versão 7. Fonte: O Autor

Como representado no diagrama da Figura 34, somente os elementos em vermelho não foram implementados na versão 7, além das classes a entidade *CidadeController* fez uso de *CidadeView*, que estava previsto no modelo de arquitetura conceitual, dessa forma a conexão entre esses elementos é representada pela cor vermelha. Além dessa técnica de visualização, a ferramenta disponibiliza a opção de detalhamento dos elementos, ao dar um duplo clique no elemento desejado, uma nova interface é aberta, que exibe os métodos da mesma, seguindo o mesmo esquema de cores para elementos existentes ou não no modelo emergente (Figura 39).

Ao selecionar a entidade *ClienteController* é exibida a janela apresentada pela Figura 39.

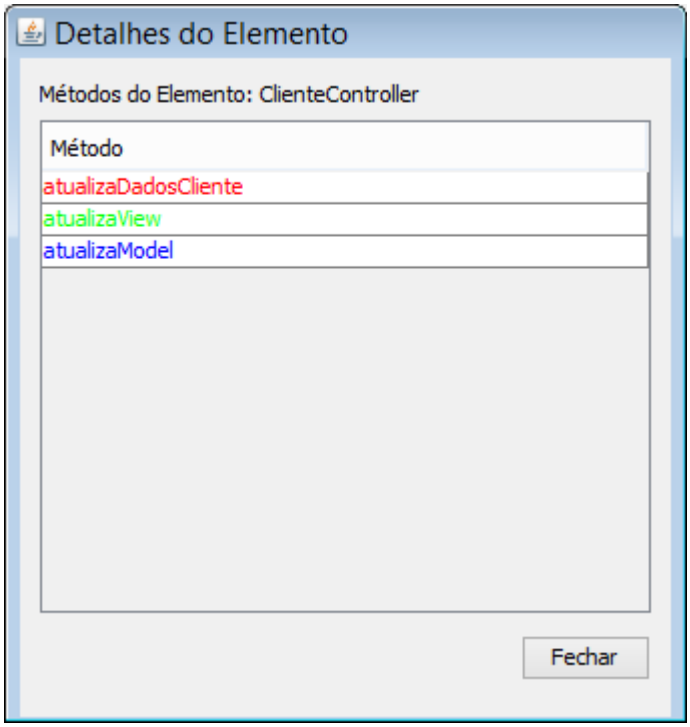

Figura 39 - Detalhes do elemento *ClienteController*. Fonte: O Autor

Como é possível visualizar, na versão 7, o elemento *ClienteController* contém implementado o método *atualizaView* (previsto no modelo de arquitetura conceitual) e o método *atualizaModel* (não previsto no modelo de arquitetura conceitual), o método *atualizaDadosCliente* (previsto no modelo de arquitetura conceitual) não foi implementado. Os métodos não são apresentados no diagrama principal devido à técnica implementada “Visão Geral + Contexto”, dessa forma somente os elementos principais do diagrama são carregados, resultado de um processamento escalável (menos informações processadas) e caso deseje visualizar detalhes de um elemento específico, somente os dados referente a esses elementos são processados.

5.5 Discussão

A abordagem AESCOM, consiste na composição de mecanismos de comparação de modelos, extração de métricas e visualização de software de forma integrada. A abordagem possibilita a percepção da aderência entre o que está sendo implementado com relação ao que foi planejado (Precisão) e prove a percepção sobre a evolução da implementação (Cobertura). A partir das informações de aderência, é possível identificar possíveis problemas no processo de elaboração da arquitetura conceitual ou na sua implementação. A abordagem não possibilita a identificação da origem do problema, deixando essa atividade aos engenheiros e arquitetos de software envolvidos no projeto.

Dados sobre a evolução da arquitetura emergente são obtidos a partir de comparações de versões armazenadas em repositórios. Para isso, o código-fonte deve estar sobre o controle de versão e tanto o modelo de arquitetura conceitual quanto emergente devem estar utilizando as ontologias para caracterização dos elementos arquiteturais. No caso do modelo emergente a abordagem apoia-se em mecanismos da MDA para transformação do código fonte em modelos da UML.

O processo de comparação dos elementos que compõe os modelos considera as ontologias atribuídas ao elemento, ou seja, se ambos os elementos tiverem a mesma ontologia com os mesmo parâmetros, são iguais.

A ferramenta desenvolvida atende a uma série de requisitos, propostos por Oliveira (2011) como requisitos necessários a uma ferramenta voltada para a avaliação da evolução arquitetural. A seguir são relacionados os requisitos propostos e uma breve descrição de como são implementados pela ferramenta:

- R1. Extração automática de dados a partir de repositórios

  A abordagem possui integração com qualquer repositório que faça uso do protocolo *subversion*. Ainda é possível, dada a forma de implementação (utilizando padrões de projeto), que novos mecanismos de integração sejam adicionados sem a necessidade de alterações estruturais na ferramenta, como já foi mostrado na seção 4.5

- R2. Geração automática de visualização;

A abordagem proposta gera automaticamente a visualização de diferenças entre modelos para cada versão;

- R3 Alta escalabilidade (de processamento de visualização), em termos dos dados a serem tratados;

Utilizando a técnica Detalhamento tem-se um ganho de tempo na obtenção dos dados iniciais, visto que somente o nome da classe é obtido *a priori*, caso queira mais detalhes sobre determinado elemento, ao clicar sobre ele, informações como nome de métodos e sua existência serão então buscados do repositório;

- R4. Capacidade de dar suporte a diversos tipos de dados

A abordagem trabalha com classes, inclusive abstratas, e interfaces. Aceita também que os métodos e atributos das classes sejam de quaisquer tipos;

- R5. Meios para comparar, correlacionar e examinar dados e revisões

No que tange ao cálculo de métricas, em ambas as interfaces (gráfico e tabela de números) é possível fazer a comparação de uma mesma métricas em várias versões do software. E na visualização da evolução, por meio do esquema de cores é possível saber as diferenças entre o modelo emergente e o modelo de arquitetura conceitual;

- R6. Integração com ferramentas existentes

A ferramenta permite interação com o *Enterprise Architect*, para modelagem do modelo de arquitetura conceitual e também, como já discutido na seção 4.5, permite a extensão desse mecanismo para agregar outras plataformas de modelagem. E no que tange ao modelo emergente, qualquer código na linguagem Java, na versão 1.5 do *Java Development Kit*, pode ser analisado;

- R7. Flexibilidade e capacidade de customização

A ferramenta possui uma interface extensível para adição de novas métricas e novos mecanismos de extração de dados, tanto para o modelo emergente quanto o conceitual, discutidas na seção 4.5;

- R8. Auxílio na identificação de problemas na evolução do software

Utilizando-se das métricas de Precisão e Cobertura que por objetivo, respectivamente, calcular a aderência e a evolução do modelo de arquitetura conceitual a abordagem proposta busca prover auxílio na identificação de problemas na evolução do software.

- R9. Facilidades na monitoração da execução do projeto (planejado × realizado)

A abordagem AESCOM possibilita o acompanhamento da execução o projeto utilizando-se de dois mecanismos, a métrica de cobertura – que visa identificar o percentual implementado do modelo de arquitetura conceitual – e o mecanismo de visualização, que permite o acompanhamento visual da evolução do projeto, em relação ao planejado.

- R10. Tratamento da dimensão de tempo (i.e., dados sensíveis ao tempo)

Os mecanismos de métricas e visualização discutidos são utilizados para a análise em alguma versão do software, que está diretamente relacionada a algum instante de tempo. A visualização possui a navegação entre versões (em forma de linha do tempo) e as métricas podem ser visualizadas também em linha do tempo – quando o usuário seleciona um período de versões para cálculo.

5.6 Análise Comparativa

Na seção 3.4, foram analisadas algumas abordagens semelhantes ao presente trabalho, que proveem apoio à visualização e compreensão da evolução arquitetural. Nesta seção, é feita uma análise comparativa entre essas abordagens e a abordagem proposta.

O trabalho de Oliveira (2011) apresenta mecanismos de visualização e métricas semelhantes ao presente trabalho, porém não provê uma distinção entre elementos arquiteturais e elementos específicos de implementação – essa distinção pode ser feita somente no momento da análise pelo usuário. A abordagem AESCOM apoia-se em ontologias para caracterização dos elementos arquiteturais, fazendo desnecessária qualquer intervenção do usuário no momento da análise para

classificação dos elementos da arquitetura emergente. Além disso, a abordagem PREViA (OLIVEIRA, 2011) utiliza a comparação sintática de elementos – apontado como falho por Vasconcelos (2007) – enquanto a abordagem AESCOM faz uso das ontologias para uma comparação semântica do elementos conceituais com os emergentes.

O trabalho de Ohst et al (2003) apresentou um mecanismo para comparação de documentos UML – apresenta em um único diagrama elementos comuns entre as versões e elementos específicos de cada versão. Dessa forma, sua abordagem prevê somente a visualização da aderência entre dois modelos UML – fazendo necessário que o usuário gere o modelo UML do código fonte. A abordagem também não prove mecanismos para tratamento da dimensão tempo e visualização da evolução do projeto.

A ferramenta SAVE (KNODEL et al, 2006) não possibilita o cálculo das métricas de Precisão e Cobertura, além de não possibilitar a visualização das diferenças em relação a dimensão tempo.

O trabalho de Patel et al (2007) desenvolveu somente uma ontologia para caracterização do domínio de aplicações, não contém mecanismos para avaliação da evolução da arquitetura do software. De forma semelhante o meta-modelo HISMO (GÎRBA et al, 2005) consiste apenas em uma representação da evolução de software, não contempla mecanismos de avaliação da evolução arquitetural. A pirâmide de ontologias SEON (WÜRSCH et al, 2012), contempla todas as etapas de um evolução do software – desde a solicitação pelo usuário até a implementação desta no software – porém a abordagem não trabalha com os mecanismo de visualização da evolução e métricas.

6 Conclusão

6.1 Considerações Gerais

Este trabalho apresentou uma nova abordagem para avaliação da evolução arquitetural, utilizando ontologias para caracterização do modelo de arquitetura conceitual do software e mecanismos da MDA para transformação do modelo emergente para conceitual, de modo a possibilitar uma comparação entre o planejado e o implementado. Foram discutidas técnicas de visualização de software para compreensão da evolução do software ao longo do tempo, e percebeu-se que os trabalhos analisados (seção 3.4) não proviam todo o suporte necessário para a tarefa de compreensão da evolução arquitetural.

Para embasar a elaboração da abordagem AESCOM, a revisão da literatura abrangeu áreas de arquitetura e evolução de software (seção 3.1 e 3.2), avaliação da evolução de arquitetura de software (seção 3.3), a abordagem MDA para transformação de modelos (seção 3.3.1) e o uso de ontologias para caracterização da arquitetura de software (seção 3.3.2). A partir da revisão da literatura, necessidades para a construção da abordagem foram identificadas. Uma ferramenta, apoiada nos mecanismos propostos, foi implementada e um estudo de caso foi realizado com objetivo de verificar se a abordagem atingiu seu objetivo – prover uma percepção da evolução do projeto e da aderência entre o que foi implementado e o que foi planejado.

Para atingir os objetivos do trabalho (capítulo 2) foram definidos:

1. Um mecanismo para avaliação da evolução de arquitetura de software baseado em mecanismos de visualização e métricas. A visualização tem por base a comparação de documentos UML, aliada a técnica de detalhamento, e tem por objetivo possibilitar a compreensão da aderência e evolução do modelo emergente;
2. Um modelo para transformação do código fonte em modelos conceituais, sendo o código fonte desenvolvido na linguagem Java e o modelo de arquitetura conceitual na UML;
3. Uma ontologia para caracterização da arquitetura de software, tanto no modelo emergente como no conceitual, de forma que seja possível o

mapeamento de elementos do código fonte para elementos da arquitetura lógica; e foi realizada também a:

4. Implementação de uma ferramenta, baseada nos mecanismos definidos propostos pela abordagem do presente trabalho, para avaliação da evolução de software.

6.2 Contribuições

As contribuições desse trabalho são:

- A abordagem AESCOM, com objetivo de possibilitar a percepção da aderência entre o projetado e o implementado e a evolução do projeto, fazendo uso de métricas, técnicas de visualização e ontologias;
- Desenvolvimento de uma ontologia para caracterização de elementos arquiteturais de um software baseado na abordagem MVC;
- Adaptação das métricas de Precisão e Cobertura, fazendo seu cálculo ter como base as ontologias propostas, com objetivo de possibilitar a distinção entre elementos conceituais e específicos de implementação e a comparação dos elementos com base em sua responsabilidade (elemento do modelo, domínio ou controladora);
- Uma ferramenta, que implementa os conceitos da abordagem; e
- Todos os mecanismos desenvolvidos no presente trabalho – código-fonte da ferramenta, estudo de caso, diagramas e inclusive esta redação – encontram-se disponíveis para consulta no *Google Code*[2]. A disponibilização desse trabalho nesse meio de compartilhamento foi realizada com o intuito de que os artefatos desenvolvidos sirvam de referência para futuros trabalhos.

[2] https://code.google.com/p/aescom

6.3 Trabalhos Futuros

A abordagem proposta apoiou-se em mecanismos bem definidos de arquitetura, plataforma de desenvolvimento e ferramentas de modelagem. Propõe-se como trabalhos futuros a extensão da AESCOM para suportar a avaliação de outros tipos de arquiteturas – baseada em componentes, camadas (não exclusivamente MVC), *pipeline*, dentre outras. Além de contemplar novas arquiteturas, futuros trabalhos podem ser desenvolvidos no intuído de agregar outras plataformas de desenvolvimento e modelagem à ferramenta produzida – atualmente com suporte a plataforma *Java*, para o modelo emergente, e ao *Enterprise Architect*, para o modelo conceitual. Novas abordagens podem ser também propostas no que tange a outras formas de visualização da evolução e novas métricas. A realização de um estudo da eficiência da abordagem proposta, juntamente com a ferramenta desenvolvida, em um ambiente acadêmico e posteriormente real de desenvolvimento, baseada em métodos estatísticos, será uma continuidade do presente trabalho.

7 Referências

ALMEIDA, M; BAX, M. Uma visão geral sobre ontologias: pesquisa sobre definições, tipos, aplicações, métodos de avaliação e de construção. **Revista Ciência da Informação**, 2003.

BAEZA-YATES, R.; RIBEIRO-NETO, B., 1999, **Modern Information Retrieval**. 1 ed. Addison Wesley.

BOOCH, G.. **Software Architecture Industry Survey.** Disponível em: <http://booch.com/architecture/blog/artifacts/SoftwareArchitecture.ppt.>. Acesso em: 07 abr. 2013.

BORST, W. **Construction of Engineering Ontologies for Knowledge Sharing and Reuse**. Tese de Doutorado, University of Twente, P.O. Box 217 - 7500 AE Enschede - The Netherlands, 1997.

CARD, S. K.; MACKINLAY, J.; SHNEIDERMAN, B., 1999, **Readings in Information Visualization: Using Vision to Think**. 1 ed. Morgan Kaufmann.

CASERTA, P.; ZENDRA, O., 2010, Visualization of the Static Aspects of Software: A Survey, **IEEE Transactions on Visualization and Computer Graphics**, p. 1-20.

CEPÊDA, R. D. S. V.; MAGDALENO, A. M.; MURTA, L. G. P.; WERNER, C. M. L., 2010, "EvolTrack: Improving Design Evolution Awareness in Software Development", **Journal of the Brazilian Computer Society**, v. 16, n. 2 (Ago.), pp. 117-131.

CHEN, C. **Information Visualization: Beyond the Horizon.** 2. ed. Springer, 2006. 336 p.

CLEMENTS, P.; SHAW, M., 2009, The Golden Age of Software Architecture Revisited, **IEEE Software**, v. 26, n. 4, p. 70-72.

COCKBURN, A.; KARLSON, A.; BEDERSON, B. B., A Review of Overview+Detail, Zooming, and Focus+Context Interfaces, **ACM Computing Surveys**, New York, NY, USA, v. 41, n. 1, pp. 1-31. 1 dez. 2008.

CORBI, T. A.. Program understanding: challenge for the 1990. **IBM Systems Journal,** Riverton, Nj, Usa, p. 294-306. 1989.

DIEHL, S. **Software Visualization: Visualizing the Structure, Behaviour, and Evolution of Software.** 2007. ed: Springer, 2007. 199 p.

ECLIPSE FOUNDATION, 2010. Eclipse. Eclipse - **The Eclipse Foundation open source community website**. Disponível em: http://www.eclipse.org/. Acesso em: 12 Dez 2010.

EELES, P. **The Benefits of Software Architecting**. Disponível em: <http://www.ibm.com/developerworks/rational/library/may06/eeles/index.html>. Acesso em: 04 abr 2013.

ERLIKH, LEN. Leveraging Legacy System Dollars for E-Business. **Journal It Professional,** Piscataway, NJ, USA, p. 17-23. 3 abr. 2000.

FALBO, R. A. **Experiences in Using a Method for Building Domain Ontologies**. Proc. of International Workshop on Ontology In Action, Banff, Alberta, Canada, Jun. 2004.

FOUAD, A. **Embedding Requirements within the Model Driven Architecture.** 2011. 333 f. Tese (Doutorado) - Bournemouth University, Bournemouth, UK, 2011.

GAMMA E.; HELM R.; JOHNSON R.; VLISSIDES J. **Design Patterns: Elements of Reusable Object-oriented Software**. Addison-Wesley, 1994.

GALLAGHER, K.; HATCH, A.; MUNRO, M., 2008, Software Architecture Visualization: An Evaluation Framework and Its Application, **IEEE Transactions on Software Engineering**, v. 34, n. 2 (Mar.), p. 260-270.

GÎRBA, T.; FAVRE, J. M.; DUCASSE, S. Using Meta-Model Transformation to Model Software Evolution. **Electronic Notes In Theoretical Computer Science (ENTCS),** Amsterdam, p. 57-64. 3 set. 2005.

GRUBER, T. R. Toward principles for the design of ontologies used for knowledge sharing. **International Journal Of Human-computer Studies,** Padova, Italy, p. 907-928. 6 nov. 1993.

GUARINO, N.; WELTY, C. Evaluating ontological decisions with OntoClean. **Communications Of The Acm**: Ontology: different ways of representing the same concept, New York, NY, USA, v. 45. p.61-65, 2 fev. 2002.

GUIZZARDI, G. **Desenvolvimento para e com reuso: Um estudo de caso no domínio de vídeo sob demanda**. Dissertação de Mestrado, Universidade Federal do Espírito Santo, 2000

IEEE. **Recommended Practice for Architectural Description of Software-intensive Systems**. 2007. DisponPractice for Architectural Description of Software-intensive , USA, v. 45. p.

FALBO, R. A. Experiences in Using a Method for Building Domain Ontologies". **Proc. of International Workshop on Ontology In Action**, Banff, Alberta, Canada, June 2004

KNODEL, J.; LINDVALL, M.; MUTHIG, D.; NAAB, M. Static Evaluation of Software Architectures. Em **Proceedings of the 10th European Conference on Software Maintenance and Reengineering** (CSMR 2006), p. 279-294, Bari, Italy. 2006.

KRASNER, G.; POPE S. A cookbook for using the model-view-controller interface paradigm in smalltalk. **Journal Of Object Oriented Programming**, 1988. p. 26-49.

KÜMMEL, G. Z. **Uma Abordagem para a Criação de Arquiteturas de Referência de Domínio a partir da Comparação de Modelos Arquiteturais de Aplicações.** 2007. 128 f. Dissertação (Mestrado) - Universidade Federal do Rio de Janeiro, Rio de Janeiro, Brasil, 2007.

LANZA M.; DUCASSE, S. Understanding software evolution using a combination of software visualization and software metrics. Em: **Langages et Modèles à Objets**, 2002. Pau, França. p. 135-149.

LEHMAN, M. Laws of software evolution revisited. Em EWSPT'96: **Proceedings of the 5th European Workshop on Software Process Technology** (London, UK, 1996), Springer-Verlag, p. 108–124

LEHMAN, MEIR M; FERNÁNDEZ-RAMIL, JUAN. Software evolution. Em: MADHAVJI, N. H.; FERNÁNDEZ-RAMIL, J. C.; PERRY, D. E. **Software evolution and feedback: Theory and practice.** Wiley, 2006. p. 7-40.

MAIA, N. E. N. **ODYSSEY-MDA: UMA ABORDAGEM PARA A TRANSFORMAÇÃO DE MODELOS.** 2006. 86 f. Dissertação (Mestrado) - Universidade Federal do Rio de Janeiro, Rio de Janeiro, Brasil, 2006.

NADIR, J. S.; FALBO, R. S. Uma Ontologia de Requisitos de Software. **Congresso Ibero-Americano em Engenharia de Software 2006**: p. 111-124.

OHST, D.; WELLE, M.; KELTER, U., 2003. Differences Between Versions of UML Diagrams, **SIGSOFT Softw. Eng. Notes**, v. 28, n. 5, p. 227-236.

OLIVEIRA, M. S. **PREVIA: Uma Abordagem para a Visualização da Evolução de Modelos De Software.** 2011. 128 f. Dissertação (Mestrado) - Universidade Federal do Rio de Janeiro, Rio de Janeiro, Brasil, 2011.

OMG. **MDA Guide Version 1.0.** 2003**.** Disponível em: <http://www.omg.org/mda/mda_files/MDA_Guide_Version1-0.pdf>. Acesso em: 07 abr. 2013.

OMG. **OMG MOF 2 XMI Mapping Specification.** 2013**.** Disponível em: < http://www.omg.org/spec/XMI/2.4.1/>. Acesso em: 07 ago. 2013.

PATEL, D.; KASOVSKI, M.; PATEL, S. **An Ontological Approach to Domain Modeling for MDA-oriented Processes.** Disponível em: <http://dspace3.mak.ac.ug/xmlui/handle/10570/710>. Acesso em: 07 abr. 2013.

SEBESTA, R. **Concepts of Programming Languages**. Addison-Wesley, 2012. 816.

SHAW, M.; GARLAN, D. **Software Architecture: Perspectives on an Emerging Discipline**. Prentice-Hall, Inc, 1996. 242p.

STANDISH, T. A.. An Essay on Software Reuse. **IEEE Transactions On Software Engineering,** Piscataway, Nj, Usa, p. 494-497. 03 abr. 1984.

TAYLOR, R. N.; MEDVIDOVIC, N.; DASHOFY, E. M. **Software Architecture: Foundations, Theory, and Practice.** Wiley, 2009. 750 p.

VASCONCELOS, A. P. V. **Uma Abordagem de Apoio à Criação de Arquiteturas de Referência de Domínio Baseada na Análise de Sistemas Legados.** 2007. 264 f. Tese (Doutorado) - Universidade Federal do Rio de Janeiro, Rio de Janeiro, Brasil, 2007.

VOGEL, O.; ARNOLD, I.; CHUGHTAI, A.; KEHRER T., 2009, **Software Achitecture: A Comprehensive Framework and Guide for Practitioners**. 1 ed. Springer.

VOINEA, STEFAN-LUCIAN. **Software Evolution Visualization.** 2007. 181 f. Tese (Doutorado) - Technische Universiteit Eindhoven, Eindhoven, 2007.

VÖLTER, M. Software Architecture Documentation in the Real World (Tutorial). Em **22º Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)**, Montreal, Canada, 2007.

WÜRSCH, M.; GHEZZI J.; HERT M.; REIF G. GALL H.C. SEON: a pyramid of ontologies for software evolution and its applications. **Computing,** Dagstuhl, Germany, p. 857-885. 11 nov. 2012.

Apêndice A

O pseudocódigo 3 apresenta o algoritmo para cálculo da métrica de Precisão.

```
Precisão(modeloConceitual, modeloEmergente)
1.   numeroDeElementosConceituaisImplementados ← 0
2.   numeroDeElementosImplementados  ← 0
3.   numeroDeTotalElementosImplementados  ← 0
4.   numeroDeElementosEspecificosDeImplementacao  ← 0
5.   PARA-CADA elemento EM modeloConceitual FAÇA
6.     SE contémOntologia(elemento) ENTÃO
7.       SE modeloEmergente.contém(elemento) ENTÃO
8.         numeroDeElementosConceituaisImplementados++;
9.         PARA-CADA metodo EM elemento.metodos FAÇA
10.          SE contémOntologia(metodo) ENTÃO
11.            SE modeloEmergente.contém(metodo) ENTÃO
12.              numeroDeElementosConceituaisImplementados++;
13.            FIM-SE;
14.          FIM-SE;
15.        FIM-PARA;
16.      FIM-SE;
17.    FIM-SE;
18.  FIM-PARA;
19.  PARA-CADA elemento EM modeloEmergente FAÇA
20.    numeroDeTotalElementosImplementados++;
21.    SE nãoContémOntologia(elemento) ENTÃO
22.      numeroDeElementosEspecificosDeImplementacao++;
23.    FIM-SE;
24.    PARA-CADA metodo EM elemento.metodos FAÇA
25.      numeroDeTotalElementosImplementados++;
26.        SE nãoContémOntologia(metodo) ENTÃO
27.          numeroDeElementosEspecificosDeImplementacao++;
28.        FIM-SE;
29.    FIM-PARA;
30.  FIM-PARA;
31.  numeroDeElementosImplementados =
32.     numeroDeTotalElementosImplementados -
33.     numeroDeElementosEspecificosDeImplementacao;
34.  Precisão = numeroDeElementosImplementados /
35.             numeroDeElementosConceituaisImplementados;
36.FIM.
```

Pseudocódigo 3 – Cálculo da métrica de Precisão

Apêndice B

O pseudocódigo 4 apresenta o algoritmo para cálculo da métrica de Cobertura.

```
Cobertura(modeloConceitual, modeloEmergente)
1.   numeroDeElementosConceituais ← 0
2.   numeroDeElementosConceituaisImplementados  ← 0
3.   PARA-CADA elemento EM modeloConceitual FAÇA
4.     SE contémOntologia(elemento) ENTÃO
5.       numeroDeElementosConceituais++;
6.       SE modeloEmergente.contém(elemento) ENTÃO
7.         numeroDeElementosConceituaisImplementados++;
8.         PARA-CADA metodo EM elemento.metodos FAÇA
9.           SE contémOntologia(metodo) ENTÃO
10.            numeroDeElementosConceituais++;
11.            SE modeloEmergente.contém(metodo) ENTÃO
12.              numeroDeElementosConceituaisImplementados++;
13.            FIM-SE;
14.          FIM-SE;
15.        FIM-PARA;
16.      FIM-SE;
17.    FIM-SE;
18.  FIM-PARA;
19.  Cobertura = numeroDeElementosConceituaisImplementados /
20.              numeroDeElementosConceituais;
21.FIM.
```

Pseudocódigo 4 – Cálculo da métrica de Cobertura

Apêndice C

O seguinte diagrama de classe foi feito utilizando a ferramenta *Enterprise Achitect* (*Trial*), representado pela Figura 40.

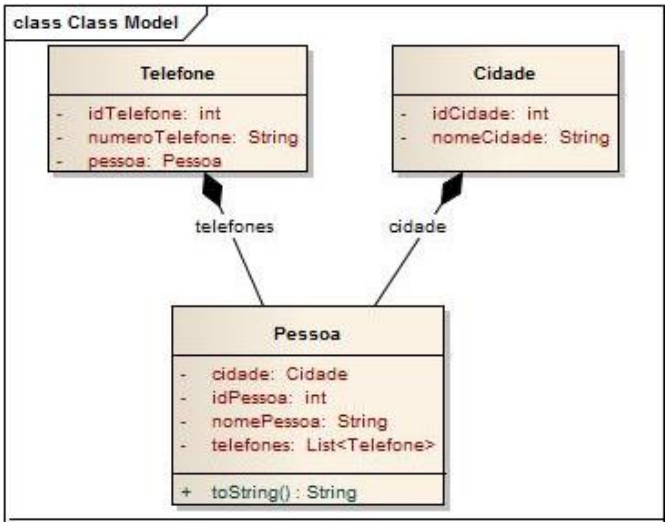

Figura 40 - Diagrama de classe feito no Enterprise Achitect. Fonte: O Autor

O diagrama representado na Figura 36, apresenta três classes (Pessoa, Cidade e Telefone) com composições entre elas. O *Enterprise Architect* permite que seja gerado o XMI de qualquer diagrama. Após gerado o XMI do diagrama, este foi importado para dentro da ferramenta *StarUML*, o resultado foi o seguinte diagrama, representado pela Figura 41:

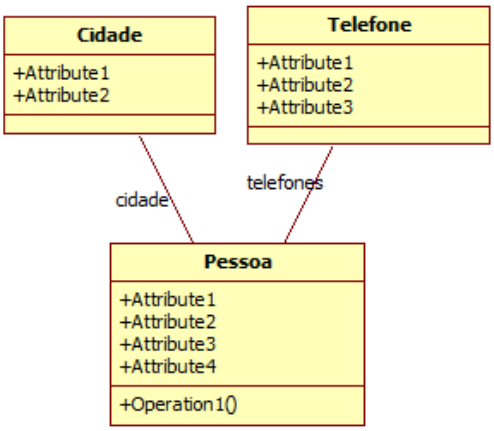

Figura 41 - Diagrama de classe importado no StarUML. Fonte: O Autor

Como pode-se observar, o resultado é diferente do especificado na plataforma Enterprise Architect. O nome dos atributos, seus tipos e visibilidade não foram exportados, o mesmo aconteceu para os métodos.