UNIVERSIDADE FEDERAL DE ALAGOAS
INSTITUTO DE COMPUTAÇÃO
PROGRAMA DE PÓS GRADUAÇÃO EM INFORMÁTICA

ITALO CARLO LOPES SILVA

Uma Solução para apoiar Processos de Desenvolvimento de Software Centrado na Arquitetura

Maceió
2014

Italo Carlo Lopes Silva

Uma Solução para apoiar Processos de Desenvolvimento de Software Centrado na Arquitetura

Dissertação apresentada como requisito parcial para obtenção do grau de Mestre pelo Programa de Pós-Graduação em Informática do Instituto de Computação da Universidade Federal de Alagoas.

Orientador: Prof. Dr. Patrick Henrique da Silva Brito

Coorientador: Prof. Dr. Baldoino Fonseca dos Santos Neto

Maceió
2014

Catalogação na fonte
Universidade Federal de Alagoas
Biblioteca Central
Divisão de Tratamento Técnico
Bibliotecário: Valter dos Santos Andrade

S586s Silva, Italo Carlo Lopes.
Uma solução para apoiar processo de desenvolvimentos de software centrado na arquitetura / Italo Carlo Lopes Silva. – Maceió.
78 f. : il.

Orientador: Patrick Henrique da Silva Brito.
Dissertação (Mestrado em Informática) – Universidade Federal de Alagoas. Instituto de Computação. Programa de Pós-Graduação em Informática. Maceió, 2014.

Bibliografia: f. 75-78.

1. Arquitetura de software. 2. Engenharia de requisitos. 3. Resolução de Tred-off. 4. Assistente baseado em regras. 5. Rastreabilidade. 6. Software – Desenvolvimento. I. Título.

CDU: 004.41

UNIVERSIDADE FEDERAL DE ALAGOAS/UFAL
Programa de Pós-Graduação em Informática – PpgI
Instituto de Computação
Campus A. C. Simões BR 104-Norte Km 14 BL 12 Tabuleiro do Martins
Maceió/AL - Brasil CEP: 57.072-970 | Telefone: (082) 3214-1401

Membros da Comissão Julgadora da Dissertação de Mestrado de Ítalo Carlo Lopes Silva, intitulada: "*Uma Solução para apoiar Processos de Desenvolvimento de Software Centrado na Arquitetura*", apresentada ao Programa de Pós-Graduação em Informática da Universidade Federal de Alagoas em 30 de dezembro de 2014, às 10h00min, no Miniauditório do Instituto de Computação da UFAL.

COMISSÃO JULGADORA

Prof. Dr. Patrick Henrique da Silva Brito
UFAL – Instituto de Computação
Orientador

Prof. Dr. Baldoino Fonseca dos Santos Neto
UFAL – Instituto de Computação
Orientador

Prof. Dr. Evandro de Barros Costa
UFAL – Instituto de Computação
Examinador

Prof. Dr. Hyggo Oliveira de Almeida
UFCG – Universidade Federal de Campina Grande
Examinador

AGRADECIMENTOS

Agradeço a Deus primeiramente pelo dom da vida e por tudo que tem me proporcionado durante todo esse tempo. Aos meus pais João de Deus e Maria Bernadete, eternos incentivadores, pelo apoio incondicional e por mostrar sempre a importância que o estudo tem na formação profissional e, principalmente, do caráter do ser humano. Agradeço também aos meus irmãos Túlio Fernando e Lígia Carla, sempre participativos e incentivadores durante toda a minha vida acadêmica. Ao amor da minha vida e futura esposa Mayara Ingred, pela paciência, incentivo, compreensão, carinho e com o sorriso que lhe é peculiar, sempre trazendo alegria para os meus dias.

Não poderia deixar de agradecer ao meu Orientador Prof. Dr. Patrick Henrique, pessoa muito importante para a realização deste sonho, sendo solicito e paciente com as minhas indagações, sempre apontando o melhor caminho para a resolução dos problemas. Ao também Orientador Prof. Dr. Baldoino Fonseca, um agradecimento especial pela sua contribuição, orientação e apoio incondicional durante todo o período do mestrado. Gostaria de agradecer também aos membros da Banca Avaliadora Prof. Dr. Evandro Costa e Prof. Dr. Hyggo Almeida pelas contribuições valiosas durante a defesa do mestrado.

Agradeço também a todos os meus amigos e parentes pelo incentivo dado durante esta difícil caminhada. E por fim, um agradecimento à Universidade Federal de Alagoas (*Campus* Arapiraca), em especial a todos que fazem o Núcleo de Tecnologia da Informação (NTI), que foram fundamentais para a concretização deste sonho.

RESUMO

O sucesso de um projeto de software está fortemente relacionado com o projeto arquitetural. No entanto, projetar a arquitetura de software correta é uma tarefa muito subjetiva e leva muito tempo, sendo muito influenciada pela experiência do arquiteto e a qualidade da engenharia de requisitos. Este conhecimento arquitetural, geralmente, não está documentado, uma vez que é considerado o conhecimento tácito dos arquitetos ou dos interessados, e, eventualmente, se dissipa. Também é essencialmente importante assegurar a consistência entre a arquitetura de software e a implementação. No entanto, esse mapeamento é feito manualmente na maioria das vezes, baseado apenas no entendimento do desenvolvedor sobre a arquitetura, exigindo disciplina por parte dele. Assim, erros podem surgir durante esta fase, comprometendo a consistência entre as decisões arquiteturais e o código fonte. Em face destas dificuldades, foi desenvolvido este trabalho, cujo o objetivo é apresentar uma ferramenta que apoie jovens arquitetos com a recomendação de um estilo arquitetural adequado, baseado nos requisitos do sistema, particularmente os atributos de qualidade do sistema. A ferramenta compreende tanto resolução trade-off sobre os atributos de qualidade e recomendação de estilos arquiteturais com base em atributos de qualidade. Por fim, com base na arquitetura recomendada, a ferramenta irá gerar o código estrutural do sistema, utilizando um modelo de implementação de componente chamado COSMOS*, proporcionando rastreabilidade entre projeto arquitetural e a implementação. A solução proposta foi avaliada no contexto de um domínio específico dos Ambientes Virtuais e Aprendizagem (AVA), a fim de ilustrar o suporte da ferramenta na execução de um processo de projeto arquitetural.

Palavras-chaves: Arquitetura de Software. Engenharia de Requisitos. Resolução de Trade-off. Rastreabilidade entre arquitetura e código. Assistente Baseado em Regras.

ABSTRACT

The success of a software project is strongly related with architectural design. However, designing the right Software Architecture is a very subjective task and takes a long time, being much influenced by architect's experience and the quality of requirements engineering. This architectural knowledge, usually, is not documented, since it is considered tacit knowledge of architects or other stakeholders, and eventually dissipates. It is also essentially important to ensure the consistency between software architecture and implementation. However, this mapping is usually made manually, based only on the developer's understanding over the software architecture, which requires high discipline. Thus, errors can arise during this phase, compromising the consistency amongst architectural decisions and source code. The objective of this work is to present a tool-based solution that supports young architects by recommending a suitable architectural style, based on the system's requirements, particularly the quality attributes of the system. The tool encompasses both trade-off resolution over quality attributes and recommendation of architectural styles based on quality attributes. Finally, based on the recommended architecture, the tool will generate the system structural source-code, using a component implementation model called COSMOS*, providing traceability between architectural design and implementation. The proposed solution has been evaluated in the context of a specific domain of Learning Management System (LMS), in order to illustrate the tool support in the execution of an architectural design process.

Keywords: Software Architecture. Architectural Decisions. Requirements Engineering. Trade-off Resolution. Traceability between software architecture and source-code. Rule-Based Assistant.

LISTA DE ILUSTRAÇÕES

Figura 1 – Notação de um Componente no padrão UML 25
Figura 2 – Representação do Componente no modelo COSMOS* 28
Figura 3 – Exemplo da Estrutura interna de um Componente no Modelo COSMOS* . . 29
Figura 4 – Exemplo Interface IManager 29
Figura 5 – Exemplo da Implementação da Classe ComponentFactory 31
Figura 6 – Exemplo Classe Implementada 31
Figura 7 – Exemplo Classe Facade 32
Figura 8 – Visão Geral da Ferramenta Proposta 39
Figura 9 – Arquitetura da Ferramenta Proposta 41
Figura 10 – Tela Principal da Ferramenta 43
Figura 11 – Atividade Preparar Ambiente 46
Figura 12 – Cadastro de um Atributo de Qualidade 47
Figura 13 – Cenário relacionado a um Atributo de Qualidade 47
Figura 14 – Cadastro de Padrão de *Trade-off* 48
Figura 15 – Meta Modelo para Representação de Regras 50
Figura 16 – Exemplo de Regra de Trade-off 50
Figura 17 – Atividade Especificar os Requisitos de Qualidade 52
Figura 18 – Exemplo da Coleta dos Requisitos de Qualidade 53
Figura 19 – Exemplo da Interação para resolução de *Trade-off* 54
Figura 20 – Atividade Projetar a Arquitetura de Software 56
Figura 21 – Atividade Gerar o Código-Fonte Estrutural do Sistema 60
Figura 22 – Componente COSMOS e suas classes 61
Figura 23 – Conector COSMOS* e suas Classes 63
Figura 24 – Exemplo de Classe do Configurador do Sistema 64
Figura 25 – Exemplo de dois estilos arquiteturais armazenados na base de conhecimento 68
Figura 26 – Fórmula para calcular a relevância de um estilo arquitetural 69
Figura 27 – Resultado do Processo de Recomendação 70
Figura 28 – Diagrama de Sequência da Arquitetura (a) 71
Figura 29 – Diagrama de Sequência da Arquitetura (b) 72

LISTA DE TABELAS

Tabela 1 – Potencial Relacionamento entre os Atributos de Qualidade 66

SUMÁRIO

1 INTRODUÇÃO 11
1.1 Problemática e Motivação da Pesquisa 11
1.2 Objetivo da Proposta 14
1.2.1 Objetivo Geral 15
1.2.2 Objetivos Específicos 16
1.3 Relevância 16
1.4 Estrutura da Dissertação 17

2 FUNDAMENTAÇÃO TEÓRICA 18
2.1 Requisitos de Qualidade 18
2.2 Arquitetura de Software 19
2.2.1 Estilos e Padrões Arquiteturais 21
2.2.2 Avaliação Arquitetural 22
2.3 Sistemas de Recomendação 22
2.4 Desenvolvimento Baseado em Componentes (DBC) 23
2.4.1 Componentes de Software 24
2.5 Processos de Desenvolvimento Baseado em Componentes 25
2.6 Desenvolvimento Centrado na Arquitetura 26
2.7 O Modelo de Implementação de Componentes COSMOS* 27
2.7.1 Modelo de Especificação 28
2.7.2 Modelo de Implementação 30
2.7.3 Modelo de Conector 32

3 TRABALHOS RELACIONADOS 34
3.1 Resolução de *Trade-offs* 34
3.2 Decisões Arquiteturais 35
3.3 Desenvolvimento Centrado na Arquitetura 36

4 UMA SOLUÇÃO PARA APOIO A PROCESSOS DE DESENVOLVIMENTO DE SOFTWARE CENTRADO NA ARQUITETURA . 38
4.1 Visão Geral do Processo 38
4.2 Arquitetura da Ferramenta Proposta 40
4.2.1 Especificação dos Componentes 42
4.3 Visão Geral da Implementação 43

5 PREPARAR O AMBIENTE (ATIVIDADE 1) 45
5.1 Registrar os Atributos de Qualidade (Atividade 1.1) 46

5.2 Registrar os Padrões de *Trade-off* (Atividade 1.2) 47
5.3 Registrar Regras para Priorizar Requisitos (Atividade 1.3) 48
5.4 Registrar Arquiteturas de Referência (Atividade 1.4) 49
5.5 Registrar as Regras para Calcular a Relevância Arquitetural (Atividade 1.5) 51

6 ESPECIFICAR OS REQUISITOS DE QUALIDADE (ATIVIDADE 2) 52
6.1 Obter os Requisitos de Qualidade (Atividade 2.1) 52
6.2 Entender os Requisitos (Atividade 2.2) 53
6.3 Gerenciar os *Trade-offs* (Atividade 2.3) 54

7 PROJETAR A ARQUITETURA DE SOFTWARE (ATIVIDADE 3) 55
7.1 Recomendar a Arquitetura de Referência baseada nos Atributos de Qualidade (Atividade 3.1) 55
7.2 Refinar a Arquitetura de Software de Referência (Atividade 3.2) . 57
7.3 Avaliar a Arquitetura de Software (Atividade 3.3) 57
7.4 Atualizar Base de Conhecimento de Arquiteturas de Referência (Atividade 3.4) 57

8 GERAR O CÓDIGO-FONTE ESTRUTURAL DO SISTEMA (ATIVIDADE 4) 59
8.1 Gerar o Código dos Componentes (Atividade 4.1) 59
8.2 Gerar o Código dos Conectores (Atividade 4.2) 62
8.3 Gerar o Código da Configuração Arquitetural (Atividade 4.3) . . . 63

9 AVALIAÇÃO DA SOLUÇÃO 65
9.1 Configuração 65
9.1.1 Planejamento 65
9.1.2 Seleção dos Sujeitos 66
9.1.3 Instrumentação 66
9.1.4 Operação 67
9.2 Resultados da Avaliação e Discussões 69
9.2.1 O assistente de elucidação de requisitos de fato melhorou a identificação correta dos requisitos pretendidos pelo interessado, se comparado ao processo manual? 69
9.2.2 O assistente de elucidação de requisitos teve um desempenho melhor na identificação dos potenciais trade-offs entre os requisitos, se comparado ao processo manual? 70
9.2.3 A recomendação da arquitetura realmente reflete os desejos (requisitos) pretendidos pelos interessados? 70
9.2.4 O código gerado atende as especificações do modelo Cosmos* utilizado? . . 75

9.2.5 O código gerado é consistente com a arquitetura de software recomendada? 75
9.2.6 A solução ajuda a estruturar e padronizar a representação do Conhecimento Arquitetural? 75
9.2.7 A solução ajuda a reduzir a Vaporização do Conhecimento Arquitetural? . . 76

10 CONCLUSÃO 77

REFERÊNCIAS 79

1 INTRODUÇÃO

1.1 Problemática e Motivação da Pesquisa

A crescente demanda por sistemas maiores e complexos tornou necessária, dentro da Engenharia de Software, a modularização do sistema em blocos abstratos denominados componentes. Além da modularização era preciso entender como estes componentes, de um sistema, estão organizados e relacionados entre si. Com o propósito de lidar com estas questões, a Arquitetura de Software surge como um elemento crítico de sucesso para os projetos de software. Ela define, em um nível alto de abstração, os componentes, suas funcionalidades e a relação entre eles materializados explicitamente através dos conectores. Assim, a Arquitetura de Software tem sido considerada cada vez mais importante como meio de entendimento e gerenciamento do desenvolvimento de sistemas complexos (FALESSI et al., 2011; SHAHIN; LIANG; LI, 2014).

Apesar da arquitetura ser projetada durante as fases iniciais do processo de desenvolvimento ela exerce influência durante todo o ciclo de desenvolvimento do sistema. Por ser uma atividade de conhecimento intensivo, grande quantidade de conhecimento é continuamente produzido e consumido.

Tal conhecimento, denominado conhecimento arquitetural, pode variar de acordo com o uso e importância. Na literatura quatro tipo de visões de conhecimento arquitetural são destacadas (FARENHORST; BOER, 2009): visão centrada em padrões [1] , visão centrada no dinamismo [2], visão centrada nos requisitos [3] e visão centrada nas decisões [4].

Dada a importância da arquitetura de software durante o projeto arquitetural, o arquiteto necessita tomar decisões importantes. Tais decisões podem facilitar ou dificultar a realização de requisitos funcionais, requisitos não-funcionais (requisitos de qualidade) e objetivos de negócio. Todo sistema possui uma arquitetura, seja ela implícita ou explícita, ou seja, documentada e especificamente projetada para atender a objetivos de negócios pré-definidos e requisitos de qualidade (FALESSI et al., 2011). Para que a arquitetura realize este requisito não-funcional, pode ser necessária a utilização de estilos arquiteturais, a fim de garantir a preservação desta propriedade durante o desenvolvimento. Um estilo arquitetural permite que sistemas que pertencem ao mesmo domínio de aplicação possam compartilhar as mesmas propriedades estruturais e semânticas.

[1] do inglês *pattern-centric*
[2] do inglês *dynamism-centric*
[3] do inglês *requirements-centric*
[4] do inglês *decision-centric*

Utilizando como base os requisitos funcionais e de qualidade, métodos são aplicados com o intuito de derivar a arquitetura do software. A atividade de projeto arquitetural pode ser guiada por três fatores principais (FALESSI et al., 2011): reuso, método e intuição. O nível de utilização de cada uma destas fontes vai depender da experiência do arquiteto, da sua formação e do caráter inovador do sistema. O reuso pode facilitar a árdua tarefa de projetar a arquitetura do sistema utilizando para isto soluções já previamente testadas e aprovadas pela comunidade. Existem diversas linguagens, métodos e modelos de processo que tentam reduzir a distância entre os requisitos e a arquitetura do sistema.

A relação entre a arquitetura de software e os atributos de qualidade é considerada bem próxima, visto que as decisões arquiteturais podem influenciar diretamente a realização dos requisitos não-funcionais. Alguns autores declaram que os *trade-offs* entre os atributos de qualidade são melhores gerenciados quando a contextualização dos impactos destes *trade-offs* sobre as decisões de projeto está relacionada à fase de projeto arquitetural (HENNINGSSON; C.WOHLIN, 2002).

Entretanto, um problema aparece quando é necessário projetar a arquitetura de software. Por ser uma tarefa muito subjetiva e demandar muito tempo, o resultado do projeto arquitetural acaba sendo muito influenciado pela experiência do arquiteto, assim como pela qualidade do levantamento de requisitos.

Cada alternativa de solução satisfaz diferentes requisitos funcionais e não-funcionais. Escolher uma solução entre as opções disponíveis envolve gerenciar *trade-offs* entre os requisitos, com relação às preferências dos interessados[5] e consequências das alternativas sobre os requisitos (ELAHI; YU, 2011).

Outro problema surge durante a fase de projeto arquitetural, quando os requisitos de qualidade não são apoiados de maneira satisfatória pela arquitetura do sistema. Isto acontece porque os requisitos são difíceis de especificar em um modelo arquitetural e os mesmos são frequentemente obtidos informalmente concomitantemente à especificação da arquitetura de software (GRÜNBACHER; EGYED; MEDVIDOVIC, 2004). Em consequência disto, os *trade-offs* entre os requisitos de qualidade não são identificados e gerenciados adequadamente durante o processo de engenharia de requisitos e as consequências só são percebidas tardiamente (SILVA et al., 2014).

Além disto, projetar a arquitetura de um software com necessidades específicas é uma tarefa difícil e muito custosa. Um dos grandes desafios para o arquiteto de software é definir a arquitetura adequada, que atenda os requisitos de qualidade esperados pelos interessados (SILVA et al., 2014). Diversos fatores têm influenciado este processo como:

1. O mercado tem sido cada vez mais exigente com projetos de qualidade e que sejam realizados em tempos cada vez menores;

[5] do inglês *stakeholder*

2. Dificuldade em conseguir identificar corretamente todos os requisitos junto aos interessados;

3. Para atender os requisitos esperados, o arquiteto necessita mesclar estilos arquiteturais diferentes, aumentando assim a sua complexidade;

4. Fazer com que a arquitetura projetada atenda de maneira satisfatória os requisitos de qualidade.

Apesar dos avanços já alcançados na academia e na indústria no tocante à preocupação sistemática com o projeto da arquitetura de software, alguns problemas ainda permanecem em aberto. Dois desses problemas são a **Representação do Conhecimento** [6] (CHE, 2013) e a **Vaporização do Conhecimento Arquitetural** [7] (SHAHIN; LIANG; LI, 2014; CHE, 2013). Tais problemas acontecem porque a maioria das decisões arquiteturais não são documentadas e não podem ser explicitamente derivadas de modelos arquiteturais. Este conhecimento normalmente existe somente na cabeça do arquiteto ou de outros interessados tendendo a se perder com o passar do tempo.

Com a vaporização do conhecimento arquitetural, outros problemas podem emergir, tais como os problemas relatados pela indústria de software (SHAHIN; LIANG; LI, 2014; CHE, 2013): (1) alto custo na evolução do sistema, (2) comprometimento na comunicação entre os interessados e (3) dificuldade de reutilização. Além disso, os arquitetos de software necessitam de um processo confiável e rigoroso para selecionar alternativas arquiteturais e garantir que as decisões que forem feitas sejam boas o suficiente em termos de redução do risco, tempo de desenvolvimento e satisfação dos requisitos de qualidade.

Uma vez que as atividades de desenvolvimento de software estão fortemente interligadas e inter-relacionadas com as atividades de engenharia de requisitos e projeto arquitetural (LUCENA et al., 2011; WOODS; ROZANSKI, 2010), outros problemas podem surgir durante a fase de implementação do sistema. É essencialmente importante garantir a consistência entre a implementação do sistema e sua arquitetura durante o ciclo de desenvolvimento, porém nem sempre ela é alcançada. Isto porque o mapeamento entre a arquitetura de software e o código fonte da aplicação não é uma tarefa direta, feito na maioria das vezes de maneira manual baseado no entendimento do programador sobre a arquitetura (ZHENG; TAYLOR, 2012), exigindo assim disciplina por parte dele. Uma das razões que dificultam este processo é a incompatibilidade de paradigma entre as abstrações do projeto arquitetural e as abstrações da implementação (WOODS; ROZANSKI, 2010; GAYARD; RUBIRA; GUERRA, 2008). Embora, a arquitetura e o projeto de um sistema sejam normalmente representados em uma linguagem semi-formal, como a UML, é difícil saber o quão perto a implementação corresponde à intenção do projeto arquitetural (WOODS; ROZANSKI, 2010). Assim, recuperar a arquitetura a

[6] do inglês *Knowledge Representation*
[7] do inglês *Architectural Knowledge Vaporization*

partir do código ou até mesmo assegurar a consistência entre arquitetura de software e código fonte torna-se um problema, sendo consequência da dificuldade de rastreabilidade entre abstrações arquiteturais e a linguagem de programação. Esta falta de informação arquitetural na implementação do sistema significa que muitas das decisões arquiteturais de um sistema podem ser ignoradas ou até desrespeitadas durante a implementação do sistema (WOODS; ROZANSKI, 2010).

O desenvolvimento de um sistema que visa reduzir a distância entre o projeto arquitetural e a implementação do sistema deve manter a consistência entre a arquitetura e o código, obedecendo na prática os requisitos de qualidade especificados pelos interessados. Sendo assim, o desenvolvimento centrado na Arquitetura surge como um elemento importante para garantir a materialização das decisões arquiteturais em código fonte aumentando assim a sua rastreabilidade.

A facilidade na especificação dos requisitos não-funcionais, melhoria na manutenção preventiva e corretiva, facilidade para entendimento do sistema e melhoria no reuso são algumas das vantagens da adoção de um processo de desenvolvimento centrado na arquitetura (BRITO et al., 2005).

Porém, realizar o desenvolvimento centrado na arquitetura objetivando garantir tal consistência leva mais tempo que a codificação usual e deixar que isto seja feito completamente pelo programador(manualmente) aumenta a possibilidade de ocorrência de eventuais ruídos no sistema, fazendo com que as decisões arquiteturais projetadas a partir dos requisitos de qualidade do sistema não sejam obedecidas plenamente.

1.2 Objetivo da Proposta

O objetivo deste trabalho é apresentar uma abordagem semi-automatizada, centrada na arquitetura, para auxiliar os jovens engenheiros e arquitetos durante o processo de desenvolvimento de software, focada, principalmente, no gerenciamento das decisões arquiteturais relacionadas à fase de projeto arquitetural. Tais decisões levarão em consideração os desejos dos interessados, capturados durante o processo de engenharia de requisitos. E por fim, as decisões arquiteturais serão mapeadas para a implementação do sistema através da utilização de um modelo de desenvolvimento centrado na arquitetura.

A solução proposta apresenta mecanismos para permitir a realização da captura e representação do conhecimento relativo as decisões arquiteturais de diversas fontes, tais como: experiência do Engenheiro e Arquiteto de Software, da literatura especializada e casos de sucesso já testados. Este conhecimento poderá ser melhorado / evoluído a qualquer momento, fazendo com que a ferramenta passe a ter uma maior eficácia na execução das suas atividades. Uma visão centrada nos requisitos (FARENHORST; BOER, 2009), focada especificamente nos requisitos de qualidade, foi adotada para realizar a captura e representação destas decisões arquiteturais.

Podemos considerar que tal solução pode ser encarada como um instrumento de aprendizagem, em que através do conhecimento retido jovens engenheiros e arquitetos aprimorariam suas habilidades à medida que as soluções para os problemas enfrentados fossem apresentadas, juntamente com as respectivas explicações relacionadas às decisões arquiteturais.

A ferramenta possui uma arquitetura baseada em regras que utiliza um motor de inferência (sistema especialista) para apoiar as atividades de engenharia de requisitos e recomendação arquitetural. As regras podem ser vistas como um repositório do conhecimento que contém soluções técnicas baseadas em padrões e experiências anteriores dos especialistas.

O enfoque principal da solução está em sugerir uma arquitetura adequada, dentro de um domínio previamente especificado, para o interessado. Para tal, é importante realizar a captura destes interesses de forma eficaz, afim de utilizá-los durante o processo de recomendação, uma vez que a qualidade da recomendação está fortemente relacionada com a qualidade do levantamento de requisitos. Esta recomendação poderá chegar em um nível de granularidade tal, que poderá ser capaz de sugerir componentes, interfaces e os métodos destas interfaces. O nível de granularidade e a qualidade desta recomendação estão fortemente relacionadas com o conhecimento armazenado no momento da execução do processo.

Uma vez que o repositório de conhecimento pode evoluir, a solução proposta é capaz de recomendar diferentes arquiteturas de software, incluindo estilos arquiteturais heterogêneos.

Vale ressaltar que algumas destas etapas anteriores tais como a sugestão de métodos serão feitas de maneira semi-automática, ou seja, em algum momento o sistema precisará ser alimentado com estas informações pelo arquiteto para que durante a execução da ferramenta ela possa realizar a sugestão do estilo arquitetural e dos elementos associados ao estilo de maneira correta. E por fim, com base na arquitetura recomendada, a ferramenta gerará o código estrutural do sistema utilizando um modelo de implementação de componentes chamado COSMOS* (SILVA-JUNIOR, 2003; GAYARD; RUBIRA; GUERRA, 2008). O modelo COSMOS* faz uso de características de linguagem de programação e padrões de projetos bem conhecidos para representar arquiteturas de software explicitas no código fonte, proporcionando assim rastreabilidade entre o projeto arquitetural e sua implementação (SILVA-JUNIOR, 2003; GAYARD; RUBIRA; GUERRA, 2008).

1.2.1 Objetivo Geral

Esse trabalho tem como objetivo principal:

1. Desenvolver uma solução semi-automática para auxiliar o processo de construção de sistemas centrados na arquitetura, de forma a evitar a vaporização do conheci-

mento arquitetural e possibilitar a reutilização desse conhecimento por arquitetos de software menos experientes.

1.2.2 Objetivos Específicos

Os objetivos específicos deste trabalho são:

1. Promover o desenvolvimento centrado na arquitetura através de uma abordagem dirigida por processo, que sistematize a integração entre as fases de elicitação de requisitos, projeto arquitetural e implementação;

2. Apoiar a atividade de engenharia de requisitos, em particular o processo de elicitação dos requisitos de qualidade dos interessados, através da identificação e gerenciamento dos *trade-offs* entre estes requisitos capturados;

3. Capturar e representar o conhecimento referente às decisões arquiteturais, ajudando na organização e reduzindo a vaporização do conhecimento arquitetural;

4. Apoiar a atividade de projeto arquitetural, realizando a recomendação da arquitetura do sistema para os interessados, com base nas preferências previamente coletadas e no conhecimento (sobre decisões arquiteturais) capturado e armazenado na base de conhecimento.

5. Apoiar a geração do código fonte do sistema, utilizando um modelo de implementação centrado na arquitetura para garantir a aderência e reduzir a distância entre a arquitetura recomendada e o código do sistema que será gerado.

1.3 Relevância

As contribuições desta pesquisa se situam na área de engenharia de software associadas as técnicas de resolução de *trade-offs*, recomendação de arquitetura de software baseada nos requisitos de qualidade e desenvolvimento centrado na arquitetura, abrangendo a utilização integrada de mecanismos para elicitação dos requisitos de qualidade, projeto arquitetural de acordo com os requisitos de qualidade desejados e rastreabilidade entre o código gerado e a arquitetura de software. Mais especificamente, este trabalho abrange a criação de uma ferramenta que apoie processos de desenvolvimento de software centrados na arquitetura.

Outras contribuições são percebidas também no que se refere a representação do conhecimento, pois faz com que as informações armazenadas estejam organizadas, facilitando a consulta e utilização efetiva para os processos de gerenciamento de requisitos e recomendação arquitetural. Podemos também considerar a redução da vaporização do conhecimento arquitetural, pois as decisões arquiteturais relativas ao domínio estarão registradas na base de conhecimento da solução.

Além de, agilizar o processo de desenvolvimento do software, a ferramenta também se propõe a aumentar a consistência entre a arquitetura e o código, através da automatização de parte do processo de geração de código estrutural, uma vez que as falhas originadas durante o processo manual de mapeamento entre arquitetura e código, processo este realizado pelo programador, serão mitigadas. Devido ao baixo acoplamento entre os componentes através do desenvolvimento centrado na arquitetura e da rastreabilidade entre arquitetura de software e código fonte, a proposta traz ganhos também para a evolução do software, em especial a evolução dos requisitos de qualidade. Por exemplo, a necessidade de se adicionar conectores para distribuição de carga, com o objetivo de aumentar a escalabilidade do sistema.

1.4 Estrutura da Dissertação

Os capítulos da dissertação estão organizados como se seguem:

O Capítulo 2 apresenta os conceitos sobre Arquitetura de Software, Requisitos de Qualidade, Sistemas de Recomendação e Desenvolvimento Centrado na Arquitetura. O Capítulo 3 apresenta os trabalhos relacionados a cada uma das áreas do trabalho que são: gerenciamento e resolução de *trade-offs* entre requisitos de qualidade, projeto arquitetural e desenvolvimento centrado na arquitetura. O Capítulo 4 apresenta uma visão geral do processo proposto e a sua implementação para a resolução dos problemas. O Capítulo 5 apresenta o funcionamento da atividade relativa ao registro do conhecimento arquitetural. O Capítulo 6 apresenta o funcionamento da atividade relativa a coleta e gerenciamento das preferências do interessado. O Capítulo 7 apresenta o funcionamento da atividade relativa a recomendação arquitetural. O Capítulo 8 apresenta o funcionamento da atividade relativa a geração do código. O Capítulo 9 apresenta a avaliação da solução proposta em um domínio real, voltado ao desenvolvimento de Ambientes Virtuais de Aprendizagem. O Capítulo 10 apresenta algumas considerações finais e direcionamentos para trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo, os conceitos, fundamentais para a realização do trabalho, relativos aos Requisitos de Qualidade, Arquitetura de Software, Sistemas de Recomendação, Desenvolvimento Baseado em Componentes e Desenvolvimento Centrado na Arquitetura são apresentados.

2.1 Requisitos de Qualidade

No ciclo de vida do desenvolvimento de software, a identificação correta dos requisitos do sistema é um processo bastante importante e que influencia diretamente os processos seguintes na construção do sistema. Os requisitos de um sistema são descrições dos serviços oferecidos pelo sistema e as suas restrições operacionais e normalmente são classificados em **requisitos funcionais** e **requisitos não-funcionais** (SOMMERVILLE, 2009).

Os **requisitos funcionais** descrevem o que o sistema deve fazer. Eles dependem do tipo de software que está sendo desenvolvido, dos usuários que o software se destina e da abordagem geral considerada (SOMMERVILLE, 2009). Já os **requisitos não-funcionais** ou **requisitos de qualidade** são aqueles não diretamente relacionados às funções específicas fornecidas pelo sistema e podem estar relacionados às propriedades emergentes do sistema como confiabilidade, tempo de resposta e espaço de armazenamento (SOMMERVILLE, 2009).

Os requisitos de qualidade do software contemplarão itens para avaliar a qualidade interna, externa e a qualidade de uso para satisfazer as necessidades dos desenvolvedores, mantenedores e usuários finais (ISO, 2001; ISO, 2003). As tecnologias utilizadas para alcançar o nível de qualidade necessário como especificação e avaliação da qualidade necessita suportar estes diferentes pontos de vista. A qualidade deve ser medida apropriadamente em cada estágio do ciclo de vida, então é extremamente necessário definir estas perspectivas de qualidade e as referidas tecnologias associadas(ISO, 2001; ISO, 2003).

O propósito é alcançar a qualidade necessária e suficiente para satisfazer as reais necessidades dos usuários. Os requisitos de qualidade externa especificam o nível da qualidade esperada sob a perspectiva externa. Eles incluem os requisitos de qualidade em uso e são usados como objetivo para validação em vários estágios de desenvolvimento. Estes requisitos devem ser declarados em especificações dos requisitos de qualidade, usando métricas externas e podem ser transformados em requisitos de qualidades internas e usados como critério quando a avaliação acontecer (ISO, 2001; ISO, 2003).

De acordo com a ISO 9126 (ISO, 2001; ISO, 2003) os principais atributos de qualidade de um software são: conformidade, segurança de acesso, confiabilidade, usabilidade, efici-

ência , manutenibilidade, testabilidade, portabilidade e interoperabilidade. Abaixo segue uma breve descrição sobre cada um dos atributos de qualidade mencionados anteiormente.

1. **Conformidade**. Estar de acordo com as normas, convenções ou regulamentações em leis e descrições similares relacionadas ao domínio da aplicação;
2. **Segurança no Acesso**. Evitar acesso não autorizado, acidental ou deliberado aos dados e programas;
3. **Confiabilidade**. O sistema deve ter uma pequena quantidade de falhas e em adição a isto, deve continuar funcionando adequadamente mesmo na ocorrência destas falhas;
4. **Usabilidade**. O sistema deve possuir interfaces bem definidas e as operações devem ser fáceis de serem encontradas, reduzindo o esforço do usuário para entender a lógica e o aprendizado para usar o sistema;
5. **Eficiência**. O sistema deve ter a habilidade de executar as atividades adequadamente em relação ao tempo e recursos alocados;
6. **Manutenebilidade**. O sistema deve ser fácil de entender e ser modificado;
7. **Testabilidade**. O sistema deve prover facilidades para execução dos casos de testes;
8. **Portabilidade**. O sistema deve ser fácil de ajustar quando transferido para outros ambientes e / ou plataformas;
9. **Interoperabilidade**. Interação com sistemas específicos.

2.2 Arquitetura de Software

À medida que, a complexidade dos sistemas computacionais crescem, o problema de projetar o sistema vai além de algoritmos e estruturas de dados: projetar e especificar a estrutura geral do sistemas surgem como novos tipos de problemas a serem resolvidos(GARLAN; SHAW, 1994). Neste contexto, projetar a arquitetura de software tornou-se um elemento muito importante e decisivo para sucesso ou falha no projeto, porque é possível descrever, observar e raciocinar sobre o comportamento do sistema e características de alto nível, ajudando na tarefa de projetar e implementar arquiteturas mais reusáveis, na definição dos componentes que farão parte do sistema, na maneira como eles serão organizados e na interação entre eles (CHEN et al., 2002). Além disto, a arquitetura representa esta interação explicitamente materializada através dos conectores (BRITO et al., 2005). Entender a arquitetura do sistema é importante porque ela nos diz como ficará a forma geral do sistema pronto e explica como diversas tecnologias serão usadas para montar este sistema (CHEESMAN; DANIELS, 2001).

A arquitetura de um sistema de software é a estrutura ou estruturas de um sistema, que é composta pelos elementos de software, as propriedades visíveis externamente destes elementos e o relacionamento entre eles (BASS; CLEMENTS; KAZMAN, 2012). Arquiteturas de software representam explicitamente a estrutura dos sistemas, como: *dependability*, confiabilidade[1],disponibilidade, *segurança no funcionamento*[2] e *segurança no acesso*[3] (BASS; KAZMAN, 1999).

Existem algumas vantagens em projetar e documentar explicitamente uma arquitetura de software, são elas:

1. **Comunicação entre os interessados**. A arquitetura é uma apresentação do sistema em alto nível, que pode ser usada para focar a discussão entre os diferentes stakeholders;

2. **Análise do Sistema**. Decisões do projeto arquitetural têm profundo impacto sobre como o sistema pode satisfazer requisitos críticos como: desempenho, confiabilidade e manutenibilidade;

3. **Reuso em Larga escala**. Sistemas que possuem requisitos similares tendem a usar a mesma arquitetura, e logo, podem suportar reuso em larga escala (BASS; CLEMENTS; KAZMAN, 2012).

Essa visão estrutural do sistema traz benefícios importantes como (BRITO; BARBOSA; RUBIRA, 2007):

- Realização explícita dos atributos de qualidade;
- Organização do sistema como uma composição de componentes lógicos;
- Antecipação da definição das estruturas de controles globais;
- Suporte sobre a definição da funcionalidade dos componentes de software.

A arquitetura de software é fundamental para as linhas de produto de software, onde múltiplos sistemas com diferentes funcionalidades são criados a partir da mesma arquitetura básica (BASS; KAZMAN, 1999). Os requisitos de qualidades do sistema como **desempenho**, **manutenibilidade** e **segurança no acesso** já devem aparecer primariamente na definição da arquitetura, pois nenhum deles pode ser alcançado separadamente da visão arquitetural. Definir bem a arquitetura, pode ajudar o arquiteto a identificar alguns riscos e possíveis problemas que possam ocorrer ainda durante o processo de projeto arquitetural, ajudando a mitigá-los cedo.

[1] do inglês *reliability*
[2] do inglês *safety*
[3] do inglês *security*

2.2.1 Estilos e Padrões Arquiteturais

Sistemas que pertencem ao mesmo domínio de aplicação tendem a utilizar a mesma organização estrutural como tipos de componentes que são utilizados e a maneira como estes componentes interagem entre si. Assim, podemos dizer que estes sistemas utilizam o mesmo estilo arquitetural. Os estilos arquiteturais mais conhecidos foram definidos informalmente e de maneira não sistematizada. A comunidade de padrões se dedica a documentar soluções já comprovadas, incluindo estilos arquiteturais, no contexto de tipos de problemas específicos em que cada solução é aplicável (SHAW; CLEMENTS, 1997).

Surgiu, assim, os padrões arquiteturais (SHAW; CLEMENTS, 1997), com o propósito de documentar os estilos arquiteturais de maneira sistematizada. A maneira como o estilo arquitetural é organizado destina-se a satisfazer determinadas propriedades não-funcionais do sistema. Então, a motivação para a escolha do estilo arquitetural depende de quais propriedades se espera do sistema. São exemplos de estilos arquiteturais:

- **Camadas:** No desenvolvimento do sistema é importante dividir o sistema em partes (chamadas subsistemas), onde cada uma das partes tenha sua funcionalidade específica, reduzindo a complexidade e consequentemente facilitando o seu entendimento. Quando o sistema é particionado em subsistemas, concorrentemente ocorre a divisão em camadas. Quando a estruturação ocorre desta maneira, a comunicação ocorre entre as camadas vizinhas, ou seja, a camada hipotética n utiliza os serviços ofertados pela camada $n + 1$ e provê serviços para a camada $n - 1$;

- **MVC:** O *Model-view-controller* (MVC) consiste na separação da informação da interação com o usuário. O *Model* representa a parte da aplicação onde a lógica de negócio é definida. A *View* é responsável pela representação dos dados modelados. O *Controller* será responsável pelas requisições dos usuários, trabalhando na interação entre a *View* e a *Model*, garantindo que as tarefas sejam delegadas corretamente. A utilização deste estilo favorece ao rápido desenvolvimento e facilidade de manutenção;

- **Repositório:** Caracteriza-se pela presença de 2 tipos de componentes: (1) o repositório onde os dados são armazenados e (2) conjunto de componentes que operam sobre estes dados;

- **Heterogêneos:** A depender da complexidade do sistema, torna-se necessário mesclar mais de um estilo arquitetural, com o propósito de satisfazer determinadas propriedades não-funcionais. Uma maneira para realizar esta combinação é através de hierarquias onde um componente, que é inserido dentro de um estilo arquitetural, tem uma estruturação própria.

2.2.2 Avaliação Arquitetural

Sendo a arquitetura de software o elemento chave para o sucesso do projeto de desenvolvimento de software, a análise arquitetural também deve ser uma prática fundamental para a organização (KAZMAN; KLEIN; CLEMENTS, 2000). A realização dos atributos de qualidade não pode ser garantida se não houver um processo de análise formal sobre tal arquitetura. Assim, Kazman *et al* (KAZMAN; KLEIN; CLEMENTS, 2000) propuseram uma técnica para análise de arquiteturas de software, chamada *Architecture Tradeoff Analysis Method* (KAZMAN; KLEIN; CLEMENTS, 2000) (ATAM).

O objetivo do ATAM é entender as consequências das decisões arquiteturais com relação aos requisitos de qualidade do sistema, verificando se tais requisitos serão alcançados pela arquitetura da maneira como a mesma foi concebida, antes que recursos organizacionais sejam comprometidos. Tal método pode ser realizado nas fases iniciais do ciclo de desenvolvimento de software, podendo ser executado de maneira rápida e barata. A análise realizada pelo ATAM será compatível com o nível de análise da especificação arquitetural.

Como principais objetivos do ATAM podemos destacar (KAZMAN; KLEIN; CLEMENTS, 2000):

1. Extrair e refinar uma declaração precisa dos requisitos de qualidade relacionados com a arquitetura;

2. Extrair e refinar uma declaração precisa das decisões arquiteturais;

3. Avaliar as decisões arquiteturais para determinar se elas satisfazem os requisitos de qualidade;

Com o intuito de avaliar a recomendação arquitetural realizada pela ferramenta, será utilizado o método de análise (ATAM) baseado em cenários para medir o nível de satisfação da arquitetura recomendada relacionada aos atributos desejados pelos interessados. Com a utilização do ATAM, pretende-se avaliar as conseqüências das decisões arquiteturais em termos de cenários que envolvem os atributos de qualidade.

2.3 Sistemas de Recomendação

Os sistemas de recomendação (SR) são ferramentas e técnicas com o objetivo de sugerir itens para serem utilizados pelos usuários (RICCI; ROKACH; SHAPIRA, 2011). Eles são direcionados principalmente para pessoas que não têm experiência pessoal suficiente ou competência para avaliar o número potencialmente enorme de itens alternativos que um site, por exemplo, pode oferecer (RICCI; ROKACH; SHAPIRA, 2011).

Nos últimos anos, o interesse em sistemas de recomendação tem aumentado drasticamente. Alguns fatores têm indicado este crescimento, tais como (RICCI; ROKACH; SHAPIRA, 2011; NEIDHARDT et al., 2014):

1. Em sites como Amazon.com, YouTube, Netflix, os sistema de recomendação têm desempenhado um papel importante. Muitas vezes eles dão suporte a comparação de uma quantidade dificilmente compreensível e entendível de opções;

2. A existência de conferências e workshops relacionados à área (RECSYS);

3. Cursos de graduação e pós-graduação inteiramente dedicados aos sistemas de recomendação;

4. Publicação de diversas edições especiais em revistas acadêmicas que cobrem pesquisas e desenvolvimento na área de sistemas de recomendação.

Os SR são sistemas que processam informação de várias fontes de dados a fim de construir suas recomendações. Os dados e o conhecimento, necessários para a realização de tal tarefa, podem ser obtidos de fontes diversas, independente da técnica que será utilizada. De uma forma geral, os dados usados pelos SR referem-se a três tipos de objetos (RICCI; ROKACH; SHAPIRA, 2011): **(1) itens** que são os objetos da recomendação ,**(2) usuários** cuja as características específicas precisam ser coletadas e **(3) transações** que é uma interação gravada entre o SR e o usuário.

No contexto deste trabalho, vamos utilizar um motor de regras como técnica para apoiar o processo de recomendação. Quando utilizamos um motor de inferência, duas técnicas de raciocínio podem ser adotadas: (i) o encadeamento para frente ou (ii) o encadeamento para trás. O encadeamento para frente inicia com as informações conhecidas pelo usuário e a partir destas informações as regras de inferência são utilizadas para extrair mais resultados, até que o objetivo seja alcançado. Já o encadeamento para trás envolve o raciocínio a partir dos objetivos, em busca de verdades que sustentem aquelas hipóteses.

2.4 Desenvolvimento Baseado em Componentes (DBC)

O desenvolvimento baseado em componentes (DBC) visa fornecer um conjunto de procedimentos, ferramentas e notações que possibilitem, ao longo do processo de software, a ocorrência tanto da produção de novos componentes quanto a reutilização de componentes existentes (SILVA-JUNIOR, 2003). O DBC é diferente das abordagens de desenvolvimento anteriores no que se refere a separação da especificação de componente de sua implementação e na divisão das especificações dos componentes em interfaces (CHEESMAN; DANIELS, 2001).

A popularização do DBC é reflexo das pressões que as empresas de software sofrem para poder disponibilizar produtos em prazos cada vez mais curtos e sem esquecer da qualidade. No DBC, a construção dos sistemas é feita através da composição de componentes que já foram previamente especificados, construídos e testados fazendo com que durante o processo de desenvolvimento do software obtenha-se um alto ganho na produtividade e confiabilidade do sistema. Os testes já realizados nos componentes não extinguem que novos testes tenham que ser realizados na construção destes novos sistemas.

2.4.1 Componentes de Software

O aumento da complexidade dos sistemas fez com que fosse necessário dividir o software em pedaços menores, facilitando assim o gerenciamento da sua complexidade. Neste sentido, uma das abordagens que pode ser utilizada é a utilização de componentes para desenvolvimento de um sistema. Um dos objetivos principais de um componente é que ele deve ser facilmente substituível ou por uma implementação completamente diferente das mesmas funções ou por uma versão atualizada da implementação atual(CHEESMAN; DANIELS, 2001).

Para que seja considerado um componente, o mesmo deve seguir alguns princípios como(CHEESMAN; DANIELS, 2001):

1. **Unificação de dados e funções**. Um objeto consiste de dados e funções para manipular os dados;

2. **Encapsulamento**. O cliente de um objeto deve depender da especificação do objeto, não interessando para o mesmo como ele é implementado;

3. **Identidade**. Cada objeto tem uma única identidade, independente do estado.

Um componente pode ser definido de acordo com as diferentes formas que ele pode tomar (CHEESMAN; DANIELS, 2001):

- **Padrão de Componente**. Componentes que estão de acordo com algum conjunto estabelecido de normas;

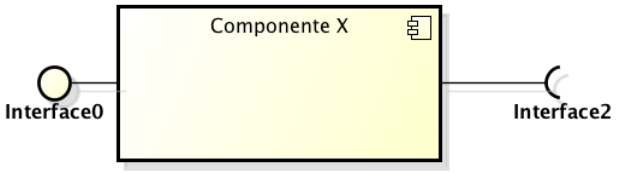
- **Especificação de Componente**. A especificação de uma unidade de software descreve o comportamento de um conjunto de objetos componentes e define uma unidade de implementação. O comportamento é definido como um conjunto de interfaces. Uma especificação de componente é realizada como uma implementação de componente;

- **Interface**. Uma definição de um conjunto de comportamentos que podem ser oferecidos por um objeto componente.

- **Implementação de Componente**. Uma realização de uma especificação de componente, que é independentemente distribuído. Isto significa que pode ser instalado e substituído independentemente de outros componentes. Isto não significa que é independente de outros componentes;
- **Componente Instalado**. Uma cópia instalada de uma implementação de componente. Uma implementação de um componente é distribuída através do registro do componente com o ambiente de execução. Isto habilita o ambiente de execução para identificar os componentes instalados para usar quando se cria uma instância do componente ou quando estiver executando uma das suas operações.

A interface identifica um ponto de interação entre o componente e seu ambiente (CHEESMAN; DANIELS, 2001). As interfaces de um componente podem ser classificadas em 2 tipos: interface provida e interface requerida. A Figura 1 mostra, usando a notação UML, um componente de software com duas interfaces: uma provida e uma requerida.

Figura 1 – Notação de um Componente no padrão UML



Fonte: Elaborada pelo autor

Uma **interface provida** identifica um ponto de acesso a serviços que o componente é capaz de prover para o ambiente. Já uma **interface requerida** identifica um ponto de acesso a serviços que o componente requer do ambiente.

2.5 Processos de Desenvolvimento Baseado em Componentes

Com o aumento da popularidade do DBC, precisou-se de novos processos que suportassem as necessidades do DBC. Mais especificamente, esses processos devem conter fases e métodos que também ofereçam técnicas que permitam o empacotamento de componentes com o objetivo específico de serem reutilizados (SOMMERVILLE, 2009). Os métodos também devem auxiliar na definição de como os componentes devem ser conectados uns aos outros para atender aos requisitos especificados, ou seja, devem auxiliar na construção das arquiteturas de software baseadas em componentes (MORONTE, 2007).

Um processo para o DBC inclui a definição de estratégias para (BRITO et al., 2005):

- **Definição explícita da Arquitetura**. Foca nos aspectos de interação entre os componentes do sistema;

- **Separação de Contextos a partir do Modelo de Domínio**. Facilita a identificação de quais componentes estão passíveis de reutilização, dependendo do domínio de aplicação do sistema que será desenvolvido;
- **Identificação das Interfaces dos Componentes**. O baixo acoplamento proporcionado pela definição de interfaces providas e requeridas é um meio de alcançar esse objetivo;
- **Identificação do comportamento Interno dos Componentes**. Por ser uma estrutura altamente encapsulada, deve haver transparência em relação à tecnologia utilizada para a sua implementação interna;
- **Montagem dos Componentes do Sistema**. Nesta fase ocorre a configuração dos componentes e a ligações entre os componentes através dos conectores para indicação dos objetos reais que implementam os serviços requeridos pelo componente;
- **Manutenção do Repositório dos Componentes**. Tem como objetivo maximizar a reutilização de componentes, através da oferta de mecanismos de busca sistemáticos.

Em um processo de DBC é possível enfatizar dois aspectos distintos (SOMMERVILLE, 2009):

- **Desenvolvimento para reuso**. Tem como foco a criação de componentes com o objetivo dos mesmos serem reutilizados.
- **Desenvolvimento com reuso**. Tem como foco a criação de sistemas a partir de componentes já previamente construídos.

2.6 Desenvolvimento Centrado na Arquitetura

A arquitetura de software é um artefato essencial no ciclo de vida do software e deve auxiliar todas as suas fases (CHEN et al., 2002). A principal motivação para isso é que a abstração oferecida pela arquitetura de software possibilita uma análise estrutural em alto nível e um melhor entendimento do sistema (BASS; KAZMAN, 1999).

Um processo de desenvolvimento centrado na arquitetura, além de usufruir dessa abstração arquitetural em todas as fases do desenvolvimento, deve fornecer meios de especificar a arquitetura do sistema, obedecendo as restrições impostas pelos requisitos não-funcionais especificados (YOU-SHENG; YU-YUN, 2003). Além disso, enquanto os processos tradicionais consideram a composição do sistema como sendo uma atividade de implementação (CHESSMAN; DANIELS, 1992) nos processos centrados na arquitetura a composição dos componentes deve ser considerada em diferentes fases do desenvolvimento.

Em outras palavras, a composição deve ser pensada nos diversos níveis de abstração (modelo abstrato, especificação detalhada e implementação) (CHEN et al., 2002).

As principais vantagens decorrentes da adoção de um processo de desenvolvimento centrado na arquitetura são: (i) facilidade de especificação de propriedades não-funcionais no sistema; (ii) reutilização em um nível maior de granularidade; (iii) redução do tempo de desenvolvimento; (iv) representação estrutural explícita; (v) compreensão facilitada do sistema; e (vi) facilidade de manutenção, tanto corretiva, quanto evolutiva do sistema. Motivadas principalmente pela reutilização de software em larga escala, é cada vez maior o número de empresas que utilizam alguma técnica de desenvolvimento centrado na arquitetura (KOSKIMIES, 2001). Porém, a ausência de processos que possibilitem a verificação formal das propriedades arquiteturais dificulta a adoção de técnicas de prevenção e remoção de falhas durante o desenvolvimento, o que pode comprometer a confiabilidade dos sistemas.

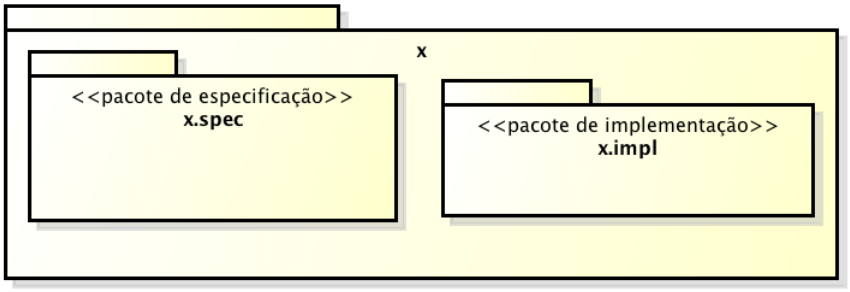
2.7 O Modelo de Implementação de Componentes COSMOS*

COSMOS* (SILVA-JUNIOR, 2003; GAYARD; RUBIRA; GUERRA, 2008) é um modelo que se concentra principalmente na estruturação, projeto e implementação dos componentes e conectores de um sistema, visando garantir a conformidade da arquitetura proposta para o sistema com relação a sua implementação. O modelo COSMOS* também visa construir componentes de software que sejam mais flexíveis à mudanças e mais facilmente adaptáveis e reutilizáveis (SILVA-JUNIOR, 2003).

No modelo COSMOS*, um componente arquitetural é mapeado em um pacote contendo outros dois subpacotes: o **pacote de especificação** (spec) de visão pública, que descreve os serviços que são oferecidos pelo componente e os serviços que ele necessita. Já o **pacote de implementação**, descreve como o componente é implementado. O pacote especificação contém um conjunto de interfaces públicas que serão providas ou requeridas pelo componente arquitetural. Este pacote estará organizado em dois subpacotes: um para guardar as interfaces providas (spec.prov) e o outro para guardar as interfaces requeridas (spec.req). Já o pacote de implementação (impl) contém as classes que implementarão os serviços ofertados pelo componente arquitetural. Todas as classes que pertencem a este pacote de implementação possuem visibilidade restrita ao pacote, com exceção da classe responsável pela instanciação das classes.

Um conector arquitetural é mapeado em um pacote que contém classes que materializam as conexões entre componentes, também chamadas de conexões de interface (SILVA-JUNIOR, 2003). No modelo COSMOS*, um conector é tratado como um componente, só que visto de maneira mais simplificada, tendo a estrutura similar ao pacote de implementação do componente. Ele será responsável por realizar a conexão entre componentes, utilizando para isto as interfaces providas e requeridas dos componentes.

Figura 2 – Representação do Componente no modelo COSMOS*



Fonte: Elaborada pelo autor

O modelo COSMOS* é formado por três modelos inter-relacionados (SILVA-JUNIOR, 2003):

- **Modelo de Especificação**. Define a visão externa do componente e corresponde ao pacote de especificação;
- **Modelo de Implementação**. Descreve como o componente é implementado e corresponde ao pacote de implementação;
- **Modelo de Conector**. Descreve as conexões entre componentes, realizadas através de conectores, e corresponde aos componentes, conectores e conexões de interface.

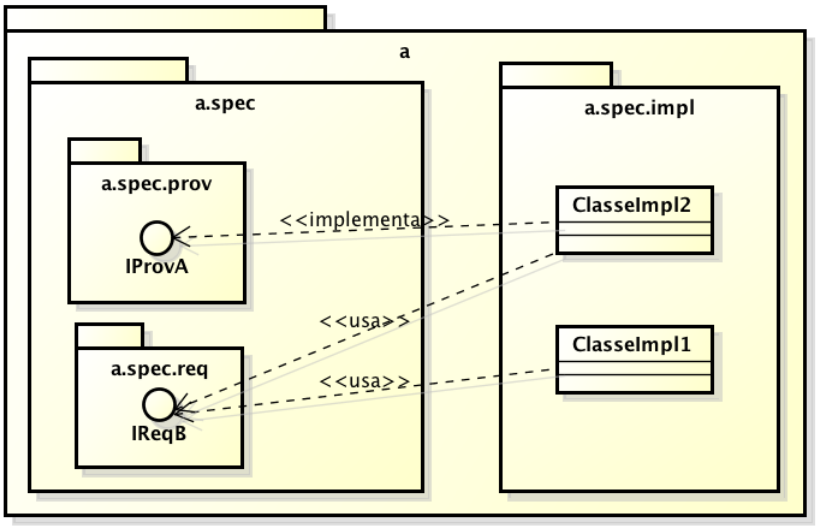
2.7.1 Modelo de Especificação

O modelo COSMOS* traz como característica a separação entre a especificação e a implementação. Tal característica traz como vantagens melhor organização do processo de desenvolvimento e divisão do componente em parte pública e privada. A especificação do componente será inteiramente pública, onde constarão as especificações dos serviços oferecidos pelo componente, através das interfaces providas (prov), bem como os serviços que o componente necessita, através das interfaces requeridas (req).

O projeto das interfaces de um componente neste modelo torna-se um importante aspecto no processo de especificação (SILVA-JUNIOR, 2003). Uma característica desejável às interfaces é que as mesmas somente disponibilizem operações estritamente relevantes para os seus clientes. Se as interfaces utilizarem tipos primitivos nos parâmetros e retornos, elas irão melhorar a adaptabilidade do componente, mas por outro lado reduz o acoplamento. Por outro lado, se as interfaces utilizarem tipos mais abstratos, elas irão aumentar o poder de legibilidade, consistência e confiabilidade do componente (SILVA-JUNIOR, 2003).

Todo componente no modelo COSMOS* possui uma interface que deve ser provida, e esta interface é denominada **IManager**. Tal interface disponibilizará serviços que per-

Figura 3 – Exemplo da Estrutura interna de um Componente no Modelo COSMOS*



Fonte: Elaborada pelo autor

mitam obter informações acerca das interfaces dos componentes, sejam elas providas ou requeridas. Além da obtenção das informações, a interface contemplará operações para ajustar as configurações das interfaces providas e requeridas, como, por exemplo, identificar o tipo da interface ou até mesmo o objeto façade que a implementa. A Figura 4 mostra um exemplo da especificação da interface **IManager** do componente COSMOS*, onde podemos observar os métodos que são responsáveis pela configuração e manipulação das interfaces providas e requeridas pelo componente.

Figura 4 – Exemplo Interface IManager

```
1  package core.spec.prov;
2  import java.util.Map;
3  public interface IManager{
4      public Object getProvidedInterface(String name);
5      public void setProvidedInterface(String name, Object facade);
6      public Object getRequiredInterface(String name);
7      public void setRequiredInterface(String name, Object facade);
8      public String[] ListProvidedInterfaceNames();
9      public String[] ListRequiredInterfaceNames();
10     public Map<String,Class> getProvidedInterfaceTypes();
11     public Map<String,Class> getRequiredInterfaceTypes();
12     public void setRequiredInterfaceType(String name, Class type);
13     public void setProvidedInterfaceType(String name, Class type);
14 }
```

Fonte: Elaborada pelo autor

A função dos objetos façades não é realizar a implementação das interfaces providas pelos componentes, mas sim realizar o redirecionamento das chamadas de métodos para as classes de implementação internas aos componentes que, de fato, implementam os serviços que os componentes oferecem aos seus clientes. Cada uma das interfaces providas pelo componente pode ser implementada por um objeto façade diferente (SILVA-JUNIOR, 2003).

2.7.2 Modelo de Implementação

O modelo de implementação define como os serviços providos por um componente são implementados (SILVA-JUNIOR, 2003). O componente possui um pacote denominado impl, que armazena todas as implementações das interfaces providas, e os objetos façades de cada interface, realizando o redirecionamento entre a interface e a classe que a implementa as operações. As classes de implementação contidas neste pacote possuem visibilidade restrita ao pacote de implementação, por outro lado, existem outras classes neste pacote necessárias no modelo COSMOS* que definem os serviços de infra-estrutura e são obrigatórias neste modelo. Tais classes também possuem visibilidade restrita ao pacote, exceto a classe **ComponentFactory**, que possui visibilidade pública.

Tais classes de infra-estrutura possibilitam a materialização dos componentes arquiteturais e propiciam a flexibilidade e evolutibilidade dos componentes (SILVA-JUNIOR, 2003). As classes de infra-estrutura de implementação dos componentes são(SILVA-JUNIOR, 2003):

1. **ComponentFactory**. Responsável pela instanciação do componente;
2. **Manager**. Realiza a interface **IManager**;
3. **Facade**. Realiza o redirecionamento entre a classe que implementa os métodos e a interface;
4. **ObjectFactory**. Responsável pela criação de instâncias de classes de implementação.

A classe **ComponentFactory** não precisa ser instanciada, pois a mesma possui um único método que é estático. A implementação do método **createInstance()** instancia um objeto do tipo **Manager** que representa um objeto que materializa o componente em tempo de execução e tem como retorno um objeto do tipo **IManager**. Ela implementa operações para configuração e uso do componente e mantém as referências para os objetos façades que implementam as interfaces providas e requeridas pelo componente (SILVA-JUNIOR, 2003). A Figura 5 ilustra a implementação da classe **ComponentFactory**, demostrando a instanciação da classe **Manager** pelo método **createInstance()**.

As metainformações sobre o componente são acessadas através das operações implementadas pela classe **Manager**. Esta classe também é responsável por manter referências

Figura 5 – Exemplo da Implementação da Classe ComponentFactory

```
1  package core.impl;
2  import cosmos.IManager;
3  public class ComponentFactory{
4      public static IManager createInstance(){
5
6          return new Manager();
7      }
8
9  }
```

Fonte: Elaborada pelo autor

entre as interfaces e os objetos façades que as implementam ou necessitam dos serviços. Para cada interface provida deve existir um façade que realize o redirecionamento para classe que efetivamente a implementa. A Figura 6 mostra a classe que implementa os serviços de uma interface provida.

Figura 6 – Exemplo Classe Implementada

```
1  package core.impl;
2  class ClassCore{
3      private Manager manager;
4
5      ClassCore(Manager mgr){
6
7           this.manager = mgr;
8      }
9
10     public void Testar( String msg){
11
12         System.out.println("Mensagem: "+msg);
13     }
14
15 }
```

Fonte: Elaborada pelo autor

Para acessar os serviços das classes de implementação, para as quais a classe façade delega as operações, o construtor da classe façade recebe como parâmetro a instância corrente da classe Manager e obtém a instância da classe ObjectFactory, criada anteriormente (SILVA-JUNIOR, 2003). A Figura 7 ilustra a implementação de um façade que realiza o redirecionamento entre a interface provida e a classe que a implementa, conforme mostrado na Figura 6.

A opção do modelo COSMOS* em diminuir as dependências diretas entre as classes de implementação e as interfaces de especificação, utilizando para isto interfaces auxilia-

Figura 7 – Exemplo Classe Facade

```
1  package core.impl;
2  class FacadeCore implements core.spec.prov.ICore{
3      private core.impl.ClassCore core;
4      private Manager manager;
5
6      FacadeCore(Manager mgr){
7        this.manager = mgr;
8        this.core = new ClassCore(this.manager);
9      }
10
11     public void Testar( String msg){
12          this.core.Testar(msg);
13     }
14
15 }
```

Fonte: Elaborada pelo autor

res chamadas de façades, tem como objetivo diminuir o acoplamento entre estas classes, fazendo com que a manutenção e evolução da implementação do componente sejam realizadas de maneira mais fácil.

As classes, responsáveis pela implementação das funcionalidades do componente, podem em algum momento necessitar de funcionalidades providas por outros componentes. Estes serviços são acessíveis através das interfaces requeridas do componente, porém para que o acesso a estas funcionalidades seja realizado, é necessário que o objeto que realize tal funcionalidade seja corretamente associado a interface requerida, operação esta realizada através do método **setRequiredInterface()** que é implementada pela classe **Manager**. Esta característica torna o componente mais flexível, porém pode torná-lo mais complexo para implementar e usá-lo.

2.7.3 Modelo de Conector

O modelo de conectores descreve as conexões entre um conjunto de interfaces providas e interfaces requeridas, permitindo assim a interação de dois ou mais componentes numa composição de software (SILVA-JUNIOR, 2003). Assim, como um componente normal, um conector é mapeado como um pacote, só que de uma maneira mais simplificada. O pacote do conector tem a sua estrutura interna similar ao pacote de implementação de um componente. Esta estrutura é formada pelas classes **ComponentFactory**, **Manager** e **Adapter**.

O conector não criará interfaces providas e nem requeridas, seu papel é utilizar as interfaces dos componentes que vão participar da composição do software para estabelecer as conexões de interface (SILVA-JUNIOR, 2003). As classes **ComponentFactory**

e **Manager** possuem o mesmo papel que no modelo anterior de implementação, já o **Adapter** tem como função realizar as conexões de interface, fazendo com que interfaces incompatíveis interajam diretamente através do conector.

Os conectores devem ser essencialmente abertos e configuráveis, de modo que os aspectos não-funcionais da interação e coordenação entre componentes possam ser inseridos ou modificados (SILVA-JUNIOR, 2003).

3 TRABALHOS RELACIONADOS

Neste capítulo, os trabalhos relacionados a cada uma das áreas em que o trabalho se propõe a atuar são apresentados e discutidos. Para cada trabalho, uma discussão é realizada sobre os pontos comuns e divergentes com a abordagem utilizada nesta pesquisa.

3.1 Resolução de *Trade-offs*

Diante da ausência típica de dados quantitativos confiáveis, algumas técnicas de Engenharia de Requisitos, como Tropos (GIORGINI et al., 2003) e i* (YU, 1996) tratam de atributos de qualidade como objetivos sem critérios claros, permitindo assim raciocinar sobre a satisfação parcial de tais metas, utilizando uma taxonomia qualitativa como parcialmente satisfeito, suficientemente satisfeito, parcialmente negado e completamente negado. Embora seja considerada uma boa evolução, a subjetividade da classificação e avaliação dos atributos de qualidade podem produzir requisitos de qualidade conflitantes. Incerteza e incompletude são características inerentes dos atributos de qualidade do software no início do desenvolvimento do software (NGO-THE; RUHE, 2005). Quando os conflitos emergem entre os atributos de qualidade, o gerenciamento dos *trade-offs* torna-se uma questão crítica (ROBINSON; PAWLOWSKI; VOLKOV, 2003).

Elahi and Yu (ELAHI; YU, 2011) apresentam uma ferramenta semi-automática que usa o processo Even Swaps (HAMMOND; KEENEY; RAIFFA, 2002) para a tomada de decisão relacionada aos requisitos conflitantes. Embora, o trabalho de Elahi and Yu (ELAHI; YU, 2011) apresente semelhanças com a nossa abordagem, a resolução de trade-off entre requisitos é feita de maneira automática e pode envolver requisitos funcionais e não-funcionais, além de não estar inter-relacionado com o projeto arquitetural. Nosso trabalho foca no gerenciamento de *trade-offs* entre requisitos de qualidade, ora resolvendo os *trade-offs* automaticamente ou através do processo de interação com o interessado, relacionado ao projeto arquitetural. Além disto, pretende-se com este trabalho esclarecer o significado dos atributos de qualidade e o relacionamento entre eles, a fim de melhorar a confiança sobre a especificação final dos atributos de qualidade.

García-Mireles *et al.* (GARCÍA-MIRELES et al., 2013) apresentam um framework conceitual para lidar com os *trade-offs* dos atributos de qualidade do software. Esta solução apresenta um conjunto de atividades para gerenciamento de *trade-offs* baseado numa comparação sistemática das especificações do CMMI e da ISO-12207. Este trabalho é compatível com a abordagem de García-Mireles *et al.* e pode ser visto como uma instância suportada por uma ferramenta deste framework conceitual.

3.2 Decisões Arquiteturais

Algumas ferramentas foram desenvolvidas para ajudar os arquitetos de software durante a fase de projeto arquitetural. Parmar *et al* (Mahesh Parmar W.U. Khan, 2011) propôs uma ferramenta para inferir os atributos de qualidade com base em cenários sobre o comportamento do sistema. Esses cenários podem ter um impacto positivo ou negativo sobre um atributo de qualidade. Depois disso, os atributos de qualidade são classificados e relacionados a um conjunto de princípios de projeto que são relevantes para o sistema.

Embora a abordagem proposta também use uma estratégia baseada em cenários para obter os atributos de qualidade e resolver os *trade-offs* entre eles, um framework de sistema especialista tem sido usado para gerenciar os *trade-offs*. Além disso, em nossa solução, as decisões de projeto relacionados à arquitetura de software são armazenadas em um repositório de conhecimento, a fim de evitar vaporização conhecimento arquitetural.

Tofan *et al* (TOFAN; GALSTER, 2014) desenvolveu uma ferramenta Web a fim de ajudar os arquitetos a capturar o conhecimento tácito e decisões arquiteturais. A ferramenta foca em três aspectos de decisões arquiteturais: captura de decisões usando a técnica *Repertory Grid* (TOFAN; GALSTER, 2014), análise de decisões e tomada de decisões em grupo. As decisões arquiteturais desta ferramenta estão relacionadas com a escolha de padrões, tecnologias ou decomposição do sistema, bem diferente da abordagem proposta nesta pesquisa que realiza recomendações baseadas nos atributos de qualidade usando gerenciamento de *trade-offs*, dando ao usuário um estilo arquitetural adequado com todos os componentes, conectores e os relacionamentos entre eles.

Outra ferramenta com suporte ao gerenciamento de conhecimento para o projeto da arquitetura de software é proposta por Babar *et al.* (BABAR; GORTON, 2007). Nesta solução, PAKME (BABAR; GORTON, 2007) serve como um repositório do conhecimento, onde arquitetos podem registrar as decisões de projeto a fim de prevenir a vaporização do conhecimento arquitetural. Contudo, os autores não utilizam este conhecimento arquitetural para recomendar uma arquitetura de software.

A solução proposta neste trabalho também pode ser vista como um repositório do conhecimento, onde decisões de projeto sobre os estilos arquiteturais em um domínio específico podem ser armazenadas. Entretanto, PAKME funciona como um manual para decisões arquiteturais, enquanto a abordagem escolhida neste trabalho usa o conhecimento capturado e armazenado para recomendar uma arquitetura adequada baseada em experiências anteriores.

Uma técnica de tomada de decisão foi proposta por Al-Naem *et al* (AL-NAEEM et al., 2005). Tal técnica consiste em uma abordagem de projeto orientada para a qualidade que promove engenharia disciplinada e um framework de raciocínio durante o projeto arquitetural. As preferências dos interessados baseadas nos atributos de qualidade são levadas em consideração para ajudar os arquitetos para determinar sistematicamente a melhor combinação de alternativas de projeto. Embora o trabalho proposto nesta dissertação

também use os atributos de qualidade para mapear as preferências dos usuários sobre um determinado software, Al-Naem *et al* (AL-NAEEM et al., 2005) foca em alternativas de projeto, diferentemente da abordagem proposta que foca em decisões de projeto.

3.3 Desenvolvimento Centrado na Arquitetura

Tomita (TOMITA, 2006) apresenta uma proposta para estender o ambiente Eclipse através de um conjunto de plugins, podendo guiar o usuário na tarefa de especificação e modelagem utilizando assistentes (*wizards*), e na geração dos artefatos propostos. Este ambiente denominado Bellatrix dá suporte a um processo de desenvolvimento baseado em componentes, chamado UML Components(CHESSMAN; DANIELS, 1992). Para utilização do mesmo, o processo de desenvolvimento tem que estar modelado em uma ferramenta de *workflow*, o que permite maior flexibilidade no uso e torna mais simples adaptar o processo à realidade de cada equipe de desenvolvimento (TOMITA PAULO A. C. GUERRA, 2006).

O ambiente proposto trazia um conjunto de editores, que permitia a modelagem gráfica da arquitetura e sua persistência. A partir desta modelagem, o ambiente gerava o esqueleto do código baseado no modelo COSMOS* para implementação dos componentes, garantindo assim a consistência entre arquitetura e código. O ambiente possuía um Repositório de Componentes integrado ao ambiente, bem como uma Ferramenta de Gestão de Configuração, como o CVS.

Assim, as características do ambiente Bellatrix são (TOMITA, 2006): (1) suporte à modelagem de interfaces; (2) suporte à modelagem e especificação de componentes; (3) suporte à modelagem e especificação da arquitetura de software e modelagem da configuração de um sistema; (4) apoio gráfico / visual para as atividades de modelagem e especificação de componentes e configurações, facilitando seu uso; (5) aderência a uma metodologia de desenvolvimento baseado em componentes, no caso o método UML Components; (6) materialização dos componentes modelados utilizando o modelo de implementação de componentes COSMOS*, inclusive facilitando seu uso através da geração de código automática; (7) código aberto (*open source*); (8) integração das ferramentas, reunindo modelagem e implementação de componentes.

O Bellatrix não tem como objetivo apoiar a fase de engenharia de requisitos, muito menos a recomendação arquitetural. Ela propõe uma ferramenta CASE, focada nos arquitetos, que propicia facilidades na criação de modelos arquiteturais. A partir deste modelo é que o código estrutural será gerado. O trabalho que está sendo proposto tem um foco mais amplo, onde as fases de desenvolvimento estão inter-relacionadas. Começa na especificação de requisitos de qualidade, onde as preferências do usuário são coletadas. Com os requisitos em mãos, uma arquitetura adequada será recomendada e por fim a geração do código será realizada. A proposta também permitirá que além do esqueleto do código

gerado, também será possível gerar implementações dos métodos dos componentes, desde que os mesmos tenham sido especificados anteriormente.

Em suma, o trabalho proposto tem como foco a identificação dos desejos que o interessado tem acerca do sistema. Utilizando o conhecimento do engenheiro de requisitos e do arquiteto armazenado na ferramenta é que será possível identificar e resolver *trade-offs* entre os requisitos, recomendar a arquitetura do sistema e gerar o código do sistema utilizando o modelo COSMOS*.

4 UMA SOLUÇÃO PARA APOIO A PROCESSOS DE DESENVOLVIMENTO DE SOFTWARE CENTRADO NA ARQUITETURA

Neste capítulo, uma visão geral da solução que visa apoiar processos de desenvolvimento centrados na arquitetura é descrita, através da apresentação das principais atividades, sua arquitetura e implementação da solução.

4.1 Visão Geral do Processo

Está sendo proposto neste trabalho a construção de um processo apoiado por ferramenta, que visa ajudar os engenheiros, arquitetos e desenvolvedores de software, durante o ciclo de desenvolvimento. O intuito é apoiar a execução das atividades, contribuindo para o aumento da eficiência na execução destas atividades, mitigando a ocorrência de erros e consequentemente aumentando a sua confiança perante os interessados. Podemos destacar como um diferencial deste processo, a inclusão de mecanismos que utilizam o conhecimento relativo à engenharia de requisitos e projeto arquitetural para apoiar as decisões durante as fases de desenvolvimento. O conhecimento adquirido e que será utilizado pela solução proposta será originado de fontes diversas, dentre elas podemos citar: experiências anteriores de arquitetos e engenheiros de software, literatura especializada e projetos já existentes.

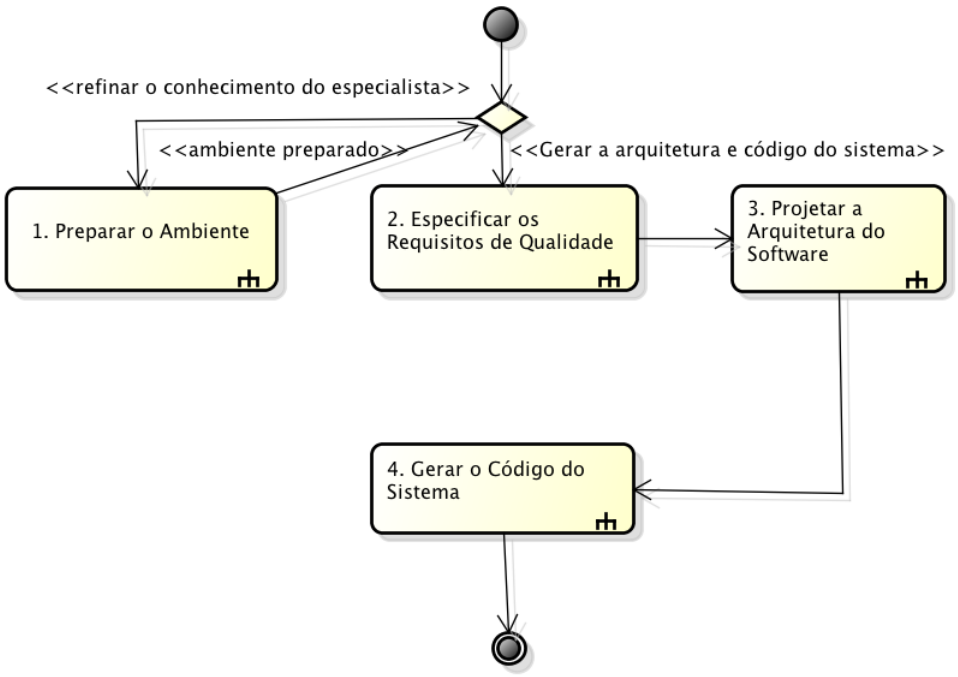
A solução proposta pode ser vista como um catálogo inteligente baseado em cenários, onde os interessados, à medida que utilizam a solução e as avaliações das recomendações aconteçam, tendem a melhorar o entendimento sobre as decisões arquiteturais relacionadas aos requisitos de qualidade especificados e a geração de código. Vale salientar que, a aplicabilidade da solução está direcionada a domínios onde apresentem variabilidades arquiteturais relacionadas aos requisitos de qualidade.

A solução atuará principalmente durante as etapas de (BRITO et al., 2005; PRESSMAN, 2001): (i) Definição do Problema, (ii) Definição de Estrutura e (iii) Desenvolvimento do Sistema. Todas estas etapas trabalharão interligadas, ou seja, a saída de uma etapa servirá como entrada para a próxima, com o propósito de garantir a consistência de informações entre elas. Na etapa de **definição do problema**, a ferramenta atua na **especificação de requisitos**, com foco principal na obtenção dos requisitos não-funcionais e na resolução de *trade-offs* entre estes requisitos (quando houver). Durante a etapa de **definição da estrutura**, a ferramenta apoia a fase de especificação arquitetural, visando dar agilidade durante o processo de construção da arquitetura, consistência através do mapeamento dos requisitos não-funcionais para os elementos que irão compor a arquitetura do sistema e ainda realizando o refinamento e avaliação de tal arquitetura. E por fim, na etapa de **desenvolvimento do sistema**, a ferramenta apoia as fases de análise, projeto

e implementação, fazendo a geração do código baseada na arquitetura recomendada pela ferramenta, utilizando um modelo de desenvolvimento centrado na arquitetura.

O gerenciamento de *trade-offs* é uma tarefa importante, que pode influenciar muitas atividades do desenvolvimento de software. Em termos de atributos de qualidade, o gerenciamento de *trade-off* é especialmente importante no contexto do projeto arquitetural. Uma visão geral do processo proposto é apresentado na Figura 8, que contextualiza atividades de gerenciamento de *trade-off* com atividades de projeto arquitetural e geração de código. Sendo assim, o processo foi concebido em 4 atividades primordiais, são elas: (i) Preparar o Ambiente, (ii) Especificar os Requisitos de Qualidade, (iii) Projetar a Arquitetura de Software e (iv) Gerar o Código do Sistema. Aqui, faremos uma breve descrição sobre o objetivo das atividades propostas. Uma visão mais detalhada sobre o funcionamento de cada uma das atividades será apresentada nos próximos capítulos.

Figura 8 – Visão Geral da Ferramenta Proposta



Fonte: Elaborada pelo autor

- **Preparar o Ambiente**. Esta atividade será responsável pelo processo de aquisição e representação do conhecimento relativo à engenharia de requisitos e decisões arquiteturais, utilizando um modelo de visão centrada nos requisitos. Tanto o engenheiro quanto o arquiteto de software poderão utilizar a ferramenta para realizar o armazenamento do conhecimento relativo ao domínio. Tarefas como registrar os requisitos de qualidade, bem como os padrões de *trade-offs* e o conhecimento para resolução destes *trade-offs*, estilos arquiteturais e seus elementos pertencentes a estes estilos arquiteturais fazem parte desta atividade.

- **Especificar os Requisitos de Qualidade**. Coletar as preferências dos interessados, levando em consideração os requisitos de qualidade do sistema é uma tarefa desta atividade. Outra atribuição desta atividade é realizar o gerenciamento dos requisitos de qualidade, identificando e resolvendo potenciais *trade-offs* entre eles.

- **Projetar a Arquitetura de Software**. Baseada na preferência dos interessados, uma recomendação arquitetural será realizada por esta atividade. Uma vez recomendada, a mesma poderá ser avaliada, levando em consideração as necessidades dos interessados.

- **Gerar o Código do Sistema**. Com a definição da arquitetura, esta atividade será responsável por materializar as decisões arquiteturais através da geração do código estrutural do sistema, utilizando como referência um modelo de desenvolvimento centrado na arquitetura, proporcionando assim a aderência entre a arquitetura e o código.

4.2 Arquitetura da Ferramenta Proposta

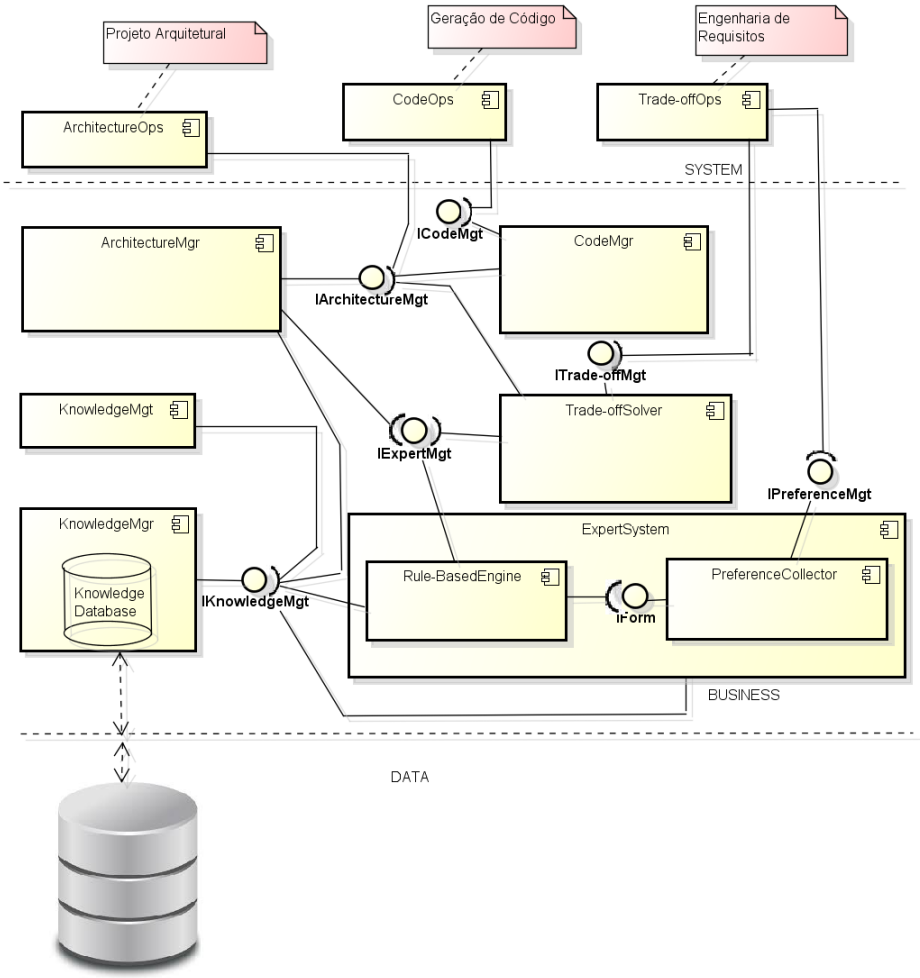
A ferramenta projetada para apoiar o processo proposto segue um estilo arquitetural heterogêneo, numa combinação de arquitetura em camadas, utilizando componentes independentes, centrada em dados e com características de sistema baseado em regras. A Figura 9 apresenta uma visão lógica (estrutural) dessa arquitetura, onde cada estilo adotado e componente utilizado são detalhados a seguir.

Dentre os benefícios da utilização do estilo **componentes independentes** na composição da arquitetura, podemos citar o baixo acoplamento entre os componentes. Ele acontece em virtude das interfaces estarem bem definidas e a interação entre eles ocorrerem através destas interfaces. Tal característica permite que o sistema tenha facilidade na troca de suas partes, facilidade na evolução e confinamento de falhas. A definição dos componentes para esta arquitetura levou em consideração a especificação das atividades presentes no processo, onde cada componente assumirá funcionalidades relacionadas a uma atividade específica.

A organização do sistema em **camadas** permite que a separação dos serviços ocorra hierarquicamente, onde cada camada oferece um serviço a uma camada imediatamente acima e utiliza os serviços da camada imediatamente abaixo. A fim de adequar o estilo ao processo proposto, o sistema foi separado em 4 camadas. Esta organização é uma variante do modelo tradicional de 3 camadas (apresentação, negócio e dados) onde uma nova camada denominada **sistema** foi incluída, com o intuito de reduzir a complexidade e promover o reuso, separando explicitamente grupos de funcionalidades básicas (negócio) dos grupos de funcionalidades mais específicas (sistema).

Somente componentes relacionados a interação com os usuários estarão presentes na camada de **Apresentação**, a qual foi omitida da Figura 9 para evitar o comprometimento

Figura 9 – Arquitetura da Ferramenta Proposta



Fonte: Elaborada pelo autor

no entendimento da arquitetura. A camada **Sistema** visa separar as funcionalidades específicas do sistema da camada de **Negócio**, que contém as funcionalidades comuns relativas ao domínio (básicas). Esta separação promove um melhor entendimento e gerenciamento, uma vez que as regras de negócio relativas ao domínio serão armazenadas em formas de regras de produção, facilitando assim a manutenção do sistema. Podemos considerar as regras como um repositório do conhecimento, onde informações relativas à verificação e resolução de *trade-offs* e o cálculo para a recomendação arquitetural estarão presentes. Como benefício desta separação podemos citar o aumento do reúso, já que os componentes de negócio são mais estáveis. Por fim, a camada de **Dados** será reponsável por armazenar parte do conhecimento arquitetural e relacionado à engenharia de requisitos.

Na adoção do modelo **centrado em dados**, o comportamento do sistema é integrado em torno dos dados que o circundam. Como característica deste estilo, é importante destacar a maior integridade entre os componentes e a separação entre dados e o proces-

samento, favorecendo assim a sua evolução, principalmente a adição de novos módulos e novas funcionalidades.

4.2.1 Especificação dos Componentes

Para promover um melhor entendimento sobre o papel de cada componente na composição da arquitetura, é importante realizar o detalhamento da sua funcionalidade, bem como a interação com os outros componentes. Na arquitetura apresentada na Figura 9, o componente **PreferenceCollector** captura as preferências dos interessados e passa para o componente **Rule-BasedEngine**, a fim de atualizar a memória de trabalho do framework. O componente **Rule-BasedEngine** consiste em um sistema baseado em regras que emula a capacidade de um especialista humano de tomar decisões (JACKSON, 1998). Tais sistemas são projetados para resolver problemas complexos através do raciocínio sobre o conhecimento, usando para isto um motor de inferência, que é um programa de computador que tenta extrair respostas de uma base de conhecimento. O componente **Trade-offSolver** pode ser visto como um adaptador que usa o componente **ExpertSystem** e realiza pós-processamento da sua saída a fim de oferecer informações mais valiosas e contextualizadas.

Com a memória de trabalho atualizada, o componente **Rule-BasedEngine** utiliza o método de raciocínio de encadeamento para frente e usará o conhecimento armazenado para identificar potenciais *trade-offs* entre os requisitos. Uma vez identificados estes *trade-offs*, o componente **ExpertSystem** tenta resolver cada um destes conflitos, apresentando questões associadas aos padrões de *trade-offs*. Tais questões, definidas pelo especialista, são consideradas parte do conhecimento arquitetural mencionado anteriormente. Esta resolução de conflitos ocorre de maneira interativa, realizando questionamentos aos interessados a cada *trade-off* listado. Uma vez que um conflito é resolvido, o componente **ExpertSystem** verifica se ainda existem conflitos pendentes ou se novos foram criados. Este processo continuará até que todos os *trade-offs* sejam resolvidos, ou os interessados interrompam o processo.

O componente **ArchitectureMgr** é responsável pela realização da recomendação arquitetural. Ele receberá como entrada a lista dos requisitos de qualidade e suas respectivas prioridades definidas pelos interessados, com os *trade-offs* entre estes requisitos resolvidos. Utilizando-se também do conhecimento do arquiteto previamente armazenado no repositório do conhecimento, ele utiliza o componente **ExpertSystem** para inferir a arquitetura adequada para os interessados. Será papel do arquiteto inserir, dentro da base de regras, a fórmula que ele considera mais adequada para realizar a recomendação. E por fim, o componente **CodeMgr** é encarregado da geração do código do sistema, seguindo o modelo de implementação de componentes COSMOS*. Ele recebe como entrada a arquitetura que foi sugerida pelo componente **ArchitectureMgr**. Ao final do processo, os códigos estruturais do sistema, relativos à cada um dos elementos presentes na recomendação

apresentada serão gerados.

4.3 Visão Geral da Implementação

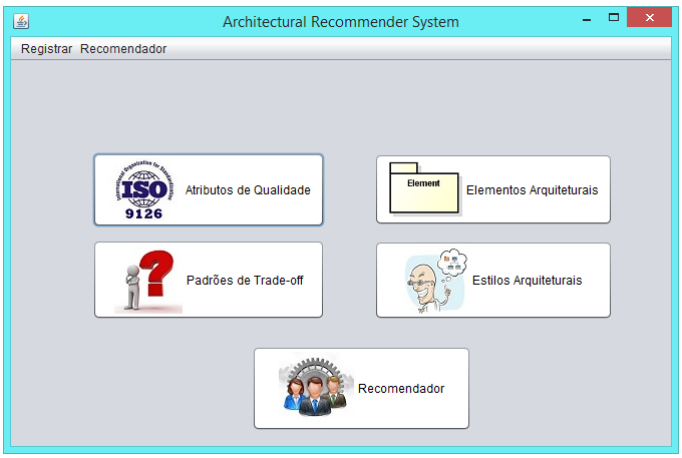
Uma ferramenta genérica que apoia as atividades propostas neste trabalho, relacionadas com a engenharia de requisitos, recomendação arquitetural e geração de código, foi desenvolvida. A implementação seguiu a arquitetura proposta e a especificação dos componentes a fim de atender os resultados esperados com a implementação do processo.

Como decisão de projeto, a linguagem de programação JAVA foi adotada, assim como o banco de dados MySQL, onde parte do conhecimento capturado é persistido utilizando o *framework* Hibernate (HIBERNATE, 2014) para realizar esta tarefa. O Hibernate é um framework *open-source* que realiza o mapeamento entre os atributos de uma entidade no banco de dados e o modelo objeto da aplicação. Isto tornou a escrita da aplicação mais legível, além de trazer benefícios como aumento da confiabilidade e desempenho.

A ferramenta desenvolvida reusou um motor de inferência baseado em regras, denominado Drools (DROOLS, 2014). Este é uma solução para Sistemas de Gerenciamento para Regras de Negócio (BRMS)[1]. A adoção desta solução levou em consideração a sua flexibilidade para incluir em suas regras de negócio os objetos utilizados na implementação do sistema. Assim, o Drools permite um acesso mais racional e prático ao conhecimento retido na ferramenta. Outro ponto positivo é que esta solução é largamente utilizada, tendo uma comunidade bastante ativa na Internet.

A primeira tela da ferramenta proposta é apresentada na Figura 10. Na tela principal do sistema as principais funcionalidades estão acessíveis a uma distância de um clique, permitindo assim um acesso rápido e sem complicação à funcionalidade desejada.

Figura 10 – Tela Principal da Ferramenta



Fonte: Elaborada pelo autor

[1] do inglês *Business Rules Management System*

Nós próximos capítulos, à medida que as atividades do processo são apresentadas, detalhes de suas respectivas implementações serão exibidos.

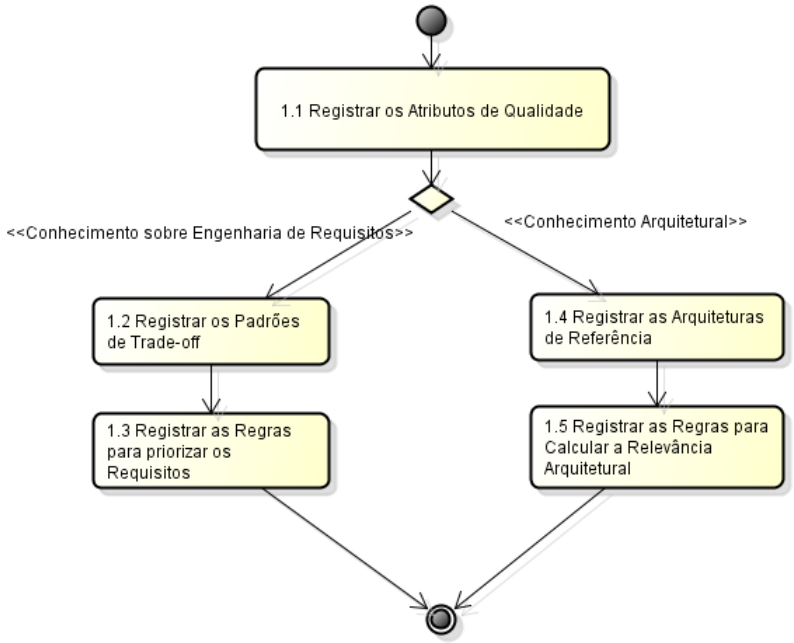
5 PREPARAR O AMBIENTE (ATIVIDADE 1)

A preparação do ambiente consiste na representação do conhecimento relacionado às atividades propostas pela solução sobre um determinado domínio. O conhecimento é um item extremamente importante e que influencia diretamente os resultados apresentados durante as etapas de engenharia de requisitos e recomendação arquitetural. Por representar e armazenar este conhecimento, a preparação do ambiente traz benefícios importantes a todo o processo, sendo o maior deles a prevenção da vaporização do conhecimento arquitetural. Como esta proposta não está atrelada a nenhum domínio de aplicação específico, caberá aos especialistas realizarem a inserção do conhecimento prévio sobre determinado domínio, a fim de utilizá-lo durante as atividades subsequentes. Esta atividade está focada em três aspectos : (i) Representação e persistência dos Elementos Arquiteturais que irão compor possíveis arquiteturas de referência para um determinado domínio; (ii) Representação e persistência do conhecimento do especialista relacionado à engenharia de requisitos; e (iii) Representação e persistência do conhecimento do especialista relacionado às Decisões Arquiteturais.

Apesar da preparação do ambiente ser executada no início, antes da primeira utilização do sistema, o conhecimento armazenado poderá ser aprimorado a qualquer momento. Este aprimoramento acontece através da reexecução da atividade. Quanto mais preciso for o conhecimento utilizado, melhor serão os resultados obtidos pela solução. Um funcionamento mais detalhado da atividade é apresentado na Figura 11. A fim de promover um melhor entendimento a atividade foi dividida em outras 4 subatividades, são elas:

- **Registrar os Atributos de Qualidade**. Permite registrar os atributos de qualidade que serão levados em consideração durante todas as etapas do processo. Cabe ao especialista definir com quais atributos ele deseja trabalhar.
- **Registrar os Padrões de *Trade-off***. Registrar os cenários em que dois ou mais atributos de qualidade entram em conflito;
- **Registrar as Regras para priorizar os Requisitos**. Registrar o conhecimento do especialista para definir, dentro de um domínio, qual o limiar para um padrão de *trade-off* ser ativado e precisar ser resolvido.
- **Registrar Arquiteturas de Referência**. Registrar a visão lógica da arquitetura, bem como de seus elementos arquiteturais formados por componentes e conectores.
- **Registrar as Regras para Calcular a Relevância Arquitetural**. Registrar o conhecimento do especialista para definir, dentro de um domínio, como será calculado a relevância arquitetural dentro do processo de recomendação.

Figura 11 – Atividade Preparar Ambiente



Fonte: Elaborada pelo autor.

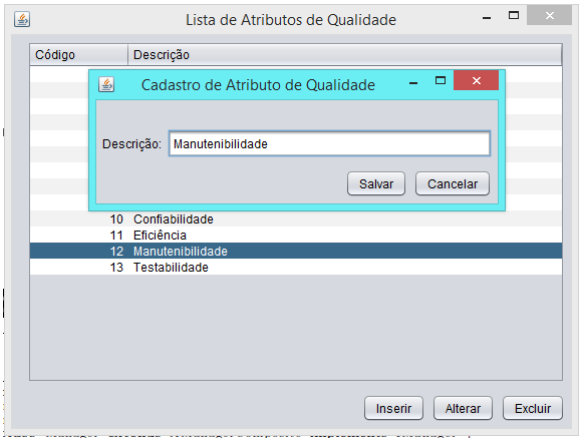
5.1 Registrar os Atributos de Qualidade (Atividade 1.1)

A definição dos atributos de qualidade que serão levados em consideração durante o ciclo de execução do processo aparece como atividade que deve ser executada *a priori*. Cabe ao especialista realizar esta definição, como também priorizar quais informações serão armazenadas sobre cada um destes atributos. Essa possibilidade de customização dos requisitos vai permitir que sejam disponibilizados atributos que efetivamente tenham relação com os estilos arquiteturais passíveis de recomendação.

Uma vez selecionados os atributos de qualidade que serão levados em consideração, eles precisam ser persistidos no repositório do conhecimento. A Figura 12 exemplifica como este procedimento acontece. Uma lista com todos os atributos já registrados é exibida ao interessado, onde funções de inclusão, alteração e exclusão estão disponíveis. Neste momento, ao utilizar a opção de inclusão ou alteração, apenas a descrição do atributo é solicitada. No contexto desse trabalho, foram cadastrados na ferramenta os requisitos de qualidade descritos na norma ISO 9126.

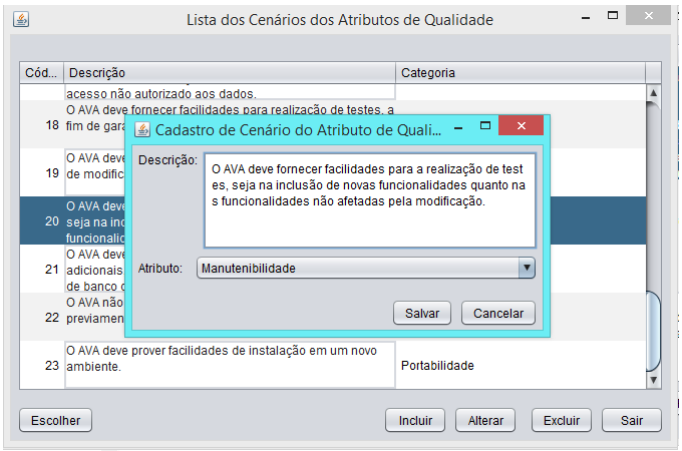
Com o propósito de realizar a captura das preferências dos interessados, minimizando a ocorrência de equívocos na escolha dos atributos, foi adotado um mapeamento da linguagem natural e dos requisitos de qualidade, em que os especialistas relacionariam cenários a cada atributo de qualidade, visando facilitar o entendimento para o usuário e dispensando dele conhecimento sobre detalhes ou termos técnicos relativos ao domínio ou à tecnologia.

Figura 12 – Cadastro de um Atributo de Qualidade



Fonte: Elaborada pelo autor

Figura 13 – Cenário relacionado a um Atributo de Qualidade



Fonte: Elaborada pelo autor

Cada cenário registrado é exibido durante o processo de coleta de requisitos, onde cada ponto associado ao cenário estará automaticamente associado ao atributo de qualidade. A Figura 13 exibe um exemplo de cadastro de cenário e sua associação ao atributo de qualidade. Podemos verificar que, um mesmo atributo de qualidade pode estar associado a vários cenários fazendo com que as prioridades sobre determinado atributo de qualidade possam ser obtidas de mais de uma fonte.

5.2 Registrar os Padrões de *Trade-off* (Atividade 1.2)

Uma vez que os atributos de qualidade estejam registrados, é necessário identificar os potenciais *trade-offs* entre eles. Os potenciais *trade-offs*, chamados padrões, consistem em cenários bem conhecidos em que dois ou mais atributos de qualidade entram em conflito.

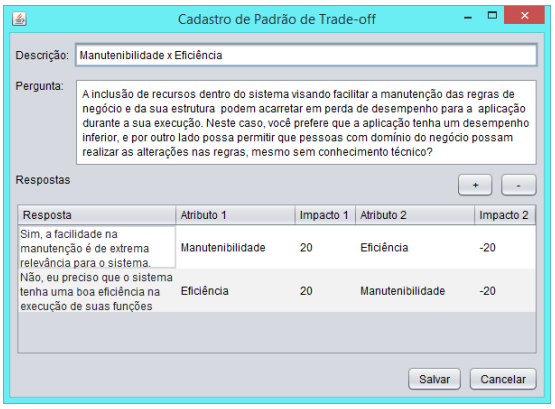
Por exemplo, em uma aplicação Web, o arquiteto de software notou que quando o número de requisições concorrentes aumenta, a aplicação torna-se lenta. Este cenário poderia gerar um padrão exemplificando um caso quando *throughput* e performance entram em conflito.

A identificação destes padrões de *trade-offs* pode ser feita, por exemplo, com base em experiências anteriores e na Literatura especializada. As fontes de informação que serão utilizadas para elaboração dos padrões de *trade-off* ficará a cargo do especialista do domínio que, a seu critério, poderá fazer uso de uma das opções citadas anteriormente, extrair informações das duas, ou até mesmo, se julgar importante, utilizar uma terceira fonte.

Cada padrão de *trade-off* será acompanhado de um pergunta. Para cada pergunta, mais de uma resposta poderá estar associada, onde cada resposta terá impacto sobre 1 ou 2 atributos de qualidade, seja ele positivo ou negativo. O limiar de prioridade entre os requisitos, que será utilizado para a ativação de cada padrão de *trade-off*, estará condicionada às regras definidas pelo especialista na **Atividade 1.3**. O nível de interesse entre dois ou mais atributos definirá a sua ativação, porque quanto mais próximos forem os níveis, maior será a chance de ocorrência do conflito.

Um exemplo de cadastro de um padrão de trade-off é apresentado na Figura 14, onde os requisitos de Eficiência e Manutenibilidade estão em conflito. No contexto deste trabalho, foram cadastrados 10 padrões de trade-off, gerados a partir de conflitos clássicos relatados na Literatura e de experiências anteriores (EGYED; GRUNBACHER, 2004; HENNINGSSON; C.WOHLIN, 2002).

Figura 14 – Cadastro de Padrão de *Trade-off*



Fonte: Elaborada pelo autor

5.3 Registrar Regras para Priorizar Requisitos (Atividade 1.3)

Um modelo baseado em regras foi adotado para armazenar as decisões sobre o processo de engenharia de requisitos. As definições dos padrões de *trade-offs* são persistidas dentro

de um banco de dados. Já as regras que definirão o limiar de quando um conflito irá ocorrer serão persistidas em forma de regras de produção.

A fim de realizar a inferência sobre quais requisitos priorizar, um motor de sistema especialista (SE) faz uso das regras de produção, bem como do conhecimento retido sobre os padrões de *trade-offs* já especificados na **Atividade 1.2**. No contexto deste trabalho, a estrutura das regras seguirá o modelo proposto pelo motor de sistema especialista Drools (DROOLS, 2014).

Tais regras visam verificar quando 2 ou mais atributos de qualidade entrarão em conflito. Estando em conflito, a depender do nível de interesse sobre cada atributo, o SE deve-se basear em suas regras para realizar a priorização de um requisito em detrimento de outro, de maneira automática. Quando isso não for possível, devem existir regras que permitam uma interação com os interessados, apresentando o padrão de *trade-off* em forma de questionamento, a fim de direcionar, através das respostas fornecidas, ajustes na prioridade de um requisito sobre os outros que estão em conflito.

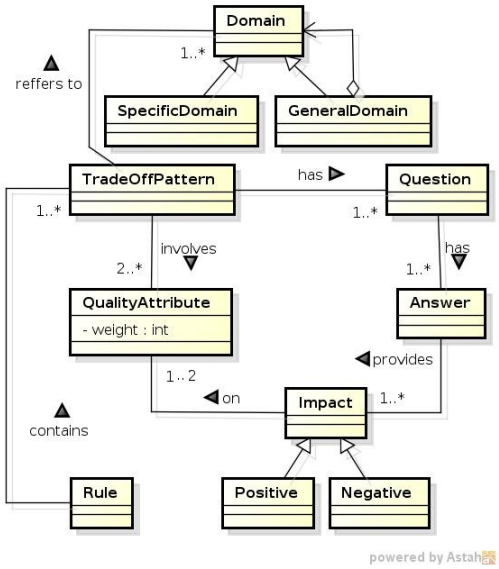
Por exemplo, no contexto do cenário de *trade-off* da hipotética aplicação Web, uma questão que poderia ser útil seria: "Você limitaria o número de requisições concorrentes a fim de prover uma melhor qualidade dos serviços em nível de performance?". Uma resposta afirmativa a tal questionamento implica em favorecer o desempenho em prol de uma limitação no *throughput*. Cada cenário de *trade-off* deve ser associado a pelo menos uma questão e cada questão deve ter um impacto associado às respostas, que pode aumentar ou diminuir o peso de determinado atributo de qualidade. A Figura 15 mostra o meta-modelo para representação de regras. Por uma questão de simplificação do diagrama, a maioria dos atributos foram omitidos. É importante enfatizar que a associação entre **TradeOffPattern** e o **Domain** permite a definição de padrões de *trade-off* de qualquer domínio, seja ele geral ou específico.

A Figura 16 exemplifica como podem ser estruturadas as regras relativas aos *trade-offs*. Neste caso, a regra verifica se dois atributos de qualidade (Eficiência e Manutenibilidade) estão presentes nas preferências do interessado e se os mesmos violam o limiar estabelecido pelo especialista (15). As condições sendo satisfeitas fazem com que o padrão de *trade-off* seja ativado. Assim, uma pergunta será feita ao usuário, em que a resposta informada influenciará nas prioridades dos atributos envolvidos.

5.4 Registrar Arquiteturas de Referência (Atividade 1.4)

Durante esta atividade é definida a visão lógica da arquitetura, que segue estilos arquiteturais e contém os elementos arquiteturais formados por componentes e conectores. Para cada estilo arquitetural, o arquiteto deve fornecer informações sobre o nível de satisfação dos atributos de qualidade. O arquiteto deve definir um percentual que expressa a forma como a arquitetura satisfaz cada atributo de qualidade.

Figura 15 – Meta Modelo para Representação de Regras



Fonte: Elaborada pelo autor

Figura 16 – Exemplo de Regra de Trade-off

```
rule "Trade-off Manutenibilidade x Eficiência"
when
  quality: ArchitectureQualityAttribute( (weight > 0) && hasDescription("Manutenibilidade"))
  quality1: ArchitectureQualityAttribute( (weight > 0) && hasDescription("Eficiência") && inConflict(quality.getWeight(),15))
  question: TradeoffQuestion(id == 4)
  qm: TradeoffQuestionMaker()
then
  qm.AskQuestion(question,quality,quality1);
  update(quality);
  update(quality1);
end
```

Fonte: Elaborada pelo autor

A definição da visão lógica das arquiteturas de referência é utilizada para recomendar para os interessados os componentes que farão parte de cada arquitetura, favorecendo a implementação através da posterior geração de esqueleto de código. Para tal, é necessário que o especialista identifique na arquitetura de referência como os componentes estão organizados, assim como o relacionamento entre eles. Tal recurso permitirá ao especialista a possibilidade de evoluir esta arquitetura à medida que for necessário, além de ajustar o detalhamento em nível de granularidade desejado, podendo chegar até as especificações dos métodos de cada interface provida e requerida.

A organização destes elementos é feita de maneira hierárquica, identificando as possíveis dependências entre eles. A descrição da visão lógica de cada arquitetura de referência é persistida em um banco de dados, assim como a informação referente ao estilo arquitetural adotado. Versões diferentes de um estilo arquitetural podem ser registradas no sistema, bem como as respectivas explicações sobre as decisões de projeto. Assim, a **Atividade 01.4** pode ser vista como a fase de aquisição do conhecimento em termos de

registrar as características de cada estilo arquitetural e as decisões de projeto relacionadas ao projeto arquitetural. O nível de detalhamento da informação influenciará diretamente o processo de geração de código (**Atividade 4**), pois quanto mais detalhada estiver a arquitetura mais próximo do projeto arquitetural estará o código apresentado.

5.5 Registrar as Regras para Calcular a Relevância Arquitetural (Atividade 1.5)

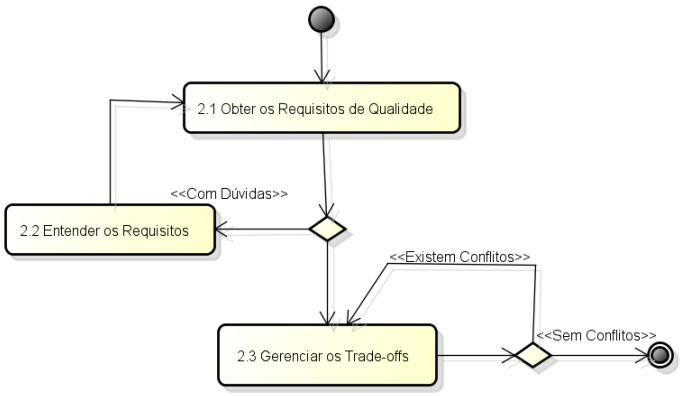
Além de descrever cada arquitetura em si, é necessário definir regras que serão utilizadas na escolha da arquitetura de referência mais adequada, a partir dos requisitos de qualidade especificados. Assim como as regras de negócio para priorizar os requisitos, as regras para calcular a relevância de cada estilo arquitetural registrado durante o processo de recomendação seguem o modelo baseado em regras de produção. Estas regras também estarão acessíveis e modificáveis sem que seja necessário mudança no código fonte da solução.

Informações sobre cada estilo arquitetural poderão ser levadas em consideração na definição da relevância, bem como os dados coletados sobre as preferências dos interessados. Para tal, foi adotada uma visão centrada nos requisitos, focada particularmente nos requisitos de qualidade. O SE adotado é flexível ao ponto de permitir que dentro da memória de trabalho sejam inseridos os objetos utilizados na aplicação, fazendo com que o especialista possa ter em mãos todo o conhecimento retido na solução para determinar como este cálculo será realizado.

6 ESPECIFICAR OS REQUISITOS DE QUALIDADE (ATIVIDADE 2)

A especificação dos requisitos de qualidade consiste em coletar os atributos de qualidade esperados pelos interessados do sistema. Para obter e gerenciar tais informações, verificando a existência de conflito entre os requisitos informados, será utilizado o conhecimento retido durante a atividade de preparação do ambiente (**Atividade 1**). A Figura 17 mostra mais detalhadamente o funcionamento desta atividade, a qual foi dividida em 3 subatividades:

Figura 17 – Atividade Especificar os Requisitos de Qualidade



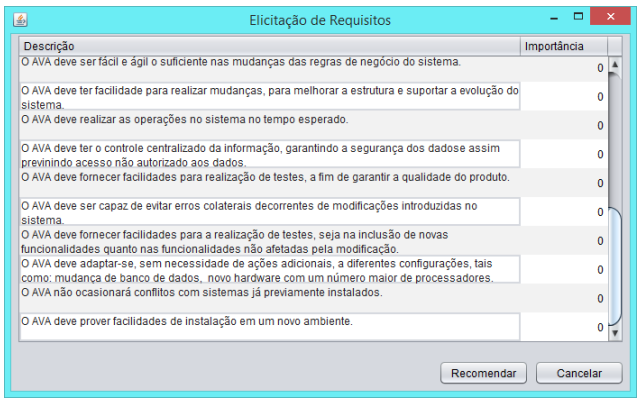
Fonte: Elaborada pelo autor

- **Obter os Requisitos de Qualidade**. Consiste em obter do interessado os requisitos de qualidade e suas prioridades.
- **Entender os Requisitos**. Consiste em fornecer detalhes adicionais, a fim de certificar que os atributos de qualidade foram escolhidos de forma correta.
- **Gerenciar os *Trade-offs***. Consiste em identificar e gerenciar os *trade-offs* entre os requisitos de qualidade.

6.1 Obter os Requisitos de Qualidade (Atividade 2.1)

Esta atividade consiste em coletar do interessado os requisitos de qualidade que são esperados para o seu sistema. Os requisitos de qualidade cadastrados no sistema (**Atividade 1.1**) serão listados e apresentados para o interessado, utilizando o mapeamento para linguagem natural. Cada item listado corresponde a um atributo de qualidade e um atributo de qualidade poderá estar presente em mais de um item na lista.

Figura 18 – Exemplo da Coleta dos Requisitos de Qualidade



Fonte: Elaborada pelo autor

A Figura 18 ilustra o processo de coleta dos requisitos de qualidade. Os cenários, registrados e vinculados aos atributos de qualidade são apresentados para que o interessado possa informar a importância de cada cenário no contexto da sua aplicação. Por padrão, todos os cenários vêm com o mesmo peso e iguais a zero. O usuário possuirá uma quantidade de 100 pontos para atribuir entre os cenários, de acordo com a importância atribuída a eles. Todos os pontos disponibilizados deverão ser distribuídos obrigatoriamente entre os itens apresentados, não sendo aceitas faltas ou sobras de pontos.

Uma vez que o interessado realize a confirmação das prioridades dos requisitos, elas serão passadas para a memória de trabalho do sistema especialista responsável pelo gerenciamento dos *trade-offs* (**Atividade 2.3**).

6.2 Entender os Requisitos (Atividade 2.2)

Durante o processo de coleta de requisitos (**Atividade 2.1**) podem ocorrer dúvidas do usuário com relação a algum requisito específico exibido na lista. É importante então fornecer informações extras sobre cada requisito de qualidade, com o propósito de aumentar a compreensão do requisito e favorecer a escolha adequada dos atributos de qualidade. Como exemplo, o sistema poderia exibir uma descrição detalhada do atributo de qualidade, ou mesmo ilustrar um exemplo ou exibir informações sobre os seus pontos fortes e fracos.

Esta atividade pode ser ativada a qualquer momento durante a coleta de requisitos (**Atividade 2.1**) e quantas vezes o usuário achar necessário, até que as informações possibilitem o entendimento mais adequado sobre o(s) atributo(s) escolhidos.

6.3 Gerenciar os *Trade-offs* (Atividade 2.3)

Uma vez coletados os requisitos esperados pelo usuário para sua aplicação e seus respectivos pesos/prioridades, é necessário agora gerenciar os *trade-offs*. Assim, a lista das preferências do interessado é passada para o Sistema Especialista que realizará este gerenciamento. Ele será baseado no conhecimento obtido previamente junto aos especialistas, tanto na definição dos padrões como nas regras de ativação destes padrões. Para isto, é necessário seguir os seguintes passos:

1. **Identificar os padrões de *trade-off*** entre os atributos de qualidades selecionados;
2. **Enquanto** existirem conflitos

   Se o conflito pode ser resolvido automaticamente

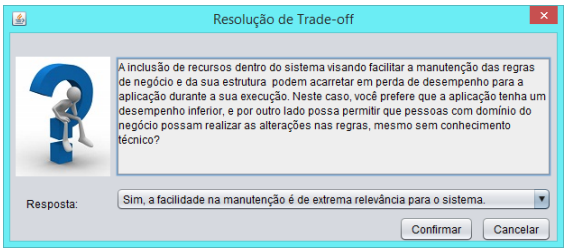
   Resolver o conflito automaticamente

   Senão

   Fazer perguntas relativas ao padrão de *trade-offs* identificado;

   Computar os impactos das repostas e atualizar o atributo do peso (positivo ou negativo) dos requisitos envolvidos;
3. **Ordenar os atributos de qualidade** baseados no seu respectivo peso;

Figura 19 – Exemplo da Interação para resolução de *Trade-off*



Fonte: Elaborada pelo autor

A Figura 19 exemplifica como acontecerá a interação com o interessado quando a regra, apresentada na Figura 16, ativar o padrão de *trade-off*. Assim que o padrão de *trade-off* for ativado, a pergunta, associada ao *trade-off*, é apresentada ao interessado, solicitando que o mesmo faça a opção por uma resposta. Tal resposta, atualizará os pesos dos atributos de qualidade envolvidos. Este processo interativo acontecerá até que não existam mais conflitos a serem resolvidos. Mas no caso de se manter o *trade-off* entre 2 ou mais requisitos mesmo depois do processo de resolução mencionado, o arquiteto de software deve ser avisado do problema e resolvê-lo manualmente.

7 PROJETAR A ARQUITETURA DE SOFTWARE (ATIVIDADE 3)

Uma vez que as preferências dos interessados foram coletadas e os *trade-offs* entre os requisitos foram resolvidos, a atividade de projeto arquitetural é iniciada. Ela é responsável pela recomendação da arquitetura, seu refinamento e, por fim, pela avaliação. A depender do resultado da avaliação, os estilos arquiteturais armazenados poderão sofrer melhorias, ou até mesmo um novo estilo pode ser criado.

A Figura 20 detalha o funcionamento da atividade **Projetar a Arquitetura de Software** que está dividida em 4 subatividades:

- **Recomendar a Arquitetura de Referência baseada nos Atributos de Qualidade**. Consiste em recomendar a arquitetura do sistema baseada nas preferências dos interessados;
- **Refinar a Arquitetura**. Consiste no refinamento da arquitetura recomendada, permitindo ao interessado adicionar ou suprimir algum elemento da arquitetura;
- **Avaliar a Arquitetura**. Consiste na validação da arquitetura, verificando se a mesma é uma arquitetura válida.
- **Atualizar Base de Conhecimento de Arquiteturas de Referência**. Consiste em registrar todas as variações arquiteturais definidas pelo especialista.

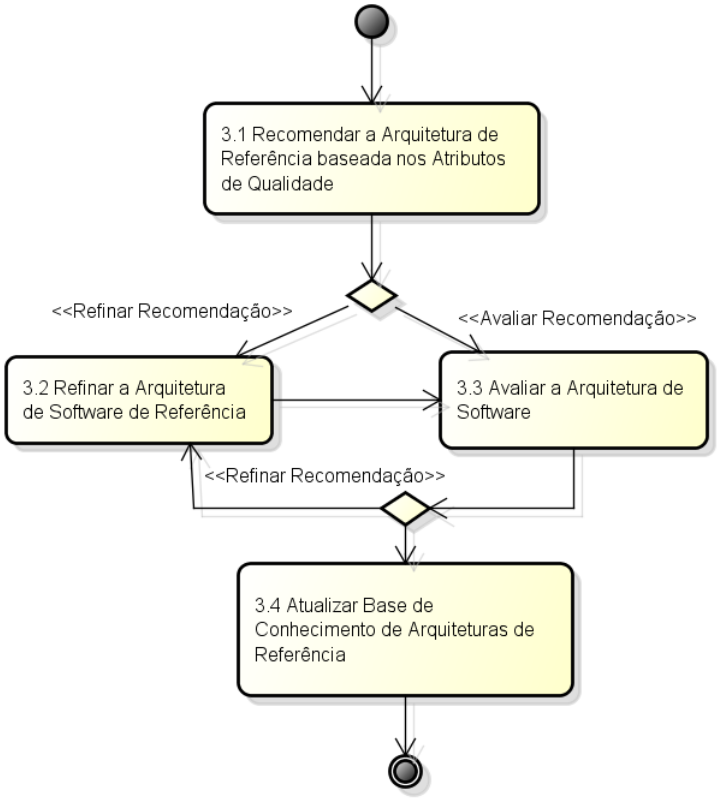
7.1 Recomendar a Arquitetura de Referência baseada nos Atributos de Qualidade (Atividade 3.1)

Esta atividade é responsável pela recomendação da arquitetura do sistema, recomendação esta baseada nos requisitos de qualidade. Como entrada, o sistema de recomendação recebe uma lista com as preferências definidas pelos interessados, resultante da execução da atividade de especificação dos requisitos (**Atividade 2**).

Um framework de SE é utilizado para encontrar um estilo arquitetural considerado adequado, entre aqueles que estão cadastrados na base de conhecimento. Vale salientar que, conforme mencionado na Seção 2.2.1, um estilo arquitetural pode ser considerado heterogêneo, combinando mais de um estilo arquitetural clássico.

Após a recomendação dos estilos arquiteturais, as informações sobre o estilo arquitetural selecionado, assim como sua relação com os atributos de qualidade também são passadas para o SE. Com todas as informações inseridas na memória de trabalho e o

Figura 20 – Atividade Projetar a Arquitetura de Software



Fonte: Elaborada pelo autor

conhecimento do especialista armazenado em forma de regras de produção, o SE se encarrega de efetuar o cálculo da relevância de cada estilo, levando em consideração o *ranking* envolvendo os requisitos do sistema.

Quando uma arquitetura é recomendada, um nível de confiança é fornecido. O nível de confiança é o percentual de confiança que a arquitetura recomendada de fato satisfaz os requisitos do sistema. Para elucidar ainda mais, a seguir o funcionamento do sistema de recomendação é apresentado na forma de algoritmo:

1. **Alimentar a memória de trabalho** com as preferências do interessado;
2. **Para** cada estilo arquitetural armazenado;

   Para cada atributo de qualidade;

   Calcular a relevância do atributo com base nas preferências;

   Atualizar a relevância total do estilo arquitetural;

   Reordenar o *ranking* de estilos arquiteturais;
3. **Exibir o resultado** da recomendação;

7.2 Refinar a Arquitetura de Software de Referência (Atividade 3.2)

Se o arquiteto desejar melhorar o nível de confiança de um estilo arquitetural recomendado, a arquitetura de referência deve ser refinada na **Atividade 3.2**. Esta atividade não é atualmente apoiada por ferramenta e deve ser executada manualmente pelo arquiteto. Embora a execução seja manual, no momento do cadastro do refinamento no sistema, é solicitado ao arquiteto uma explicação contendo decisões de projeto que justifiquem as diferenças entre a arquitetura de referência original e a sua versão refinada. O objetivo desta atividade é registrar estas mudanças e usá-las para prevenir a **Vaporização do Conhecimento Arquitetural** e melhorar a qualidade da documentação da arquitetura de software.

À medida que o refinamento arquitetural acontece, é possível detalhar os elementos arquiteturais em sub-componentes internos para realização de requisitos funcionais. Para isso, pode-se utilizar estratégias de agrupamento funcional. Um exemplo de tal agrupamento ocorre com o processo UML Components (CHEESMAN; DANIELS, 2001), que sistematiza a descoberta de componentes a partir dos casos de uso do sistema. A ferramenta desenvolvida permite que o refinamento seja realizado em diversos níveis de detalhe, podendo chegar até ao detalhamento das interfaces e a implementação dos seus métodos. Quanto mais detalhada a definição da arquitetura, mais detalhado será o código a ser gerado automaticamente.

7.3 Avaliar a Arquitetura de Software (Atividade 3.3)

A fim de avaliar a arquitetura recomendada, esta atividade usa o método ATAM para medir quanto a arquitetura de software satisfaz os atributos de qualidade desejado. Como apresentado na Seção 2.2.2, o ATAM é um método baseado em cenários, que tem por objetivo avaliar a consequência das decisões arquiteturais em termos de cenários específicos da aplicação, envolvendo cada um dos atributos de qualidade desejado.

7.4 Atualizar Base de Conhecimento de Arquiteturas de Referência (Atividade 3.4)

Uma vez que a arquitetura de software foi refinada **(Atividade 3.2)** ou avaliada no contexto específico dos cenários da aplicação **(Atividade 3.3)**, essas informações adicionais devem ser utilizadas para alimentar a base de conhecimento do sistema. A **Atividade 3.4**, consiste no cadastro ou atualização de todas as variações arquiteturais definidas pelo especialista para um domínio específico, assim como cenários de avaliação, que serão utilizados para refinar a recomendação em utilizações posteriores. Estes estilos serão baseados em experiências anteriores de especialistas sobre esse domínio e podem ser complementados com a literatura especializada.

Para cada estilo arquitetural, o arquiteto deve fornecer informações sobre o nível de satisfação dos atributos de qualidade. Para isso, o resultado da avaliação utilizando ATAM é considerada primordial, sendo essa a principal ferramenta utilizada pelo arquiteto para justificar a atribuição do nível de satisfação de cada atributo de qualidade. Dependendo dos resultados obtidos durante a avaliação, o arquiteto deve definir um percentual que expressa a forma como a arquitetura satisfaz cada atributo de qualidade.

Para descrever os estilos arquiteturais, o arquiteto deve registrar os elementos arquiteturais (componentes e conectores), bem como suas interfaces providas e requeridas e as dependências entre elas. Conforme apresentado na Seção 7.2, versões diferentes de um estilo arquitetural podem ser registradas no sistema, bem como as respectivas explicações sobre as decisões de projeto que justificam as diferenças. Assim, a **Atividade 3.4** pode ser vista como a fase de aquisição de conhecimento em termos de registrar / atualizar as características de cada estilo arquitetural e as decisões de projeto relacionadas ao projeto arquitetural.

8 GERAR O CÓDIGO-FONTE ESTRUTURAL DO SISTEMA (ATIVIDADE 4)

A última etapa do processo de desenvolvimento que é apoiada pela ferramenta proposta é a geração do código-fonte estrutural do sistema. Com base na arquitetura recomendada (**Atividade 3**) e seus elementos arquiteturais, é gerado o código-fonte estrutural do sistema, seguindo uma abordagem de desenvolvimento baseada em componentes. É adotado o modelo COSMOS*, apresentado na Seção 2.7, favorecendo assim a rastreabilidade entre arquitetura de software e o código-fonte. As principais características do código gerado são (SILVA-JUNIOR, 2003): (1) rastreabilidade entre arquitetura de software e o código-fonte do sistema; (2) uso de conectores explícitos, que proporcionam baixo acoplamento entre os componentes; (3) alta granularidade dos componentes, que combinado ao baixo acoplamento favorecem o reuso em larga escala; e (4) permitir que os componentes sejam mais flexíveis à mudanças, com uma maior capacidade de adaptação e reuso. A linguagem de programação adotada foi a linguagem JAVA, na qual todo o código fonte será gerado. O nível de granularidade deste código fonte gerado dependerá fortemente do nível de detalhamento da arquitetura que foi registrada na ferramenta, podendo inclusive gerar as assinaturas de todos os métodos dos componentes, desde que estejam devidamente especificadas no estilo arquitetural.

A Figura 21 mostra como funcionará esta atividade, que será dividida em 3 subatividades:

- **Gerar o Código dos Componentes**. Consiste na geração do código fonte dos componentes do sistema e seus elementos hierarquicamente dependentes seguindo o modelo de implementação COSMOS*;
- **Gerar o Código dos Conectores**. Consiste na geração do código fonte dos conectores, que serão identificados de acordo com os elementos presentes na arquitetura e as associações entre eles usando o modelo COSMOS*;
- **Gerar o Código da Configuração Arquitetural**. Consiste na geração do código da Classe que será responsável por inicializar o sistema, realizando a instanciação dos componentes, conectores e realizando também a associação efetiva entre os componentes através dos conectores (*binding*).

8.1 Gerar o Código dos Componentes (Atividade 4.1)

A geração de código inicia com a geração do código dos componentes da arquitetura. Para cada componente arquitetural presente no primeiro nível da arquitetura, a ferramenta irá realizar o mapeamento destes componentes direcionado para o modelo

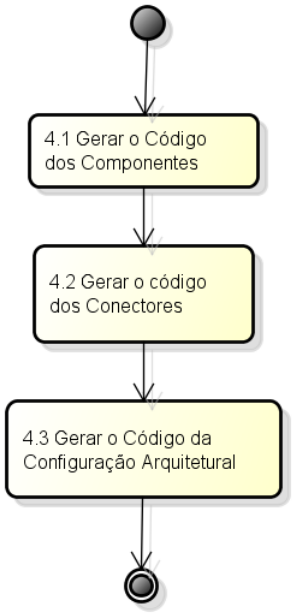


Figura 21 – Atividade Gerar o Código-Fonte Estrutural do Sistema

Fonte: Elaborada pelo autor

de pacotes utilizado no COSMOS*. Conforme mencionado na Seção 2.7 e mostrado na Figura 2, este pacote será composto por 2 subpacotes:

- **Pacote de Especificação**, cujos elementos possuem visibilidade pública (ver Seção 2.7.1). Este pacote consiste na especificação dos serviços que o componente ofertará e os que ele necessitarás. As **interfaces providas** e **interfaces requeridas** serão geradas com base na configuração definida pelo especialista do estilo arquitetural registrado. Já a especificação dos métodos será feita a partir do nível de definição realizado na arquitetura, ou seja, se o interessado conseguiu definir na especificação da arquitetura de referência os métodos de cada componente, a ferramenta gerará também as especificações de cada um destes métodos. Além destas interfaces providas e requeridas, será especificada também neste subpacote a interface **IManager**, que será responsável por fornecer informações sobre as interfaces providas e requeridas do pacote.

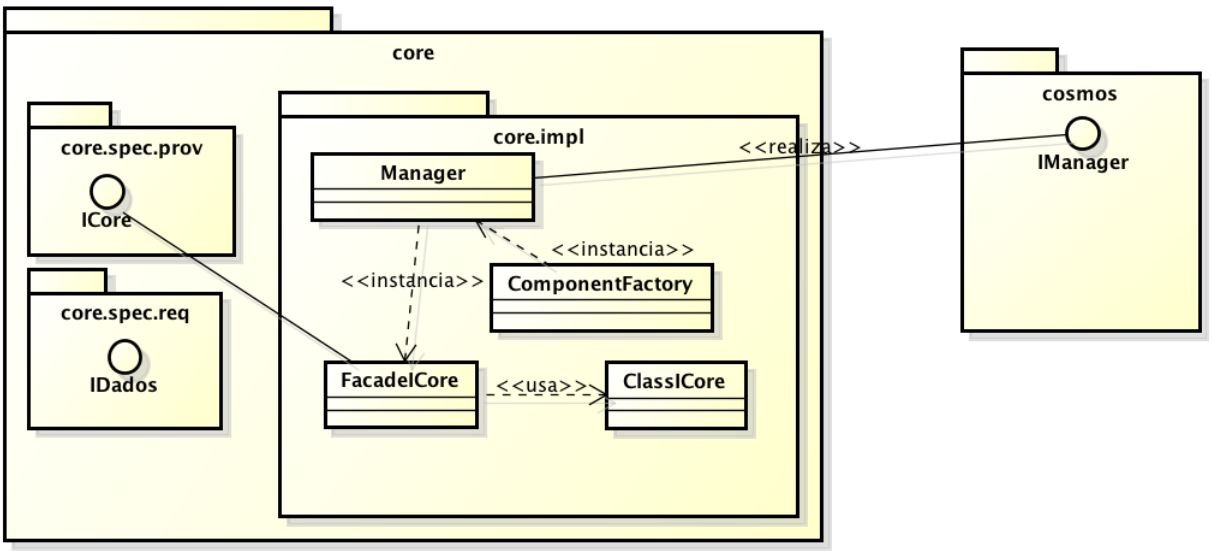
- **Pacote de Implementação** (ver Seção 2.7.2). Consiste na descrição da implementação das funcionalidades do componente. Para cada interface provida pelo componente, será gerado uma classe que realizará a efetiva implementação dos métodos e uma classe de façade que fará o redirecionamento entre a interface e a implementação da classe. Completando o conjunto de classes presentes neste pacote, também

serão geradas as classes **Manager**, que é a implementação da interface **IManager** e **ComponentFactory** que será responsável pela instanciação dos componentes.

A Figura 22 exemplifica como ficará a estrutura de um componente gerado no modelo COSMOS*, composto por uma interface provida e uma interface requerida depois de ter o seu código fonte gerado. Ela mostra todas as classes que serão geradas no caso de um componente simples, gerando um total de 6 classes. Podemos imaginar a implementação de um componente mais complexo, ou até mesmo quando tivermos a necessidade de gerar vários componentes seguindo este mesmo modelo. Será uma tarefa que tomará um tempo considerável para construir e garantir que o código gerado seja compatível com o modelo, ou por ser feito manualmente ocorrer na existência de falhas durante a especificação e implementação do componente, por ausência de algum item importante ou até mesmo falha na codificação, fazendo com que o código não consiga refletir efetivamente a arquitetura recomendada pela ferramenta.

Estes fatos atestam a importância que esta subatividade tem durante o processo de desenvolvimento do sistema, que através do processo de automatização parcial da codificação garantirá que o código do componente arquitetural gerado siga fielmente o modelo COSMOS* adotado, garantindo assim a agilidade na codificação do modelo e na aderência entre o código gerado e a arquitetura recomendada, trazendo também todas as outras vantagens inerentes ao modelo, que podem ser vistas mais detalhadamente na Seção 2.7.

Figura 22 – Componente COSMOS e suas classes



Fonte: Elaborada pelo autor

Os componentes gerados pela ferramenta basicamente possuirão a seguinte estrutura:

- **Interfaces providas** de acordo com os serviços ofertados pelo componente;

- **Interfaces requeridas** de acordo com a necessidade do componente para execução dos serviços ofertados por ele;
- Classe **Facade** para cada interface provida;
- **Classe** que implementa efetivamente cada interface. Será utilizada pelo **Facade** para realizar a associação com a **interface provida**.
- Classe **Manager** responsável por configurar as interfaces e instanciar os **Facades**;
- Classe **ComponentFactory**, responsável por instanciar a classe **Manager**;

8.2 Gerar o Código dos Conectores (Atividade 4.2)

Uma vez gerado o código fonte dos componentes, o próximo passo é a geração do código dos conectores que, conforme mencionado na Seção 2.7.3, tem como papel realizar a interação entre dois ou mais componentes. Para realizar a comunicação entre eles, são utilizadas as interfaces providas e requeridas dos próprios componentes que serão interligados. Em outras palavras, um conector implementa ao menos uma interface requerida de um componente e utiliza ao menos uma interface provida de outro componente.

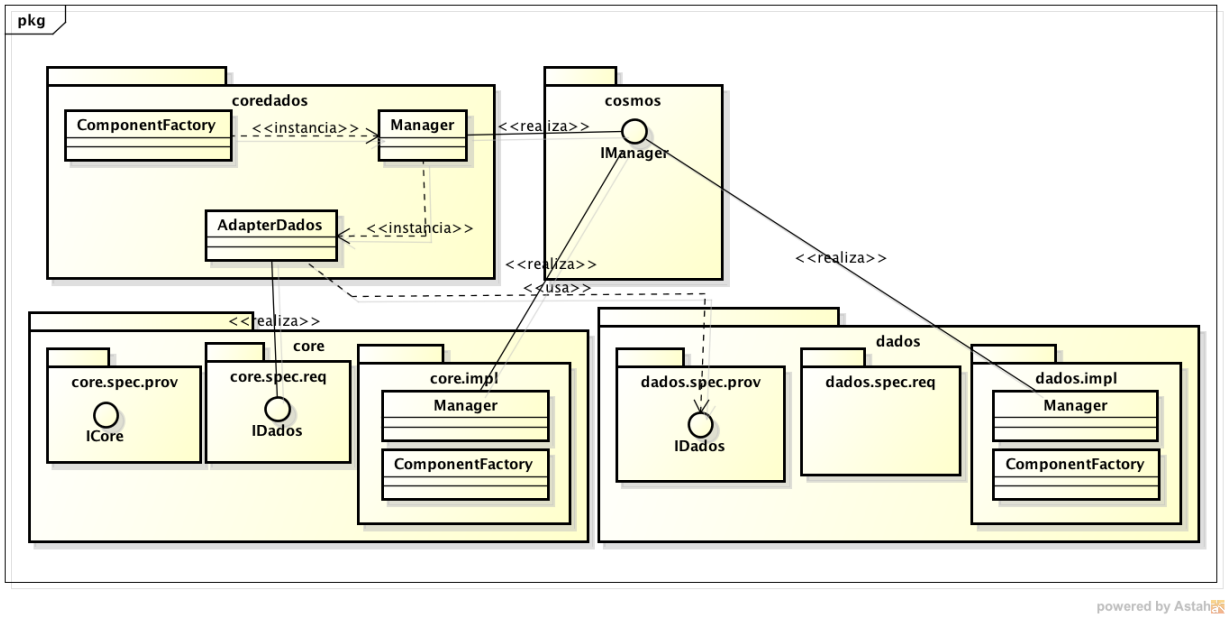
O primeiro passo nesta subatividade é realizar a coleta dentro da arquitetura de referência de quais conectores serão gerados, uma vez que cada arquitetura de referência define todas as associações entre os componentes. Similar ao componente arquitetural, o conector também será mapeado em um pacote, sendo este pacote mais simples do que o componente arquitetural e similar a estrutura do seu pacote de implementação.

Estes conectores precisam existir porque as interfaces providas e requeridas não podem ser conectadas diretamente, isto ocorre porque os componentes são autônomos e o nome completo das interfaces são diferentes (incluindo o *namespace*). Imaginemos um componente **Core** que possui uma interface provida **core.spec.req.ICore** e um componente **Dados** que possui uma interface requerida **dados.spec.req.ICore**. Apesar delas possuírem as mesmas especificações, elas possuem nomes completos diferentes, o que inviabiliza a conexão direta. Para realizar esta conexão entre as interfaces incompatíveis, o conector implementa o padrão de projeto **Adapter**, permitindo a conexão indireta entre estas interfaces (GAYARD; RUBIRA; GUERRA, 2008).

A Figura 23 exemplifica como ficará estruturado um conector gerado pela ferramenta, de acordo com o modelo COSMOS*. Neste exemplo, chamamos o conector de **Dados-Core** e podemos verificar também como é a associação entre as classes para que aconteça efetivamente esta comunicação entre os componentes. Na Figura 23 este conector referencia as interfaces definidas nos componentes **Dados** e **Core**, fazendo com que haja um grande acoplamento entre o conector e os componentes. Por outro lado, permite que haja a comunicação entre os componentes **Dados** e **Core**, sem que necessite de um compo-

nente referenciando interfaces definidas no outro componente, fazendo assim com que os componente tenham um baixo acoplamento entre si.

Figura 23 – Conector COSMOS* e suas Classes



Fonte: Elaborada pelo autor

Os conectores gerados pela ferramenta basicamente possuirão a seguinte estrutura básica:

- Classe **Adapter** que irá adaptar as interfaces providas e requeridas dos componentes antes incompatíveis. Esta classe que implementará o padrão de projeto **Adapter** dentro do componente;
- Classe **Manager** responsável por configurar as interfaces e instanciar a classe **Adapter**;
- Classe **ComponentFactory**, responsável por instanciar a classe **Manager**;

8.3 Gerar o Código da Configuração Arquitetural (Atividade 4.3)

Com o código dos componentes e conectores gerados, resta agora como etapa final no processo de geração do código a geração da classe principal do sistema. Esta classe será responsável por realizar a configuração arquitetural da arquitetura, constituindo a instanciação dos componentes (objetos das classes **Manager**) e conectores. Deve ser realizado também a ligação entre componentes e conectores (*binding*), através das interfaces requeridas e providas de cada um.

A Figura 24 mostra um exemplo de uma classe principal que se pretende gerar com a ferramenta, onde o sistema possui 2 componentes e um conector que realiza a conexão entre eles. Na figura vê-se a instanciação de cada objeto do tipo **Manager** dos componentes e no método principal a configuração das **interfaces requeridas** do conector e do componente Core. Enquanto o conector é ligado ao componente Dados, o componente Core é ligado ao conector fechando o ciclo.

Figura 24 – Exemplo de Classe do Configurador do Sistema

```
1  class ArchConfigure{
2      private static cosmos.IManager managercoredados =
3        coredados.ComponentFactory.createInstance();
4      private static cosmos.IManager managerdados =
5        dados.impl.ComponentFactory.createInstance();
6      private static cosmos.IManager managercore =
7        core.impl.ComponentFactory.createInstance();
8
9   public static void main(String args[]){
10
11        managercoredados.setRequiredInterface("IDados",
12           managerdados.getProvidedInterface("IDados"));
13
14        managercore.setRequiredInterface("IDados",
15           managercoredados.getProvidedInterface("IDados"));
16
17   }
18
19 }
```

Fonte: Elaborada pelo autor

Uma vez que a classe principal foi gerada, o sistema estará pronto para que o desenvolvedor possa utilizar os serviços dos componentes que foram gerados e configurados para implementar as funcionalidades necessárias para a conclusão do sistema. Até este ponto, a ferramenta terá ajudado efetivamente ao desenvolvedor a montar a estrutura necessária para que o código seja consistente com a arquitetura recomendada, fazendo com que o seu trabalho ganhe em agilidade, reduzindo a probabilidade de falhas que possam vir a ocorrer na especificação dos componentes que comporão o sistema.

9 AVALIAÇÃO DA SOLUÇÃO

9.1 Configuração

O objetivo desta avaliação é verificar se, de fato, a solução proposta contribui para melhoria no processo de desenvolvimento de software centrado na arquitetura, apoiando as atividades de engenharia de requisitos, projeto arquitetural e geração de código. Esta seção está organizada como segue: a Seção 9.1.1 coloca as sete questões de pesquisa que conduzem a avaliação. A Seção 9.1.2 apresenta os sujeitos que foram escolhidos para participar da avaliação. A Seção 9.1.3 indica em que domínio foi instanciada a solução, e como o conhecimento foi armazenado para a execução da avaliação e finalmente a Seção 9.1.4 apresenta os detalhes sobre a execução do experimento.

9.1.1 Planejamento

A Questão de Pesquisa [1](RQ)geral que pretendemos responder é:

RQ. A solução proposta, de fato, ajuda os jovens engenheiros e arquitetos durante o processo de desenvolvimento de software centrado na arquitetura?

Esta Questão de Pesquisa foi organizada em outras sete questões de pesquisa com escopos mais específicos, a saber:

- **RQ 1.**O Assistente de elucidação de requisitos de fato melhorou a identificação correta dos requisitos pretendidos pelo interessado, se comparado ao processo manual?
- **RQ 2.**O Assistente de elucidação de requisitos teve um desempenho melhor na identificação dos potenciais trade-offs entre os requisitos, se comparado ao processo manual?
- **RQ 3.**A Recomendação da arquitetura realmente reflete os desejos (requisitos) pretendidos pelos interessados?
- **RQ 4.**O Código gerado atende as especificações do modelo Cosmos* utilizado?
- **RQ 5.**O Código gerado é consistente com a arquitetura de software recomendada?
- **RQ 6.**A Solução ajuda a estruturar e padronizar a representação do Conhecimento Arquitetural?
- **RQ 7.**A Solução ajuda a reduzir a Vaporização do Conhecimento Arquitetural?

[1] do inglês *Research Question*

9.1.2 Seleção dos Sujeitos

Com o propósito de participar da avaliação, quatro voluntários (educadores), com experiência no domínio dos AVAs, foram selecionados, a fim de realizar a especificação das suas necessidades (requisitos de qualidade) para o sistema AVA pretendido. Além dos educadores, outros quatro voluntários (desenvolvedores) também foram escolhidos para participar da avaliação. Dois destes desenvolvedores são analistas de tecnologia, e os outros dois possuíam conhecimento inicial em Engenharia de Requisitos e Arquitetura de Software.

9.1.3 Instrumentação

Para avaliar a viabilidade da solução proposta instanciamos a ferramenta no contexto do domínio dos Ambientes Virtuais de Aprendizagem (AVA). Para isso, utilizou-se um trabalho que apresenta variabilidades arquiteturais relacionadas ao domínio dos AVAs (ROCHA et al., 2013). Essas variabilidades envolvem os atributos de qualidade descritos na ISO-9126 (ISO, 2001; ISO, 2003), são eles: eficiência, manutenibilidade, usabilidade, confiabilidade, testabilidade, interoperabilidade, segurança, portabilidade e conformidade.

Baseando-se na literatura especializada (EGYED; GRUNBACHER, 2004; HENNINGSSON; C.WOHLIN, 2002) e em experiências anteriores de um arquiteto de software, foi realizado um mapeamento inicial de *trade-offs*. Os resultados dessa análise estão representados na Tabela 1, que considera três tipos de relacionamentos entre os atributos de qualidade: conflito (−), neutro (**0**) ou cooperação (+).

Tabela 1 – Potencial Relacionamento entre os Atributos de Qualidade

	Eficiência	**Manutenibilidade**	**Usabilidade**	**Confiabilidade**	**Testabilidade**	**Interoperabilidade**	**Segurança no Acesso**	**Portabilidade**
Eficiência	+	−	+	−	−	−	−	−
Manutenibilidade	−	+	0	−	0	−	0	+
Usabilidade	+	0	+	0	−	0	−	+
Confiabilidade	−	−	0	+	+	0	+	0
Testabilidade	−	0	−	+	+	+	+	+
Interoperabilidade	−	−	0	0	+	+	0	0
Segurança no Acesso	−	0	−	+	+	0	0	0
Portabilidade	−	+	+	0	+	0	0	+

Fonte: Elaborada pelo autor

Essas informações foram então analisadas e regras para ativação de *trade-offs* foram

geradas. Tais regras contêm: (1) **um par de atributos de qualidades conflitantes** (chamados padrões de *trade-off*); (2) **perguntas** que visam resolver o *trade-off* através da priorização de um requisito em detrimento do outro; (3) **definição do impacto**, que diz o impacto de cada possível resposta para as perguntas em termos da priorização dos atributos de qualidade.

Estilos arquiteturais também foram registados para avaliar o processo de recomendação. Estes estilos foram definidos pelo especialista baseado em um modelo de referência voltado para os AVAs (ROCHA et al., 2013). O modelo de referência tem variabilidades e permite diferentes configurações arquiteturais. Quatro variações deste modelo foram criadas, onde cada variação satisfaz mais de um atributo de qualidade em diferentes níveis de satisfação. Cada variação foi registrada no banco de dados de conhecimento como um estilo arquitetural. Neste banco de dados, um estilo arquitetural é definido em termos de: (1) componentes arquiteturais; (2) conectores arquiteturais; (3) configuração arquitetural; (4) cenários arquiteturais envolvendo atributos de qualidade e (5) nível de satisfação para cada atributo de qualidade. Para avaliar a adequação de cada estilo aos atributos de qualidade foram definidos cenários relativos ao domínio de AVAs, no intuito de identificar conflito de interesses e relações de *trade-off*. Fatores como a experiência do arquiteto e o envolvimento de especialistas no domínio de educação a distância foram fundamentais nessa tarefa, pois tais relações puderam ser representadas mais fielmente no domínio escolhido para a avaliação.

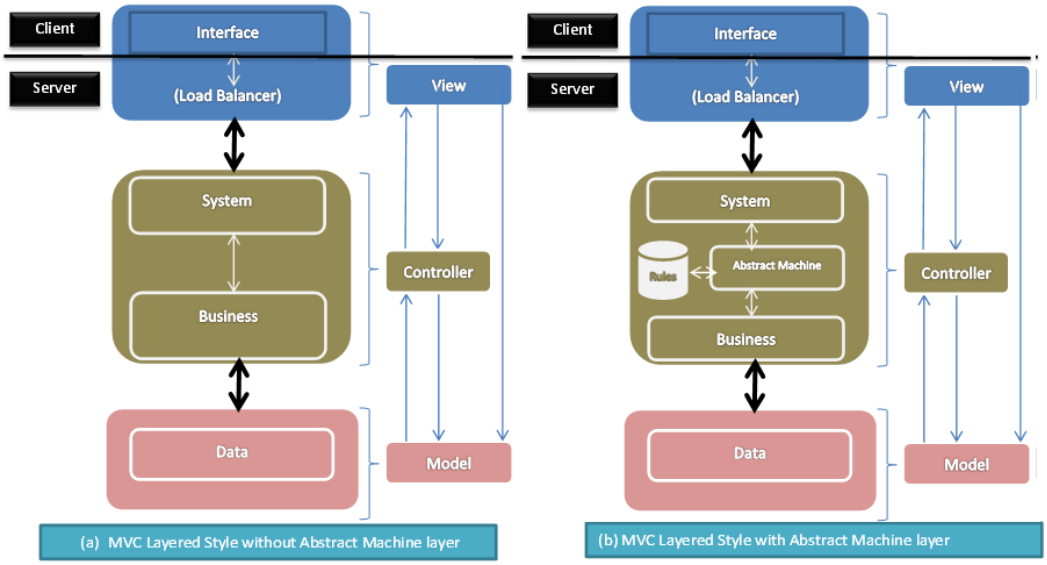
A Figura 25 ilustra dois estilos arquiteturais armazenados na base de conhecimento. A principal diferença entre os estilos (a) e (b) é a inclusão de mais uma camada baseada no estilo arquitetural de máquina abstrata (virtual) (BASS; CLEMENTS; KAZMAN, 2012). O objetivo dessa evolução foi representar a lógica de negócio através de regras externas, independentes da linguagem de programação. Note que comparado ao estilo (a), o estilo (b) tem maior indireção, o que tende a interferir negativamente na eficiência e no *throughput*. Em contrapartida, o menor acoplamento entre as regras de negócio e o código-fonte do sistema tendem a melhorar a manutenibilidade, possibilitando inclusive a atualização das regras diretamente por especialistas do negócio, ainda que não sejam programadores. Do ponto de vista de disponibilidade e escalabilidade, a satisfação dos requisitos permanece a mesma para ambos os estilos arquiteturais, uma vez que o conector **balanceador de carga** está presente em ambos os estilos.

9.1.4 Operação

Depois de preparar a base de conhecimento, os quatro voluntários (educadores) especificaram uma lista de atributos de qualidade para um AVA adequado ao seu curso. Cada voluntário especificou os atributos de qualidade duas vezes: manualmente e com a ferramenta proposta.

Inicialmente, os quatro voluntários receberam uma lista dos atributos de qualidade

Figura 25 – Exemplo de dois estilos arquiteturais armazenados na base de conhecimento



Fonte: Elaborada pelo autor

da ISO-9126 e foi pedido que priorizassem os atributos entre eles. Enquanto, dois voluntários (a) e (b) foram orientados, inicialmente, a classificar de acordo com sua intuição e conhecimento no domínio. Os outros dois (c) e (d) foram orientados a classificar utilizando o assistente de elicitação de requisitos apresentado na Seção 6. Os avaliadores foram orientados a colocar até três atributos de qualidade no mesmo nível de prioridade.

Após a primeira fase da classificação, os voluntários fizeram uma segunda classificação. Enquanto os voluntários (a) e (b) foram orientados a utilizar o assistente proposto, os voluntários (c) e (d) foram orientados a classificar de acordo com sua intuição e conhecimento no domínio.

Uma vez que as especificações foram feitas pelos educadores, os desenvolvedores executaram o processo de gerenciamento dos requisitos, projeto arquitetural e geração de código. Dois desenvolvedores (e) e (f), sendo um deles analista experiente e o outro com pouca experiência, foram orientados a executar inicialmente o processo manualmente e em seguida utilizando o assistente. Já os outros dois desenvolvedores (g) e (h), com perfis semelhantes aos dois primeiros, executaram a especificação na ordem inversa: primeiramente utilizando o assistente e depois repetiram o processo sem o auxílio da solução.

Como apresentado na Seção 6, o assistente proposto guiou tanto o processo de engenharia de requisitos, quanto o gerenciamento dos possíveis *trade-offs* entre eles. Uma vez que os requisitos foram especificados, eles foram usados como entrada para o processo de recomendação arquitetural. Foram considerados apenas os requisitos finais, isto é, requisitos que os voluntários julgaram ser mais confiáveis. Vale salientar que em todos os casos, os voluntários optaram por utilizar os requisitos especificados de forma iterativa, utilizando o assistente.

Figura 26 – Fórmula para calcular a relevância de um estilo arquitetural

$$\sum_{i=1}^{n} UQi * AQi$$

Fonte: Elaborada pelo autor

A fórmula apresentada na Figura 26 é usada pelo componente **ArchitectureMgr** para calcular a adequação do estilo arquitetural ao conjunto de estilos arquiteturais desejados. Esta fórmula considera os requisitos dos usuários em termos dos atributos de qualidade e o nível de satisfação de cada requisito, por parte do estilo arquitetural. **UQi** representa a importância dada pelo usuário para um atributo de qualidade **i** e **AQi** é a satisfação deste atributo de qualidade **i** no estilo arquitetural avaliado. O sistema especialista realiza este cálculo para cada estilo arquitetural registrado na base de conhecimento. Esta fórmula pode ser melhorada dinamicamente, por exemplo, para refinar o processo de recomendação através da introdução de novas variáveis e pesos. A melhoria pode ser realizada de uma forma dinâmica, sem a necessidade de recompilar a aplicação, já que a fórmula é armazenada na forma de regras de produção. Porém, o ajuste dinâmico da fórmula a partir de dados empíricos está fora do escopo do presente trabalho, podendo ser considerado um trabalho futuro.

Após o cálculo da relevância sobre cada estilo arquitetural, o assistente guiou os desenvolvedores através das explicações sobre cada estilo, permitindo também que o refinamento da recomendação fosse realizado. Uma vez concluído este processo, o especialista realizou a avaliação arquitetural sobre as duas recomendações mais relevantes em cada execução. Por fim, o código estrutural foi gerado e apresentado aos desenvolvedores.

Finalmente, depois de utilizar a ferramenta seguindo o processo apresentado na Seção 4, os voluntários foram questionados sobre o seu nível de confiança na especificação, comparando os dois resultados e experiências.

9.2 Resultados da Avaliação e Discussões

Nesta seção, os principais resultados da avaliação mencionada na Seção 9 são apresentados. As Questões de Pesquisa listadas na seção 9.1.1 são respondidas.

9.2.1 O assistente de elucidação de requisitos de fato melhorou a identificação correta dos requisitos pretendidos pelo interessado, se comparado ao processo manual?

O primeiro ponto que merece destaque é a diferença entre a especificação dos requisitos de qualidade realizada manualmente e a especificação utilizando o assistente. Dentre os métodos empregados durante a avaliação, todos os voluntários concordaram que o processo

Figura 27 – Resultado do Processo de Recomendação

Fonte: Elaborada pelo autor

de elicitação de requisitos utilizando o assistente torna a identificação dos requisitos de qualidade mais fácil, compreensível e confiável.

9.2.2 O assistente de elucidação de requisitos teve um desempenho melhor na identificação dos potenciais trade-offs entre os requisitos, se comparado ao processo manual?

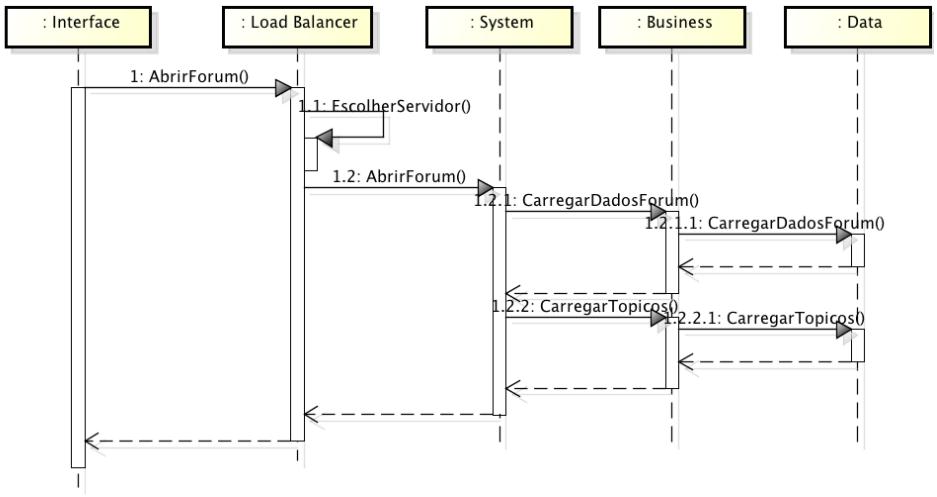
Durante o processo manual, alguns *trade-offs* foram destacados pelos desenvolvedores. Contudo, o índice de acerto destes *trade-offs* foi muito baixo (falsos positivos). Além disto, nem todos os *trade-offs* foram identificados (falsos negativos). Em contrapartida, com a utilização do assistente, ocorreu um alto índice de detecção dos conflitos. Todos os *trade-offs* identificados foram resolvidos através da interação com os interessados.

9.2.3 A recomendação da arquitetura realmente reflete os desejos (requisitos) pretendidos pelos interessados?

A Figura 27 ilustra um possível resultado do processo de recomendação. Cada item sugerido contém um estilo arquitetural, uma descrição do estilo arquitetural, o nível de confiança da recomendação (relevância) e uma explicação das principais decisões de projeto que justificam a escolha daquele estilo em termos dos atributos de qualidade presentes nas preferências do usuário.

Ao término da recomendação arquitetural, o especialista realizou a avaliação para os dois estilos arquiteturais com uma maior taxa de relevância. Para tal, o método ATAM foi empregado, levando em consideração dois tipos de cenários na avaliação dos estilos arquiteturais. O primeiro cenário avalia o atributo de qualidade “eficiência”, considerando o tempo de resposta estimado para cada elemento arquitetural durante a execução de um recurso do AVA. Já o segundo cenário avalia o atributo “manutenibilidade”, considerando

Figura 28 – Diagrama de Sequência da Arquitetura (a)



Fonte: Elaborada pelo autor

dois aspectos importantes no contexto de AVAs: custo de manutenção e esforço para realização da manutenção.

Para ilustrar o processo realizado, vamos apresentar a avaliação realizada para os estilos arquiteturais (a) e (b) da Figura 25, considerados durante o processo de recomendação arquitetural. As Figuras 28 e 29 mostram os diagramas de sequência, apresentando a interação entre os objetos dos estilos arquiteturais (a) e (b), respectivamente. Tal interação refere-se ao cenário de execução do recurso de visualização do fórum de postagens. Levando em consideração esses diagramas de sequência, os cenários descritos a seguir foram executados.

O primeiro cenário avaliado leva em consideração o atributo de qualidade "eficiência" para efetuar a avaliação das arquiteturas, verificando o tempo médio gasto na execução de uma rotina. Para tal, o especialista, com base na sua experiência no domínio dos AVAs, estimou possíveis valores para o tempo gasto na execução da rotina em cada elemento arquitetural. Tal estimativa foi realizada com base na experiência do arquiteto quanto à utilização das tecnologias empregadas nos elementos arquiteturais, tais como algoritmos e *frameworks* utilizados.

Cenário 1: Eficiência

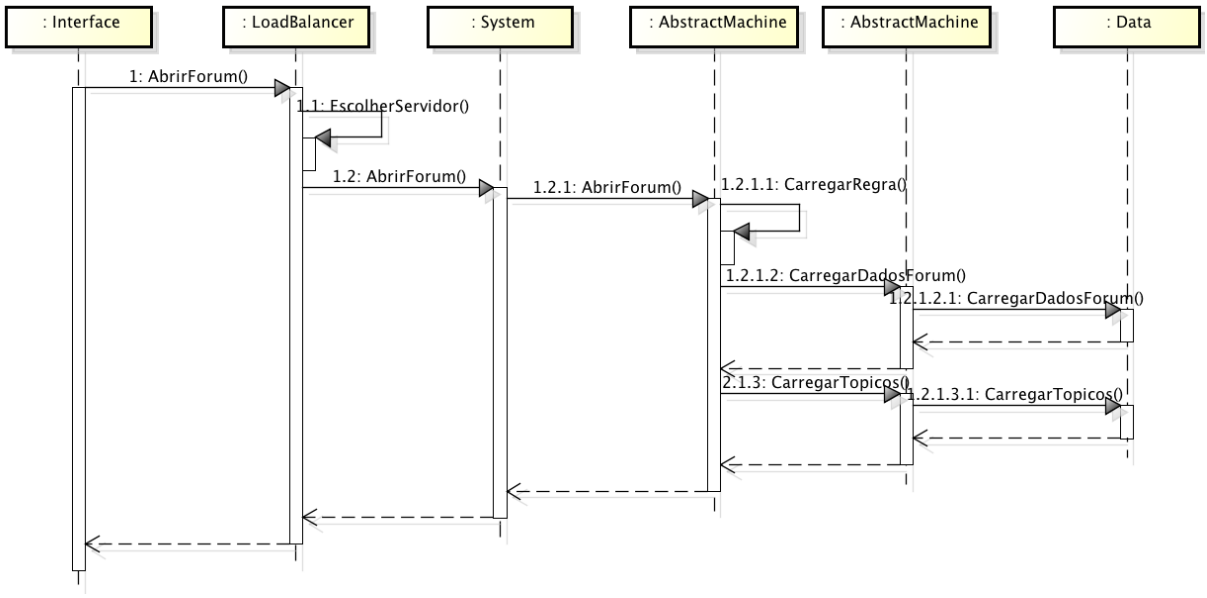
1. **Arquitetura:** (a) Estilo heterogêneo combinando Camadas e MVC com *load balancer*

2. **Descrição:** Tempo de resposta estimado para cada elemento arquitetural do estilo arquitetural.

3. **Elementos Arquiteturais:** Total 950ms

    a) **Interface:** desconsiderada, pois depende do usuário

Figura 29 – Diagrama de Sequência da Arquitetura (b)



Fonte: Elaborada pelo autor

b) **Load Balancer:** 100ms

c) **System:** 200ms

d) **Business:** 150ms

e) **Data:** 500ms

Cenário 1: Eficiência

1. **Arquitetura:** (b) Estilo heterogêneo combinando Camadas, MVC e Abstract Machine com *load balancer*

2. **Descrição:** Tempo de resposta estimado para cada elemento arquitetural do estilo arquitetural.

3. **Elementos Arquiteturais:** Tempo médio Total 1350ms

   a) **Interface:** desconsiderada, pois depende do usuário

   b) **Load Balancer:** 100ms

   c) **System:** 100ms

   d) **Abstract Machine:** 500ms

   e) **Business:** 150ms

   f) **Data:** 500ms

Ao fim da execução deste cenário, percebe-se que o estilo arquitetural (b) possui um desempenho inferior, se comparado ao estilo (a) de mais de 42% . A principal razão para esse declínio no desempenho foi a inclusão do componente *Abstract Machine*, que utiliza *framework* de carregamento e interpretação de regras em tempo de execução. Tal componente fez com que a eficiência fosse afetada significativamente em detrimento da facilidade da manutenção das regras de negócio.

O segundo cenário objetiva avaliar o atributo de qualidade manutenibilidade nos dois estilos mencionados. Alguns aspectos, considerados importantes pelo especialista, foram destacados durante esta avaliação, são eles:

- Disponibilidade da aplicação para realizar a alteração;
- Exigência de conhecimento técnico de programação;
- Tempo médio necessário para realizar a alteração (horas);
- Custo médio necessário para realizar a alteração (horas * valor-hora);

Cenário 2: Manutenibilidade

1. **Arquitetura:** (a) Estilo heterogêneo combinando Camadas e MVC com *load balancer*
2. **Descrição:**Deseja-se alterar a funcionalidade de envio de mensagem em Fórum. A alteração consiste em exigir confirmação de envio através de um código captcha. Nesse cenário de alteração, a funcionalidade de exibição e validação do código captcha está disponível e é provida pelo componente *Business*. Porém, tal funcionalidade não era utilizada pela funcionalidade de postagem em fórum.
3. **Elementos Arquiteturais:**
   a) **Interface:** desconsiderada, pois a mudança na regra de negócio não afeta o componente.
   b) **Load Balancer:** desconsiderada, pois a mudança na regra de negócio não afeta o componente.
   c) **System:**
      i. **Disponibilidade da aplicação para realizar a alteração:** O servidor necessita ser reiniciado após efetivação da alteração;
      ii. **Exigência de conhecimento técnico de programação:** Toda a manutenção será realizada por desenvolvedores (programadores), sendo necessário contratar mão de obra especializada;
      iii. **Tempo médio necessário para realizar a alteração (horas):** Entre 2 e 3 horas de trabalho de um programador, incluindo testes;

iv. **Custo médio necessário para realizar a alteração (horas * valor-hora):** Considerando o custo horário de R$ 150,00, o custo fica entre R$ 300,00 e R$ 450,00.

d) **Business:**desconsiderada, pois a mudança na regra de negócio não afeta o componente.

e) **Data:** desconsiderada, pois a mudança na regra de negócio não afeta o componente.

Cenário 2: Manutenibilidade

1. **Arquitetura:** (b) Estilo heterogêneo combinando Camadas, *Abstract Machine* e MVC com *load balancer*

2. **Descrição:**Deseja-se alterar a funcionalidade de envio de mensagem em Fórum. A alteração consiste em exigir confirmação de envio através de um código captcha. Nesse cenário de alteração, a funcionalidade de exibição e validação do código captcha está disponível e é provida pelo componente *Business*. Porém, tal funcionalidade não era utilizada pela funcionalidade de postagem em fórum.

3. **Elementos Arquiteturais:**

   a) **Interface:** desconsiderada, pois a mudança na regra de negócio não afeta o componente.

   b) **Load Balancer:** desconsiderada, pois a mudança na regra de negócio não afeta o componente.

   c) **System:** desconsiderada, pois a mudança na regra de negócio não afeta o componente.

   d) **Abstract Machine:**

      i. **Disponibilidade da aplicação para realizar a alteração:** O servidor NÃO necessita ser reiniciado após efetivação da alteração;

      ii. **Exigência de conhecimento técnico de programação:** A alteração pode ser realizada pelo especialista do negócio sem exigência de conhecimento técnico de programação;

      iii. **Tempo médio necessário para realizar a alteração (horas):** Aproximadamente 1 hora de trabalho de um especialista do negócio, incluindo testes;

      iv. **Custo médio necessário para realizar a alteração (horas * valor-hora):** Considerando custo horário de R$ 100,00, o custo fica em torno de R$100,00. Porém, considerando que nesse domínio o especialista está disponível na instituição além do tempo de sala de aula essa tarefa poderia ser realizada sem custo adicional.

e) **Business:**desconsiderada, pois a mudança na regra de negócio não afeta o componente.

f) **Data:** desconsiderada, pois a mudança na regra de negócio não afeta o componente.

O estilo arquitetural (b) comparado ao estilo (a), além de possuir um menor custo para mudanças nas regras de negócio, permite que as regras sejam modificadas sem que a aplicação necessite ser interrompida ou até mesmo precise de conhecimento técnico de computação para realizar tal modificação.

9.2.4 O código gerado atende as especificações do modelo Cosmos* utilizado?

Ao fim da avaliação das arquiteturas, os códigos fontes estruturais da arquiteturas recomendadas foram gerados. Todos esses códigos foram inspecionados, analisando a existência de erros sintáticos além de inconsistências entre o código gerado e o modelo de implementação utilizado (modelo Cosmos*). Foram analizadas tanto a estrutura dos componentes e conectores arquiteturais, quanto a configuração arquitetural definida pelo código de *binding* entre componentes e conectores arquiteturais. Todos os códigos gerados foram importados e avaliados na IDE NetBeans no qual tais itens foram devidamente inspecionados e depurados manualmente, constatando-se que atendiam todas as especificações esperadas.

9.2.5 O código gerado é consistente com a arquitetura de software recomendada?

Todos os desenvolvedores consideraram que o processo de geração de código acelerou e melhorou o mapeamento entre a arquitetura e o código. Uma vez que o código gerado atende as especificações do modelo Cosmos* utilizado. Podemos então, assumir que o código possui rastreabilidade com a arquitetura de software recomendada (GAYARD; RUBIRA; GUERRA, 2008).

9.2.6 A solução ajuda a estruturar e padronizar a representação do Conhecimento Arquitetural?

Foi consenso entre os voluntários que a ferramenta ajuda a estruturar e padronizar a representação do conhecimento arquitetural. Embora os voluntários tenham considerado a solução útil (todos responderam que gostariam de utilizá-la em outros projetos), consideraram a ferramenta, especialmente, importante no contexto dos arquitetos de software menos experientes.

9.2.7 A solução ajuda a reduzir a Vaporização do Conhecimento Arquitetural?

Foi consenso entre os voluntários participantes da avaliação que a ferramenta ajuda a prevenir a Vaporização do Conhecimento Arquitetural. Uma das vantagens da solução proposta está relacionada à evolução da base de conhecimento e a consequente generalização do domínio. O conhecimento pode evoluir através da inclusão de novos estilos arquiteturais, novos cenários envolvendo atributos de qualidade, novos cenários de avaliação usando ATAM, novas explicações sobre as decisões de projeto, novos padrões de *trade-off* e novas regras para ativação destes *trade-offs*. Assim, quanto mais a ferramenta for utilizada e mais conhecimento for representado, mais precisa a recomendação deve se tornar.

10 CONCLUSÃO

Este trabalho apresentou uma abordagem para favorecer o desenvolvimento centrado na arquitetura de software. A solução compreende três aspectos principais: (1) um processo sistemático apoiado por uma ferramenta para ajudar jovens engenheiros de requisitos e arquitetos de software durante a especificação dos requisitos do sistema, detectando e resolvendo os *trade-offs* entre eles; (2) apoio ao arquiteto de software na fase de projeto arquitetural através de recomendações de estilos arquiteturais com base nas preferências do usuário; e (3) geração de esqueleto de código, a partir da arquitetura de software, seguindo um modelo baseado em componentes. A ferramenta desenvolvida tem uma arquitetura baseada em regras que utiliza um sistema especialista que se comporta como um engenheiro de requisitos e um arquiteto de software, mantendo, assim, o conhecimento técnico e a experiência na empresa de software.

Embora outras abordagens existentes tratem do mesmo objetivo, o presente trabalho focou no projeto arquitetural interligado com o gerenciamento de *trade-off*, envolvendo especificamente atributos de qualidade de software. Além disso, também visa esclarecer o significado de atributos de qualidade e da relação entre eles, e como tais atributos são conectados com decisões arquiteturais. Este fato tende a melhorar a confiança do interessado na especificação final atributo de qualidade, bem como o projeto arquitetural e o código-fonte desenvolvido.

Essa integração também visa aumentar a qualidade da resolução *trade-off*, uma vez que os *trade-offs* são mais bem resolvidos quando contextualizado com artefatos de projeto, como a arquitetura de software e código-fonte (HENNINGSSON; C.WOHLIN, 2002).

Os resultados preliminares mostraram que é uma abordagem viável com resultados promissores relacionados com a satisfação sobre o processo de recomendação. Houve consenso entre os voluntários de que a abordagem sistemática fez com que o processo de especificação se tornasse mais fácil e reprodutível, além de ter proporcionado melhor compreensão dos atributos de qualidade durante a especificação. Além disso, uma vez que as decisões de projeto são associadas a recomendações de estilos arquiteturais, os voluntários apontaram que a solução proposta promove a reutilização justificada de arquiteturas e favorece a prevenção da Vaporização do Conhecimento Arquitetural.

Como fruto do trabalho desenvolvido, duas publicações foram aceitas em conferências internacionais. O trabalho intitulado *A Tool for Trade-off Resolution on Architecture-Centered Software Development*, que foca as atividades de engenharia de requisitos com gerenciamento dos trade-offs entre os atributos de qualidade foi aceito e publicado na *Twenty-Sixth International Conference on Software Engineering and Knowledge Engineering* (SEKE - B1) , realizada no mês de Julho 2014 em Vancouver - Canadá. O segundo

trabalho, intitulado *A decision-making tool to support architectural designs based on Quality Attributes*, foca a recomendação arquitetural relacionada com o processo de engenharia de requisitos e foi aprovado e será apresentado no *The 30th ACM/SIGAPP Symposium On Applied Computing* (SAC - A1), que será realizado no mês de Abril de 2015 em Salamanca - Espanha. Pretende-se submeter uma nova publicação focando todo o processo, incluindo a geração do código estrutural, para conferências internacionais ainda no ano de 2015.

Como um trabalho futuro imediato, a abordagem deve ser avaliada em cenários reais e domínios, com um elevado número de voluntários, no contexto de um experimento empírico estatisticamente significativo. Tal trabalho terá como foco a publicação em um periódico. Além disso, outro trabalho futuro consiste na utilização de técnicas de aprendizado de máquina para ajustar o cálculo da relevância de cada estilo, a partir de experiências positivas e negativas de recomendações catalogadas na base de conhecimento. Para isso, poderiam ser utilizados, por exemplo, algoritmos de mineração de dados. Este trabalho será realizado durante os meses de Março a Junho de 2015, durante a participação no projeto *DEsign, Verification and VAlidation of large scale, dynamic Service SystEmS* (DEVASSES). Tais atividades serão desempenhadas, neste período, na Universidade de Coimbra - Portugal, parceiros do projeto.

Como uma evolução da solução proposta pretende-se adotar, nas futuras versões, a inclusão de recursos que considerem as restrições ambientais durante o processo de recomendação arquitetural. Estas restrições podem estar relacionadas, por exemplo, a infraestrutura já existente na empresa. Além disto, o processo de gerenciamento de *trade-off* também deve ser aprimorado com a inclusão de lógica *fuzzy*.

Finalmente, acredita-se que a solução apresentada, neste trabalho, possui características compatíveis com o conceito de desenvolvimento dirigido por modelos (MDD[1]). Nesse sentido, um outro trabalho futuro seria contextualizar a solução proposta aos conceitos de MDD. Para tal, faz-se necessário maior esforço nas etapas de transformação de modelos, com o objetivo de favorecer ainda mais a automação do desenvolvimento e separação entre artefatos independentes e dependentes de plataforma.

[1] Sigla do inglês *Model-Driven Development*

REFERÊNCIAS

AL-NAEEM, T. et al. A quality-driven systematic approach for architecting distributed software applications. In: *In Proceedings of the 27th International Conference on Software Engineering (ICSE)*. [S.l.]: ACM Press, 2005. p. 244–253.

BABAR, M. A.; GORTON, I. A tool for managing software architecture knowledge. In: *Proceedings of the Second Workshop on SHAring and Reusing Architectural Knowledge Architecture, Rationale, and Design Intent*. Washington, DC, USA: IEEE Computer Society, 2007. (SHARK-ADI '07), p. 11–. ISBN 0-7695-2951-8. Disponível em: <http://dx.doi.org/10.1109/SHARK-ADI.2007.1>.

BASS, L.; CLEMENTS, P.; KAZMAN, R. *Software Architecture in pratice 3rd Edition*. Reading, Massachusetts: Addison-Wesley Publishing Company, 2012.

BASS, L.; KAZMAN, R. *Architecture-Based Development*. Pittsburgh,PA, 1999. v. 99, n. 007, 50 p. Disponível em: <http://www.sei.cmu.edu/library/abstracts/reports/99tr007.cfm>.

BRITO, P. H. da S. et al. *Um processo para o Desenvolvimento Baseado em Componentes com Reuso de Componentes*. Campinas, 2005. v. 05, n. 22, 27 p.

BRITO, P. H. da S.; BARBOSA, M. A. M.; RUBIRA, C. M. F. *Um metodo para Modelagem de Arquitetura a Partir dos Requisitos do Sistema*. Campinas, 2007. v. 07, n. 11, 40 p.

CHE, M. An approach to documenting and evolving architectural design decisions. In: *Proceedings of the 2013 International Conference on Software Engineering*. Piscataway, NJ, USA: IEEE Press, 2013. (ICSE '13), p. 1373–1376. ISBN 978-1-4673-3076-3. Disponível em: <http://dl.acm.org/citation.cfm?id=2486788.2487009>.

CHEESMAN, J.; DANIELS, J. *UML Components - A Simple Process for Specifying Component-Based Software*. [S.l.]: Addison-Wesley, 2001. (Component Software Series).

CHEN, F. et al. An architecture-based approach for component-oriented development. *26th Annual International Computer Software and Applications Conference*, August 2002.

CHESSMAN, J.; DANIELS, J. *UML Components: A Simple Process for Specifying Component-Based Software*. [S.l.]: Paperback, 1992.

DROOLS. *Drools - Overview*. 2014. Disponível em <http://www.drools.org/>. Accessado em Setembro de 2014.

EGYED, A.; GRUNBACHER, P. Identifying requirements conflicts and cooperation: How quality attributes and automated traceability can help. *IEEE Software*, IEEE Computer Society, Los Alamitos, CA, USA, v. 21, n. 6, p. 50–58, 2004. ISSN 0740-7459.

ELAHI, G.; YU, E. A semi-automated decision support tool for requirements trade-off analysis. In: *COMPSAC*. [S.l.]: IEEE Computer Society, 2011. p. 466–475. ISBN 978-0-7695-4439-7.

FALESSI, D. et al. Decision-making techniques for software architecture design: A comparative survey. *ACM Comput. Surv.*, ACM, New York, NY, USA, v. 43, n. 4, p. 33:1–33:28, out. 2011. ISSN 0360-0300. Disponível em: <http://doi.acm.org/10.1145/1978802.1978812>.

FARENHORST, R.; BOER, R. C. Knowledge Management in Software Architecture: State of the Art. In: BABAR, M. A. et al. (Ed.). *Software Architecture Knowledge Management*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009. cap. 2, p. 21–38. ISBN 978-3-642-02373-6. Disponível em: <http://dx.doi.org/10.1007/978-3-642-02374-3_2>.

GARCÍA-MIRELES, G. A. et al. A framework to support software quality trade-offs from a process-based perspective. In: MCCAFFERY, F.; O'CONNOR, R. V.; MESSNARZ, R. (Ed.). *EuroSPI*. [S.l.]: Springer, 2013. (Communications in Computer and Information Science, v. 364), p. 96–107. ISBN 978-3-642-39178-1.

GARLAN, D.; SHAW, M. *An Introduction to Software Architecture*. Pittsburgh,PA, 1994. v. 94, n. 166, 42 p.

GAYARD, L. A.; RUBIRA, C. M. F.; GUERRA, P. A. de C. *COSMOS*: a COmponent System MOdel for Software Architectures*. [S.l.], 2008.

GIORGINI, P. et al. Formal reasoning techniques for goal models. *J. Data Semantics*, Springer, v. 1, p. 1–20, 2003. Disponível em: <http://dblp.uni-trier.de/db/journals/jods/jods1.html#GiorginiMNS03>.

GRÜNBACHER, P.; EGYED, A.; MEDVIDOVIC, N. Reconciling software requirements and architectures with intermediate models. *Software and Systems Modeling*, Springer-Verlag, v. 3, n. 3, p. 235–253, 2004. ISSN 1619-1366. Disponível em: <http://dx.doi.org/10.1007/s10270-003-0038-6>.

HAMMOND, J. S.; KEENEY, R. L.; RAIFFA, H. *Smart choices : a practical guide to making better life decisions*. USA: Broadway Books, 2002.

HENNINGSSON, K.; C.WOHLIN. Understanding the relations between software quality attributes - a survey approach. In: *In Proceedings of 12th International Conference for Software Quality*. Ottawa - Canada: [s.n.], 2002.

HIBERNATE. *Hibernate*. 2014. Disponível em <http://hibernate.org/orm/>. Acessado em Novembro 2014.

ISO. *ISO Standard 9126: Software Engineering - Product Quality, part 1*. 2001.

ISO. *ISO Standard 9126: Software Engineering - Product Quality, parts 2 and 3*. 2003.

JACKSON, P. *Introduction To Expert Systems*. 3rd. ed. Harlow, England: Addison Wesley, 1998.

KAZMAN, R.; KLEIN, M.; CLEMENTS, P. *ATAM: Method for Architecture Evaluation*. [S.l.], 2000.

KOSKIMIES, K. Towards architecture-oriented programming environments. *First ASERC Workshop on Software Architecture*, August 2001.

LUCENA, M. et al. Stream: A strategy for transition between requirements models and architectural models. In: *Proceedings of the 2011 ACM Symposium on Applied Computing*. New York, NY, USA: ACM, 2011. (SAC '11), p. 699–704. ISBN 978-1-4503-0113-8. Disponível em: <http://doi.acm.org/10.1145/1982185.1982337>.

Mahesh Parmar W.U. Khan, B. K. An Architectural Decision Tool Based on Scenarios and Non-functional Requirements. *International Journal of Advanced Computer Science and Applications(IJACSA)*, v. 2, n. 2, 2011. Disponível em: <http://ijacsa.thesai.org/>.

MORONTE, T. C. *Uma Infra-estrutura de Software para Apoiar a Construção de Arquiteturas de Software Baseadas em Componentes*. Dissertação (Mestrado), Campinas, São Paulo. Brasil, 2007. Dissertação de Mestrado.

NEIDHARDT, J. et al. Eliciting the users' unknown preferences. In: *Proceedings of the 8th ACM Conference on Recommender Systems*. New York, NY, USA: ACM, 2014. (RecSys '14), p. 309–312. ISBN 978-1-4503-2668-1. Disponível em: <http://doi.acm.org/10.1145/2645710.2645767>.

NGO-THE, A.; RUHE, G. Decision support in requirements engineering. In: AURUM, A.; WOHLIN, C. (Ed.). *Engineering and Managing Software Requirements*. Heidelberg: Springer, 2005.

PRESSMAN, R. S. *Software Engineering: a Practitioner's Approach*. 5th. ed. [S.l.]: McGraw-Hill, 2001.

RICCI, F.; ROKACH, L.; SHAPIRA, B. Introduction to Recommender Systems Handbook. In: RICCI, F. et al. (Ed.). *Recommender Systems Handbook*. Boston, MA: Springer US, 2011. cap. 1, p. 1–35. ISBN 978-0-387-85819-7. Disponível em: <http://dx.doi.org/10.1007/978-0-387-85820-3_1>.

ROBINSON, W. N.; PAWLOWSKI, S. D.; VOLKOV, V. Requirements interaction management. *ACM Comput. Surv.*, ACM, New York, NY, USA, v. 35, n. 2, p. 132–190, jun. 2003. ISSN 0360-0300. Disponível em: <http://doi.acm.org/10.1145/857076.857079>.

ROCHA, H. B. et al. A reference model for virtual learning environments web: Bringing the perspectives of author and publisher. In: *EDULEARN13 Proceedings*. [S.l.]: IATED, 2013. (5th International Conference on Education and New Learning Technologies), p. 1014–1021. ISBN 978-84-616-3822-2. ISSN 2340-1117.

SHAHIN, M.; LIANG, P.; LI, Z. Do architectural design decisions improve the understanding of software architecture? two controlled experiments. In: *Proceedings of the 22Nd International Conference on Program Comprehension*. New York, NY, USA: ACM, 2014. (ICPC 2014), p. 3–13. ISBN 978-1-4503-2879-1. Disponível em: <http://doi.acm.org/10.1145/2597008.2597139>.

SHAW, M.; CLEMENTS, P. A field guide to boxology: Preliminary classification of architectural styles for software systems. In: *Proceedings of COMPSAC97, First International Computer Software and Applications Conference*. [S.l.: s.n.], 1997.

SILVA, I. C. L. et al. A tool for trade-off resolution on architecture-centered software development. In: *Proceedings of the Twenty-Sixth International Conference on Software Engineering and Knowledge Engineering*. Skokie, IL, USA: Knowledge Systems Institute

Graduate School, 2014. (SEKE 2014), p. 35–38. ISBN 1-891706-35-7. Disponível em: <http://ksiresearchorg.ipage.com/seke/seke14paper/seke14paper_190.pdf>.

SILVA-JUNIOR, M. C. da. *COSMOS - Um Modelo de Estruturação de Componentes para Sistemas Orientados a Objetos*. Dissertação (Mestrado), Campinas, São Paulo. Brasil, 2003. Dissertação de Mestrado.

SOMMERVILLE, I. *Software Engineering*. Reading, Massachusetts: Addison-Wesley Publishing Company, 2009.

TOFAN, D.; GALSTER, M. Capturing and making architectural decisions: An open source online tool. In: *Proceedings of the 2014 European Conference on Software Architecture Workshops*. New York, NY, USA: ACM, 2014. (ECSAW '14), p. 33:1–33:4. ISBN 978-1-4503-2778-7. Disponível em: <http://doi.acm.org/10.1145/2642803.2642836>.

TOMITA PAULO A. C. GUERRA, C. M. F. R. R. T. *Bellatrix: Um Ambiente para Suporte Arquitetural ao Desenvolvimento Baseado em Componentes*. [S.l.], 2006.

TOMITA, R. T. *Bellatrix: Um Ambiente para Suporte Arquitetural ao Desenvolvimento Baseado em Componentes*. Dissertação (Mestrado), Campinas, São Paulo. Brasil, 2006.

WOODS, E.; ROZANSKI, N. Unifying software architecture with its implementation. In: *Proceedings of the Fourth European Conference on Software Architecture: Companion Volume*. New York, NY, USA: ACM, 2010. (ECSA '10), p. 55–58. ISBN 978-1-4503-0179-4. Disponível em: <http://doi.acm.org/10.1145/1842752.1842767>.

YOU-SHENG, Z.; YU-YUN, H. Architecture-based software process model. *Software Engineering Notes*, v. 28, n. 2, March 2003.

YU, E. S.-K. *Modelling strategic relationships for process reengineering*. Dissertação (Mestrado), Toronto, Ont., Canada, Canada, 1996. UMI Order No. GAXNN-02887 (Canadian dissertation).

ZHENG, Y.; TAYLOR, R. N. Enhancing architecture-implementation conformance with change management and support for behavioral mapping. In: *Proceedings of the 34th International Conference on Software Engineering*. Piscataway, NJ, USA: IEEE Press, 2012. (ICSE '12), p. 628–638. ISBN 978-1-4673-1067-3. Disponível em: <http://dl.acm.org/citation.cfm?id=2337223.2337297>.