Universidade Federal de Alagoas
Instituto de Computação
Programa de Pós-Graduação em Informática

Dissertação de Mestrado

Um processo para monitoração de requisitos de qualidade de software utilizando informações arquiteturais

Suzy Kamylla de Oliveira Menezes
suzy.kamylla@gmail.com

Orientador:
Prof. Dr. Patrick Henrique da Silva Brito

Maceió, dezembro de 2016

Suzy Kamylla de Oliveira Menezes

Um processo para monitoração de requisitos de qualidade de software utilizando informações arquiteturais

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

Orientador:

Prof. Dr. Patrick Henrique da Silva Brito

Maceió, dezembro de 2016

Catalogação na fonte
Universidade Federal de Alagoas
Biblioteca Central
Divisão de Tratamento Técnico
Bibliotecária Responsável Helena Cristina Pimentel do Vale

M543u Menezes, Suzy Kamylla de Oliveira.
Um processo para monitoração de requisitos de qualidade de software utilizando informações arquiteturais / Suzy Kamylla de Oliveira Menezes. – 2016.
103 f. : il.

Orientador: Patrick Henrique da Silva Brito.
Dissertação (mestrado em Informática) - Universidade Federal de Alagoas. Instituto de Computação. Maceió, 2016.

Bibliografia: f. 97-102.
Apêndice: f. 103.

1. Arquitetura de software. 2. Estilos arquiteturais. 3. Monitoração de requisitos. 4. Requisitos de qualidade. 5. Requisitos não-funcionais. 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 Suzy Kamylla de Oliveira Menezes, intitulada: "*Um processo para monitoração de requisitos de qualidade de software utilizando informações arquiteturais*", apresentada ao Programa de Pós-Graduação em Informática da Universidade Federal de Alagoas em 16 de dezembro de 2016, às 09h45min, na Sala de Reuniões do Instituto de Computação da UFAL.

COMISSÃO JULGADORA

Prof. Dr. Patrick Henrique da Silva Brito
UFAL – Campus Arapiraca
Orientador

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

Profª. Drª. Cecília Mary Fischer Rubira
Unicamp – Instituto de Computação
Examinadora

Resumo

Os avanços na construção de sistemas computacionais possibilitaram que sistemas cada vez mais sofisticados surgissem. Nesse contexto, os desenvolvedores de software deparam-se com problemas complexos que podem ser difíceis de solucionar. Um dos passos fundamentais ao construir um software é definir as principais características dele, isto é, definir os requisitos. Contudo, nem sempre é trivial definir quais são esses requisitos. Normalmente, os requisitos são classificados como *funcionais* e *não-funcionais*. Os requisitos não-funcionais definem propriedades do sistema ou como esse deve operar. Eles também são chamados de requisitos de qualidade, pois são importantes para garantir que o sistema seja viável para uso e atenda às necessidades dos usuários. Apesar da sua importância, eles são difíceis de serem avaliados e mensurados de forma a gerar evidências do seu pleno atendimento. Uma das razões para essa dificuldade é a necessidade de se avaliar o software em tempo de execução, o que é uma tarefa complexa que envolve grandes volumes de dados a serem analisados. O objetivo dessa dissertação é sistematizar e viabilizar a monitoração de requisitos de qualidade a partir de características dos estilos arquiteturais utilizados no projeto da arquitetura de software. Nesse sentido, a questão de pesquisa que direcionou esse estudo foi "como monitorar e analisar requisitos de qualidade em um sistema de software". Com base nisso, foi proposto um processo de monitoração de requisitos de qualidade utilizando informações arquiteturais, tais como estilos arquiteturais, denominado *ArMoni*. Para avaliar a utilização prática do referido processo foi realizado um experimento avaliativo com alunos do curso de Ciência da Computação. Os alunos realizaram a monitoração de uma aplicação real denominada *Falibras*. Esse sistema realiza tradução *online* de textos em língua portuguesa para a Língua Brasileira de Sinais (LIBRAS). Através do experimento, foi possível perceber que os alunos consideraram o processo proposto viável para direcionar as decisões de monitoração. Por outro lado, os alunos que não tiveram acesso ao processo proposto e não conheciam a arquitetura do Projeto Falibras sentiram mais dificuldades para analisar o sistema. Desse modo, é importante para o(a) desenvolvedor(a) de software ter recursos que o(a) auxiliem na compreensão dos requisitos de qualidade relevantes para o software que está sendo monitorado, bem como processos que direcionem a monitoração desses requisitos. Os resultados preliminares apontam que o processo *ArMoni* contribui para guiar a monitoração e analisar os requisitos de qualidade a partir de informações arquiteturais, tais como os estilos arquiteturais que compõem

o sistema e nota-se que ele beneficia principalmente desenvolvedores menos experientes. Assim, essa dissertação apresenta relevância acadêmica na área de engenharia de software e poderá auxiliar desenvolvedores(as) e engenheiros(as) de software a monitorar requisitos de qualidade, tais como confiabilidade e eficiência, além de identificar possíveis gargalos estruturais que podem comprometer a satisfação desses requisitos.

Palavras-chave: Arquitetura de Software. Estilos Arquiteturais. Monitoração de Requisitos. Requisitos de Qualidade. Requisitos Não-Funcionais.

Abstract

Advances in building computer systems allowed the development of increasingly sophisticated software. In this context, software developers are faced with complex problems that can be difficult to resolve. One of the fundamental steps to build a software is to define its main features, that is, define the requirements that can be satisfied, which is not an easy task. Typically, requirements are classified as functional and non-functional. The non-functional requirements define system's properties or how it should operate. They are also called quality requirements, since they are important to ensure that the system is feasible to use and meets the needs of users. Despite their importance, they are difficult to assess and be measured in order to generate evidence of its full service. One reason for this difficulty is the need to evaluate the software at run-time, which is a complex task that involves large volumes of data to be analyzed. This work aims to systematize and facilitate the monitoring of quality requirements from characteristics of architectural styles used in the software architecture design. In this sense, the research question that guided this study was "how to monitor and analyse quality requirements for a software system". On that basis, it was proposed a process, called ArMoni, for monitoring quality requirements using architectural information, such as architectural styles. To assess the practical use of the proposed process, an evaluation experiment with students from the Computer Science course was performed. Students performed the monitoring of a real application called Falibras. This target system performs online text translation from Portuguese to the Brazilian Sign Language (Libras). Through the experiment, it was observed that the students considered the proposed process feasible to direct monitoring decisions. On the other hand, students who did not have access to the proposed process and do not know the architecture of Falibras have related more difficult to analyze the system. Thus, we have realized that it is important for the software developer to have guidelines that assist him/her in understanding the relevant quality requirements for the software that is being monitored, such as a processes that address the monitoring of these requirements. Preliminary results indicate that the ArMoni process contributes to guide the monitoring and analysis of the system based on architectural information, such as the architectural styles and we have noticed that it benefits mainly less experienced developers. Thus, this work presents academic relevance in software engineering and can assist developers and software engineers to monitor quality requirements, such as reliability and efficiency, as

well as to identify possible structural bottlenecks that could compromise the satisfaction of these requirements.

Keywords: Software Architecture. Architectural Styles. Requirements Monitoring. Quality Requirements. Non-Functional Requirements.

À minha família,
pelo amor e apoio.

Agradecimentos

Agradeço, primeiramente, a Deus, pois sem um alicerce espiritual, enfrentar as dificuldades seria uma jornada impossível. Ele é meu conforto e guia-me mesmo na minha infinita ignorância quanto à complexidade da vida.

Aos meus pais, Josineise e Agilson, pelo amor, dedicação, carinho, parceria, esforços para me ajudar em meus projetos pessoais e profissionais, além de seus ensinamentos que são essenciais para a minha construção como ser. Amor incondicional.

Aos meus irmãos, Kenneth e Samara, com os quais partilho bons momentos e são tão importantes na minha vida.

Ao meu esposo, Mario Diego, pelo amor, paciência, confiança e, principalmente, por ter aceitado estar comigo nos momentos bons e ruins. Agradeço por acreditar em mim e incentivar meus objetivos.

Ao meu orientador, Prof. Dr. Patrick Henrique da Silva Brito, que durante a elaboração desta dissertação contribuiu para direcionar meus estudos, aperfeiçoar minhas pesquisas e concretizar esse trabalho. Agradeço por colaborar para a realização desse projeto profissional, que para mim é a realização de um sonho.

Ao Prof. Dr. Evandro de Barros Costa e à Profa. Dra. Cecília Mary Fischer Rubira por aceitarem o convite de fazer parte da banca examinadora.

Aos professores do Instituto de Computação, do Programa de Pós-Graduação em Informática, pela dedicação e conhecimentos passados nas aulas ministradas, que muito contribuíram para a minha formação acadêmica e pessoal.

À Fundação CAPES pela concessão da bolsa de mestrado, que possibilitou a realização desta pesquisa.

Aos meus colegas e amigos do mestrado.

Aos alunos do curso de Ciência da Computação - Campus Arapiraca pela participação na pesquisa desenvolvida nesta dissertação.

Agradeço a André pela amizade, confiança e lealdade, a Ciência da Computação uniu nossas histórias.

Às minhas amigas Danielle, Caroline Maria, Alyne, Thayse e Nathália, mesmo na distância a amizade floresce.

Ao meu amigo Rusanil. Mais que nunca, estamos cientes que a vida de quem deseja ser mestre requer muito trabalho e persistência.

Agradeço ainda a todos que, direta ou indiretamente, contribuíram para a realização desse trabalho.

"Se você julgar um peixe por sua habilidade de escalar uma árvore, ele vai passar a vida inteira acreditando que é estúpido."

Albert Einstein

Lista de Figuras

Lista de Tabelas

Lista de Códigos

Lista de Siglas

ACM - *Association for Computing Machinery*
ATAM - *Architecture Tradeoff Analysis Method*
API - *Application Programming Interface*
BPEL - *Business Process Execution Language*
C&C - Componente-e-Conector
COSMOS - *Component Structuring Model for Object-oriented Systems*
CPU - *Central Processing Unit*
ESB - *Enterprise Service Bus*
GQM - *Goal-Question-Metric*
HTTP - *Hypertext Transfer Protocol*
IEEE - *Institute of Electric and Electronic Engineers*
ISO/IEC - *Internacional Organization of Standardization/International Electrotechnical Commission*
JPPF - *Java Parallel Processing Framework*
LECC - Laboratório de Ensino em Ciência da Computação
LTS - *Long Term Support*
MOP - *Monitoring Oriented Programming*
MTBF - *Mean Time Between Failures*
MTF - *Mean Time to Failure*
MTTR - *Mean Time to Repair*
MVC - *Model-View-Controller*
MySQL - Sistema de gerenciamento de banco de dados relacional *open source*
NBR - Norma Brasileira
NTI - Núcleo de Tecnologia da Informação
OSI ISO - *Open Systems Interconnection - Internacional Organization of Standardization*
POA - Programação Orientado a Aspectos
POO - Programação Orientado a Objetos
QoS - *Quality of Service*
QP - Questão de Pesquisa
QuAM - Framework para monitoramento de software orientado a aspectos e objetos
RAM - *Random Access Memory*

REST - *Representational State Transfer*
RNF - Requisitos Não-Funcionais
SGBD - Sistema de Gerenciamento de Base de Dados
SLA - *Service Level Agreement*
SOA - *Service-Oriented Architecture*
SOAP - *Simple Object Access Protocol*
SQuaRe - *System and Software Quality Requirements and Evaluation*
SoS - *Systems of Systems*
TDD - *Test Drive Development.* UFAL - Universidade Federal de Alagoas
UML - *Unified Modeling Language*
WSDL - *Web Service Definition Language*

Sumário

Capítulo 1

Introdução

1.1 Contextualização

Os(As) desenvolvedores(as) de software deparam-se com problemas complexos que podem ser difíceis de solucionar. Ao construir sistemas computacionais uma tarefa desafiadora é definir precisamente o que o sistema deve fazer. Para caracterizar o sistema e determinar o que ele deve apresentar são estabelecidas funções e restrições que são chamados de *requisitos*. Nesse contexto, a atividade de descoberta, elicitação, desenvolvimento, análise, determinação de métodos de verificação, validação, comunicação, documentação e gerenciamento de requisitos é chamada de *engenharia de requisitos* (ISO/IEC, 2011).

É preciso ressaltar que devido à complexidade dos sistemas, nem sempre é tão simples distinguir os requisitos e não há um consenso sobre esse termo. Ele pode ser representado como uma declaração abstrata de alto nível ou apresentar uma definição detalhada de uma função do sistema, dependendo do tipo de requisito. De forma simplificada, pode-se distinguir os requisitos utilizando os termos: *requisitos do usuário* e *requisitos de sistema*.

Requisitos do usuário incluem observações requeridas de entradas e informações que os usuários necessitam para usar o sistema, saídas que eles requerem do sistema para desempenhar suas tarefas, e quaisquer condições ou restrições que envolvem a interação deles com o sistema. Esses requisitos são usados para descrever os cenários operacionais que especificam como reuni-los quando interagindo com o sistema. Os *requisitos do sistema* referem-se a funções, desempenho, restrições de projeto e atributos do sistema, seus ambientes operacionais e interfaces externas (ISO/IEC, 2011).

Independentemente do nível de detalhes em que são especificados, os requisitos são normalmente classificados como *funcionais* e *não-funcionais*. Os *requisitos funcionais* descrevem o sistema, funções de elementos do sistema ou tarefas a serem desempenhadas. Os *requisitos não-funcionais* especificam como o sistema deve operar ou propriedades dele. Eles definem como supostamente o sistema deve ser. Definem restrições sobre os serviços e funções oferecidas, como, por exemplo restrições de tempo ou utilização de padrões. Um

requisito não-funcional pode também referir-se a um processo utilizado para desenvolver o sistema (ISO/IEC, 2011; SOMMERVILLE, 2003).

Nas definições sobre requisitos não-funcionais encontradas na literatura, basicamente surgem os termos propriedade ou característica, atributo, qualidade, restrição e desempenho. Propriedade ou característica é usado de modo geral para denotar algo que o sistema deve ter, que inclui qualidades específicas como usabilidade e confiabilidade. Atributo é um termo usado para se referir a uma coleção de qualidades específicas. Além disso, todo requisito pode ser relacionado com qualidade, pois, conforme a ISO 9000:2000, a qualidade é um grau em que um conjunto de características satisfazem os requisitos (GLINZ, 2007; ISO/IEC, 2000).

Embora na literatura existam vários trabalhos que abordam sobre requisitos não-funcionais, não há um consenso sobre esse termo. Observa-se nos trabalhos de Glinz (2007), Chung e Leite (2009) e Silva, Albuquerque e Barroso (2016) que os requisitos não-funcionais são nomeados como atributos de qualidade, características de qualidade, restrições, por exemplo. Nesta dissertação utilizou-se o termo *requisitos de qualidade* para referir-se aos *requisitos não-funcionais*.

Os requisitos de qualidade apesar de não designarem funcionalidades representam aspectos de qualidade que ao menos uma funcionalidade do sistema deve ter, tais como usabilidade, segurança, flexibilidade, integridade, confiabilidade, portabilidade, desempenho. Eles devem ser projetados no sistema quando ele está sendo desenvolvido. Para aferir a eficácia de suas implementações, devem ser definidas métricas apropriadas para mensurá-los (ISO/IEC, 2011; CHUNG; LEITE, 2009)

Para se avaliar o requisito de qualidade de desempenho, pode-se, por exemplo, medir a velocidade analisando as transações processadas por segundo ou o tempo de resposta ao usuário. Para o requisito de facilidade de uso, por exemplo, pode-se considerar o tempo médio de treinamento necessário para utilizar o sistema. Já para aferir a confiabilidade poderia-se considerar o tempo médio de execução normal do sistema antes de falhar, a taxa de ocorrência de falhas, ou até mesmo o seu percentual de disponibilidade.

Pode-se notar que de maneira geral os requisitos de qualidade destacam aspectos que podem estar relacionados com o funcionamento geral do sistema e com os *stakeholders*. *Stakeholders* incluem as partes interessadas direta ou indiretamente no desenvolvimento, funcionamento e uso do software, como por exemplo, usuários, organizações de usuários finais, desenvolvedores, produtores, consumidores, operadores, instituições reguladoras (ISO/IEC, 2011).

No passado, os desenvolvedores de software concentravam-se principalmente nas propriedades funcionais dos sistemas. Contudo, nos sistemas de software modernos, as propriedades não-funcionais estão se tornando cada vez mais importantes. Devido à sua importância e complexidade, as propriedades não-funcionais de uma arquitetura de software têm um grande impacto no seu desenvolvimento, manutenção e extensão. Quanto maior

e mais complexo um sistema de software for e quanto maior for seu tempo de vida, maior será a importância das suas propriedades não-funcionais (BRITO; GUERRA; RUBIRA, 2007). Desse modo, a área de arquitetura de software tem recebido uma atenção considerável, especialmente pela relação estreita entre arquitetura de software e requisitos de qualidade (LUNDBERG et al., 1999)

Requisitos de qualidade são frequentemente cruciais para vários sistemas e devem ser endereçados tão cedo quanto possível no ciclo de vida do software, com atenção especial ao projeto arquitetural do software antes de um projeto detalhado prosseguir para um caminho não desejado (CHUNG; NIXON; YU, 1994a). Projetar a arquitetura correta é uma tarefa muito subjetiva e requer tempo, sendo muito influenciada pela experiência do(a) desenvolvedor(a) bem como a engenharia de requisitos de qualidade. Um grande problema surge na fase de projeto arquitetural quando *trade-offs* entre atributos de qualidade não são identificados e gerenciados durante a engenharia de requisitos (SILVA et al., 2014).

Os *requisitos de qualidade* são muitas vezes mais críticos que os requisitos funcionais, uma vez que referem-se à estruturação do sistema como um todo e não a características isoladas e pontuais do sistema. Quando um requisito funcional individual falha ele pode comprometer parcialmente o sistema, contudo violar um requisito de qualidade como, por exemplo, confiabilidade ou desempenho, pode tornar todo o sistema inviável.

Nessa perspectiva, a arquitetura de um sistema é o primeiro artefato que pode ser analisado para determinar como seus requisitos de qualidade serão alcançados e serve como uma planta baixa. Ela é importante, pois permite descrever, observar e pensar sobre características qualitativas estratégicas do comportamento do sistema no início do ciclo de vida (BASS; CLEMENTS; KAZMAN, 2003).

O tratamento de requisitos de qualidade no desenvolvimento de software ainda não é tão dominado quanto o tratamento de requisitos funcionais. Existem várias razões para essa situação: a) requisitos de qualidade devem ser elicitados, analisados e documentados tanto quanto os requisitos funcionais; b) engenharia de requisitos e projeto arquitetural devem ser integrados de forma que o conhecimento adiquirido na fase de engenharia de requisitos seja usado sistematicamente no desenvolvimento da arquitetura de software; c) as técnicas atuais para incorporação de requisitos de qualidade dentro de arquiteturas de software não são tão maduras quanto as que estão concentradas somente em requisitos funcionais (ALEBRAHIM; HATEBUR; HEISEL, 2011).

Além desses aspectos, Silva, Albuquerque e Barroso (2016) citam outros fatores que estão associados à negligência na elicitação e tratamento desses requisitos, tais como: a diversidade de requisitos de qualidade; a natureza subjetiva desses requisitos; a possibilidade de conflitos entre os próprios requisitos de qualidade e requisitos funcionais; a falta de consciência dos desenvolvedores de sistemas quanto à importância desses requisitos; a falta de conhecimento do domínio do software; os clientes muitas vezes não conseguem

definir os requisitos de qualidade que seriam desejáveis para o sistema.

Por meio de Alebrahim, Hatebur e Heisel (2011) e Silva, Albuquerque e Barroso (2016) percebe-se que essa diversidade de aspectos, por vezes, conduzem o(a) desenvolvedor(a) de software ou equipe de desenvolvimento a deixar em segundo plano o levantamento de requisitos de qualidade, aspecto que pode fortemente contribuir para uma baixa qualidade do software produzido.

Dessa forma, deve-se pensar nas alternativas para realizar a elicitação dos requisitos de qualidade ao desenvolver um software. O projeto arquitetural de um software é normalmente guiado pela escolha de um ou mais estilos arquiteturais. Um estilo arquitetural define uma família de sistemas em termos de um padrão de organização estrutural. Ele determina o vocabulário de componentes e conectores que podem ser usados em instâncias desse estilo, junto com um conjunto de restrições de como podem ser combinadas. Esses podem incluir restrições topológicas em descrições arquiteturais. Outras restrições podem fazer parte da definição do estilo (GARLAN; SHAW, 1994). Embora exista uma relação estreita entre arquitetura de software e requisitos de qualidade, tais requisitos só são materializados definitivamente no código-fonte (LUNDBERG et al., 1999).

Por isso, faz-se necessário ter uma preocupação constante com a preservação desses requisitos em tempo de execução. A monitoração constante, além de servir como evidência de satisfação dos requisitos de qualidade, pode servir para identificar padrões de degradação do sistema, ajudando na difícil tarefa de prever eventuais cenários de violação dos requisitos. Assim, o(a) desenvolvedor(a) de software é capaz de agir por antecipação e evitar prejuízos maiores. Dependendo da natureza e da criticidade do sistema, tais prejuízos podem ser incalculáveis. Por exemplo, falhas nos requisitos de desempenho e confiabilidade em um sistema de controle de caldeiras pode implicar na explosão da caldeira e, consequentemente, na perda de vidas humanas.

Apesar da sua importância, requisitos de qualidade são difíceis de serem avaliados e mensurados, de forma a gerar evidências do seu pleno atendimento. Uma das razões para essa dificuldade é a necessidade de se avaliar o software em tempo de execução, o que é uma tarefa complexa que envolve grandes volumes de dados a serem analisados.

No presente estudo é proposto um processo para monitoração de requisitos de qualidade de software baseado em estilos arquiteturais, denominado *ArMoni*. Por essa razão, é crucial a compreensão de conceitos de engenharia de software, dentre eles ter clara a definição de alguns estilos arquiteturais amplamente utilizados. Pretende-se, para cada estilo arquitetural considerado, identificar suas potencialidades em termos de requisitos de qualidade, além de identificar seus elementos críticos, a serem monitorados para cada requisito. Dessa forma, a monitoração de requisitos de qualidade deve ser considerada de forma sistemática e criteriosa. Para isso, o processo proposto utilizará o *framework* para monitoração de requisitos de qualidade denominado *QuAM Framework* em um experimento avaliativo (LIMA, 2016; SILVA, 2015).

Para avaliar o processo proposto, o trabalho de Silva (2015) foi relevante para o desenvolvimento dessa dissertação, uma vez que a ferramenta por ele desenvolvida foi aprimorada no trabalho de Lima (2016) e possibilitou uma alternativa para a monitoração de requisitos de qualidade. Com o trabalho desenvolvido por Lima (2016) foi obtido um *framework* de monitoração que será o instrumento de monitoração utilizado para analisar um sistema real, por usuários com diferentes níveis de experiência e de conhecimento do sistema alvo de monitoração.

1.2 Objetivos

1.2.1 Objetivo Geral

O objetivo geral desse trabalho é sistematizar e viabilizar tecnicamente a monitoração de requisitos de qualidade a partir de características dos estilos arquiteturais utilizados no projeto da arquitetura de software.

1.2.2 Objetivos Específicos

Os objetivos específicos são:

- Definir diretrizes para identificar pontos do software que devem ser monitorados, a partir dos estilos arquiteturais adotados.
- Definir um processo para sistematizar a monitoração do software a partir do mapeamento entre estilos arquiteturais e requisitos de qualidade.
- Automatizar atividades do processo proposto utilizando um *framework* de monitoração já existente.
- Definir diretrizes para análise dos *logs* gerados na monitoração do software.

1.3 Visão Geral da Solução Proposta

Há uma diversidade de estilos arquiteturais e o interesse em monitorar os requisitos de qualidade apoiados por esses. Para cada um deles há aspectos particulares que exigem formas diferenciadas para realizar a monitoração de um requisito. Nesse contexto é importante delimitar critérios para realizar a monitoração dos requisitos de qualidade do sistema, uma vez que isso proporciona maior qualidade para o software desenvolvido.

Foi utilizado o *framework* desenvolvido no trabalho de Lima (2016), denominado *QuAM Framework*, para realizar a avaliação da solução proposta aplicando-a na monitoração de uma aplicação real. Através do reúso de software Lima (2016) elaborou um

framework, a partir da ferramenta de Silva (2015), que realiza a monitoração intrusiva de requisitos de qualidade. Os referidos trabalhos são resultantes de pesquisas realizadas na UFAL dentro da área de Engenharia de Software, onde o primeiro é um trabalho de conclusão de curso e o segundo uma dissertação de mestrado.

O *QuAM Framework* é baseado em anotações e permite anotar todas as classes do sistema. Contudo, anotar todas as classes do sistema para realizar a monitoração não é viável e pode interferir diretamente no comportamento do sistema. A monitoração de todas as classes do sistema o tornaria demasiadamente lento, além de dificultar consideravelmente a análise dos dados de monitoração.

O presente trabalho foca nessa limitação quanto à monitoração. De modo que é proposto um processo centrado na arquitetura de software a fim de identificar pontos estratégicos que precisam ser monitorados. Dessa maneira, é possível obter dados significativos de monitoração, mas sem comprometer o comportamento do sistema.

O intuito do processo proposto nesta dissertação é mostrar no contexto de cada estilo arquitetural os componentes que serão monitorados e como isso será realizado, além de identificar as classes desses componentes que devem ser monitoradas a partir de padrões de projeto normalmente utilizados na codificação de componentes de software. Essa é uma tarefa importante, uma vez que o mapeamento entre arquitetura de software e código não é uma tarefa simples.

Os trabalhos de Silva (2015) e Lima (2016) oferecem o meio para monitorar os requisitos de confiabilidade e eficiência no software, mas não explicitam com monitorar, quando monitorar e como analisar os dados gerados na monitoração.

A partir dos estilos arquiteturais adotados são identificados elementos potencialmente críticos, os quais devem ser monitorados e o que deve ser monitorado em cada um desses componentes. Ao identificá-los é possível realizar análises sobre os dados obtidos. Contudo, este trabalho não abrange ferramentas de visualização dos dados. Através do processo proposto pretende-se analisar como monitorar confiabilidade e eficiência em cada um dos estilos arquiteturais que foram especificados dentro deste trabalho. Para isso, a discussão focará em 6 (seis) estilos arquiteturais, segundo Garlan et al. (2010), comumente utilizados na construção de sistemas computacionais.

1.4 Relevância do Trabalho

Considerando a evolução dos sistemas computacionais e a complexidade inerente a construção desses, garantir qualidade ao processo de desenvolvimento de software é fundamental para construir sistemas que atendam não somente a requisitos funcionais, mas também requisitos de qualidade.

Dessa forma, este trabalho apresenta relevância acadêmica na área de engenharia de

software, com ênfase na monitoração de requisitos de qualidade através de estilos arquiteturais. É proposto um processo para avaliar se alguns requisitos estão sendo efetivamente satisfeitos pelo sistema. Assim, a solução desenvolvida neste estudo poderá auxiliar desenvolvedores(as) e engenheiros(as) de software a monitorar requisitos, tais como confiabilidade e eficiência, além de identificar possíveis gargalos estruturais que podem comprometer o atendimento dos requisitos.

1.5 Metodologia

Para desenvolver esta dissertação, inicialmente foi conduzida uma revisão da literatura. Foram utilizados os sistemas da ACM (*Association for Computing Machinery*) e IEEE (*Institute of Electrical and Electronics Engineers*) que são fontes importantes de artigos científicos e cujos materiais são acessíveis através do ambiente acadêmico da UFAL. Foram utilizadas algumas palavras-chave para a busca dos artigos como: *quality attributes, quality requirements, non-funcional requirements, software monitoring, architectural design, software architecture, architectural style* que são relevantes para a pesquisa. Os *abstracts* dos artigos encontrados foram analisados a fim de identificar se eram pertinentes para o tema pesquisado.

Foi realizado um levantamento para buscar soluções que tratem da avaliação de software centrada na arquitetura e modelos de implementação de componentes consistentes com abstrações arquiteturais. Também foram selecionados artigos sobre processos existentes com o intuito de: a) auxiliar na identificação das atividades do processo proposto para avaliação e monitoração de requisitos de qualidade; b) identificar os pontos-chave de cada estilo arquitetural considerado, do ponto de vista da monitoração; c) identificar atividades sistemáticas para apoiar o mapeamento entre arquitetura de software e código-fonte, a partir dos principais modelos de componentes atuais.

Foi realizado um levantamento e triagem de trabalhos a fim de identificar aqueles que se aproximam de uma perspectiva crítica sobre o tema escolhido. Na pesquisa bibliográfica foram selecionados 62 trabalhos nas temáticas de arquitetura de software, estilos arquiteturais, requisitos de qualidade e estratégias e ferramentas de monitoração de requisitos de qualidade. Além disso, foram consultadas as normas ISO/IEC (2003) e ISO/IEC (2011).

Em paralelo com a revisão de literatura, foi feito um estudo do *framework* de monitoração utilizado como base para obter os dados de monitoração. Além disso, foi planejado um experimento utilizando a técnica *Goal-Question-Metric* (GQM). A solução proposta foi avaliada através da monitoração de uma aplicação real e foi executado o experimento planejado com voluntários. Após o experimento, foram utilizados questionários para obter o *feedback* deles quanto ao experimento realizado.

1.6 Organização da Dissertação

O restante da dissertação está organizado da seguinte forma: o Capítulo 2 apresenta a revisão da literatura; o Capítulo 3 mostra os trabalhos relacionados; o Capítulo 4 aborda o processo de monitoração de requisitos de qualidade proposto nesta pesquisa; o Capítulo 5 apresenta o método de avaliação do processo de monitoração proposto; e, finalmente, o Capítulo 6 apresenta as conclusões e direcionamentos para trabalhos futuros.

Capítulo 2

Revisão da Literatura

Neste capítulo, será apresentada a revisão da literatura realizada. Esta revisão é baseada no guia proposto por Kitchenham (2004). Neste processo foram levantadas questões de pesquisa e estabelecido um protocolo de busca determinando critérios de inclusão e exclusão para a seleção de estudos. Esses passos são importantes para classificar aqueles que são mais relevantes em relação ao estudo desenvolvido nesta dissertação.

Os resultados obtidos são organizados em quatro grupos que abordam conceitos referentes à Qualidade de software, Desenvolvimento centrado na arquitetura, Relação entre requisitos de qualidade e arquitetura de software e Monitoração de software. Essa discussão é oportuna para compreender como decisões arquiteturais podem favorecer o alcance dos requisitos de qualidade no software. Também são apresentados estilos arquiteturais e suas principais características. Além disso, posteriormente, no Capítulo 3 são discutidos os trabalhos relacionados obtidos por meio da revisão da literatura.

2.1 Metodologia

2.1.1 Questões de Pesquisa

Apesar da monitoração ser uma solução usual na indústria, existem alguns desafios ao realizá-la. Dessa forma, surgem alguns questionamentos relevantes: a) Como minimizar a interferência da monitoração na própria execução do sistema? b) O que monitorar em um sistema para aferir determinados requisitos de qualidade? c) Como realizar tal monitoração, de forma a acoplar e desacoplar facilmente os monitores? d) Como analisar os dados de *log*?

Tais perguntas nos levam a buscar formas confiáveis de realizar a monitoração de um software e garantir a qualidade dele. Desse modo, pretende-se responder à questão de pesquisa principal:

- Como monitorar e analisar requisitos de qualidade em um sistema de software?

Com base na questão de pesquisa principal, algumas questões de pesquisa (QP) mais específicas foram elaboradas:

- QP1 - Quais os principais requisitos de qualidade dos estudos analisados?
  - QP1.1 Como são determinados esses requisitos?
  - QP1.2 Quais os objetivos de monitoração que esses estudos estabelecem?
- QP2 - De qual forma esses requisitos são monitorados?
  - QP2.1 Como o alvo de monitoração é determinado?
  - QP2.2 Os requisitos de qualidade são monitorados de forma dinâmica ou estática?
- QP3 - É possível uma solução genérica para identificar aspectos de monitoração baseada em estilos arquiteturais?
  - QP3.1 Quais os requisitos de qualidade apoiados por cada cada um dos principais estilos arquiteturais?
  - QP3.2 Quais os requisitos de qualidade que podem ser monitorados na arquitetura de software?
  - QP3.3 Como analisar os dados da monitoração?

Nesta dissertação, a revisão da literatura apresenta a literatura existente sobre monitoração de requisitos de qualidade de software. Também são consideradas de grande relevância para esse estudo as soluções propostas que apresentam abordagens para avaliação de requisitos de qualidade de software.

Para responder a QP1 tem-se como base a ISO/IEC 9126. Essa é fundamental para compreender quais são os requisitos de qualidade que sistemas de monitoração buscam avaliar. Assim, tem-se nela a referência para determinar quais requisitos de qualidade são alvo de monitoração a partir da arquitetura.

Sobre a QP2 há uma diversidade de estilos e padrões arquiteturais e o interesse em monitorar os requisitos de qualidade a partir das características estruturais e comportamentais de cada um deles. Para cada uma deles há aspectos particulares que exigem formas diferenciadas para realizar a monitoração de um requisito.

A QP3 aponta para a busca de uma solução que possa atender de forma genérica a monitoração de requisitos de qualidade. Visto que a diversidade de arquiteturas posta em análise na QP2 exige abordagens diferenciadas para realizar a monitoração, o intuito é construir uma abordagem que determine critérios para identificar requisitos de qualidade genericamente.

2.1.2 Protocolo utilizado na revisão da literatura

A partir das questões de pesquisa, o próximo passo foi estabelecer os critérios de seleção para realizar a revisão da literatura e obter estudos relacionados as questões estabelecidas. Utilizando uma *string* de busca, a consulta automática foi realizada nas seguintes bases de dados: *ACM Digital Library*[1] e *IEEEXplore Digital Library*[2]. Os estudos foram obtidos através de busca automática nas bases de dados mencionadas através da utilização de alguns termos de pesquisa agrupados em forma de uma *string* de busca, que está no idioma Inglês, descrita na Figura 2.1.

Foram considerados os seguintes descritores: atributos de qualidade, requisitos de qualidade, requisitos não-funcionais, monitoração de software, projeto arquitetural, arquitetura de software e estilos arquiteturais. A busca apenas por meio da *string* na Figura 2.1 não apresentou uma grande quantidade de materiais para estudo e, desse modo, também foram realizadas combinações entre os descritores citados para obter maior variedade de artigos.

Figura 2.1: *String* de busca automática.

(“quality attributes” OR “quality requirements” OR “non-functional requirements”) AND (“software monitoring” OR “architectural design” OR “software architecture” OR “architectural styles”)

Fonte: Elaborada pela autora.

O período de tempo considerado nas pesquisas foi os últimos 26 anos. Dessa forma, foram selecionados artigos publicados entre 1990 e 2016. Foram considerados relevantes artigos em inglês ou em língua portuguesa que apresentassem estudos relevante sobre a monitoração de requisitos de qualidade dentro de sistemas ou que tivessem relação com requisitos não-funcionais. Além disso, foi importante considerar se apresentavam uma metodologia clara e possuiam dados suficientes para embasar as conclusões obtidas. Foram considerados artigos de conferência, revista ou *survey*.

No processo de seleção dos artigos foi utilizada a ferramenta StArt [3]. Alguns critérios de inclusão e exclusão determinados para selecionar os artigos filtrados com a busca automática. Os critérios de inclusão admitidos foram: 1) Estudos que após a leitura do título e *abstract* estavam relacionados ao tema e podiam ser relevantes; 2) Estudos publicados entre 1990 e 2016; 3) Estudos que atendem alguma das questões de pesquisa; 4) Estudos que abordam algum requisito de qualidade estabelecido pela NBR ISO/IEC 9126; 5) Estudos que relacionam estilos arquiteturais e requisitos de qualidade; 6) Estudos que apresentam algum procedimento para verificar e validar processos de software (por exem-

[1] http://dl.acm.org/
[2] http://ieeexplore.ieee.org/Xplore/home.jsp
[3] http://lapes.dc.ufscar.br/tools/start_tool

plo, estudo de caso, experimentos); 7) Estudos que a solução proposta consegue realizar a monitoração de requisitos. Também foram adicionados artigos clássicos e já conhecidos pelo grupo de pesquisa, a fim de complementar o embasamento teórico do trabalho.

Os critérios de exclusão admitidos foram: 1) Estudos que não são artigos completos (*slides* de *Power point*, resumo, etc.); 2) Estudos que não estão relacionados com as questões de pesquisa; 3) Estudos publicados antes de 1990; 4) Estudos duplicados; 5) Estudos que apesar de terem título e/ou *abstract* relacionado não tinham conteúdo relevante para esse estudo. A extração de dados foi realizada com a leitura integral dos artigos selecionados.

Uma vez definidos alguns passos importantes realizados para a revisão de literatura, é importante frisar que a concepção de sistemas complexos requer planejamento para obter um sistema aceitável que apresente as propriedades desejadas pelos *stakeholders*. Nesse contexto, a arquitetura de software é um artefato importante desde as fases iniciais do desenvolvimento de software, principalmente se tratando dos requisitos de qualidade pretendidos para o sistema. Visando trazer uma melhor compreensão, foram incluídos artigos que apresentam alguns conceitos pertinentes sobre a relação entre qualidade de software e arquitetura de software; além de artigos sobre os principais estilos arquiteturais e algumas das vantagens e desvantagens que esses apresentam e, por fim, artigos sobre monitoração de software. Ainda, foram considerados livros que abordam especificamente sobre qualidade de software e documentação de arquitetura de software.

O resultado da revisão da literatura foi agrupado em quatro grupos: (1) Qualidade de software; (2) Desenvolvimento Centrado na Arquitetura; (3) Relação entre Requisitos de Qualidade e Arquitetura de Software e (4) Monitoração de Software. Os trabalhos relacionados, que também foram fruto da revisão de literatura, são apresentados no Capítulo 3.

2.2 Resultados da pesquisa

Para realizar as buscas foi tomada como base a *string* de busca da Figura 2.1. Tendo em vista que as interfaces das bases de dados utilizadas não apresentam exatamente a mesma forma de inserir os dados para busca, foram realizadas diferentes combinações dos descritores apresentados na Subseção 2.1.2. Foi possível perceber nos resultados uma grande quantidade de materiais que foram disponibilizados pelo mecanismo de busca, mas que não tiveram relevância para o objetivo da pesquisa. Uma das razões para isso foi o mecanismo sugerir muitos materiais que estavam relacionados com a palavra "software". Desse modo, será observado mais adiante que poucos foram os estudos que, de fato, foram analisados. Essa consideração é importante, pois quanto aos estudos relacionados a monitoração de software percebeu-se a escassez de material. Foi notado que a maioria dos

estudos que apresentaram o descritor "software monitoring" estavam relacionados com QoS e "Web Services". Esses estudos não foram considerados por não ser o foco desta pesquisa. Por essa razão, foram inseridos manualmente outros materiais que contribuíram para esta pesquisa. Tais materiais foram obtidos, por exemplo, através da leitura das referências de artigos selecionados pelo mecanismo de busca, além de relatórios técnicos e livros que tratam sobre arquitetura de software, requisitos de qualidade e estilos arquiteturais. Na Figura 2.2 estão dispostos os resultados.

Figura 2.2: Resultado da busca no IEEE e ACM

	String	IEEE		ACM		Manualmente
		Encontrados	Relevantes	Encontrados	Relevantes	
1	"software monitoring"	108	2	23	1	3
2	"quality attributes" AND "software monitoring"	14	2	0	0	3
3	"quality requirements" AND "software monitoring"	8	0	0	0	3
4	"non-functional requirements" AND "software monitoring"	10	1	0	0	-
5	("quality attributes" OR "quality requirements" OR "non-functional requirements") AND ("software monitoring")	29	2	**	**	3
6	("quality attributes" OR "quality requirements" OR "non-functional requirements") AND ("software monitoring" OR "architectural design" OR "software architecture" OR "architectural styles")	3,881	29	**	**	33

Fonte: Elaborada pela autora.

Na Tabela 2.2 estão as combinações utilizadas nas buscas. Com a primeira *string* utilizando o descritor **"software monitoring"** foram obtidos 108 resultados no IEEE, contudo poucos foram significativos para o tema da pesquisa e foram selecionados 2 artigos. No site da ACM foram obtidos 23 resultados, onde apenas um foi considerado relevante. Manualmente, foram inseridos 3 artigos que se relacionam a monitoração de software. Esses mesmos artigos foram contabilizados na inserção manual relacionada as *strings* 2, 3 e 5, já que tratam-se das buscas sobre monitoração de software.

A segunda *string* **"quality attributes" AND "software monitoring"** proporcionou 14 resultados, dos quais 2 artigos foram selecionados. Com a terceira *string* **"quality requirements" AND "software monitoring"** foram obtidos 8 resultados e nenhum artigo foi selecionado. Sobre a quarta *string* **"non-functional requirements" AND "software monitoring"** foram obtidos 10 resultados, sendo apenas 1 resultado selecionado. No ACM as *strings* de busca 2, 3 e 4 não geraram resultados.

Com a quinta *string* **("quality attributes" OR "quality requirements" OR "non-functional requirements") AND ("software monitoring")** foram obtidos 29 resultados e foram selecionados 2 artigos. Esses dois artigos são iguais a 2 dos 3 artigos obtidos a partir da *strings* 2 e 4. Essa busca não foi aplicada no ACM seguindo essa mesma construção da *string* devido a forma de inserção dos dados, por isso o uso dos asteriscos. Contudo, isso não configurou um problema, pois com as *strings* 1, 2, 3 e 4 foi possível fazer combinações que equivalem a esse *string*.

Por último, a sexta *string* no IEEE forneceu 3,905 resultados. Essa *string* foi útil para ter uma visão geral quanto aos estudos que envolvem todos os descritores utilizados nessa pesquisa. Foram selecionados 29 artigos, os quais foram base para a revisão de literatura. Ainda, foram inseridos manualmente 33 materiais dentre artigos, relatórios técnicos e livros. Devido a forma como os descritores são inseridos no ACM a sexta *string* não pôde ser aplicada.

A Figura 2.3 apresenta a distribuição temporal dos resultados para a *string* 6, onde houve maior quantidade de estudos selecionados. Na *string* 1 houve dois estudos de 2012 no IEEE; 1 estudo de 2016 no ACM e manualmente foi inserido 1 estudo de 2014, 1 estudo de 2015 e um do ano de 2016. A *string* 2 resultou em 1 estudo de 2009 e 1 de 2010, no IEEE; nenhum estudo no ACM e manualmente foram considerados os mesmos estudos da *string* 1, bem como para a *string* 3 e 5. A *string* 3 não apresentou resultados no IEEE e ACM. A *string* 4 apresentou 1 estudo de 2004 e nenhum no ACM. A *string* 5 apresentou um estudo de 2004 e um de 2010.

2.3 Qualidade de Software

Segundo Koscianski e Soares (2007) uma das primeiras questões relacionadas à qualidade é como julgá-la. Apesar de cotidianamente a qualidade de diversos produtos ser avaliada, não temos um referencial padrão para estabelecer critérios claros que sirvam para julgar um produto. Então, para resolver esse impasse, normalmente, são levantados requisitos. O problema de definir e avaliar qualidade não está totalmente resolvido pelos requisitos, mas uma abordagem importante é relacionar os dois conceitos. Isso é necessário para haver um ponto de referência para julgar um produto. Na definição de requisitos de software um ponto importante a ser considerado é o papel de diferentes clientes em um mesmo projeto. Essa não é uma tarefa trivial, principalmente em grandes projetos que envolvem muitas pessoas e funções, os chamados *stakeholders*. A qualidade do produto tem o propósito de satisfazer o cliente e isso implica ao(à) desenvolvedor(a) de software habilidade para gerenciar conflitos entre os diferentes *stakeholders* quando está definindo os requisitos do software.

Nessa perspectiva, uma das tarefas fundamentais de um(a) desenvolvedor(a) é cons-

Figura 2.3: Distribuição Temporal dos Estudos da String 6

Distribuição Temporal dos Estudos		
String 6		
Ano	**IEEE**	**Manualmente**
1994	4	-
1995	1	-
1996	1	-
1997	1	-
1998	0	-
1999	1	1
2000	2	0
2001	0	0
2002	3	1
2003	2	0
2004	0	1
2005	0	4
2006	1	0
2007	3	0
2008	1	0
2009	2	0
2010	3	1
2011	1	5
2012	1	4
2013	1	4
2014	2	2
2015	2	3
2016	1	3

Fonte: Elaborada pela autora.

truir uma arquitetura que englobe o mais próximo possível os requisitos funcionais e requisitos de qualidade dentro das restrições especificadas para o sistema. Por essa razão, o(a) desenvolvedor(a) deve entender os impactos que os elementos arquiteturais possuem sobre a capacidade do sistema reunir tais requisitos e manipular os elementos apropriadamente (CARRIERE; KAZMAN; WOODS, 1999).

Logo, é necessário considerar um conjunto de fatores que influenciam na construção do produto e influenciam no julgamento do cliente. Alguns fatores a serem considerados são: tamanho e complexidade do software a ser construído, número de pessoas envolvidas no projeto, ferramentas utilizadas, custos associados à existência de erros e custos associados à detecção e remoção de erros (KOSCIANSKI; SOARES, 2007).

É preciso ressaltar que devido à complexidade dos sistemas, nem sempre é tão simples distinguir os requisitos e não há um consenso sobre esse termo. Ele pode ser representado como uma declaração abstrata de alto nível ou apresentar uma definição detalhada de uma função do sistema, dependendo do tipo de requisito. Conforme Koscianski e Soares (2007), “os requisitos de um software são as descrições sobre seu comportamento, funções

e especificações das operações que deve realizar e especificações sobre suas propriedades e atributos". Os requisitos definem o que o software deve fazer. Dois pontos de vista, de modo geral, são o do(a) cliente(a) e o do(a) desenvolvedor(a). O(A) cliente detalha o trabalho que ele(a) realiza e como ele(a) deseja que o software o(a) auxilie; e o(a) desenvolvedor(a) define a maneira como o software deve funcionar internamente.

2.3.1 Requisitos de Qualidade

O desenvolvimento de software na indústria tem mostrado que os requisitos de qualidade são de grande relevância para o sucesso dos projetos. A percepção sobre a importância desses requisitos têm se delineado pela recorrência de problemas como: decisões arquiteturais inadequadas, insatisfação dos clientes, prazo e custos acima do esperado, entre outros (SILVA; ALBUQUERQUE; BARROSO, 2016). Os problemas apontados são aspectos que mesmo quando em menor nível dentro do desenvolvimento de software geram produtos com baixa qualidade, consequentemente comprometendo o uso deste e, em casos mais críticos, levando ao descarte desse tipo de software.

Nessa perspectiva, surgem trabalhos que discutem sobre a integração de requisitos não-funcionais à arquitetura e buscam construir modelos e abordagens para tornar mais clara a definição de requisitos de qualidade (ou requisitos não-funcionais). A seguir são apontados alguns trabalhos com esse propósito. O intuito é conhecer quais os objetivos dos trabalhos que vêm discutindo sobre esse tema.

No estudo de Ameller et al. (2012) são realizadas entrevistas com arquitetos de software para analisar como eles decidem quais requisitos de qualidade são mais relevantes nos projetos que executam, como determinam esses requisitos e como são documentados e validados. Como respostas a esses questionamentos, foi observado que os tipos de requisitos considerados mais relevantes foram desempenho e usabilidade e que os arquitetos não compartilhavam um vocabulário comum dos tipos de requisitos de qualidade, além de mostrar confusão quanto aos conceitos. Sobre a documentação, os requisitos de qualidade normalmente não são documentados e mesmo quando documentados, a documentação não é precisa. Foi observado que os arquitetos não utilizam ferramentas específicas para gerenciar esses requisitos. Além disso, a maioria dos entrevistados não validavam efetivamente os requisitos, mas alegavam que esses eram satisfatoriamente atingidos ao fim dos projetos.

Ainda, no estudo de Condori-Fernandez e Lago (2015) é realizada uma revisão sistemática de literatura sobre a priorização de requisitos de qualidade em equipe de desenvolvimento. Um dos pontos observados foi quais requisitos são considerados mais importantes, onde foi levantada uma lista dos dez requisitos de qualidade mais importantes de acordo com a revisão. Os requisitos de qualidade considerados foram: modificabilidade, reusabilidade, adaptabilidade, escalabilidade, desempenho, usabilidade, confiabilidade,

interoperabilidade, disponibilidade e segurança.

Silva, Albuquerque e Barroso (2016) apresentam uma revisão sobre a situação atual de organizações no que se trata da elicitação dos requisitos não-funcionais. São analisados diferentes papéis dentro da equipe de desenvolvimento e os motivos pelos quais muitas vezes esses requisitos não são elicitados. Lagerstedt (2014) aborda sobre a experiência prática do uso de ferramentas para realizar testes automatizados e verificadores que proporcionem um *feedback* rápido sobre erros relacionados ao requisitos não-funcionais.

Khatter e Kalia (2013) discutem sobre os desafios de integrar requisitos não-funcionais em desenvolvimento dirigido a modelo de sistemas e fornece uma abordagem baseada em decisões de projeto arquitetural para integrar os requisitos não-funcionais à arquitetura. Ameller et al. (2013) apresentam uma pesquisa que aborda sobre práticas relevantes para arquitetos(as) de software lidarem com requisitos não-funcionais. A pesquisa é dividida em duas partes. A primeira analisa os requisitos não-funcionais do ponto de vista das atividade de engenharia de software, como: elicitação, documentação e validação. A segunda parte investiga como tais requisitos influenciam decisões arquiteturais. Liu et al. (2012) propõem uma abordagem baseada em padrão para o projeto de requisitos não-funcionais e integrar o resultado desse projeto a modelos funcionais UML. O objetivo é obter modelos mais compreensíveis sobre requisitos funcionais e não-funcionais. Bajpai e Gorthi (2012) tratam sobre a importância dos requisitos não-funcionais em vários domínios, onde traz uma visão geral sobre os mesmos em termos de conceitos e quais são mais presentes em cada domínio apontado no estudo. Saadatmand, Cicchetti e Sjödin (2012) apresentam uma abordagem genérica onde analisa modelos de projeto de sistema no que se trata da satisfação de requisitos não-funcionais para permitir a avaliação de *trade-offs*.

Iqbal (2011) faz uma revisão de literatura onde trata sobre as diferentes abordagens disponíveis sobre modelagens de requisitos não-funcionais. São abordadas as abordagens: orientada a objetivos, orientada a aspectos e orientada a objetos. Wang, Song e Chung (2005) apresentam uma metodologia para detalhar projetos e selecionar padrões de projeto que sejam mais adequados para atender a determinado requisitos não-funcionais. Cysneiros e Leite (2004) apresentam um processo para elicitar requisitos não-funcionais, analisar sua interdependências e traçá-los em modelos conceituais. O foco é em modelos conceituais utilizado UML (*Unified Modeling Language*).

Após essa breve apresentação de estudos pertinentes aos requisitos não-funcionais (denominados também requisitos de qualidade), vale ressaltar sobre a importância de estar atento ao fato que requisitos de qualidade pode estar em harmonia, mas também pode existir em um estado de tensão mútua. Por exemplo, um sistema que busca maximizar a portabilidade pode sacrificar o desempenho; um sistema que tem alta escalabilidade pode obter esse requisito tendo como resultado perda de eficiência. Desse modo, percebe-se que decisões de *trade-offs* precisam ser tomadas e essas decisões devem estar baseadas em um conjunto explícito de prioridades. Contudo, é importante frisar a impossibilidade

de obter todos os requisitos de qualidade simultaneamente. Além disso, para se alcançar um determinado requisito de qualidade, outros requisitos podem estar diretamente ou indiretamente relacionados. Sendo assim importante refletir sobre a interdependência entre eles (KAZMAN; BASS, 1994; DABBAGH; LEE, 2013; HENNINGSSON; WOHLIN, 2002; BOEHM, 2015).

Nessa perspectiva, o sucesso de um projeto está fortemente relacionado com o projeto arquitetural. Contudo, projetar uma arquitetura não é uma tarefa simples e, por vezes, pode ser também subjetiva, quando se leva em consideração a experiência que o(a) desenvolvedor(a) de software possui. Dentre os desafios que surgem no projeto arquitetural está o *trade-off* entre atributos de qualidade. As soluções para resolver esse tipo de impasse satisfazem diferentes requisitos funcionais e de qualidade em diferentes proporções (SILVA et al., 2015). Dessa forma, talvez a atividade mais complexa durante o desenvolvimento de uma aplicação é transformar a especificação de requisitos em uma arquitetura do software (LUNDBERG et al., 1999).

Uma forma para articular meios de alcançar um requisito de qualidade é identificar *cenários* em que esse pode ser mensurado ou observado. Os cenários são úteis para descrever situações particulares de um sistema. Um cenário consiste nos seguintes aspectos: o *estímulo* que requisita uma resposta da arquitetura; a *fonte do estímulo*; o *contexto* dentro do qual o estímulo ocorre; os tipos de *elementos do sistema* envolvidos na resposta; *possíveis respostas* e as *métricas* usadas para caracterizar a resposta da arquitetura (BASS; KLEIN; BACHMANN, 2002; BASS; CLEMENTS; KAZMAN, 2003; KLEIN; KAZMAN, 1999).

Para tornar mais claro, será exemplificado um cenário para o requisito de desempenho. O cenário é o seguinte: Eventos externos chegam no sistema periodicamente durante uma operação normal. Em média, o sistema deve responder dentro de um intervalo de tempo específico. No cenário descrito os eventos externos que chegam no sistema são o estímulo e fica expresso que eles vem de fora do sistema. O contexto é a operação que ocorre normalmente, ou seja, sem saída inesperadas. O próprio sistema caracteriza o elemento do sistema envolvido nesse cenário. Por fim, o intervalo de tempo para resposta representa a resposta que o sistema deve retornar e isso implica definir métricas para esse tempo de resposta (BASS; KLEIN; BACHMANN, 2002).

Além disso, com base em Bachmann et al. (2005), existem três abordagens na arquitetura de software com relação a requisitos de qualidade: (1) Abordagem de requisitos não-funcionais (RNF) que se originou na Universidade de Toronto; (2) Abordagem de modelo de atributos de qualidade; e (3) Abordagem intuitiva. A *abordagem RNF* criada por Chung, Nixon, Yu e Mylopoulos[4] tem como premissa que não é precisamente determinado que uma arquitetura de software proverá requisitos de qualidade. Em outras palavras, o

[4]Mais informações podem ser obtidas em Chung et al. (2000)

que se pode determinar são as decisões arquiteturais que podem ajudar ou prejudicar os requisitos de qualidade. Essa abordagem apresenta dois problemas a serem observados. O primeiro é por não haver uma precisão quanto as decisões de projeto, o(a) desenvolvedor(a) deve confiar em sua intuição sobre as decisões tomadas. O segundo aspecto é que avaliar se as decisões ajudam ou prejudicam um requisito de qualidade depende da análise do contexto e dos demais atributos que estão envolvidos.

Por sua vez, a *abordagem de modelo de atributo de qualidade* desenvolve métodos para mensurar os atributos. As ISO 9126-1, 9126-2 e 9126-3 fornecem descrições e métricas para atributos de qualidade. Sobre essa abordagem algumas restrições são observadas, tais como: 1) Nem todo atributo de qualidade tem um modelo preditivo que relacione decisões arquiteturais com métricas de qualidade de software; 2) Nem toda arquitetura possibilita a análise de vários modelos de requisitos de qualidade (BACHMANN et al., 2005).

Por último, a *abordagem de projeto intuitivo* envolve métodos de projeto de arquitetura de software centrado em requisitos de qualidade. Alguns exemplos de trabalhos nessa perspectiva são o método *Attribute Driven Design* (ADD) e método QASAR de Bosch[5]. No método ADD, dentre outros passos, os cenários são utilizados para especificar requisitos de qualidade. O ADD define os seguintes aspectos para determinar a arquitetura de software de um sistema: Requisitos de qualidade, Restrições de projeto e Requisitos Funcionais (BACHMANN et al., 2005; PETROV; BUY, 2011).

Desse modo, o desafio de trabalhar com requisitos de qualidade é que esses geralmente são difíceis de ser verificados no produto. Quando o usuário determina algumas das características desejadas para o produto, o problema com esses requisitos surge quando se abre a possibilidade para as interpretações sobre eles. Uma forma de lidar com isso é utilizar um modelo que direcione a especificação de requisitos. O SQuaRE é um modelo que permite a especificar características de qualidade de software e pode contribuir para tornar os requisitos mais precisos (KOSCIANSKI; SOARES, 2007).

2.3.2 ISO/IEC 9126-1 e SQuaRE: ISO/IEC 25000

A NBR ISO/IEC 9126-1 (Qualidade do produto de software), juntamente com a NBR ISO/IEC 14598 (Avaliação de produto de software), substituem a NBR 13596 (Tecnologia de informação - Avaliação de produto de software - Características de qualidade e diretrizes para o seu uso). Isso ocorreu, pois as características de qualidade e as métricas associadas podem ser úteis não só a avaliação do produto de software, mas também para a definição de requisitos de qualidade e outros usos.

A elaboração da 9126-1 e da 14598 procurou criar um conjunto de documentos que

[5]Mais informações podem ser obtidas em (BASS; CLEMENTS; KAZMAN, 2003) e através Bosch, J.: *Design & use of software architectures* (Addison-Wesley, London, 2000).

abrangem aspectos relevantes de um programa de qualidade com foco em produtos de software. A ISO 9126-1 aborda os instrumentos necessários para realizar uma avaliação e determina um modelo de qualidade para o produto, bem como descreve como aderir qualitativa e quantitativamente à qualidade. A ISO 14598 envolve aspectos gerenciais, tais como uma metodologia na empresa e documentação relevante que deve acompanhar o processo. As perspectivas de desenvolvedor(a) e cliente são levadas em consideração (KOSCIANSKI; SOARES, 2007).

Contudo, foi elaborada a norma ISO/IEC 25000 que é uma evolução das normas ISO/IEC 9126-1 e ISO/IEC 14598, a qual é chamada SQaRE (*Software product Quality Requirements and Evalution*) que significa Requisitos de Qualidade e Avaliação de Software. O projeto SQuaRE reorganizou o material existente das duas normas anteriores, mas não fez mudanças radicais nelas. O modelo hierárquico de qualidade proposto na ISO 9126 continuou válido (KOSCIANSKI; SOARES, 2007; OUHBI et al., 2013).

O projeto SQuaRE resulta de um esforço e consenso de centenas de pesquisadores, desse modo representa a soma de experiências em torno do assunto. Ele adotou uma nova divisão de assuntos em cinco tópicos: Requisitos de Qualidade, Modelo de Qualidade, Gerenciamento de Qualidade, Medições e Avaliação. Cada divisão é composta por um conjunto de documentos que tratam de um assunto específico. Os *Requisitos de Qualidade* já estavam presentes na ISO 9126-1 e são retomados pela SQuaRE para estabelecer objetivos de qualidade para um produto. Para garantir a qualidade de um produto, não é suficiente apenas realizar medidas, mas também determinar valores-alvo que tenham sido especificados antecipadamente. Esses valores-alvo fazem parte da especificação de requisitos de qualidade. Contudo, as normas 9126-1 e 14598 continuam válidas e podem ser utilizadas, o SQuaRE surgiu para reorganizá-las (KOSCIANSKI; SOARES, 2007).

O *Gerenciamento de Qualidade* apresenta documentos voltados a todos aqueles que possivelmente fazem uso dela, como gerentes, programadores, avaliadores ou compradores. O *Modelo de Qualidade* corresponde principalmente à ISO/IEC 9126-1. A referida ISO define conceitos de qualidade externa, interna e em uso que possibilitam guiar diferentes perspectivas de avaliação. A *Medição* consiste em definir propriamente o que é uma medição e aspectos relacionados à realização dessa tarefa, além de propor uma série de métricas que podem ser utilizadas ou adaptadas pelos usuários das normas. Por último, a *Avaliação* é a realização de uma avaliação de qualidade na qual os resultados são confrontados com os modelos definidos pelos diferentes públicos da norma, como usuário, desenvolvedores e compradores, entre outros (KOSCIANSKI; SOARES, 2007).

A SQuaRE identifica três tipos de qualidade que são relativas a diferentes situações envolvendo um produto de software: *Qualidade de uso, Qualidade externa* e *Qualidade interna*. Esses tipos de qualidade referem-se à Qualidade no ciclo de vida do software. A *Qualidade de uso* corresponde ao ponto de vista de um usuário e refere-se a um programa sendo executado, isso depende de fatores como o hardware utilizado, o treinamento do

usuário, a tarefa realizada, etc. A *Qualidade externa* considera o produto como uma caixa-preta, pois sua arquitetura, código e modo de funcionamento não são explicitamente apresentados. Os testes de funcionamento do produto são formas de verificar a qualidade externa. Esses testes envolvem desde sub-rotinas até módulos ou sistemas completos (KOSCIANSKI; SOARES, 2007).

Por último, a Qualidade interna refere-se a avaliação estática do produto e é analisada a arquitetura interna do produto, ao contrário dos tipos de qualidade mencionados acima. A organização do código ou a complexidade algorítmica são critérios que podem indicar, prováveis custos de manutenção e velocidade de execução (KOSCIANSKI; SOARES, 2007).

Há situações distintas em que a qualidade de software pode ser avaliada. A *qualidade interna* pode ser analisada em relação à representação estática, como códigos, diagramas e especificações. Logo, esses são aspectos que não envolvem o software funcionando. A *qualidade externa* relaciona-se com a execução do software e essa pode ser medida através das partes que o compõe. Execuções em ambientes controlados também são consideradas como avaliações da qualidade externa. Além disso, tem-se a *qualidade de uso* que se refere a situações reais de execução (KOSCIANSKI; SOARES, 2007).

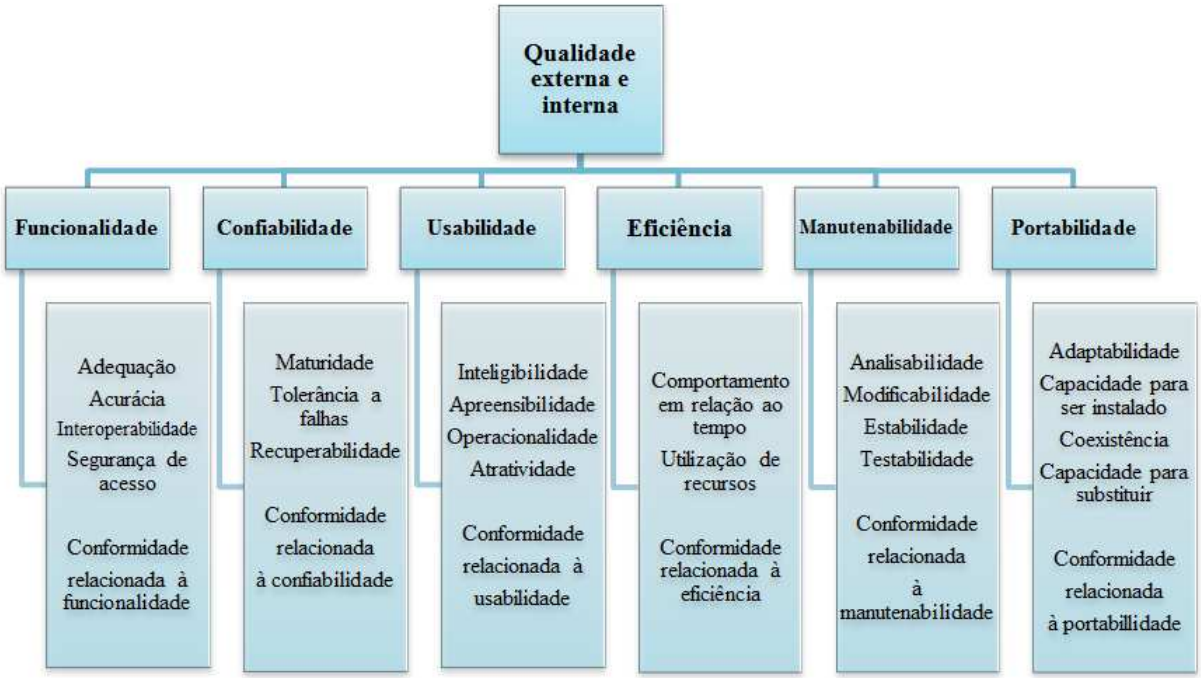
É importante salientar que os tipos de qualidade são interdependentes, isto é, ao definir requisitos de qualidade em uso direciona-se a decisão sobre os requisitos de qualidade externa e interna, onde por sua vez quando os requisitos de qualidade externa e interna são obtidos isso é refletido na qualidade de uso. Para definir os requisitos de qualidade é importante ter um ponto de partida que forneça exemplos de métricas externas e internas que podem ser utilizadas como ponto de partida para definir um sistema de avaliação. O modelo de qualidade SQuaRE é hierárquico (KOSCIANSKI; SOARES, 2007).

A qualidade em uso apresenta apenas um nível de divisão e contém quatro características: efetividade, produtividade, segurança e satisfação. Por sua vez, a qualidade interna ou externa de um produto é descrita por seis características que, por sua vez, são divididas em subcaracterísticas e estas em atributos. Esse modelo está em conformidade com o que já estava definido na ISO 9126-1 e define características de qualidade hierarquicamente. A ISO 9126-1 não define atributos, pois estes variam de produto a produto. Desse modo, os usuários da norma devem identificar os atributos relevantes no projeto quando for definir o modelo de qualidade. Essa divisão hierárquica permite estabelecer uma descrição de requisitos de qualidade (KOSCIANSKI; SOARES, 2007).

As características de qualidade do produto de software definidas na ISO 9126-1 podem ser usadas para especificar requisitos funcionais e não-funcionais do usuário. Ela apresenta um modelo de qualidade externa e interna que categoriza os atributos de qualidade de software em seis características (*funcionalidade, confiabilidade, usabilidade, eficiência, manutenabilidade* e *portabilidade*) as quais são subdivididas em subcaracterísticas. As subcaracterísticas podem ser medidas por meio de métricas externas e internas (ISO/IEC, 2003;

ME; CALERO; LAGO, 2016; CANESSANE; SRINIVASAN, 2013; UM et al., 2011). Na Figura 2.4 estão organizadas essas características de qualidade.

Figura 2.4: Modelo de qualidade para qualidade interna e externa.



Fonte: Adaptada de ISO/IEC (2003).

As seis características de qualidade presentes na Figura 2.4 serão definidas conforme o que está especificado na (ISO/IEC, 2003). *Funcionalidade* é a capacidade do software de prover funções que atendam às necessidades explícitas e implícitas, quando o software estiver sendo utilizado sob condições especificadas. *Confiabilidade* refere-se à capacidade do produto de software de manter determinado nível de desempenho, quando usado em condições especificadas. *Usabilidade* é a capacidade do produto de software de ser compreendido, aprendido, operado e atraente ao usuário, quando usado sob condições especificadas. *Eficiência* é a capacidade do produto de software de apresentar desempenho apropriado, relativo à quantidade de recursos usados, sob condições especificadas. *Manutenabilidade* refere-se à capacidade do produto de software de ser modificado. As modificações podem incluir correções, melhorias ou adaptações do software devido a mudanças no ambiente e nos seus requisitos ou especificações funcionais. Por último, *Portabilidade* é a capacidade do produto de software de ser transferido de um ambiente para outro.

Assim, a NBR ISO/IEC 9126-1 é um elemento importante para determinar requisitos de qualidade. Por isso, é considerada dentro desse estudo para determinar quais requisitos de qualidade serão monitorados.

2.4 Desenvolvimento Centrado na Arquitetura

A avaliação e projeto de arquitetura de software envolve tarefas complexas, pois requer decisões para resolver os *tradeoffs* e satisfazer os *stakeholders*. O conhecimento para tomar as decisões arquiteturais adequadas podem estar além das capacidades do(a) desenvolvedor(a). Com o reconhecimento da importância que as decisões arquiteturais apresentam para o desenvolvimento de software, várias abordagens tem sido desenvolvidas para dar suporte a processos arquiteturais. Algumas delas são: Architecture Tradeoff Analysis Method (ATAM), Visões 4+1, Rationale Unified Process (RUP) e Desenvolvimento centrado na arquitetura (BABAR; GORTON; JEFFERY, 2005).

O desenvolvimento de software centrado na arquitetura apresenta alguns aspectos críticos. Um desses aspectos é a seleção de um estilo ou padrão arquitetural adequado. Domínios computacionais cada vez mais complexos tornam mais desafiante a tarefa de desenvolver aplicações que atendam às características desejadas para solucionar um problema. Uma alternativa para lidar com essa complexidade é realizar a decomposição do problema em uma estrutura hierárquica que possibilite sua compreensão e descrição através de suas partes (XAVIER; WERNER; TRAVASSOS, 2002).

A atividade de projeto arquitetural baseia-se na conversão de um conjunto de requisitos na arquitetura de software que os preenche, ou pelo menos facilita o preenchimento deles (CHUNG; NIXON; YU, 1994b). É importante ressaltar que a atividade de projeto arquitetural baseada apenas na função que o software realiza não é suficiente para garantir que esse seja considerado bom. Embora ele atenda o que realmente foi especificado nos requisitos funcionais, ele pode não apresentar algumas características de qualidade de natureza não-funcional.

2.5 Arquitetura de Software e Requisitos de Qualidade

Requisitos de qualidade devem ser considerados ao longo do projeto, implementação e desenvolvimento de software. Um aspecto importante é compreender que nenhum requisito de qualidade é dependente apenas do projeto, nem apenas da implementação ou desenvolvimento. A satisfatória interação entre as diferentes fases de desenvolvimento proporcionam que o software realmente atenda aos requisitos de qualidade pretendidos. Assim, bons resultados dependem de uma arquitetura bem elaborada, bem como dos detalhes de implementação corretos (BASS; CLEMENTS; KAZMAN, 2003).

As arquiteturas de software podem seguir alguns estilos ou padrões específicos que irão guiar a especificação arquitetural. A seleção de um estilo ou padrão é um dos aspectos críticos no desenvolvimento de software centrado na arquitetura. A observância de determinado estilo arquitetural possibilita que soluções conhecidas sejam reaplicadas a muitos projetos reutilizando modelos de composição e regras de aplicação (SHAW; GARLAN,

1996). A reutilização de modelos arquiteturais de software é uma alternativa para reduzir investimentos para desenvolver aplicações do mesmo domínio computacional.

É possível encontrar benefícios e dificuldades na utilização de determinado estilo ou padrão arquitetural com relação a algumas características de qualidade. Não há um critério único que apóie a escolha de um conjunto mínimo de características para a avaliação da utilização dos estilos e padrões arquiteturais. Um modelo de qualidade padrão para a avaliação de produtos de software é a ISO/IEC 9126 (XAVIER; WERNER; TRAVASSOS, 2002).

Conforme Chung, Nixon e Yu (1994a) uma abordagem mais disciplinada para projeto arquitetural é necessária para desenvolver a habilidade para entender a interação de restrições de sistema em alto nível e a análise relacional por trás das escolhas arquiteturais, para reusar conhecimento arquitetural relativo aos requisitos de qualidade e para analisar o projeto com respeito às restrições não relacionais.

A técnica de modularização de software é uma abordagem que permite a representação estrutural de inter-relação entre módulos. Através dela é possível a identificação de características importantes, por exemplo, a observação de que sistemas complexos apresentam um padrão comum de organização dos módulos que os compõem. Os trabalhos nessa área contribuíram para a formalização do conceito que atualmente é denominado *arquitetura de software* (STAA, 2000).

A arquitetura de software e os requisitos de qualidade estão interligados. Nessa relação destaca-se a conexão entre a arquitetura do sistema e a habilidade de alcançar requisitos de qualidade dentro do sistema. Diferentes arquiteturas são capazes de satisfazer requisitos de qualidade de acordo com as características próprias que cada uma delas apresentam. As decisões de projeto incorporadas na arquitetura de software são fortemente influenciadas pelos requisitos de qualidade que se deseja alcançar. Desse modo, percebe-se que os requisitos de qualidade dirigem o projeto de arquitetura de software (KAZMAN; BASS, 1994; BASS; KLEIN; BACHMANN, 2002; KOZIOLEK, 2012; ZAYARAZ; THAMBIDURAI, 2005).

Além disso, os requisitos de qualidade não são apenas abstratos, eles manifestam-se no software e hardware dentro de um ambiente que o usuário ou a instituição necessita. Dessa forma, quando determinado requisito é considerado importante não se pode apenas afirmar que tal requisito seja importante, mas também deve-se tomar decisões determinando atributos de qualidade e determinando as prioridades em termos de cenários de uso (KAZMAN; BASS, 1994).

Logo, percebe-se o quanto a atividade de elaborar uma arquitetura de software pensando na perspectiva de alcançar requisitos de qualidade pode ser complexa. Segundo Bachmann et al. (2005), projetar uma arquitetura que atinja requisitos de qualidade é uma das tarefas mais exigentes que um(a) desenvolvedor(a) enfrenta. É exigente por incluir a falta de especificidade nas exigências sobre os requisitos, uma escassez de conhe-

cimento documentado de como projetar para um determinado atributos de qualidade e os *trade-offs* envolvidos para alcançar atributos de qualidade.

Para muitos sistemas, requisitos de qualidade como desempenho, confiabilidade, segurança e modificabilidade são tão importantes quanto garantir que o software retorne as respostas corretas. A capacidade do software de produzir respostas corretas não é útil se esse leva muito tempo responder, ou se o sistema não fica "acordado" tempo suficiente para entregá-la, ou, ainda, se ele revela seus resultados para os concorrentes ou inimigos. Na arquitetura questões como essas serão endereçadas. Nesse sentido, faz-se necessário refletir quais os aspectos relevantes quando se quer fazer esse endereçamento sobre cada requisito de qualidade. As soluções para essas preocupações são de natureza arquitetural e ao(à) desenvolvedor(a) cabe buscar as soluções mais adequadas e torná-las acessíveis aos demais envolvidos no processo (GARLAN et al., 2010).

Na documentação da arquitetura há três obrigações relacionadas aos requisitos de qualidade. Primeiramente, ela deve indicar quais requisitos de qualidade dirigiram o projeto. Segundo, deve capturar as soluções encontradas para satisfazer os requisitos de qualidade. Por fim, deve ser capaz de ser uma solução convincente para os requisitos necessários. Esses aspectos tem por objetivo capturar informação suficiente para que o(a) desenvolvedor(a) seja capaz de analisar se o sistema construído a partir dela será capaz de fornecer os requisitos esperados (GARLAN et al., 2010).

A princípio, alguns pontos podem ser considerados ao se pensar na arquitetura e em como essa pode se relacionar com os requisitos de qualidade. Para ilustrar pode-se pensar sobre características que o sistema requer para cumprir um determinado requisito. Conforme Garlan et al. (2010), para obter *alto desempenho* é necessário explorar potencial paralelismo pela decomposição do trabalho em processos cooperando ou sincronizando; gerenciar o volume de comunicação de rede e entre processos e frequência de acesso a dados; ser capaz de estimar latências esperadas e taxa de transferência; e identificar potenciais gargalos de desempenho. Se o sistema necessita de *alta precisão* é preciso ficar atento para como os elementos são definidos e usados e como seus valores fluem pelo sistema. Ao pensar em *segurança* é importante definir precisamente relacionamentos e comunicações restritas entre as partes; identificar as partes do sistema onde acesso não autorizado será mais danoso; e possibilidade de introduzir elementos especiais que apresentam alto grau de confiança. Se o requisito necessário se trata de *modificabilidade* e *portabilidade*, é preciso separar interesses cuidadosamente entre as partes do sistema para que uma mudança em um elemento não tenha uma repercussão através do sistema. Por último, se o objetivo é desenvolver um sistema incrementalmente é necessário manter a relação de dependência entre as partes que são entregues, mas evitar a síndrome "nada funciona até que tudo funcione".

Nesse sentido, desenvolver um método que guie o(a) desenvolvedor(a) de modo a pensar em requisitos de qualidade dentro da arquitetura deve apresentar alguns característi-

cas, tais como: incluir requisitos conhecidos e uma das fontes que descreve atributos de qualidade é a ISO 9126-1; prover gerenciamento de *trade-offs*; possibilitar a interpretação de conhecimento de requisitos de qualidade em termos de arquitetura de software e interpretar arquitetura de software em termos de requisitos de qualidade (BACHMANN et al., 2005).

2.5.1 Conceitos de Arquitetura de Software

Conforme Bass, Clements e Kazman (2003), a arquitetura de software de um programa ou sistema de computação é a estrutura ou estruturas de sistema que envolve elementos de software, as propriedades externamente visíveis e os relacionamentos entre eles. Ela é uma abstração do sistema que suprime detalhes de elementos que não afetam como eles usam, são usados, relacionam com, ou interagem com outros elementos. A arquitetura está preocupada com o lado público da interação entre os elementos, desse modo detalhes privados de implementação são não arquiteturais. Os elementos podem envolver implementação e processos, por exemplo. Esse podem ser um objeto, um processo, uma biblioteca, uma base de dados, etc. Arquitetura não é definida por uma única estrutura, por isso um projeto de um sistema pode ser dividido em unidades que são distribuídas para que equipes de desenvolvimento se encarreguem de implementá-las.

Nas definições comuns de arquitetura de software são encontrados alguns conceitos que descrevem a arquitetura como sendo um projeto em alto nível, uma estrutura global do sistema, estrutura de componentes de um programa ou sistema e seus interrelacionamentos, por exemplo. Tais definições são genéricas e devem ser analisadas cuidadosamente. Primeiramente, nem todas as tarefas relacionadas com projeto são arquiteturais, por exemplo, decidir estruturas de dados que serão encapsuladas, isto é, a escolha das estruturas de dados. Contudo, a escolha das interfaces para essas estruturas de dados é decididamente arquitetural (BASS; CLEMENTS; KAZMAN, 2003).

Considerar a arquitetura uma estrutura global induz a um conceito incorreto de que ela é única. As diferentes estruturas fornecem pontos sobre a engenharia que proporcionam para o sistema requisitos de qualidade que podem render o sucesso ou fracasso dele. A arquitetura de um sistema de software pode ser representada por visões diferentes e ortogonais entre si: **visão lógica, visão de desenvolvimento, visão de processos, visão física** e **visão de casos de uso**. A seguir será abordado sobre cada uma delas (BASS; KLEIN; BACHMANN, 2002; BASS; CLEMENTS; KAZMAN, 2003; BRITO; GUERRA; RUBIRA, 2007; PETROV; BUY, 2011).

A **visão lógica** relaciona-se com os requisitos funcionais do sistema. O sistema é decomposto em um conjunto de abstrações que fazem parte do domínio do problema. Essas abstrações são objetos e classes e são descritos através de diagramas, na UML, de classes e diagramas dinâmicos (sequência, colaboração, atividades, etc). A **visão de desen-**

volvimento volta-se para a estruturação do sistema em módulos de código. É possível representá-la na UML através de diagramas de componentes contendo módulos, bibliotecas e as dependências entre esses elementos. A **visão de processos** trabalha com a divisão do sistema em processos e processadores. Também chamada de *visão de concorrência*. Através dela requisitos não-funcionais como desempenho, tolerância a falhas e disponibilidade são tratados. A visão de processos é expressa na UML através de diagramas dinâmicos, assim como diagramas de componentes e implantação (BRITO; GUERRA; RUBIRA, 2007).

A **visão física** foca principalmente nos requisitos não-funcionais do sistema, de modo particular os que estão relacionados com sua organização física, como tolerância a falhas, escalabilidade e desempenho. Também conhecida como *visão de implantação*. A UML expressa a visão física através de diagramas de implantação. Por fim, a **visão de casos de uso**, também conhecida como *visão de cenários*, faz a integração das outras visões para descrever a funcionalidade para o usuário. Ela é utilizada desde o início do desenvolvimentos para expressar requisitos, até a fase em que os testes de aceitação são feitos. Na UML essa visão é representada por diagramas de casos de uso (BRITO; GUERRA; RUBIRA, 2007).

Além disso, é necessário compreender alguns termos relacionados com a arquitetura de software, que, às vezes, são confundidos com essa. Serão abordados brevemente os termos padrões arquiteturais, modelos de referência e arquiteturas de referência, afim de entender com esses termos estão relacionados com a arquitetura.

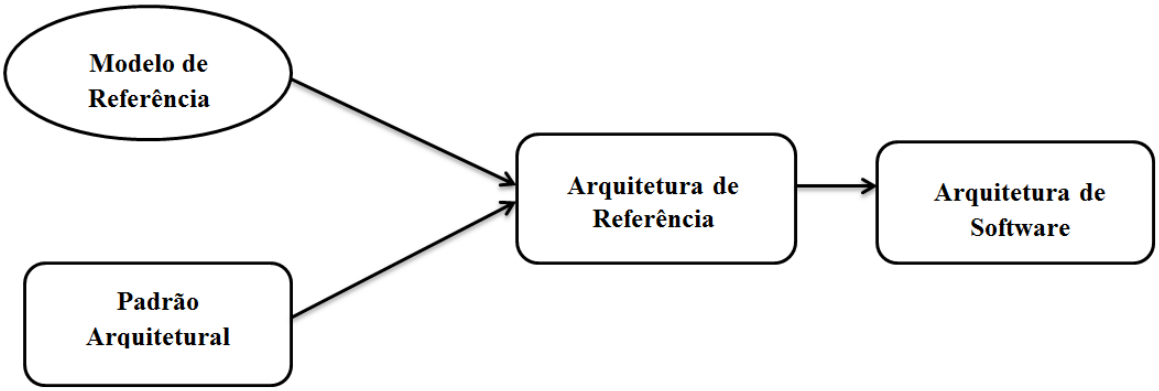
Primeiramente, uma padrão arquitetural envolve uma descrição de elementos e tipos de relação entre esses que possuem um conjunto de restrições de como eles podem ser usados. Então, um padrão pode ser pensado como um conjunto de restrições na arquitetura, em seus elementos e padrões de interação, que definem um conjunto ou família de arquiteturas que os satisfazem. Um padrão arquitetural não é uma arquitetura, mas pode proporcionar uma visualização do sistema que impõe restrições úteis na arquitetura, e consequentemente no sistema (BASS; CLEMENTS; KAZMAN, 2003). *Estilo arquitetural* é utilizado com o conceito aproximado ao de padrão arquitetural. Contudo, há diferenças entre esses dois termos. A discussão sobre os conceitos de padrão arquitetural e estilo arquitetural será retomada mais adiante, na Seção 2.5.2.

Modelo de referência é uma divisão de funcionalidade junto com o fluxo de dados entre as partes. Ele é um padrão de decomposição de um problema conhecido em partes que cooperativamente resolvem o problema. Eles advém de domínios maduros, resultando da experiência em resolver um determinado problema. Quando se aprende as partes padrão de compilador ou de um sistema de gerenciamento de banco de dados, está se explicitando o modelo de referência que essas aplicações possuem. Por sua vez, uma *arquitetura de referência* é um modelo de referência mapeado dentro de elementos de software que cooperativamente implementam a funcionalidade definida em um modelo de

referência e o fluxo de dados entre eles. Então, um elemento de software deve implementar parte de uma função ou de várias funções (BASS; CLEMENTS; KAZMAN, 2003).

Logo, percebe-se que padrões de arquitetura, modelos de referência e arquiteturas de referência não são arquiteturas de software. Esses constituem conceitos úteis que capturam elementos de uma arquitetura. Na Figura 2.5 é apresentado os relacionamentos entre modelos de referência, padrões de arquiteturais, arquiteturas de referência e arquiteturas de software. As flechas em cada forma na imagem indicam que conceitos subsequentes contêm mais elementos de projeto.

Figura 2.5: Termos relacionados com arquitetura de software.



Fonte: Adaptada de Bass, Clements e Kazman (2003).

Uma arquitetura de software deve descrever os componentes do sistema, suas conexões e suas interações, e a natureza das interações entre o sistema e o ambiente. Avaliar um projeto de sistema antes dele ser construído é uma boa prática de engenharia. Uma técnica que permite a avaliação de uma arquitetura candidata antes do sistema ser construído é de grande valor (BARBACCI et al., 1995).

Para que haja a interação entre componentes é necessária alguma forma de comunicação entre eles. A responsabilidade pelo estabelecimento dessa comunicação é atribuída a elementos arquiteturais denominados *conectores*. Um conjunto de componentes interligados pelos conectores define uma *configuração arquitetônica*. Criar, manter e evoluir a arquitetura de um sistema é uma tarefa difícil, nesse sentido há alguns princípios básicos que devem ser almejados: encapsulamento, baixo acoplamento, alta granularidade e alta coesão (BRITO; GUERRA; RUBIRA, 2007).

É importante ressaltar que a arquitetura do sistema influencia o desempenho e manutenção do mesmo. Desse modo, é possível perceber que ela está diretamente relacionada com o planejamento dos requisitos de qualidade desejados para um software. Segundo Kazman e Bass (1994), existe uma conexão íntima entre uma arquitetura de um sistema e a habilidade para alcançar qualidades particulares dentro do referido sistema. Como o

sistema cresce em tamanho e complexidade, a realização de requisitos de qualidades reside nas decisões arquiteturais.

A construção de uma arquitetura de software de um sistema endereçará determinados requisitos de qualidade necessários para o sistema operar adequadamente. Ao considerar diferentes requisitos, tais como, portabilidade, modificabilidade, usabilidade, segurança, desempenho e eficiência, entra-se em uma questão crítica, pois se deve refletir sobre os *trade-offs* entre esses requisitos.

O(A) desenvolvedor(a) de software necessita analisar *trade-offs* entre múltiplos atributos conflitantes para satisfazer requisitos do usuário. O principal objetivo é quantitativamente avaliar múltiplos atributos para chegar a um sistema global melhor. Deve-se não olhar apenas uma métrica universal, mas para a quantificação desses atributos individuais e os *trade-offs* entre essas diferentes métricas, começando pela arquitetura de software (BARBACCI et al., 1995).

Assim, deve-se considerar que alguns requisitos do sistema podem entrar em conflito. Requisitos funcionais podem entrar em conflito com requisitos de qualidade, ou mesmo requisitos de qualidade podem entrar em conflito com outros requisitos de qualidade. Essas situações exigem muita habilidade e planejamento dos desenvolvedores de software na determinação de quais requisitos serão atendidos e qual estilo arquitetural seria mais viável. Em alguns casos, pode ser utilizada a combinação de estilos arquiteturais para conciliar diferentes requisitos dentro do sistema.

2.5.2 Estilos Arquiteturais

O(A) desenvolvedor(a) de software necessita de meios que o(a) orientem para definir uma estrutura. Um dos métodos mais utilizados é o reuso de soluções conhecidas que foram consideradas adequadas para domínios específicos. Essas soluções bem-sucedidas são conhecidas como **padrões arquiteturais** (BRITO; GUERRA; RUBIRA, 2007). Um padrão arquitetural é formado a partir de experiências bem sucedidas na utilização de um **estilo arquitetural**. Fazendo analogia com o conceito de estilo arquitetural utilizado em construções como o Gótico, Grego ou Vitoriano, um estilo arquitetural de software apresenta características chave e regras para combiná-las de forma a determinar se a integridade arquitetural está preservada. Sendo assim um padrão arquitetural define um conjunto de tipos de elementos arquiteturais (tais como um repositório de dados ou um componente que computa uma função matemática); um projeto topológico de elementos indicando seus relacionamentos; um conjunto de restrições semânticas e um conjunto de mecanismos de interação que determinam a coordenação dos elementos através da topologia permitida (BASS; CLEMENTS; KAZMAN, 2003).

Os estilos arquiteturais estão crescendo em complexidade, com uma riqueza de vocabulário e numerosas restrições. Apesar de existirem várias ferramentas que ajudam

a analisar as arquiteturas de sistemas individuais, poucas pesquisas ajudam no projeto de estilos arquiteturais. Um dos aspectos mais importantes da arquitetura de software moderna é o uso sistemático de estilos arquiteturais, através da catalogação de padrões arquiteturais. Um estilo arquitetural define uma família de sistemas relacionados fornecendo um vocabulário de projeto arquitetural de domínio específico junto com restrições de como as partes devem trabalhar juntas. Alguns estilos definem padrões de desenvolvimento, definem alguma estrutura ou questões de projeto, outros podem definir fatores de comunicação. Geralmente, os sistemas usam uma combinação de um ou mais estilos (KIM; GARLAN, 2006; KIM; GARLAN, 2010; CHAVAN; MURUGAN; CHAVAN, 2015).

A escolha de padrões arquiteturais inclui o conhecimento em requisitos de qualidade, pois cada um deles influencia de diferentes formas tais requisitos (ME; CALERO; LAGO, 2016). Dessa maneira, a qualidade do projeto arquitetural tem um impacto direto sobre o sistema, de forma que as decisões de quais estilos arquiteturais serão adotados é de grande relevância. O uso de padrões ou estilos arquiteturais depende da própria experiência que o(a) desenvolvedor(a) possui em resolver problemas e familiarização com os estilos. Por essa razão a recomendação de um estilo arquitetural está atrelada ao requisitos de qualidade que são desejados para o software (CHEN et al., 2014).

Há inúmeros benefícios que são considerados quando estilos arquiteturais são utilizados, dentre eles o fato dos estilos fornecerem um vocabulário comum para arquitetos(as), consequentemente facilitando a rotina de projeto arquitetural. Outros benefícios são: 1) Os estilos favorecem o reúso de arquiteturas para muitos produtos; 2) Suportam interoperabilidade entre componentes e 3) Permitem análises especializadas conforme o(s) estilo(s) que a arquitetura apresenta. Além do mais, um grande número de estilos vem sendo criados, onde esses também derivam de elaborações de estilos já existentes. Contudo, definir um novo estilo não é uma tarefa trivial. Por isso, definir estilos se torna também um desafio intelectual e empírico (KIM; GARLAN, 2010).

Há alguns estilos arquiteturais amplamente usados. Para apresentar adequadamente os estilos arquiteturais considerados na definição do processo proposto, é necessário apresentar alguns conceitos comumente presentes na definição de estilos arquiteturais: *elementos*, *relação* e *propriedades*.

Primeiramente, um *elemento* é um bloco de construção da arquitetura que é característico de um estilo. Ele pode ser um módulo, um componente, um conector ou um elemento no ambiente do sistema. Ao descrever um elemento também se fala dos papéis que esse desempenha na arquitetura e as propriedades que ele possui. Uma *relação* define como os elementos cooperam para realizar o trabalho dentro do sistema, consequentemente também determina regras de como os elementos podem se relacionar ou não. Por sua vez, uma *propriedade* possui informação adicional sobre elementos e relações. Uma definição de um estilo contém o nome da propriedade e a descrição. Ao realizar a docu-

mentação de uma visão arquitetural são atribuídos valores às propriedades, os quais são frequentemente utilizados como uma forma de analisar a arquitetura e sua capacidade de prover os requisitos de qualidade esperados (GARLAN et al., 2010).

Com base em Garlan et al. (2010), os estilos arquiteturais podem ser classificados em três categorias: **estilos de módulos, estilos componente-conector** e **estilos de alocação**. Nos **estilos de módulos** os elementos básicos são os módulos. Esses são uma unidade de implementação que fornece um conjunto coerente de responsabilidades e pode ter a forma de uma classe, uma coleção de classes, uma camada, um aspecto ou qualquer coisa decomposição de uma unidade de implementação. Além disso, apresentam propriedades que podem expressar informações importantes e restrições associadas a eles. Algumas dessas propriedades são responsabilidades, visibilidade e autor ou proprietário. As relações que os módulos têm com outros módulos incluem: *é parte de*, *depende de* e *é um*.

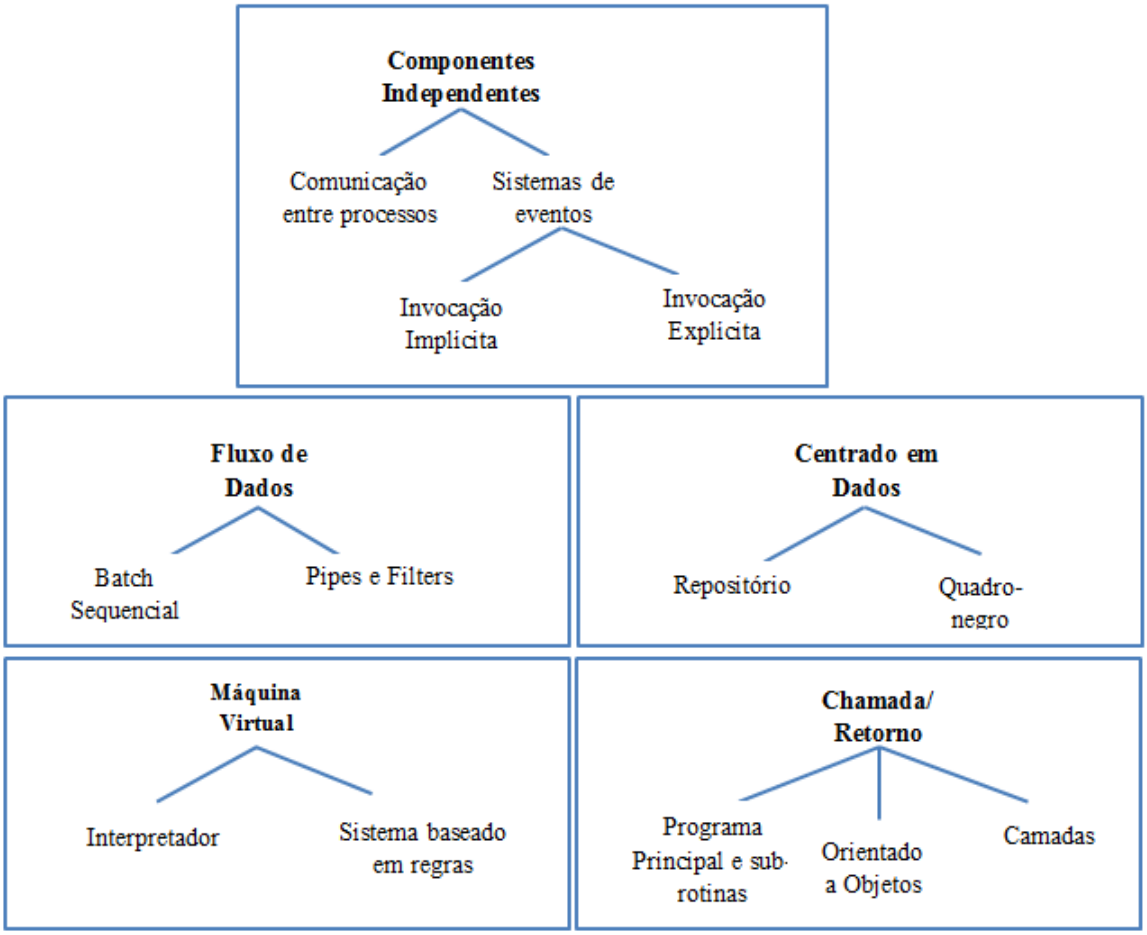
Os **estilos componente-e-conector** expressam comportamento em tempo de execução e seus elementos são os componentes e conectores. Um *componente* é uma das unidades principais de processamento de um sistema em execução e armazenamento de dados. Componentes podem ser serviços, processos, *threads*, *filters*, repositórios, *peers*, ou clientes e servidores, por exemplo. Eles apresentam interfaces em tempo de execução chamadas portas, que definem pontos de interação entre os componentes e o ambiente deles. Os *conectores* identificam os caminhos e papéis de interação entre componentes. Componentes e conectores se relacionam através de uma configuração arquitetural que define um modelo de como um conjunto de componentes estão conectados um ao outro por meio dos conectores. Conectores incluem *pipes*, filas, *request-reply*, protocolos, invocação direta, invocação dirigida a evento, chamadas de procedimentos, transmissão de evento e consultas a banco de dados, dentre outros. Componentes e conectores podem ser decompostos em outros componentes e conectores. Ainda, juntamente com a configuração arquitetural há propriedades associadas com os elementos. Por exemplo, se um componente executa tarefas periódicas, as propriedades podem definir prioridade e uso de CPU. Nesse caso os conectores, poderiam incluir latência e protocolo de interação. Arquiteturas componente-e-conector podem ser estruturadas em camadas, que são agrupamentos lógicos de componentes. O estilo multicamada é encontrado em Java EE. A forma como os componentes de um sistema estão organizados é conhecida como configuração arquitetural (GARLAN; SHAW, 1994; BRITO; GUERRA; RUBIRA, 2007; GARLAN et al., 2010; KIM; GARLAN, 2010; SHARAFI; GHAZVINI; EMADI, 2010; CHAVAN; MURUGAN; CHAVAN, 2015).

Por último, **estilos de alocação** descrevem o mapeamento de unidades do software para elementos de um ambiente no qual o software é desenvolvido ou executa. Esse ambiente pode ser *hardware* ou os sistemas de arquivos que suportam desenvolvimento ou implantação, por exemplo. Dentro desse trabalho será dada ênfase aos estilos componente-e-conector. De modo que, serão vistos mais detalhes sobre os estilos que estão nessa

categoria.

Na Figura 2.6 pode-se ver um pequeno catálogo de padrões arquiteturais que são categorizados em uma hierarquia de herança. Por exemplo, um sistema baseado em evento é um subestilo de componentes independentes. Sistemas de eventos apresentam dois subpadrões: invocação implícita e explícita. Por meio dessa imagem pode-se fazer uma representação mental sobre como os estilos arquiteturais estão organizados e relacionados.

Figura 2.6: Catálogo de padrões arquiteturais.



Fonte: Adaptada de Bass, Clements e Kazman (2003).

Com base em Garlan e Shaw (1994), a seguir serão tratados alguns estilos explicitando seus pontos principais. Conhecer as diferenças entre eles ajuda a ter um *framework* comum para visualizá-los. Tal *framework* adotado trata uma arquitetura de um sistema específico como uma coleção de *componentes computacionais* ou simplesmente componentes, juntos com uma descrição das interações entre esses componentes - os *conectores* (GARLAN; SHAW, 1994).

No estilo de **invocação implícita**, ao invés de invocar o procedimento diretamente, um componente pode anunciar um ou mais eventos. Outros componentes podem registrar interesse no evento associando um procedimento a ele. Quando o evento é anunciado, o

sistema invoca todos os procedimentos que foram registrados para ele. Então, um evento anunciado implicitamente causa a invocação de procedimentos em outros módulos. Os componentes nesse estilo são módulos cujas interfaces fornecem uma coleção de procedimentos e um conjunto de eventos. Os conectores em um sistema de invocação implícita incluem chamada de procedimento tradicional bem como as ligações entre os anúncios de eventos e chamadas de procedimento (GARLAN; SHAW, 1994).

Um benefício importante da invocação implícita é que ela fornece forte suporte para reúso. Qualquer componente pode ser introduzido no sistema registrando-o para os eventos daquele sistema. Além disso, componentes podem ser substituídos por outros sem afetar as interfaces de outros componentes no sistema. Contudo, quando muda a identidade de um componente que fornece alguma função, todos os módulos que importam aquele componente também devem mudar (GARLAN; SHAW, 1994).

Por outro lado, algumas desvantagens podem ser observadas. Os anunciadores de eventos não sabem quais componentes serão afetados por tais eventos. Desse modo, os componentes não podem fazer suposições sobre a ordem de processamento ou o que ocorrerá como resultado de seus eventos. Outro problema é em relação à mudança de dados. Às vezes, o dado pode ser passado com o evento, mas em outras situações sistemas de eventos podem confiar em um repositório compartilhado para realizar interação. Em casos de desempenho global e gerenciamento de recursos isso pode tornar-se um sério problema (GARLAN; SHAW, 1994).

No estilo denominado **pipes-and-filters** cada componente tem um conjunto de entradas e um conjunto de saídas. Um componente lê cadeias de dados em sua entrada e produz cadeias de dados em suas saídas, entregando uma instância completa do resultado em uma ordem padrão. Isso é geralmente realizado aplicando uma transformação local na cadeia de entrada e a saída então começa a ser computada incrementalmente antes da entrada ser consumida. Assim, os componentes são chamados *filters*. Os conectores desse estilo servem para conduzir cadeias, transmitindo saídas de um *filter* para a entrada de outro. Consequentemente, os conectores são chamados *pipes*. Os compiladores e sistemas distribuídos são exemplos bem conhecido de programa com essa arquitetura (GARLAN; SHAW, 1994; FERNANDEZ; ORTEGA-ARJONA, 2009).

Algumas vantagens dessa arquitetura são: permitir entender o comportamento global de entrada e saída do sistema como uma simples composição de comportamentos de filtros individuais; suporte a reúso; sistemas podem ser facilmente mantidos (*filters* novos podem ser adicionados a sistemas existentes e *filters* velhos são substituídos); suportam execução concorrente (cada *filter* pode ser implementado como uma tarefa separada e potencialmente executado em paralelo com outros *filters*).

Por outro lado, temos algumas desvantagens. Frequentemente realizam processamento em lote, embora possam processar dados incrementalmente o projetista é forçado a pensar em cada *filter* promovendo uma transformação completa dos dados de entrada e saída. Por

causa dessa característica sistemas de *pipes and filters* não são indicados para aplicações interativas. Outro aspecto é a dificuldade para manter correspondência entre dois *filters* separados, mas com fluxos relacionados. Por último, dependendo da implementação, eles podem forçar um menor denominador comum sobre a transmissão de dados, resultando em trabalho adicional para cada *filter* para analisar seus dados. Desse modo, ocasionando perda de desempenho.

No estilo **repositório** existem dois tipos distintos de componentes: uma estrutura de dados central representa o estado corrente e a coleção de componentes independentes opera no armazenamento de dados central. Interações entre o repositório e os componentes externos podem variar significativamente entre os sistemas. A escolha de controle distingue outras subcategorias: se os tipos de transações em uma *stream* de entrada de transações engatilha a seleção de processos para executar, o repositório pode ser uma *base de dados tradicional*. Se o estado corrente da estrutura de dados central está no gatilho principal para a seleção de processos para executar, o repositório pode ser um *blackboard* (GARLAN; SHAW, 1994; STIGER; GAMBLE, 1997).

Sistemas **blackboard** são tradicionalmente usados para aplicações que requerem interpretações complexas de processamento de sinal e reconhecimento de padrão. Outro exemplo de sistema de repositório são alguns ambientes de programação. Esses são frequentemente organizados como uma coleção de ferramentas junto com um repositório compartilhado de programas (GARLAN; SHAW, 1994).

Outro tipo de estilo arquitetural apresenta **abstração de dados e orientação a objetos**. Nesse estilo as representações de dados e suas operações primitivas associadas são encapsuladas em um tipo de dado abstrato ou objeto. Os componentes desse estilo são os objetos ou instâncias de tipos de dados abstratos. Os objetos são responsáveis por preservar a integridade de um recurso e interagem através de funções ou invocações de procedimentos (GARLAN; SHAW, 1994).

Sistemas orientados a objeto apresentam as seguintes vantagens: os objetos escondem a representação de seus clientes, de modo que é possível mudar a implementação sem afetá-los; o empacotamento de um conjunto de rotinas de acesso com os dados que eles manipulam permite aos *designers* decompor problemas dentro de coleções de agentes interagindo (GARLAN; SHAW, 1994).

Contudo, também algumas desvantagens podem ser apontadas. Para um objeto interagir com outro, por chamada de procedimento, ele deve conhecer a identidade desse outro objeto. Quando a identidade de um objeto muda, é necessário mudar todos os objetos que o invocam explicitamente. Além disso, existem problemas causados por efeitos colaterais. Por exemplo, um objeto A é usado pelos objetos B e C, se B realiza mudanças em A, em algum momento elas podem causar efeitos colaterais em C, e vice versa (GARLAN; SHAW, 1994).

O **estilo baseado em camadas** apresenta uma estrutura organizada hierarquica-

mente. Nele cada camada prove serviço para a camada acima dela e serve como cliente para a camada que está abaixo. Em alguns sistemas em camadas, camadas internas são escondidas das outras exceto da camada adjacente a outra. Os conectores são definidos por protocolos que determinam como as camadas irão interagir. Exemplos desse estilo arquitetural é o modelo OSI ISO. Sistemas em camadas possuem vantagens, tais como: oferece suporte para *design* baseado em níveis crescente de abstração e suporte para reuso (diferentes implementações de uma mesma camada podem ser utilizadas). Dentre as desvantagens pode-se apontar: nem todos os sistemas podem ser facilmente estruturados no modelo em camadas e dificuldade em encontrar os níveis certos de abstração (GARLAN; SHAW, 1994).

Outro estilo arquitetural caracteriza os **Interpretadores dirigidos por tabela**. Em uma organização por interpretador uma máquina virtual é produzida no software. Um interpretador inclui um pseudoprograma sendo interpretado e o próprio mecanismo de interpretação. O mecanismo de interpretação inclui a definição do interpretador e o estado corrente de sua execução. Desse modo, o interpretador geralmente tem quatro componentes: um mecanismo de interpretação para fazer o trabalho, a memória que contém o pseudocódigo sendo interpretado, uma representação do estado de controle do mecanismo de interpretação e a representação do estado atual do programa sendo simulado.

Após essa breve explanação sobre os estilos componente-e-conector com base em Garlan e Shaw (1994). Ainda é interessante analisar a classificação proposta em Garlan et al. (2010), na Figura 2.7, onde pode-se ver também como a junção de estilos pode gerar outros estilos, bem como amplia e torna mais compreensível a classificação já vista na Figura 2.6.

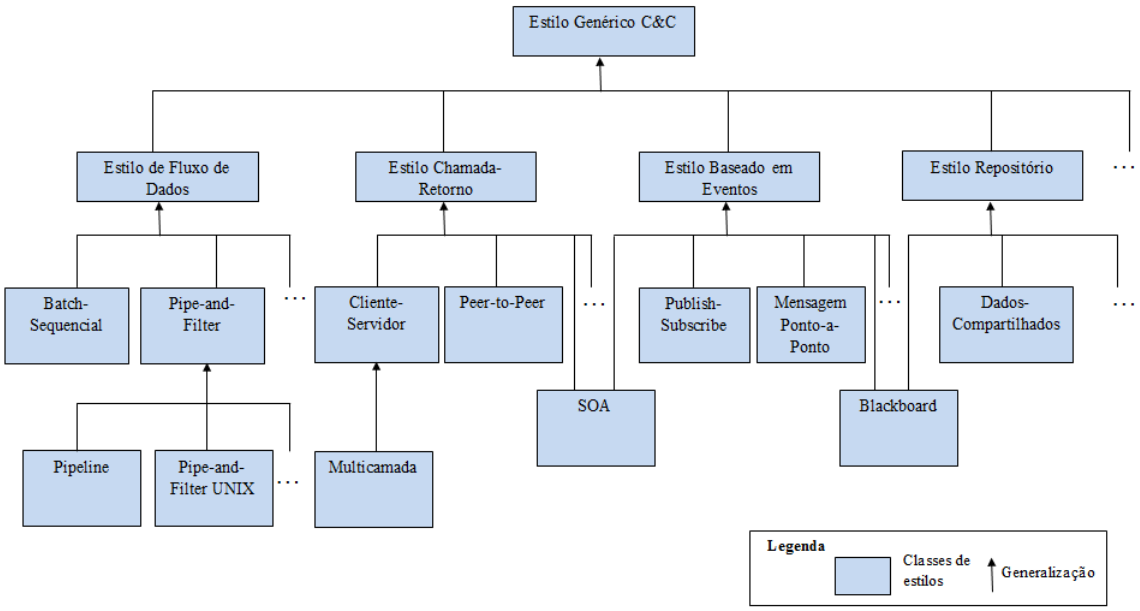
Um *estilo componente-conector* (C&C) apresenta um conjunto específico de componentes e conectores e as regras específicas de como esses elementos podem ser combinados. Uma de suas funções é capturar aspectos em tempo de execução de um sistema, por essa razão esse tipo de estilo também está associado com um modelo computacional que prescreve como dados e controle de fluxo acorre dentro do sistema projetado em um determinado estilo (GARLAN et al., 2010).

Escolher um estilo C&C depende da natureza das estruturas em tempo de execução do sistema. Por exemplo, se o sistema necessita acessar um conjunto de bases de dados legadas, o estilo deve ser provavelmente baseado em estilos de dados compartilhados. Em outro caso, se o sistema objetiva transformação de cadeias de dados, o estilo de fluxo de dados será provavelmente escolhido. Devido à diversidade de estilos, eles podem diferir bastante quanto às especificidades que apresenta. Desse modo, pode-se considerar várias categorias amplas de estilos, diferindo especialmente pelo modelo computacional subjacente. De modo geral, são apresentadas quatro categorias: **estilos chamada-retorno** nos quais os componentes interagem através de invocação síncrona de capacidades fornecidas por outros componentes; **estilos de fluxo de dados** onde a computação é dirigida

pelo fluxo de dados através do sistema; **estilos baseado em eventos** são aqueles em que os componentes interagem através de eventos assíncronas ou mensagens; **estilos repositório** são estilos nos quais componentes interagem através de grandes coleções de dados compartilhados ou persistentes (GARLAN et al., 2010).

A Figura 2.7 mostra uma representação geral de estilos C&C, onde há outros estilos que são especializações dessas categorias. Na maioria dos sistemas reais, vários estilos podem ser utilizados juntos.

Figura 2.7: Diagrama de Estilos C&C

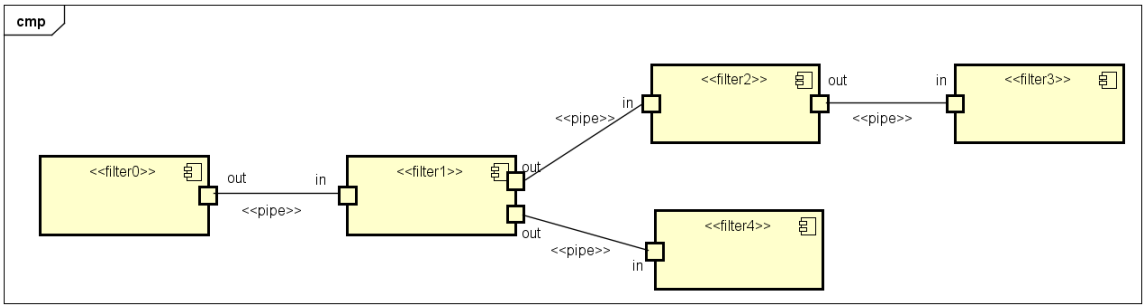


Fonte: Adaptada de Garlan et al. (2010).

Estilos de fluxo de dados apresentam um modelo computacional em os componentes atuam como transformadores de dados e os conectores transmitem dados de saída de um componente e entrada de outro. Cada componente no estilo de fluxo de dados tem um número de portas de entrada e portas de saída, onde seu trabalho é consumir dados em suas portas de entrada e escrever dados transformados para suas portas de saída. Na prática, há uma variedade de estilos de fluxo de dados. No início da computação, um estilo comum era o *batch sequencial*, no qual cada componente transforma todos os dados que ele possui antes de passar para o próximo componente consumi-los. Posteriormente, um novo estilo foi criado chamado de estilo *pipe-and-filter*, Figura 2.8.

Estilos de fluxo de dados são utilizados em domínios onde há processamento de *stream* e onde uma computação pode ser quebrada em um conjunto de passos de transformação desses dados. Exemplos *pipe-and-filter*: sistemas de processamento de sinais, sistemas construídos usando UNIX *pipes*, arquitetura de processamento de requisição do *Apache Web server*; o paradigma de *map-reduce* de mecanismos de busca, Yahoo! Pipes para processamento de *RSS feeds* (GARLAN et al., 2010).

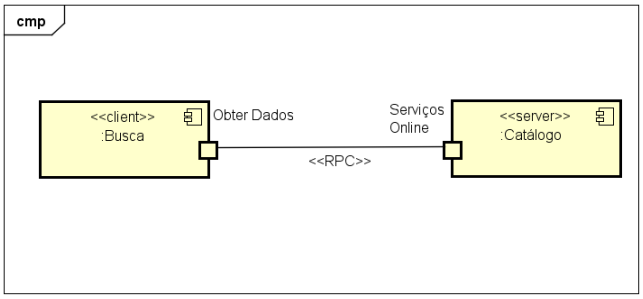
Figura 2.8: Diagrama UML do Estilo *Pipe-and-Filter*



Fonte: Elaborada pela autora.

Estilos de chamada-retorno apresentam um modelo computacional no qual componentes fornecem um conjunto de serviços que podem ser invocados por outros componentes. O termo serviço nesse contexto se refere à operação genérica ou uma função que pode ser invocada via conector chamada-retorno, não se referindo a serviços como em arquiteturas orientadas a serviços. Um componente invocando um serviço para ou fica bloqueado até que o serviço esteja completo. Desse modo, o estilo chamada-retorno é a analogia arquitetural de uma chamada de procedimento em linguagens de programação. Os conectores são responsáveis por transmitir a requisição de serviço do solicitante para o provedor e para retornar os resultados. São exemplos de estilos chamada-retorno os estilos *peer-to-peer* (Figura 2.8), cliente-servidor (Figura 2.9) e REST (*Representational State Transfer*) (GARLAN et al., 2010).

Figura 2.9: Diagrama UML do Estilo Cliente-Servidor



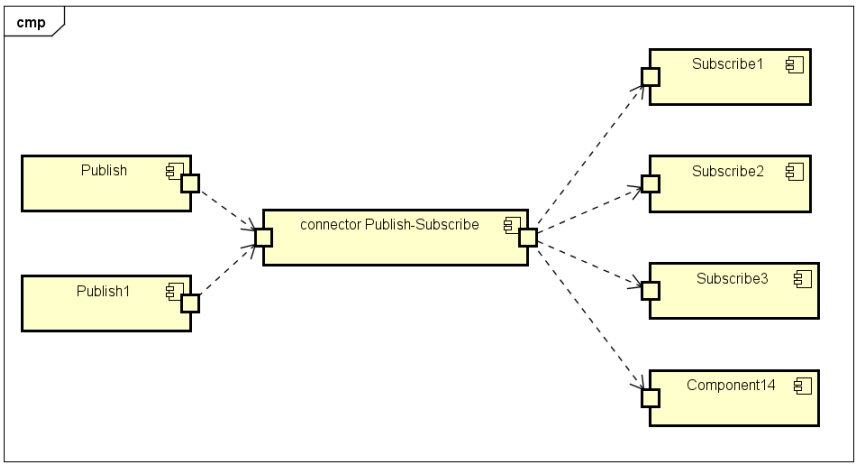
Fonte: Elaborada pela autora.

Exemplos de sistemas que utilizam o estilo cliente-servidor: 1) Sistemas de informação executando em redes locais, onde os clientes são aplicações GUI (*Graphical User Interface*) e o servidor é um sistema de gerenciamento de base de dados (SGBD); 2) Aplicações *Web* onde os clientes executam em navegadores *Web* e os servidores são componentes executando em um servidor *Web* (como o Tomcat). O *World Wide Web* é o exemplo melhor conhecido de um sistema que apresenta o estilo cliente-servidor. Ele é um sistema baseado em hipertexto que permite clientes (navegadores *Web*) acessarem informações sobre servidores distribuídos através da Internet. Clientes acessam a informação que está escrita

em *Hypertext Markup Language* (HTML), provida por um servidor *Web* usando *Hypertext Transfer Protocol* (HTTP). HTTP é uma forma de invocação *request/reply*. A conexão entre o cliente e o servidor termina depois que o servidor responde (GARLAN et al., 2010).

Estilos baseados em eventos possibilitam aos componentes se comunicarem através de mensagens assíncronas. Os elementos desses estilos são fracamente acoplados e há elementos que apresentam o comportamento de engatilhar eventos para outros elementos através de eventos. Em alguns estilos desse tipo os conectores são ponto-a-ponto transmitindo mensagens de modo similar ao estilo de chamada-retorno, mas permite maior concorrência pelo fato do remetente não precisar ficar bloqueado enquanto o evento é processado pelo destinatário. Em outros estilos os conectores podem enviar eventos para vários componentes e são chamados de *publish-subscribe*. Exemplos de sistemas que empregam o estilos *publish-subscribe*, ver Figura 2.10, são: 1) As interfaces gráficas de usuários, onde as ações de entrada de dados são tratadas como eventos que são direcionadas para os manipuladores adequados; 2) Aplicações baseadas em MVC (*Model-View-Controller*), quando os componentes *view* são notificados quando um estado de componente *model* muda; 3) Redes sociais, quando os "amigos" são notificados quando mudanças ocorrem na página *Web* da pessoa (GARLAN et al., 2010).

Figura 2.10: Diagrama UML do Estilo *Publish-Subscribe*

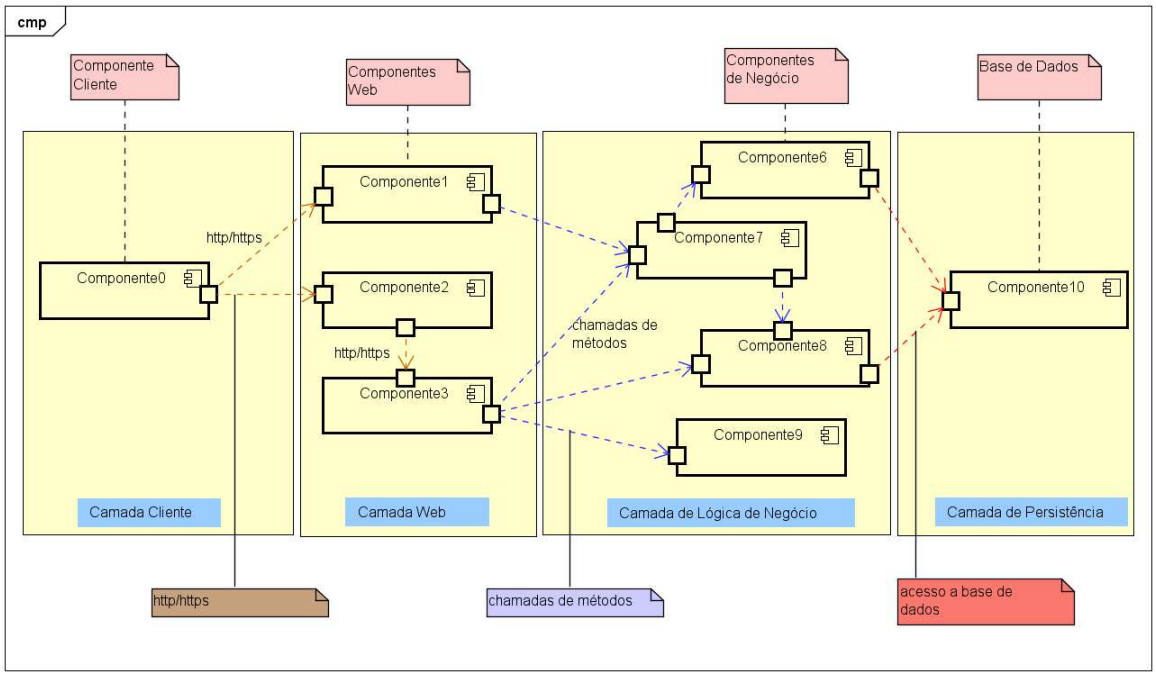


Fonte: Elaborada pela autora.

A SOA (*Service-Oriented Architecture*) une aspectos de estilos chamada-retorno e estilos baseado em eventos, como visto na Figura 2.7. Um conceito chave de SOA são os serviços. Serviços referem-se à abstração e modelagem de atividades de negócios do mundo real, fornecendo componentes reusáveis, ver Figura 2.11. Algumas características desse tipo de arquitetura são: orientada a processos; centrada nos negócios, baixo acoplamento, reuso e flexibilidade (YUE et al., 2010).

Estilos repositório contém um ou mais componentes chamados de repositórios, os

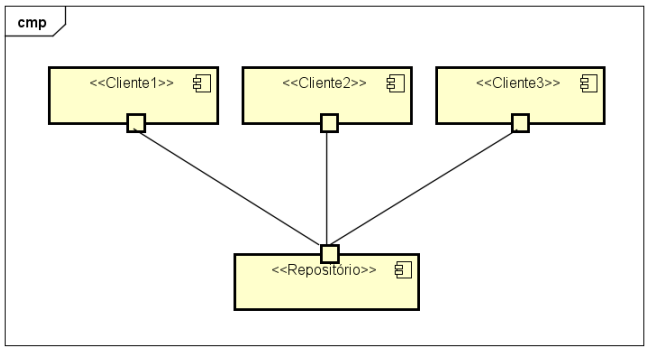
Figura 2.11: Diagrama UML do Estilo SOA



Fonte: Adaptado de Garlan et al. (2010)

quais normalmente armazenam uma grande quantidade de dados persistentes. Além desses componentes há outros que lêem e escrevem dados no repositórios. Frequentemente, o acesso aos repositórios é mediado por um software chamado sistema de gerenciamento de base de dados (SGBD) que fornece uma interface chamada-retorno para recuperar e manipular dados. MySQL é um exemplo de SGBD. Os sistemas em que os acessadores de dados são responsáveis por iniciar a interação com o repositório são chamados de *estilos de dados compartilhados*, ver Figura 2.12. Já em sistemas em que o repositório é responsável por notificar outros componentes quando os dados são modificados são chamados de estilos *blackboard*.

Figura 2.12: Diagrama UML do Estilo de Dados Compartilhados



Fonte: Elaborada pela autora.

Entendida as classificações dos estilos C&C também pode-se ver na Figura 2.7 que

estilos podem trabalhar juntos e formar outros estilos como, por exemplo, os estilos SOA, ver Figura 2.11, que agrega os estilos chamada-retorno e baseado em eventos; e o estilo *blackboard* que une os estilo baseado em eventos e estilo repositório.

Após uma visão mais detalhada dos estilos C&C pode-se ter uma compreensão de como esses estilos podem cooperar entre si para o funcionamento de um sistema, bem como os componentes e conectores nativos de cada estilo estão configurados arquitetonicamente.

Vale a pena ressaltar que estilos arquiteturais podem ser combinados entre si, a fim de compor uma arquitetura heterogênea, predominante em sistemas de software reais, especialmente os de médio e grande porte. No Capítulo 5 é apresentada a arquitetura de software de um sistema de tradução automática real, utilizado durante o experimento de avaliação da solução proposta. Nela são evidenciados os estilos arquiteturais que compõem o sistema.

Em suma, a seguir são apresentados pontos principais pertinentes a cada estilo arquitetural, com o intuito de facilitar a compreensão desses. As tabelas resumo estão baseadas em Garlan et al. (2010) e mostram os elementos de cada estilo apresentado, as relações entre eles, modelo computacional do estilo, restrições e a utilidade do ponto de vista de para quais finalidades seriam opções mais adequadas. As tabelas são referentes aos estilos *Pipe-and-Filter* (Tabela 2.1), Cliente-servidor (Tabela 2.2), *Peer-to-Peer* (Tabela 2.3), SOA (Tabela 2.4), Baseado em eventos (Tabela 2.5) e Dados compartilhados (Tabela 2.6).

2.6 Monitoração de Software

Monitoração de software objetiva obter informações relacionadas ao estado, comportamento e ambiente de um software em tempo de execução, bem como lidar com desvios no comportamento do sistema em relação aos requisitos o mais cedo possível. Ações relacionadas à tomada de decisão e adaptação do software podem se basear em informação extraída através da monitoração de software (WANG et al., 2009). Além disso, é importante ponderar sobre os impactos que a monitoração pode ter sobre o funcionamento de um sistema, alguns desses estão relacionados aos custos que a monitoração pode ter e impacto no desempenho do sistema ou serviços. Outros pontos que devem ser analisados é método de monitoração utilizado, localização de monitores, capacidade de processamento do monitor e modo de monitoração (HEWARD et al., 2010).

Uma das abordagens utilizadas para realizar a monitoração de software é o uso da Programação Orientada a Aspectos (POA). A POA, desenvolvida entre 1980 e 1990, não tem o intuito de substituir os paradigmas existentes, mas trabalhar conjuntamente para a resolução de problemas que tanto a programação orientada a objetos quanto programação procedimental não resolvem sozinhas eficientemente. Programação Orientada a Objetos (POO) apresenta muitos avanços que permitiram novas formas de representar a

Tabela 2.1: Resumo do Estilo *Pipe-and-Filter*

	Estilo Pipe-and-Filter
Elementos	*Filter* - componente que transforma dados lidos na sua porta de entrada para dados escritos em sua porta de saída. Geralmente, executam concorrentemente e incrementalmente. Propriedades podem especificar taxas de processamento, formato de entrada e saída de dados, e a transformação executada pelo *filter*. *Pipe* - conector que transmite dados da porta de saída de um *filter* para outra porta de entrada do *filter*. Um *pipe* tem um único papel de entrada de dados e saída de dados, preserva a sequência dos dados e não altera a passagem de dados. Propriedades podem especificar o tamanho do *buffer*, protocolo de interação e formato dos dados que passam através do *pipe*.
Relações	A relação associa portas de saída do *filter* com papéis de entrada de dados de um *pipe*, e portas de entrada de um *filter* com papéis de saída de dados de um *pipe*.
Modelo Computacional	O dado é transformado de uma entrada externa do sistema para uma sua saída externa através de uma série de transformações desempenhadas por seus *filters*.
Restrições	- *Pipes* conectam portas de saída de um *filter* a portas de saída de um outro *filter*. - *Filters* conectados devem estar de acordo com o tipo de dado passado ao longo dos *pipes*. - Especializações do estilo podem restringir a associação de componentes para um grafo acíclico ou uma sequência linear, às vezes chamado *pipeline*.
Utilidade	- Aperfeiçoar reúso devido à independência de filtros. - Aperfeiçoar a taxa de transferência com paralelização de processamento de dados. - Simplificar o raciocínio sobre o comportamento global.

Fonte: Adaptada de Garlan et al. (2010).

realidade em abstrações que tornaram possível produzir sistemas mais flexíveis, com baixo acoplamento, alta granularidade, entre outras características importantes. Contudo, não constitui a solução pra todos os problemas. Há muitos problemas de programação onde técnicas de POO não são suficientes para capturar claramente todas as decisões de projeto que o programa deve implementar (KICZALES et al., 1997).

Processos de projeto detalhado software e linguagens de programação apresentam uma relação de mapeamento, isto é, devem utilizar unidades de abstração similares. Para isso,

Tabela 2.2: Resumo do Estilo Cliente-Servidor

	Estilo Cliente-Servidor
Elementos	*Cliente* - componente que invoca serviços de um componente servidor. *Servidor* - componente que fornece serviços para os componentes clientes. As propriedades geralmente incluem informação sobre a natureza das portas do servidor (tais como quantos clientes irão se conectar) e características de desempenho (tais como taxas máximas de invocação de serviço). *Conector Request /Reply* - usado para um cliente invocar serviços em um servidor. Ele tem dois papéis: um papel de requisição e um papel de resposta. Propriedades do conector podem incluir se as chamadas são locais ou remotas, e se o dado é criptografado.
Relações	A relação associa portas de requisição-serviço do cliente com o papel de requisição do conector e portas de resposta-serviço do servidor com o papel de resposta do conector.
Modelo Computacional	Clientes iniciam interações invocando serviços dos servidores e esperam o resultado dessas requisições.
Restrições	- Clientes são conectados aos servidores através de conectores *request/reply*. - Componentes servidor podem ser clientes de outros servidores. - Componentes podem ser organizados em camadas.
Utilidade	- Promover modificabilidade e reuso. - Aperfeiçoar escalonamento e disponibilidade em caso de replicação do servidor. - Analisar dependabilidade, segurança e taxa de transferência.

Fonte: Adaptada de Garlan et al. (2010).

os processos de projeto detalhado de software dividem o sistema em unidades menores e as linguagens de programação oferecem mecanismos que permitem ao programador definir abstrações de subunidades do sistema e compor essas abstrações de diferentes modos para produzir um sistema completo. Um processo de projeto e uma linguagem de programação trabalham bem juntas quando uma linguagem oferece mecanismos de abstração e composição que dão suporte aos tipos de unidades que processo de projeto divide dentro do sistema (KICZALES et al., 1997).

Há muitas linguagens de programação, incluindo linguagens orientadas a objetos, linguagens procedurais e linguagens funcionais que podem ser compreendidas como tendo uma raiz comum e que seus mecanismos de abstração e composição chaves são enraizados em um mesmo procedimento generalizado. Vale ressaltar que isso não reduz os muitos

Tabela 2.3: Resumo do Estilo *Peer-to-Peer*

	Estilo Peer-to-Peer
Elementos	*Peer* - componente que troca serviços com outro *peer*. *Conector Call-return* - usado para conectar a rede de *peers*, buscar outros *peers* e invocar serviços de outros *peers*.
Relações	A relação associa associa *peers* com conectores *call-return*.
Modelo Computacional	A computação é efetivada pela cooperação entre os *peers* que requisitam serviços uns dos outros.
Restrições	- Restrições podem ser colocadas no número de ligações permitidas para qualquer porta ou papel. - Componentes *peer* podem fornecer roteamento, indexação e capacidade de buscar *peers*. - Especializações podem impor restrições de visibilidade nas quais os componentes podem saber sobre outros componentes.
Utilidade	- Prover disponibilidade. - Prover escalabilidade. - Habilitar sistemas distribuídos, tais como compartilhamento de arquivos, mensagens instantâneas e *desktop grid computing*.

Fonte: Adaptada de Garlan et al. (2010).

avanços de POO, mas ressalta-se o que há de comum entre as linguagens. Os métodos de projeto que envolvem linguagens de procedimento generalizado tendem a dividir os sistemas em unidades menores de comportamento ou função. Esse estilo é chamado decomposição funcional. A natureza da decomposição difere entre os paradigmas de linguagem, mas cada unidade é encapsulada em um procedimento/função/objeto. Cada unidade encapsulada é uma unidade funcional de um sistema maior (KICZALES et al., 1997).

Para resolver problemas relativos à complexidade de um software ou sistema uma das soluções é a modularização. Ao quebrar um problema em partes menores torna-se possível manuseá-lo mais facilmente. Quando se lida com requisitos de software complexos, pode-se quebrá-los em partes como funcionalidade do negócio, acesso a dados e lógica de apresentação. Essas funcionalidades são chamadas de *interesses* do sistema. Segurança, *logging*, monitoração de desempenho, entre outros, são interesses transversais a outros módulos de um sistema. Por isso, pode-se denominá-los de *interesses transversais* (LADDAD, 2010).

A POA pode fornecer boas soluções para esses **interesses**, onde podem ser implementadas classes específicas para segurança, *logging*, acesso a dados. Contudo, seria mais

interessante se houvesse um módulo que identificasse Segurança, Auditoria ou Monitoração de Desempenho. Para esse propósito surgiu a POA. Esse tipo de programação oferece uma abordagem que permite a separação de interesses transversais introduzindo unidades de modularização chamadas de **aspectos**.

No Capítulo 3, serão apresentadas trabalhos relacionados que utilizam a POA, através de AspectJ, uma linguagem de programação, para construir softwares que realizam a monitoração de requisitos de qualidade.

Tabela 2.4: Resumo do Estilo de Arquitetura Orientada a Serviços

	Estilo de Arquitetura Orientada a Serviços
Elementos	*Provedores de Serviços* - fornecem um ou mais serviços através de interfaces publicadas. Propriedades variam com a tecnologia de implementação (tais como EBJ ou ASP.NET), mas podem incluir desempenho, restrições de autorização, disponibilidade e custo. Em alguns casos essas propriedades são especificadas em *service level agreement* (SLA). *Consumidores de Serviços* - invocam serviços diretamente ou através de um intermediário. *ESB* - é um elemento intermediário que pode rotear ou transformar mensagens entre provedores de serviços e consumidores. *Registro de Serviços* - pode ser usado por provedores para registrar seus serviços e pelos consumidores para consultar e descobrir serviços em tempo de execução. *Servidor de Orquestração* - coordena as interações entre consumidores de serviços e provedores baseado em *scripts* que definem *business workflows*. *Conector SOAP* - usa o protocolo SOAP para comunicação síncrona entre *Web Services*, tipicamente sobre HTPP. Portas de componentes que usam SOAP são frequentemente descritas em WSDL. *Conector REST* - depende de operações *request/reply* básicas do protocolo HTTP. *Conector de mensagens* - usa um sistema de mensagens para oferecer trocas de mensagens assíncronas *publish-subscribe* ou ponto-a-ponto.
Relações	Ligação de diferentes tipos de portas disponíveis para os respectivos conectores.
Modelo Computacional	A computação é efetuada por um conjunto de componentes cooperando que fornecem e/ou consomem serviços sobre a rede. A computação é frequentemente descrita como um tipo de modelo de *workflow*.
Restrições	- Consumidores de serviços são conectados aos provedores de serviços, mas componentes intermediários (tais como ESB, registro ou servidor BPEL) podem ser usados. - ESBs conduzem uma topologia *hub-and-spoke*. - Provedores de serviço podem ser consumidores de serviços. - Padrões SOA específicos impõem restrições adicionais.
Utilidade	- Permitir interoperabilidade de componentes distribuídos executando em diferentes plataformas ou através da Internet. - Integrar sistemas legados. - Permitir reconfiguração dinâmica.

Fonte: Adaptada de Garlan et al. (2010).

Tabela 2.5: Resumo do Estilo Baseado em Eventos

	Estilo Baseado em Eventos
Elementos	Qualquer componente C&C com ao menos uma porta *publish* ou *subscribe*. As propriedades variam, mas elas devem incluir quais eventos são anunciados e assinados, e as condições em que o anunciador é bloqueado. *Conector publish-subscribe* - terá papéis para anunciar e escutar componentes que desejam publicar e/ou inscrever-se em eventos.
Relações	A relação associa componentes com conector *publish-subscribe* prescrevendo quais componentes anunciam eventos e quais componentes se registram para receber eventos.
Modelo Computacional	Componentes se inscrevem em eventos. Quando um evento é anunciado por um componente, o conector envia o evento para todos os inscritos.
Restrições	- Todos os componentes são conectados a um distribuidor de evento que pode ser visualizado tanto como um veículo (que é o conector) ou como um componente. Portas *publish* são anexadas a papéis de anúncio, e portas *subscribe* são ligadas a papéis de ouvintes. Restrições podem limitar quais componentes podem escutar quais eventos, se um componente pode escutar seus próprios eventos, e quantos conectores *publish-subscribe* podem existir dentro do sistema. - Um componente pode ser tanto um *publish* quanto um *subscribe* e por isso ter portas de ambos os tipos.
Utilidade	- Enviar eventos para destinatários desconhecidos, separando os produtores de eventos dos consumidores de eventos. - Fornecer funcionalidades básicas para *frameworks* GUI, listas de discussão (por exemplo, listas de nomes de pessoas interessadas em receber determinado conteúdo), quadros de avisos e redes sociais.

Fonte: Adaptada de Garlan et al. (2010).

Tabela 2.6: Resumo do Estilo de Dados Compartilhados

	Estilo de Dados Compartilhados
Elementos	*Componente repositório* - propriedades incluem tipos de dados armazenados, propriedades de dados orientados a desempenho, distribuição de dados, número de acessadores permitidos. *Componente de acesso a dados.* *Conector de escrita e leitura de dados* - Uma importante propriedade é se o conector é transacional ou não.
Relações	A relação determina quais acessadores de dados estão conectados a quais repositórios de dados.
Modelo Computacional	A comunicação entre acessadores de dados é mediada por um armazenamento de dados compartilhados. O controle pode ser iniciado por um dos acessadores de dados ou pelo armazenamento de dados. Os dados são persistidos pelo armazenamento de dados.
Restrições	- Acessadores de dados interagem com o(s) armazenador(es) de dados.
Utilidade	- Permite que múltiplos componentes acessem dados persistidos. - Fornece modificabilidade aperfeiçoada pelo desacoplamento dos produtores de dados dos que são os consumidores de dados.

Fonte: Adaptada de Garlan et al. (2010).

Capítulo 3

Trabalhos Relacionados

Foi possível perceber através da revisão sistemática da literatura que existe uma escassez quanto a presença de estudos que realizem propriamente a monitoração de requisitos de qualidade em sistemas após a construção desses. Foram estabelecidas duas categorias quanto aos estudos levantados que estão relacionados ao tema dessa dissertação. As categorias foram: *Estilos arquiteturais e requisitos de qualidade* e *Monitoração de requisitos de qualidade*.

Na primeira categoria estão estudos que relacionam a escolha dos estilos arquiteturais com a obtenção efetiva de requisitos de qualidade, de modo que há uma reflexão sobre quais estilos podem ser mais adequados quando se pretende alcançar um determinado requisito de qualidade. Na segunda categoria estão estudos que apresentam *frameworks* que têm o intuito de prover a monitoração de requisitos de qualidade em *softwares*.

3.1 Estilos Arquiteturais e Requisitos de Qualidade

Estudos seminais como os realizados em Chung, Nixon e Yu (1994a) e Chung, Nixon e Yu (1994b) mostram abordagens para definir requisitos de qualidade ainda nas fases iniciais de construção do sistema. Chung, Nixon e Yu (1994a) em seu trabalho apresentam uma abordagem orientada a processo onde os requisitos de qualidade são explicitamente representados como objetivos a serem endereçados e alcançados durante o processo de projeto arquitetural. A premissa fundamental dessa abordagem é que os requisitos de qualidade têm a propriedade de potencialmente interagir com os demais, em conflito ou sinergia. Essa propriedade é usada para sistematicamente guiar a seleção entre alternativas de projeto arquitetural e relacioná-las com o processo de projeto arquitetural geral.

Xavier, Werner e Travassos (2002), Alebrahim, Hatebur e Heisel (2011), Soares et al. (2012), Dwivedi e Rath (2014), Silva et al. (2015) e Me, Calero e Lago (2016) apresentam abordagens que relacionam estilos e padrões arquiteturais com requisitos de qualidade para através da análise de qual padrão arquitetural pode alcançar os requisitos de quali-

dade desejados. Trabalhos com esse objetivo estão alinhados com o intuito do presente processo de monitoração, pois consideram a relação entre os requisitos de qualidade e os estilos arquiteturais que estruturam o sistema. Essa relação é fundamental para que o sistema apresente qualidade e coerência.

Xavier, Werner e Travassos (2002) apresentam uma abordagem para apoiar o processo de seleção de padrões arquiteturais no contexto do projeto arquitetural de um software. Essa abordagem baseia-se na relação entre os padrões e um conjunto de características de qualidade de natureza não-funcional.

Em Alebrahim, Hatebur e Heisel (2011) é apresentado um método baseado em padrão e modelo que permite aos desenvolvedores de software considerarem requisitos de qualidade no início do processo de desenvolvimento do software. Requisitos de qualidade são incorporados em uma arquitetura de software inicial utilizando mecanismos ou padrões de desempenho e segurança e através de uma solução proposta é possível os desenvolvedores checarem condições de integridade semântica em diferentes modelos.

Soares et al. (2012) realizam um estudo para identificar subrequisitos de qualidade mais específicos descritos direta ou indiretamente entre requisitos de software de diferentes domínios. Esses requisitos estão relacionados com o atendimento dos atributos de qualidade nos sistemas desenvolvidos. Esses subrequisitos são denominados de Parâmetros de Atributos de Qualidade (PAQs). Na seleção de padrões, a identificação dos parâmetros contribui para buscar padrões que atendam mais especificamente esses parâmetros e aos atributos de qualidade relacionados. No estudo de Dwivedi e Rath (2014) o intuito foi classificar diferentes estilos arquiteturais com base em atributos de qualidade tais como eficiência, complexidade, escalabilidade, heterogeneidade, adaptabilidade, portabilidade, confiabilidade e segurança. Os mesmos construíram uma tabela onde foi expresso o quanto cada estilo arquitetural pode proporcionar determinado atributo de qualidade. Essa classificação é útil para compreender como cada estilo se relaciona com determinado atributo de qualidade. Por meio dessa análise sobre os diferentes estilos, eles escolheram o estilo que apresentava melhor relação com o que era desejado para desenvolver uma aplicação para roteamento de cargas. No caso, foi selecionado o estilo C2 (componente e conector) por ser usado em sistemas distribuídos, heterogêneos e reativos.

Na Tabela 3.1 de Dwivedi e Rath (2014) o símbolo "++" representa que um estilo trabalha bem com algum atributo de qualidade. O símbolo "+" significa que o estilo provê algum suporte para um atributo de qualidade em particular. O símbolo "0" indica que o estilo não afeta o atributo de qualidade. Por fim, o símbolo "-" mostra que o estilo tem um impacto negativo para o atributo de qualidade. Os estilos apresentados nas colunas são respectivamente, *Batch-Sequencial* (BS), *Pipe-and-Filter* (PF), *Virtual Machine* (VM), *Client-Server* (CS), *Publish-Subscriber* (PS), *Event-Based* (EB), *Peer-to-Peer*, *Component and connector* (C2) e *Common Object Request Broker Architecture*(CORBA).

Muitos estilos arquiteturais, como cliente-servidor, máquina virtual, *pipe-and-filter*,

Tabela 3.1: Categorização e avaliação de estilos arquiteturais.

RQs/Estilos	BS	PF	VM	CS	PS	EB	PP	C2	CORBA
Eficiência	0	0	-	-	+	+	0	0	-
Complexidade	0	0	0	0	+	+	++	+	++
Escalabilidade	0	+	+	+	0	+	+	+	0
Heterogeneidade	-	-	+	-	+	+	0	++	++
Adaptabilidade	0	-	0	0	+	0	0	+	++
Portabilidade	0	0	++	0	++	+	0	++	+
Confiabilidade	0	0	0	-	-	-	++	+	0
Segurança	0	0	0	++	+	0	+	0	-

Fonte: Adaptada de Dwivedi e Rath (2014).

baseado em eventos, baseado em regras, *blackboard, publish-subscriber, peer-to-peer* são aplicados para desenvolver diferentes sistemas. Diferentes estilos arquiteturais são utilizados para propósitos distintos (DWIVEDI; RATH, 2014).

Me, Calero e Lago (2016) apresentam uma pesquisa sobre a relação entre padrões ou estilos arquiteturais e atributos de qualidade que são encontrados na literatura. Uma das bases para o estudo foi a ISO/IEC 9126 como modelo de qualidade, assim como nesta dissertação, onde uma das bases é a referida ISO. Um dos resultados das análises realizadas foi a seguinte Tabela 3.2. Os padrões arquiteturais identificados estão de acordo com o trabalho de Buschmann [1] e os padrões podem ser caracterizados de acordo com o impacto negativo ou positivo sobre os atributos de qualidade. O símbolo "+" significa o quão forte está relacionado com o padrão com o atributo. O símbolo "-" significa que há menor relação entre o padrão e o atributo. O símbolo "=" significa neutralidade. Todos os atributos na Tabela estão presentes na ISO 9126, menos Implementabilidade. São apresentados, respectivamente, os atributos Usabilidade, Segurança, Manutenibilidade, Eficiência, Confiabilidade, Portabilidade e Implementabilidade.

Silva et al. (2015) apresentam um processo de recomendação integrado com uma ferramenta que baseada nos requisitos de qualidade escolhidos pelo(a) usuário(a) ajuda arquitetos(as) de software na fase de projeto arquitetural. A ferramenta utiliza mecanismos de Inteligência Artificial através da abordagem de Sistemas Especialistas para recomendar estilos arquiteturais de acordo com os requisitos especificados previamente. Essa solução também pode ser considerada um repositório de conhecimento onde decisões de projeto sobre estilos arquiteturais em um domínio específico podem ser armazenadas. Ela permite a resolução de *trade-off* sobre requisitos de qualidade e oferece suporte aos(às) arquitetos(as) pela recomendação de estilo arquitetural adequado, baseado nos requisitos do sistema, particularmente os requisitos de qualidade do sistema. A solução de *trade-off*

[1] F. Buschmann et al. *Pattern-Oriented software architecture: A system of patterns.* Inglaterra: Wiley, West Sussex, 1996.

Tabela 3.2: Padrões Arquiteturais e Atributos de Qualidade.

Padrões Arquit./AQs	Usab.	Secur.	Manut.	Efic.	Confiab.	Portab.	Implem.
Camadas	=	++	++	-	+	+	-
Pipes and Filters	-	-	+	+	–	++	-
Blackboard	=	–	++	-	=	=	–
MVC	++	=	-	-	=	-	-
Apresentação	+	=	+	–	=	+	–
Microkernel	=	=	++	–	+	++	–
Reflecção	=	=	++	-	–	+	-
Broker	+	+	+	=	=	++	+

Fonte: Adaptada de Me, Calero e Lago (2016).

é um aspecto de grande importância no desenvolvimento de software, embora às vezes não receba a devida atenção.

Em linhas gerais, em Silva et al. (2015) para recomendar um estilo arquitetural, os requisitos de qualidade fornecidos à ferramenta são classificados de acordo com a prioridade que o(a) desenvolvedor(a) atribuiu a eles, então a relevância de cada estilo arquitetural é calculada com base em casos de uso de sucesso ou fracasso contidos na base de conhecimento. Esse mecanismo que recomendação também é utilizado para aprimorar a própria base de conhecimento. Os requisitos de qualidade analisados pela ferramenta foram levantados com base na ISO 9126-1: eficiência, manutenibilidade, usabilidade, confiabilidade, testabilidade, interoperabilidade, segurança e portabilidade.

Com base na Tabela 3.3 adaptada no trabalho de Silva et al. (2015) pode-se observar como ocorre o *trafe-off* entre os requisitos de qualidade já mencionados, mostrando a relação pontencial entre eles. Os dois requisitos que estão em foco para esta dissertação são a eficiência e a confiabilidade. Nessa tabela os requisitos são relacionados entre si e a relação potencial possui as seguintes classificações: conflituosa (-), neutra (0) ou cooperativa (+).

As tabelas de Dwivedi e Rath (2014), Me, Calero e Lago (2016) e Silva et al. (2015) são interessantes para a presente pesquisa, pois permitem visualizar a relação entre os requisitos de qualidade que são alvo do processo de monitoração desenvolvido nessa dissertação. Ainda, permite analisar quais estilos arquiteturais são mais interessantes para os requisitos de qualidade elencados.

Nota-se que os trabalhos até então discutidos apresentam propostas semelhantes e são relevantes para essa dissertação para pensar as relações entre requisitos de qualidade e os estilos arquiteturais. Além disso, essas abordagens estão centradas em etapas iniciais do desenvolvimento no momento em que o software ainda não está sendo implementado. No processo de monitoração proposto nesse trabalho o intuito é dar continuidade ao processo monitoração quanto o software já está em funcionamento.

Tabela 3.3: Relação potencial entre atributos de qualidade.

RQs	Efic.	Manut.	Usab.	Conf.	Test.	Interop.	Seg.	Port.
Eficiência	+	-	+	-	-	-	-	-
Manutenibilidade	-	+	0	-	0	-	0	+
Usabilidade	+	0	+	0	-	0	-	+
Confiabilidade	-	-	0	+	+	0	+	0
Testabilidade	-	0	-	+	+	+	+	+
Interoperabilidade	-	-	0	0	+	+	0	0
Segurança	-	0	-	+	+	0	0	0
Portabilidade	-	+	+	0	+	0	0	+

Fonte: Adaptada de Silva et al. (2015).

3.2 Monitoração de Requistos de Qualidade

Nesta seção são apresentados os trabalhos selecionados com foco na monitoração de software. Nas revisão de literatura foram encontrados muitos estudos que tratam de monitoração voltados para QoS, contudo esse não era o objetivo de pesquisa nessa dissertação. A seguir serão discutidos alguns trabalhos que apresentam *frameworks* ou abordagens de monitoração e que foram importantes para pensar sobre as atividades que seriam necessárias para desenvolver um processo de monitoração com foco em informações arquiteturais do software.

Os estudos de Sundmark, Moller e Nolin (2004), Wang et al. (2009), Jayathilake (2012), Dharam e Shiva (2012) e Thanhofer-Pilisch et al. (2016) trazem *frameworks* ou abordagens de monitoração, embora o foco não seja em requisitos de qualidade e também não se baseiam na ISO 9126. Porém, é importante analisar trabalhos que também se propõe a realizar a monitoração de software.

Sundmark, Moller e Nolin (2004) propõem a monitoração de componentes de software para engenharia de sistemas de computador embarcados. A abordagem apresentada considera a perspectiva do ciclo de vida no processo de engenharia e identifica áreas chave de engenharia de sistemas baseado em componentes onde benefícios podem ser obtidos através da monitoração. Embora, não seja proposto um *framework* que realize a monitoração, o trabalho é relevante pelo foco na monitoração de componentes de software, que é um dos pontos estabelecidos no processo proposto nesta dissertação.

Em Wang et al. (2009) é introduzida uma abordagem de especificação de restrição baseada em padrão para monitoração de requisitos de *Web Services*, um modelo de monitoração e um *framework* de monitoração baseado a abordagem e modelo propostos. O *framework*, que foi um protótipo preliminar, foi baseado em Java e monitorou serviços *Web* desenvolvidos em plataforma baseada em Java. Além disso, o protótipo utilizou AspectJ para determinar pontos de monitoração.

No estudo desenvolvido por Jayathilake (2012) foi criado um *framework* para automatizar a monitoração de aspectos de qualidade do software, onde foram utilizadas várias ferramentas para essa finalidade. Esse *framework* fornece uma linguagem de *script* para automatizar análise de procedimentos, uma linguagem declarativa para interpretar a saída das ferramentas de monitoração e uma API em C++ para implementar o código de controle. Esse *framework* difere do que é utilizado no experimento realizado nesta dissertação, pois este não resulta de uma adaptação de um conjunto de ferramentas de monitoração.

Em Dharam e Shiva (2012) é proposto um *framework* que utiliza informações obtidas pelo desempenho de duas técnicas de teste de software em pré-desenvolvimento: Teste de fluxo de dados e Teste de caminho básico. Elas são úteis para desenvolver monitores em tempo de execução para acompanhar a monitoração de software em tempo de execução pós-desenvolvimento. O Teste de fluxo de dados é uma abordagem que detecta uso inapropriado de dados que pode ser desempenhada estaticamente ou dinamicamente. O Teste de caminho básico é um técnica caixa branca que identifica um conjunto mínimo de todos os caminhos legais de execução do programa. O objetivo do *framework* é identificar um conjunto de caminhos críticos a serem monitorados no software durante sua execução para a detecção e prevenção de ataques de caminho transversal. Esse *framework* tem foco em aplicações Java e é integrado a uma ferramenta denominada MOP (*Monitoring Oriented Programming*) para integrar monitores no código fonte. Esses monitores são localizados nos caminhos considerados críticos para o monitoramento pós-desenvolvimento. Nesta dissertação o experimento para o processo de monitoração proposto utiliza um *framework* pode fazer o mapeamento do código fonte e identifica aspectos estruturais do mesmo, e em comum com a abordagem do estudo de Dharam e Shiva (2012) determina pontos de monitoração no código fonte do software, sendo também voltado para aplicações Java.

Thanhofer-Pilisch et al. (2016) propõem uma abordagem baseada em um *framework* desenvolvido em um trabalho anterior denominado REMINDS para fornecer suporte para capturar e comparar vestígios de eventos em SoS. Esse *framework* foi aperfeiçoado e denominado REMINDS/CC o qual endereça evolução-consciente com uma abordagem de gerenciamento de variabilidade. Ele faz comparação dos eventos e dados fornecidos pelo sistema, e possui mecanismo que permite a visualização dos resultados comparados. Esse trabalho é interessante para esta dissertação no referente ao conhecimento de diferentes técnicas de monitoração, bem como pensar sobre as atividades que um processo de monitoração necessita.

Nos trabalhos de Lima (2016), Silva (2015) e Franco (2014) são apresentadas soluções que visam a monitoração de requisitos de qualidade. O trabalho de Franco (2014) visa a monitoração de atributos de Qualidade de Serviço (QoS). Na literatura é possível perceber que existem mais estudos voltados para medir e monitorar atributos de QoS e uma escassez quanto a monitoração de requisitos de qualidade dentro do software. Além disso, o foco do estudo de Silva (2015) e Lima (2016) está em aspectos estruturais do software não

se estendendo a analisar atributos de QoS que é mais específico da área de Sistemas Distribuídos.

O trabalho de Lima (2016) estende o trabalho desenvolvido por Silva (2015), onde a ferramenta de Silva (2015) foi organizada como um *framework* e a monitoração pode ser realizada através de anotações de código em Java (*Java Annotation*). Os registros de monitoração são armazenados em arquivos de *log* separados por comportamento. Por meio do *framework* proposto por Lima (2016), denominado, *QuAM Framework*, o processo de monitoração proposto nessa dissertação será avaliado utilizando uma aplicação real.

No trabalho de Silva et al. (2015) é apresentada uma ferramenta que permite a resolução de *trade-off* sobre requisitos de qualidade e oferece suporte aos(às) arquitetos(as) pela recomendação de estilo arquitetural adequado, baseado nos requisitos do sistema, particularmente os requisitos de qualidade do sistema. A solução de *trade-off* é um aspecto de grande importância no desenvolvimento de software, embora às vezes não receba a devida atenção. Em nosso trabalho por meio do processo de monitoração também são considerados os *trade-offs* em relação aos requisitos de qualidade.

Silva (2015) aborda sobre a monitoração de atributos de qualidade referenciados pela norma ISO/IEC 9126-1. São apresentadas árvores de decisão que relacionam elementos arquiteturais a questões de monitoração. É apresentada uma ferramenta para o processo de monitoração dos requisitos de confiabilidade e eficiência, através da geração e análise de *logging* de auditoria e registro de exceções. A ferramenta proposta utiliza a linguagem AspectJ de POA para monitorar software densenvolvido na linguagem de programação Java.

Franco (2014) foca na monitoração de atributos de Qualidade de Serviço (do inglês, *Quality of Service* - QoS), os quais são usados para estabelecimento de Acordos de Nível de Serviço (do inglês, *Service Level Agreement* - SLA) mediante o uso desses atributos. A partir dessa demanda foi desenvolvida uma solução para realizar a monitoração de serviços *Web*. A solução proposta chama-se *FlexMonitorWS* e utiliza Desenvolvimento Baseado em Componentes. Ela também faz uso de POA utilizando Java e AspectJ.

Diferentemente de todos os trabalhos identificados nessa Seção, esta dissertação busca contribuir com a sistematização do processo de monitoração e interpretação dos dados de *log*, visando auxiliar os desenvolvedores em como utilizar os *frameworks* de monitoração e como sistematizar os passos da análise dos dados gerados.

Capítulo 4

ArMoni: Processo de Monitoração Baseado em Estilos Arquiteturais

Neste capítulo, é trabalhado um caso ilustrativo para visualizar como os estilos arquiteturais estão presentes na arquitetura de um sistema, evidenciando que a escolha de cada estilo pode favorecer determinado aspecto no funcionamento do mesmo. Além disso, é explanado o funcionamento do QuAM Framework, um sistema para monitoração de requisitos de qualidade e é apresentado o processo ArMoni, onde são destrinchadas as atividades que compõem o mesmo.

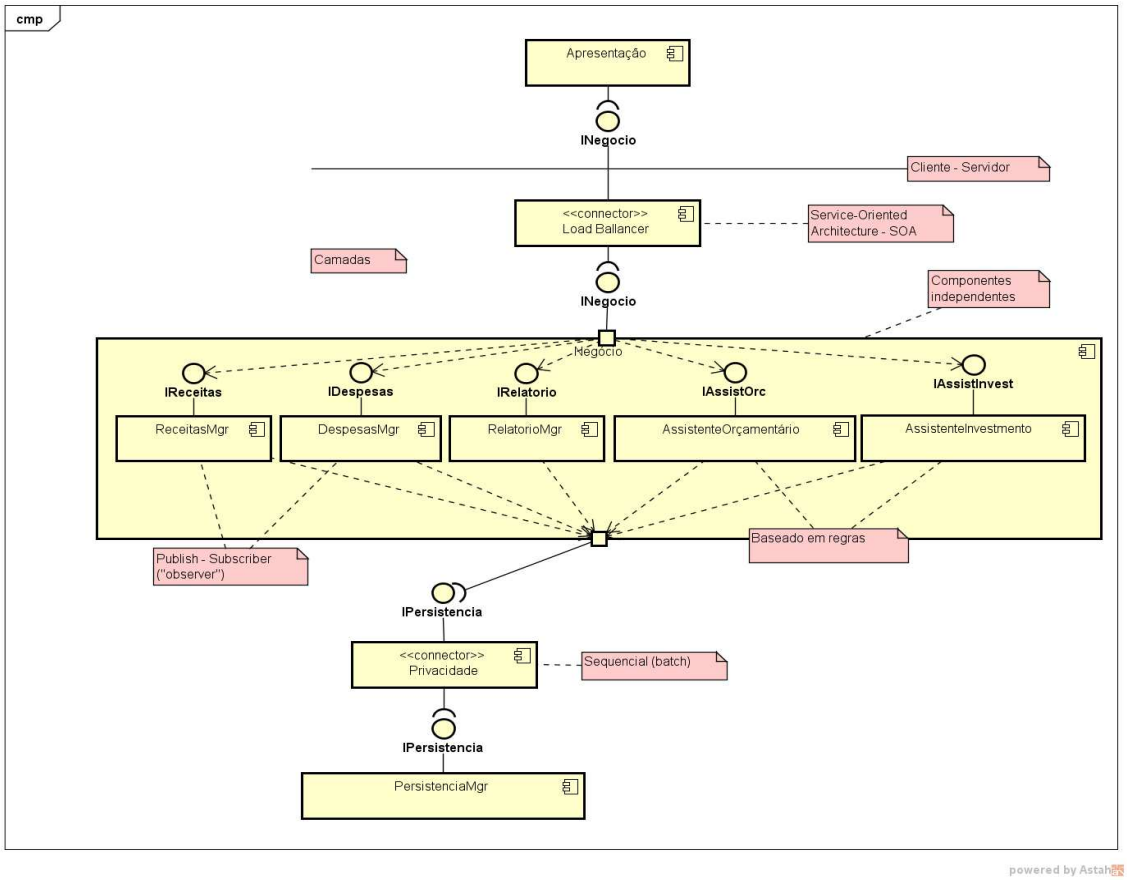
4.1 Caso Ilustrativo

Com intuito de ilustrar os estilos arquiteturais dentro de uma aplicação, será analisada uma estrutura de um sistema financeiro. Por meio desse exemplo pode-se apreender alguns aspectos de como estão organizados os componentes e conectores presentes no modelo arquitetural da Figura 4.1 que descreve o referido sistema. O sistema possui uma arquitetura de software heterogênea, que combina de forma simples vários estilos arquiteturais.

Inicialmente pode-se observar que a arquitetura é baseada no **estilo em camadas** onde há a *camada de apresentação, a camada de aplicação* e a *camada de base de dados*. A camada de apresentação se refere ao componente que permite a interação entre o usuário e o sistema através da Internet por meio do computador ou celular, ou seja, é um sistema responsivo, à medida que esse se adapta ao meio em que sua interface principal é acessada. A camada de aplicação apresenta a lógica de funcionamento do sistema e a camada de base de dados apresenta os componentes responsáveis pela persistência de dados.

Outro estilo arquitetural que se pode observar é o **cliente-servidor**. O componente *Apresentação* possui uma interface para comunicar-se com a camada de aplicação decorrente da maneira como as camadas estão organizadas. Ela tem como mediador de

Figura 4.1: Arquitetura de um Sistema Financeiro com diferentes estilos arquiteturais.



Fonte: Elaborada pela autora.

requisições um conector responsável pelo balanceamento de carga chamado *Load Balancer*. Dentro da camada de aplicação, pode-se observar um conjunto de componentes que recebem requisições da camada de apresentação. Há os componentes: *ReceitasMgr, DespesasMgr, RelatorioMgr, AssistenteOrçamentário* e *AssistenteInvestimento*. Nessa camada visualizamos o **estilo de componentes independentes** que permite baixo acoplamento e alta coesão entre eles, possibilitando o reúso de componentes e facilitando a evolução do sistema. Consequentemente, deixa claro quais os papéis atribuídos a cada um dos componentes.

Ainda, pode-se trabalhar outros estilos arquiteturais com base na arquitetura em questão. Na camada de aplicação pode ser identificado o estilo **publish-subscriber** que estrutura a relação entre os componentes *ReceitasMgr* e *DespesasMgr*. As mudanças que ocorrem em algum dos componentes podem ser utilizadas como base para o funcionamento do outro componente, de modo que há essa interação entre os ganhos e gastos que o sistema identifica. Por isso, emprega-se também o padrão de projeto de *software observer*, a fim de realizar essa comunicação de atualização de dados.

Além disso, há o estilo **baseado em regras** onde os componentes *AssistenteOrçamentário* e *AssistenteInvestimento* interagem com o objetivo de viabilizar o direcionamento

de recursos financeiros. Por exemplo, se o sistema identifica que o sistema apresenta um orçamento favorável a investimentos, o *AssistenteInvestimento* proporciona "dicas" de investimentos que seriam favoráveis para o usuário.

Passando para a camada de base de dados, é identificado o estilo de **batch-sequential** que pode ser aplicado em transações em sistemas financeiros. Esse estilo é aplicado com a finalidade de proporcionar segurança ao processo de armazenamento dos dados financeiros. Através dele a pretensão é promover um requisito básico e indispensável a um sistema desse tipo que é a privacidade, e por consequência, a segurança. Percebe-se nesse aspecto, a preocupação com requisitos de qualidade, uma vez que apenas o funcionamento adequado dos componentes isoladamente não garante a qualidade do software.

Dependendo do tipo de requisito de qualidade que seja alvo da monitoração, deve-se pensar qual ou quais componentes podem prover o referido requisito, aliando essa informação com o estilo arquitetural em que os mesmos estão dispostos. Essa visão permite estabelecer com maior critério pontos que devem ser monitorados para garantir um dado requisito de qualidade.

4.2 QuAM Framework - Monitoração de Requisitos de Qualidade

Como ferramenta para avaliar e validar o processo de monitoração proposto foi utilizado *framework* desenvolvido por Lima (2016), baseado no trabalho de Silva (2015), denominado *QuAM Framework*. Essa solução monitora atributos de qualidade, possuindo implementação finalizada para os requisitos de confiabilidade e eficiência, e podendo ser estendido para outros requisitos.

Esse *framework* permite o mapeamento de elementos arquiteturais tais como, pacotes, classes, interfaces, métodos, construtores, atributos, exceções, parâmetros, modificadores, tipos de retorno e comportamentos implementados por aspectos (LIMA, 2016). A Figura 4.2 mostra um exemplo de uma classe do *QuAM Framework* mapeada pelo próprio *framework*.

Segundo Lima (2016), o monitoramento padrão registra os seguintes dados:

- *datetime* - Data e hora, acompanhados do fuso horário, indicando o momento de registro quando os dados do monitoramento foram registrados;
- *executable* - Nome do executável, em conjunto com o nome do pacote e nome da classe;
- *args* - Objeto com pares de argumentos e parâmetros, caso existam;
- cpuTime - Total de tempo em segundos pelo qual a Unidade Central de Processamento (CPU) foi usada para processar as instruções do elemento executável;

- wallTime - Total de tempo real em segundos decorrido desde o início até o término da execução do elemento;
- returnType - Tipo do retorno do elemento (caso seja um método).
- result - Retorno do elemento (caso seja um método e seu tipo de retorno não seja *void*).

A monitoração realizada pelo QuAM Framework é intrusiva, de modo que os monitores são dispostos diretamente nas áreas de código que se deseja monitorar. Conforme a monitoração for definida, a localização das anotações pode ter maior ou menor granularidade conforme o(a) desenvolvedor(a) desejar. Em outras palavras, pode-se monitorar interfaces dos componentes e conectores, ou aumentar o nível de especificidade localizando as anotações em métodos e funções. Vale ressaltar que o processo ArMoni não depende do referido *framework*, desse modo também pode ser utilizado para definir atividades de monitoração em cenários não intrusivos.

Figura 4.2: Mapeamento de uma classe do próprio *QuAM Framework*

```
{
  "pkg" : "com.quam.fw.element",
  "modifiers" : [ "final" ],
  "type" : "class",
  "name" : "Class",
  "interfaces" : [ ],
  "constructors" : [ ],
  "methods" : [ {
    "name" : "newInstance",
    "modifiers" : [ "static" ],
    "parameters" : [ {
      "type" : "java.lang.Class",
      "name" : "clazz",
      "isFinal" : false
    } ],
    "returnType" : "com.quam.fw.element.Class",
    "exceptions" : [ ]
  } ],
  "attributes" : [ {
    "name" : "superClass",
    "type" : "java.lang.String",
    "modifiers" : [ "private" ]
  } ],
  "innerClasses" : [ ],
  "innerInterfaces" : [ ],
  "innerEnums" : [ ],
  "superClass" : "com.quam.fw.element.Element"
}
```

Fonte: Lima (2016).

Para monitorar um elemento é utilizada a anotação @*Loggable*, como mostra a Figura 4.3. Com @*Loggable* todos os elementos executáveis declarados por ele são monitorados. Quando é desejado não monitorar algo específico, é utilizada a anotação @*NotLoggable*. Essa anotação só pode ser utilizado para métodos (LIMA, 2016).

Figura 4.3: Anotações @*Loggable* e @*NotLoggable*

```
1  package com.sample;
2
3  import com.quam.fw.annotations.Loggable;
4  import com.quam.fw.annotations.NotLoggable;
5
6  @Loggable
7  class SampleClass {
8    ...
9    SampleClass() {
10     ...
11   }
12
13   int sampleMethod(String foo) {
14     ...
15     return anyInteger;
16   }
17
18   @NotLoggable
19   void anotherSampleMethod() {
20     ...
21   }
22 }
```

Fonte: Lima (2016).

Os dados coletados são registrados no arquivos de *log*, ver Figura 4.4, de acordo com a classificação de comportamento atribuída pelo *framework*. Os dados das execuções normais, onde não há retorno de uma exceção, são armazenados no arquivo *normal.log*. As execuções com retorno excepcional ficam registradas no arquivo *abnormal.log* (LIMA, 2016). Os dados registrados para retornos excepcionais são, conforme (LIMA, 2016): .

- *name* - Exceção que foi lançada;
- file - Nome do arquivo onde a exceção foi lançada;
- line - Número da linha do arquivo onde o erro foi lançado;
- message - Caso existente, mensagem produzida pela exceção.

4.3 Confiabilidade

Para avaliar a confiabilidade de um software se faz necessário adotar métricas para mensurar a satisfação do requisito de confiablidade em um processo de monitoração. *Confiabilidade* refere-se à capacidade do produto de software de manter um determinado nível de funcionamento, quando usado em condições especificadas. Esse requisito está subdividido em: *maturidade, tolerância a falhas* e *recuperabilidade* (ISO/IEC, 2003).

Sobre a tolerância a falhas, espera-se que o programa tenha, dentro de certos limites, capacidade de continuar a funcionar quando algo inesperado ocorrer. Uma situação

Figura 4.4: Exemplo de *log* gerado pelo *QuAM Framework*

```
{
  "datetime" : "2015-07-05T19:10:54-0300",
  "executable" : "com.sample.SampleClass.sampleMethod",
  "args" : {
    "foo" : "sample string"
  },
  "cpuTime" : 0.062892658,
  "wallTime" : 0.067617342,
  "returnType" : "int",
  "result" : 100
}
```

Fonte: Lima (2016).

que ilustra essa característica é quando o espaço em disco necessário para uma operação torna-se menor que o especificado nos requisitos do software. Caso isso ocorra, a solução esperada seria que o programa contornasse o problema evitando falhas ou, pelo menos, encerrasse a execução controladamente. Sobre a recuperabilidade, o comportamento esperado é que o programa tenha capacidade de voltar a operar corretamente após a ocorrência de uma falha (KOSCIANSKI; SOARES, 2007).

A maturidade é uma característica difícil de medir e refere-se a um programa que é robusto. Um programa robusto deve ser capaz de evitar as falhas ocasionadas por defeitos. Para isso, é importante a realização de testes que abrangem condições de erro pouco prováveis e tratamento de exceções. Além disso, maturidade também é uma característica atribuída a produtos que foram testados por vários usuários e falhas já foram encontradas e corrigidas. Dessa maneira, a probabilidade que ocorram falhas torna-se menor.

Além disso, no que se refere ao comportamento de um software é importante entender alguns conceitos, tais como *falha*, *erro* e *defeito*. Esses, por vezes, são utilizados como sinônimos, contudo essa é uma ideia equivocada. *Falha* refere-se a uma imperfeição de um produto. Dessa forma, ele faz parte do produto e é algo que foi implementado incorretamente no código. Uma falha não significa apenas algo que faz o programa não funcionar, mas também pode ser considerado algo que faz o programa não funcionar adequadamente. Consequentemente, uma falha não necessariamente ocasiona a interrupção do programa. As falhas podem estar presentes, mas não serem perceptíveis ao usuário em determinados cenários de uso.

Porém, quando uma falha é ativada, isto é, executada, o sistema pode alcançar um estado errôneo (*erro*). Um erro, quando detectado, é normalmente sinalizado na forma de exceções e podem ser tratados através de blocos redundantes de código, que podem estar presentes nos blocos de tratamento de erro (*catch*) ou ainda em elementos arquiteturais redundantes. Quando um erro não é tratado adequadamente e os efeitos da falha se tornam perceptíveis ao usuário, afirma-se que foi identificado um *defeito*. Sendo assim,

um defeito pode ser visto como o resultado errado que foi ocasionado por uma falha. Além dos fatores internos, há fatores externos que podem ocasionar as falhas, como uma base de dados corrompida e invasões de memória por outros programas. Um exemplo comum de defeito é quando o programa trava (LEE; ANDERSON, 1990).

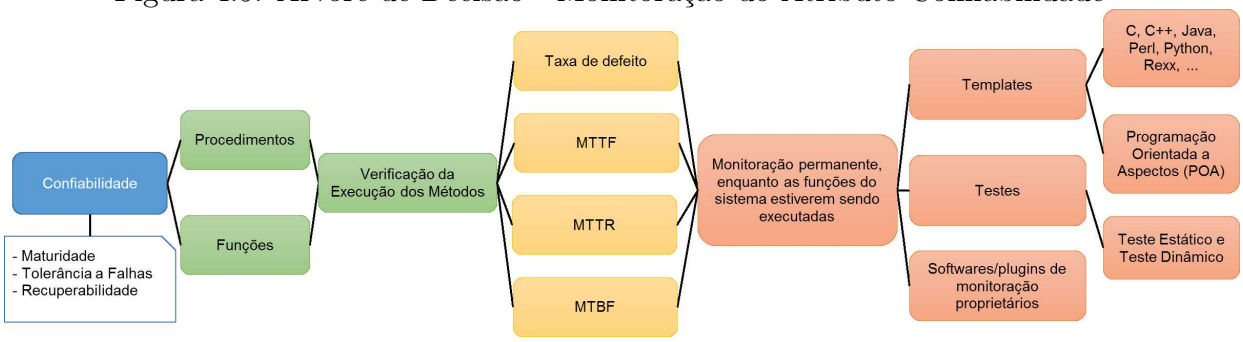
Os defeitos que podem ocorrer em um software são notadas quando esse está operando, ou seja, podem haver problemas na execução de procedimentos, funções ou componentes que o compõem, de forma pontual; ou até mesmo comprometer o funcionamento global desse software. Por essa razão, o processo de monitoração deve analisar os elementos que realizam determinada funcionalidade (SILVA, 2015).

Além disso, é preciso salientar que há situações em que é possível que os usuários utilizem o software e seja possível conviver com situações de defeito, visando a identificação e tratamento de erros (exceções). Sendo assim, do ponto de vista funcional, um retorno excepcional inesperado pode ser considerado como um defeito local, uma vez que não representa o retorno esperado da operação. Do ponto de vista arquitetural, pode ser visto como um erro que, se não for tratado pela arquitetura, será propagado até o usuário, onde enfim será considerado um defeito. Contudo há outros casos em que o defeito representa o fracasso do produto e ele é descartado. Atualmente, há uma diversidade de aplicações críticas em que um defeito pode representar riscos e/ou grandes prejuízos para os indivíduos que estão direta ou indiretamente utilizando um software ou sistema.

Nessa perspectiva, é fundamental mensurar a confiabilidade, pois através dela é possível estimar a probabilidade de ocorrência de falhas. Uma vez que um defeito torna-se visível e há probabilidade de recuperação do sistema após a identificão de erros, pode-se reportá-lo e corrigir a falha que a causou. Em determinados casos as falhas não são corrigidos ou sequer detectadas, por não conseguir reproduzir o defeito que a explicitou, não conseguir isolar a falha ou por não considerar como não sendo tão grave e/ou a correção dessa é dispendiosa (KOSCIANSKI; SOARES, 2007).

As métricas consideradas no trabalho de Silva (2015) são: taxa de defeito, tempo médio para falhar (*Mean Time to Failure* - MTF), tempo médio para correção (*Mean Time to Repair* - MTTR) e tempo médio entre ocorrências de falhas (*Mean Time Between Failures* - MTBF). A monitoração pode ser realizada através de *templates* implementados em linguagens de programação convencionais ou utilizando POA; testes estáticos e/ou dinâmicos; e através de *softwares/plugins* de monitoração (SILVA, 2015). Na Figura 4.5, Silva (2015) descreve uma árvore de decisão para mostrar o fluxo necessário para realizar uma monitoração do requisito de confiabilidade.

Figura 4.5: Árvore de Decisão - Monitoração do Atributo Confiabilidade



Fonte: Silva (2015).

4.4 Eficiência

Eficiência é a capacidade do produto de software de apresentar desempenho apropriado, relativo à quantidade de recursos usados, sob condições especificadas. Este requisito está subdividido em: comportamento temporal e utilização de recursos (ISO/IEC, 2003). *Medidas de comportamento temporal* podem ser facilmente definidas, contudo a principal dificuldade é definir o contexto de uso ao aplicá-las. Um exemplo bastante conhecido de medida de comportamento temporal é o número de transações por segundo. *Medições de uso de recurso* são referentes aos elementos físicos que um software usa durante sua execução. Alguns tipos de recursos são o espaço em disco para instalação e operação, volume de tráfego de rede e carga de CPU. Assim como o comportamento temporal, medir o uso de recursos também depende do contexto da aplicação (KOSCIANSKI; SOARES, 2007).

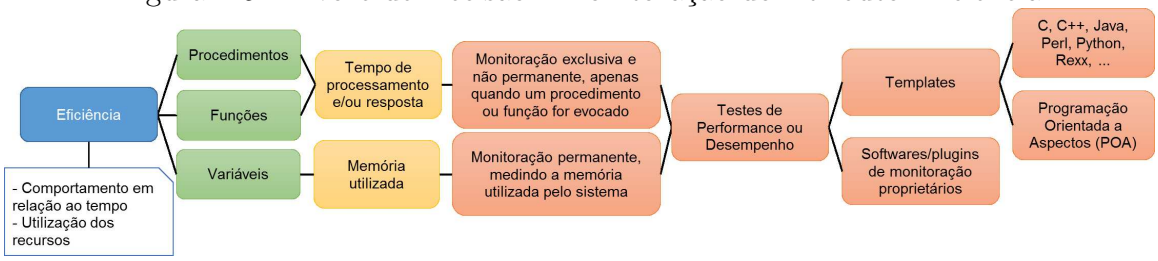
A monitoração da eficiência analisa o sistema focando no tempo e recursos que são consumidos pelo software durante a realização de alguma tarefa. O intuito da monitoração é identificar pontos em que há gargalos de processamento os quais podem prejudicar o desempenho do software. Dessa forma, a monitoração envolve a análise do tempo de resposta das requisições realizadas pelos procedimentos e funções, juntamente com o processamento e memória necessários para a realização de uma tarefa (SILVA, 2015). Assim como a confiabilidade, a eficiência pode ser mensurada por meio de *templates* implementados em linguagens de programação convencionais ou utilizando POA; testes estáticos e/ou dinâmicos; e através de *softwares/plugins* de monitoração.

A velocidade de operação de um software sofre interferência de diferentes fatores, tais como, velocidade da CPU, quantidade de memória cache e memória RAM, desempenho de disco rígido, volume de tráfego de rede, interação com outros softwares e com o sistema operacional (SO), configurações do SO, entre outros fatores. A utilização de recursos envolve todo recurso que não seja tempo de CPU, como quantidade de memória, carga de CPU, ocupação de disco, entre outros (KOSCIANSKI; SOARES, 2007).

São necessários procedimentos cuidadosos para realizar com eficácia a medida de um software e especificar condições sob as quais as medidas serão válidas. Embora sejam realizados vários testes para aferir determinado comportamento desejado, não é possível controlar todas as condições em que o software pode operar. Desse modo, pode-se estabelecer condições mínimas e máximas que são consideradas adequadas para caracterizar que o software é eficiente (KOSCIANSKI; SOARES, 2007).

Na Figura 4.6, Silva (2015) descreve uma árvore de decisão para mostrar o fluxo necessário para realizar uma monitoração do requisito de eficiência.

Figura 4.6: Árvore de Decisão - Monitoração do Atributo Eficiência



Fonte: Silva (2015).

4.5 Atividades do Processo ArMoni

O processo de monitoração proposto tem por objetivo estabelecer atividades que podem guiar o(a) desenvolvedor(a) de software tornando mais intuitiva a monitoração de requisitos de qualidade e para isso toma como base a arquitetura de software do sistema e as características dos estilos arquiteturais clássicos. A sequência lógica entre as atividades será apresentada no contexto de um diagrama de atividades UML que visa facilitar a visualização desse processo.

O *processo ArMoni* é independente do *QuAM Framework* e pode ser utilizado para guiar as atividades de monitoração em outros cenários onde ela seja intrusiva ou não. Em *frameworks* onde a monitoração não seja intrusiva pode-se considerar a escolha dos pontos de monitoração com base nas interfaces dos componentes que estão disponíveis. Desse modo, a monitoração não acessará trechos mais específicos do código, como ocorre com a monitoração realizada pelo *QuAM Framework*.

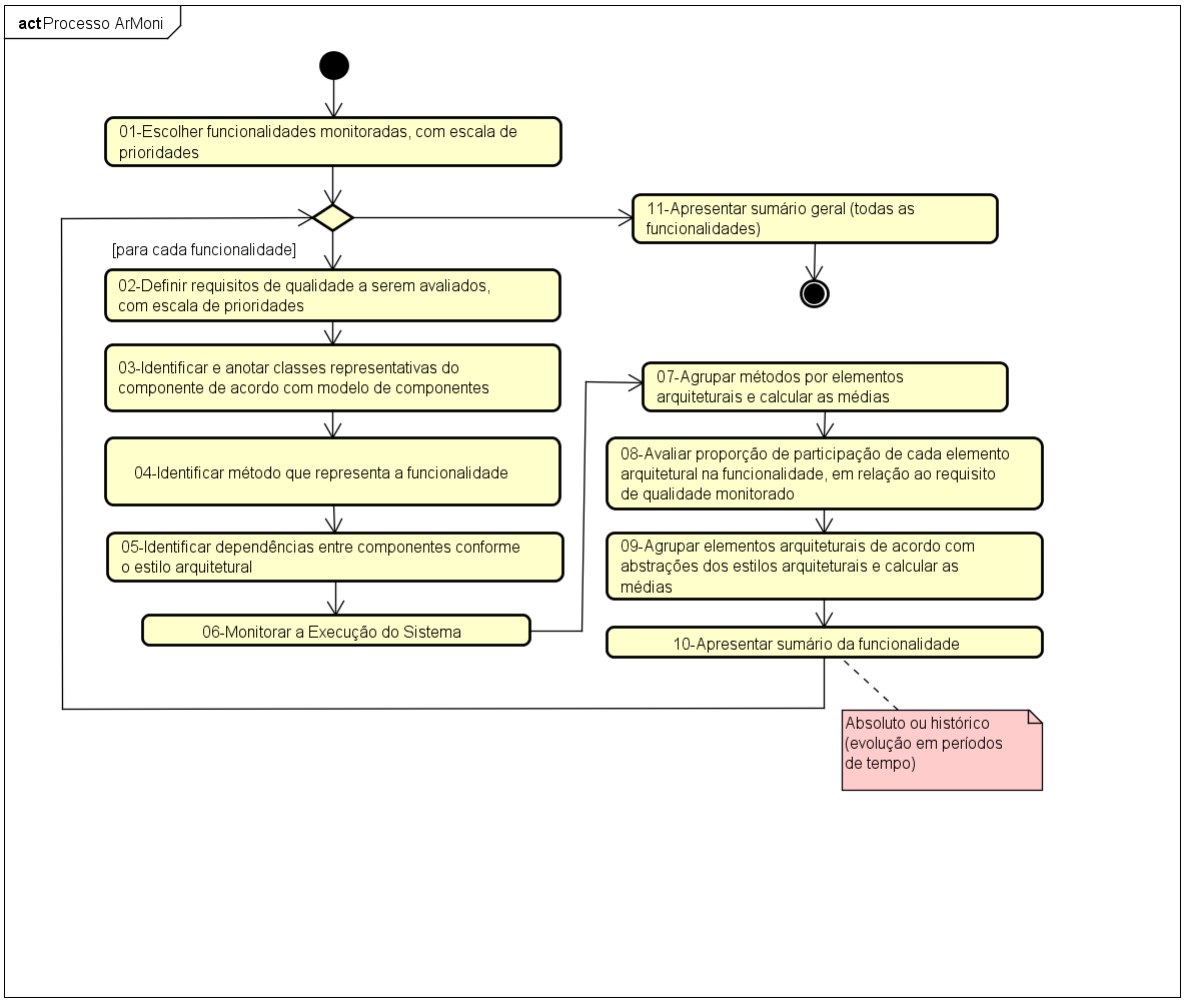
A fluxo de atividades do ArMoni apoia a monitoração de uma funcionalidade de cada vez e para ela escolhe-se o requisito de qualidade que será monitorado. Essa característica do ArMoni considera o fato da existência de *tradeoffs*, onde a monitoração de um determinado requisito pode interferir em outro requisito caso a monitoração ocorra simultaneamente. Esse aspecto é considera tanto na monitoração intrusiva ou não intrusiva.

Após o período de monitoração de uma funcionalidade verificando um dado requisito de qualidade, o objetivo é organizar um sumário dos dados obtidos. Esse processo pode ser

repetido com a mesma funcionalidade para monitorar outros requisitos ou passar para uma nova funcionalidade. Ao finalizar a monitoração de todas as funcionalidades e requisitos de qualidade escolhidos, o processo propõe um sumário de todas as monitorações realizadas no sistema.

Inicialmente, o(a) desenvolvedor(a) deve escolher as funcionalidades que deseja monitorar no sistema por ele(ela) escolhido. Com base nessa lista ele(ela) deve escolher uma das funcionalidades de cada vez para dar prosseguimento às atividades com indicação da relevância de cada uma delas (Atividade 1). O processo sugere que a indicação da relevância pode ser definida através de uma escala de pesos, que variam de 1 a 9. A Figura 4.7 mostra as atividades definidas pelo ArMoni (Processo de Monitoração Baseado em Estilos Arquiteturais).

Figura 4.7: Atividades do Processo ArMoni



Fonte: Elaborada pela autora.

Para cada uma das funcionalidades identificadas na Atividade 01, o(a) desenvolvedor(a) deve indicar quais requisitos ele(ela) deseja e refletir de modo mais amplo quais

são fundamentais para o sistema. Na Atividade 02, o processo sugere que a indicação da importância seja determinada através de uma escala de prioridades definida através de pesos, que variam de 1 a 9. Realizados esses primeiros passos, em que o(a) desenvolvedor(a) analisa as funcionalidades e requisitos de qualidade, ele(ela) deve identificar e anotar as classes representativas para identificar os componentes do sistema (Atividade 03).

O processo assume que há consistência entre arquitetura e código-fonte. A identificação da(s) classe(s) que encapsula(m) o componente vai depender dos padrões de projeto utilizados para implementar as classes dos pacotes do sistema ou do modelo de componentes utilizado, caso a implementação seja baseada em componentes como, por exemplo, o modelo COSMOS (JR.; GUERRA; RUBIRA, 2003; GAYARD; RUBIRA; GUERRA, 2008). Esse modelo define diretrizes para estruturar os pacotes e classes que implementam componentes de software. Nele as classes indicadas para a monitoração são as classes de fachada (padrão de projeto *Façade*) que implementam as interfaces providas dos componentes. Após a identificação das classes, é necessário identificar o método que representa (ou coordena) a execução da funcionalidade (Atividade 04).

Depois de determinar os métodos, é necessário compreender os aspectos arquiteturais do sistema em questão. Em outras palavras, identificar quais são as dependências entre os elementos arquiteturais (componentes e conectores). Embora nem todos os sistemas possuam uma arquitetura claramente definida, o(a) desenvolvedor(a) pode buscar através do código entender aspectos estruturais com o intuito de definir como está organizada a arquitetura do sistema (Atividade 05).

Como resultado dessa análise, ele pode identificar a configuração arquitetural do sistema, dessa forma conhecer quais são os componentes, conectores e estilo arquitetural ou estilos arquiteturais que o compõem. Ter esse conhecimento bem estabelecido auxilia na identificação das dependências entre os elementos arquiteturais. Por exemplo, em uma arquitetura que possui o estilo arquitetural cliente-servidor um dos passos seria identificar as chamadas de métodos de um componente para outro: o componente que faz requisição (cliente) para o componente que recebe a requisição (servidor).

Uma vez escolhidos os métodos ou componentes que serão monitorados, o(a) desenvolvedor(a) utilizará o *framework* para realizar a monitoração onde ele poderá utilizar as anotações Java para determinar locais no código em que essas anotações poderão coletar os dados (Atividade 06). Para facilitar a análise dos dados gerados na monitoração, o processo proposto sugere uma maior contextualização com a arquitetura de software. Para isso, os dados obtidos devem ser agrupados e apresentados por cada elemento arquitetural (Atividade 07). Após agrupar por elementos arquiteturais, o processo proposto sugere o cálculo da média dos valores registrados no *log*.

Além da avaliação isolada de cada elemento arquitetural, é recomendada a avaliação proporcional da participação de cada elemento arquitetural na execução da funcionalidade

em questão (Atividade 08). Outro agrupamento que pode oferecer informações relevantes para o arquiteto em eventuais refatorações da arquitetura é o agrupamento de acordo com abstrações do estilo (Atividade 09). Esse agrupamento consiste em reunir componentes de uma mesma abstração de estilo como, por exemplo, uma camada da arquitetura, e oferecer dados mais abstratos. No caso do MVC, analisar todos os componentes do modelo juntos, todos os do controle juntos e todos os da visão juntos, de modo a fazer a análise conforme a granularidade dos elementos arquiteturais.

Finalmente, uma síntese dos dados relevantes coletados deve ser apresentada, tanto no contexto de cada funcionalidade (Atividade 10), quanto no contexto de todas as funcionalidades (Atividade 11). Além da contextualização da informação de acordo com cada elemento arquitetural, o processo proposto ressalta a importância dessa síntese apresentar também variações das métricas em períodos de tempo. A justificativa para essa preocupação é o fato de padrões de variação poderem representar degradação de uso, isto é, quando um elemento arquitetural entra em decadência, com tendência de violação dos requisitos de qualidade. Para avaliar tendências de degradação, o(a) desenvolvedor(a) deve também calcular as métricas com os dados agrupados por intervalos de tempo.

Esse intervalo é relativo e deve ser definido preferencialmente pelo(a) desenvolvedor(a) de software. Por exemplo, em um sistema de alta demanda, um intervalo de um minuto de monitoração pode gerar dados suficientes para obtenção de métricas com significância estatística, enquanto em sistemas de uso exporádico, esse tempo poderia ser de dias ou semanas. Tendo em mãos os valores de variação das métricas no decorrer do tempo, o(a) desenvolvedor(a) pode se antecipar preventivamente à violação do requisito.

Outro modo relevante de análise de dados, recomendado pelo processo, é que além da síntese dos dados "isolados" de cada requisito de qualidade, pode-se apresentar sínteses gerais do sistema, a partir da ponderação dos resultados de cada requisito de qualidade e de cada funcionalidade, conforme definição do(a) desenvolvedor(a) nas Atividades 01 e 02.

A seguir cada uma das atividades do processo é apresentada de forma sucinta:

1. **Escolher as funcionalidades monitoradas, com escala de prioridades** - dentro das funcionalidades apresentadas pelo sistema, construir a lista das funcionalidades que deseja monitorar. A execução dirigida por funcionalidade é uma característica importante para a avaliação da arquitetura, a exemplo de métodos tradicionais como o ATAM (Architecture Tradeoff Analysis Method).

2. **Definir requisitos de qualidade a serem avaliados com escala de prioridades** - indicação da importância pode seja definida através de uma escala de prioridades definida através de pesos, que variam de 1 a 9.

3. **Identificar e anotar classes representativas do componente de acordo com**

o modelo de componentes - com base no estilo arquitetural ou estilos que o software apresenta serão selecionadas as classes que representam o componente que se deseja monitorar. A visão arquitetural do sistema por meio dos estilos arquiteturais permitirá identificar quais componentes são mais relevantes para a funcionalidade que está sendo monitorada. O processo assume que há consistência entre arquitetura e código-fonte. A identificação da(s) classe(s) que encapsula(m) o componente vai depender dos padrões de projeto utilizados para implementar as classes dos pacotes do sistema ou do modelo de componentes utilizado, caso a implementação seja baseada em componentes.

4. **Identificar método que representa a funcionalidade** - estabelecer quais métodos, funções ou componentes são responsáveis pela funcionalidade em questão através da identificação do possível mapeamento entre o documento de requisitos e as interfaces providas dos componentes. Por exemplo, o processo *UML Components* (CHEESMAN; DANIELS, 2000), define regras explícitas de mapeamento entre casos de uso e interfaces (com operações). De maneira geral, components de controle possuem pelo menos uma operação relativa a cada funcionalidade do sistema. A identificação da operação que representa a funcionalidade a ser monitorada é um passo fundamental, uma vez que tal operação é o alvo principal da monitoração.

5. **Identificar dependências entre componentes conforme o estilo arquitetural** - essa atividade busca analisar as interações que o componente selecionado para a monitoração realiza com outros componentes, em outras palavras, através do estilo ou estilos que compõem o sistema identificar como os componentes se relacionam. Com isso, pode-se definir se o componente é crítico com relação a monitoração, isto é, se a monitoração dele é fundamental no processo. Essa atividade também é importante do ponto de vista da identificação de componentes que são críticos para a monitoração conforme o estilo arquitetural no qual ele está classificado.

6. **Monitorar a execução do sistema** - uma vez definidas as classes representativas dos componentes, assim como a operação que representa a funcionalidade a ser monitorada, serão determinados no código os monitores para capturar os dados referentes a esses métodos. Executar o *QuAM Framework* para as classes identificadas. Serão colocadas as anotações Java para coletar os dados de funcionamento dos elementos alvo da monitoração.

7. **Agrupar métodos por elementos arquiteturais e calcular as médias** - conforme os componentes identificados para as funcionalidades monitoradas, os métodos serão agrupados de acordo com cada elemento arquitetural. Se métodos diferentes importantes para uma funcionalidade estiverem dentro do mesmo componente, eles

serão agrupados para proporcionar a visualização de como aquele componente participa na execução da funcionalidade.

8. **Avaliar proporção de participação de cada elemento arquitetural na funcionalidade, em relação ao requisito de qualidade monitorado** - conforme a funcionalidade monitorada deve-se analisar o quanto cada elemento definido na monitoração influencia no seu funcionamento. Vários elementos arquiteturais podem trabalhar conjuntamente que um requisito de qualidade seja alcançado, contudo a participação de cada um pode afetar em maior ou menor escala a satisfação do requisito monitorado.

9. **Agrupar elementos arquiteturais de acordo com abstrações dos estilos arquiteturais e calcular as médias** - complementarmente à atividade anterior, os métodos monitorados serão identificados de acordo com os estilos arquiteturais identificados, aos quais os componentes monitorados pertencem. Análise conforme a granularidade dos elementos arquiteturais.

10. **Apresentar sumário da funcionalidade** - apresenta uma síntese dos dados monitorados, contendo os dados críticos sobre cada um dos elementos monitorados para facilitar a análise do processo de monitoração para avaliar se o requisito não-funcional esperado está adequado para a funcionalidade. Essa visão abstrata da monitoração favorece a análise arquitetural do sistema e pode ser útil para o arquiteto tomar decisões relacionadas à refatoração da arquitetura de software, como por exemplo, a mudança do estilo arquitetural utilizado. O histórico pode ser absoluto ou por períodos de tempo.

11. **Apresentar sumário geral (todas as funcionalidades)** - apresenta um resumo de todas as funcionalidades e dos requisitos de qualidade monitorados. Uma vez que o processo conduz a análise de uma funcionalidade e de um determinado requisito por vez, o resumo geral é importante para avaliar diferentes fluxos de monitoração. Essa atividade é necessária para avaliar tendências de degradação, o(a) desenvolvedor(a) deve também calcular as métricas com os dados agrupados por intervalos de tempo.

Capítulo 5

Avaliação da Solução Proposta

O presente capítulo discorre sobre a avaliação da solução proposta e está dividido em cinco seções. Na primeira seção é apresentada uma visão geral da proposta dessa dissertação e do que foi realizado em sua avaliação. Na segunda seção será abordado o planejamento da avaliação, onde a mesma apresenta o método utilizado para realizar o experimento avaliativo do processo ArMoni, desenvolvido nesta dissertação.

Na terceira seção é realizada uma explanação breve sobre o sistema utilizado para realizar os testes, o Falibras. Essa descrição traz as características principais sobre o referido sistema. Na quarta seção é abordada a execução do processo proposto, onde são apresentados os participantes do experimento avaliativo, os recursos disponibilizados para as duplas, o ambiente de execução do experimento e as etapas do GQM.

Na quinta seção é descrita a forma de obtenção dos dados, tornando evidente quais as variáveis analisadas no experimento. Por fim, a sexta seção traz a análise dos resultados, onde estão presentes as reflexões e observações sobre os resultados do experimento avaliativo utilizando o GQM.

5.1 Visão Geral

O presente trabalho propõe um processo para apoiar o(a) desenvolvedor(a) de software na tarefa de monitorar os requisitos de qualidade do software a partir da arquitetura. Para isso, o processo define uma sistemática para saber o que monitorar, de acordo com os estilos arquiteturais utilizados e os requisitos a serem monitorados.

Conforme cada estilo arquitetural e cada requisito que se deseja monitorar, há pontos de monitoração que são considerados críticos dentro do sistema. Com o auxílio do processo de monitoração, é possível sistematizar atividades necessárias para identificar quais pontos dentro do sistema são importantes para a monitoração.

No Capítulo 2, subseção 2.5.2, foram apresentados seis estilos arquiteturais, de modo a explicitar quais requisitos cada um deles favorece e quais os componentes e conectores que

os compõe. Essa visão permite compreender quais pontos críticos determinado sistema pode ter e, assim, direcionar onde ele deve ser monitorado.

Foi considerado que para a realização da avaliação do processo proposto seria viável um experimento avaliativo e, desse modo, o referido processo foi aplicado na prática utilizando como cenário um sistema real, o sistema Falibras. Esse sistema foi desenvolvido na UFAL e alunos participantes desse projeto colaboraram nas atividades de avaliação do processo, além de outros alunos que não participam do referido projeto. A participação desses dois grupos de alunos foi importante para analisar como o processo pode ser viável em um contexto onde as pessoas conhecem o sistema que deseja monitorar e também por pessoas que desejam monitorar um sistema o qual ainda não tem conhecimento sobre seu funcionamento interno.

Nas duas próximas subseções será descrito o método escolhido para realizar o experimento avaliativo descrito nesta dissertação, bem como a organização adotada no presente trabalho para cada uma das partes que esse método propõe.

5.2 Planejamento da Avaliação

5.2.1 O Método Goal-Question-Metric

Para realizar a avaliação do processo proposto foi considerada a utilização do método GQM. O método GQM (*Goal-Question-Metric*) é bastante utilizado na literatura, tendo sido também empregado na ISO/IEC 9126. Esse método organiza o planejamento de uma medição de software em etapas (KOSCIANSKI; SOARES, 2007). A cada etapa deve-se definir um dos seguintes elementos (KOSCIANSKI; SOARES, 2007; WANGENHEIM, 2000):

- **Objetivos** - estabelecidos de acordo com as necessidades dos *stakeholders*. Os objetivos de medição devem ser estabelecidos conforme os requisitos do software.

- **Questões** - definidas para realizar o trabalho de medição. São as perguntas que se espera responder com o estudo. As questões fazem uma ligação entre os objetivos planejados e as métricas que podem definir o sucesso ou fracasso do produto.

- **Coleta de dados e métricas** - Consiste na obtenção dos dados e definição das métricas a serem utilizadas na comparação. Se subdivide em duas subfases:

  Categorias - particionam o conjunto de dados obtidos, isto é, podem definir diferentes tipos de informação.

  Formulários - conduzem o trabalho dos avaliadores. Essa abordagem é útil para evitar que cada avaliador utilize seu pròprio formato, o que pode induzir a erros na coleta de dados.

A seguir estão definidos os objetivos, as questões e métricas que guiaram o planejamento do experimento avaliativo.

5.2.2 GQM - *Goal(G), Question (Q), Metric (M)*

- **G1: Verificar se o processo ajuda no planejamento da monitoração do sistema**

  Q1. O processo ajuda a guiar arquitetos menos experientes na monitoração do sistema?

  M1. *Tempo necessário para anotar o sistema com os monitores.*

  M2. *Quantidade de dúvidas sobre monitoração.*

  Q2. O processo ajuda a reduzir o *overhead* dos monitores no sistema?

  M3. *Comparar o tempo de monitoração com o tempo de execução do serviço geral (um único método sendo monitorado).*

  M4. *Comparar o tempo de monitoração com o tempo de execução do serviço (todos os métodos do sistema sendo monitorados).*

- **G2: Analisar a capacidade de identificar módulos problemáticos ("gargalos" arquiteturais)**

  Q1. O processo ajuda a guiar arquitetos menos experientes na identificação de módulos problemáticos?

  M1. *Tempo necessário para identificar a(s) falha(s).*

  M2. *O(A) desenvolvedor(a) consegue ou não identificar a(s) falha(s).*

5.3 Sistema Alvo: Sistema de Tradução Automática Falibras

Para realizar a avaliação do processo ArMoni, foi escolhido o sistema *Falibras*. O sistema em questão será descrito com o intuito de compreender a arquitetura em que esse está baseado.

O referido sistema realiza a tradução *online* de texto em língua portuguesa para Língua Brasileira de Sinais (LIBRAS). A arquitetura desse sistema inicialmente apresentava uma arquitetura cliente-servidor, onde o módulo cliente consistia em uma extensão do navegador *Web Firefox*. Esse módulo realizava requisições para o módulo servidor responsável pelo processo de tradução. Contudo, essa arquitetura apresentava problemas estruturais que comprometiam a viabilidade técnica do sistema, tais como o desempenho do tradutor

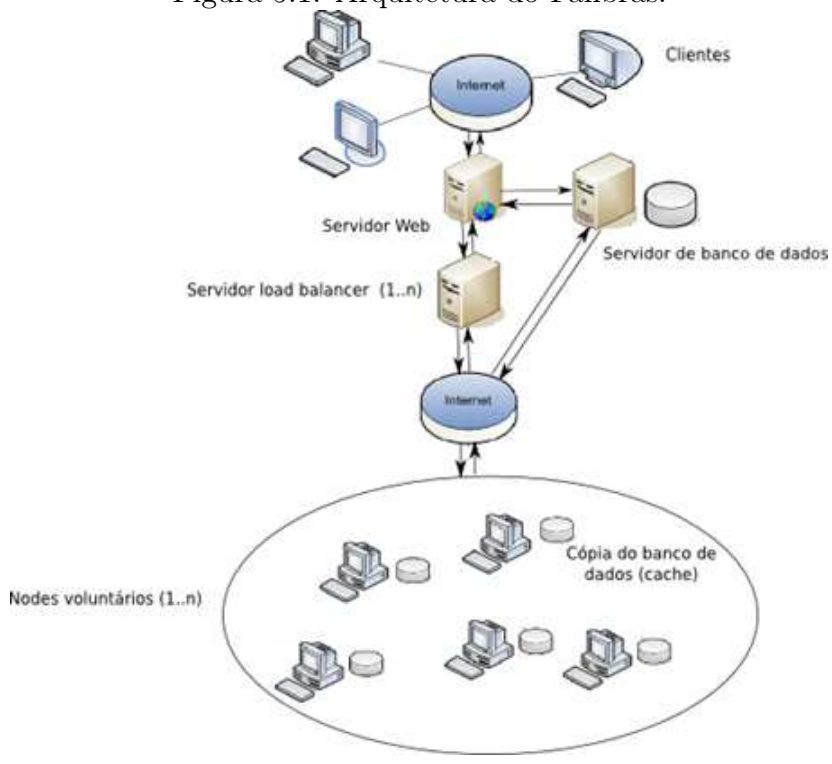
e a baixa escalabilidade que limitava o número de requisições simultâneas que o servidor conseguia atender (SILVA; BRITO, 2015).

Esse cenário gerou a necessidade de projetar uma nova arquitetura. Foi definida uma arquitetura distribuída para o Falibras utilizando *Grid Computing*. A principal motivação para *Grid Computing* foram as restrições de recursos existentes no projeto, que foram contornadas com a utilização do conceito de *Volunter Computing*. *Grid* foi adotado pela necessidade de se ter recursos distribuídos em vários locais utilizando recursos computacionais heterogêneos e um número variável e dinâmico de nós, que facilita a entrada e saída de voluntários na rede de processamento colaborativo. A arquitetura proposta adota o conceito de replicação de dados com o uso de cache. Em suma, o novo servidor do Falibras possui uma arquitetura distribuída utilizando *Grid Computing* e segue o estilo "Componentes Independentes" juntamente com o estilo "Cliente-Servidor" (SILVA; BRITO, 2015).

O *framework* utilizado para o desenvolvimento da nova arquitetura definida para o *Falibras* foi o JPPF 5.0.4, o qual é escrito em Java é *open source*. Foram utilizadas as abstrações *Tasks* e *Jobs*. As *Tasks* são utilizadas para representar qualquer tarefa que será executada no *grid*, e o *Job* é composto por um conjunto de *Tasks*. Cada requisição de tradução é considerada uma *Task*. Ambos são executados usando o algoritmo de escalonamento *Round-Robin* (SILVA; BRITO, 2015).

Segundo Silva e Brito (2015), os componentes da arquitetura, ver Figura 5.1, são:

Figura 5.1: Arquitetura do Falibras.



Fonte: Silva e Brito (2015).

- **Nodes** - Usuários voluntários que contribuem com seus recursos computacionais ao *grid* processando as traduções;
- **Clientes** - Requisitam as traduções ao sistema via serviços *Web*;
- **Servidor Web** - Responsável por receber as solicitações dos clientes e encaminhá-las ao servidor de tradução;
- **Servidor de banco de dados** - Responsável por armazenar o banco de dados de tradução, além de disponibilizar os dados de cache para os *nodes*;
- **Servidor *load balancer*** - tem o papel de escalonar tarefas de tradução aos *nodes* utilizando o algoritmo *Round-Robin*, com a possibilidade de ter mais de um servidor.

Segundo Brito, Franco e Coradine (2012), o *Falibras* apresenta sua arquitetura baseada nos seguintes estilos:

- **Centrada em Dados** - Integração entre módulos geradores das regras e os respectivos executores, pois trata-se de um sistema baseado em regras.
- **MVC** - Separação entre a visão e o controle, ver Figura 5.2. Desse modo, as funcionalidades e os dados ficam separados. Possibilita a especificação de interfaces distintas para o Falibras (por exemplo, uma interface *desktop* e uma interface *Web* integrada ao navegador *Firefox*).
- **Cliente-Servidor** - Descentralização da execução para aumentar a escalabilidade da arquitetura.

Em linhas gerais, quando o cliente requisita uma tradução de uma ou mais frases ao sistema Falibras, a frase é inserida em uma *task* dentro de um *job*. Esse é transferido para o servidor *load balancer*, atuando como um escalonador, e enviando para algum *node* ocioso conectado ao *grid*. Então, o *node* processa a tradução da frase e retorna uma lista de *glosas*. A mesma percorre o caminho inverso e é exibida a frase traduzida para o cliente. Para um usuário se tornar um *node* voluntário, o mesmo baixa e executa a aplicação denominada "Falibras *node*" que é uma extensão do componente "JPPF *node*" (SILVA; BRITO, 2015).

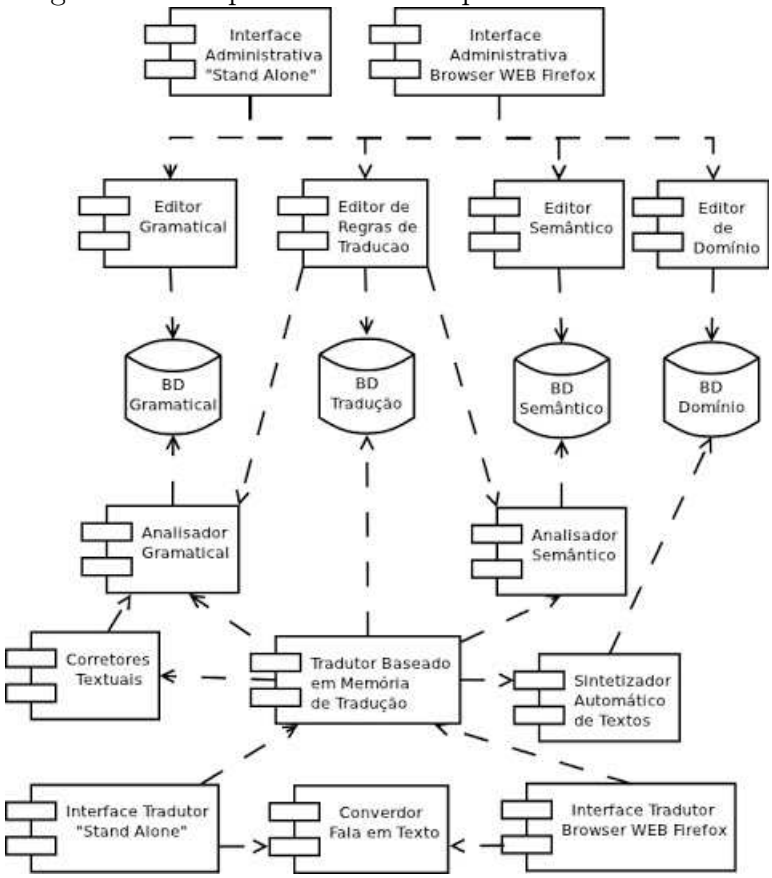
Na Figura 5.3 pode-se ver como está composta a arquitetura do Falibras em termos de modularização funcional, baseada em componentes.

Na Figura 5.4 é apresentado um diagrama com a arquitetura do sistema Falibras explicitando os estilos arquiteturais que ele possui. Esse diagrama nos permite compreender como os estilos estão organizados dentro do sistema e é um artefato importante para pensar sobre decisões arquiteturais, bem como para auxiliar no planejamento da monitoração dos componentes do sistema.

Figura 5.2: Diagrama do Falibras baseado no modelo MVC.

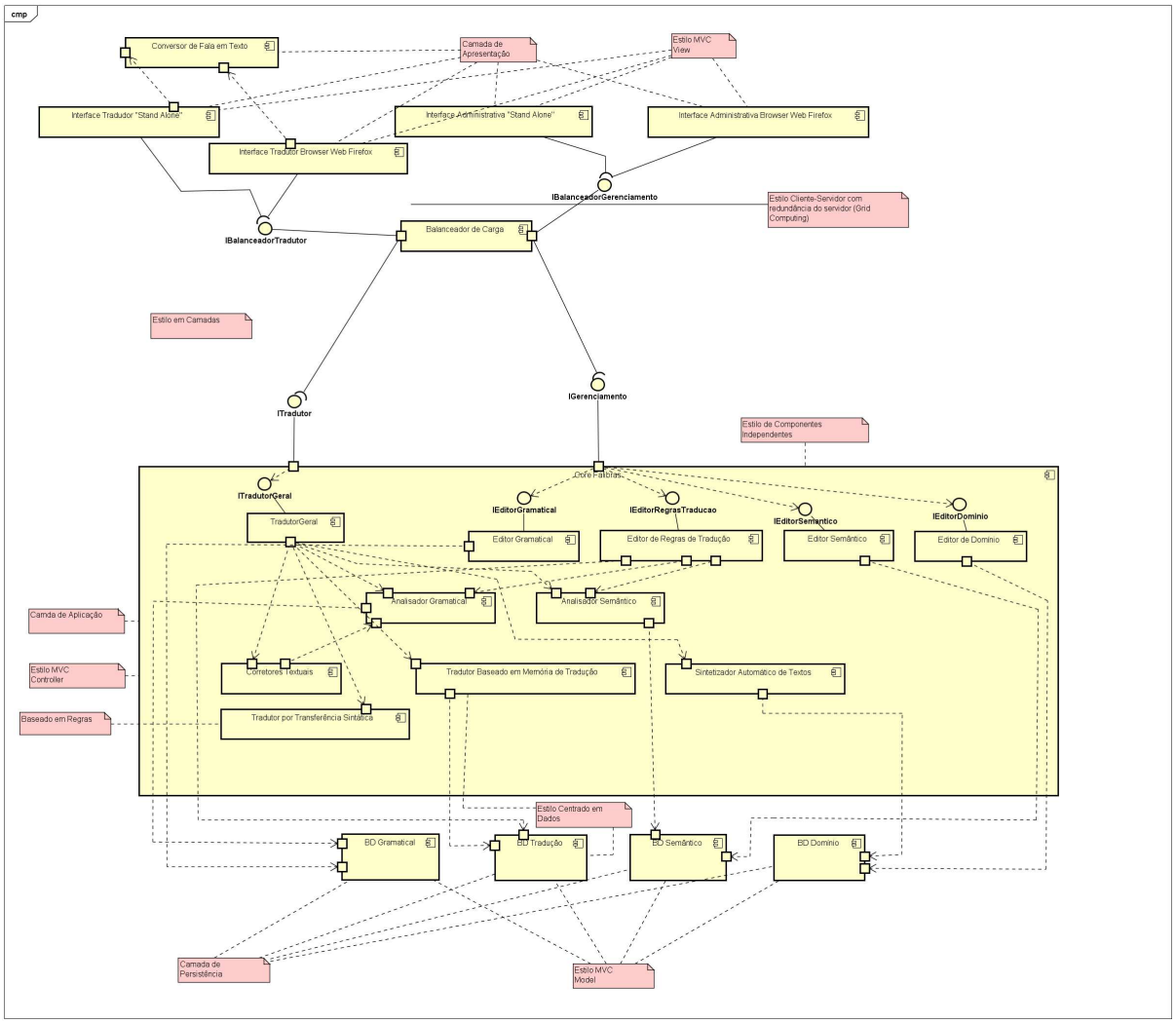
Fonte: Brito, Franco e Coradine (2012).

Figura 5.3: Arquitetura de Componentes do Falibras.



Fonte: Brito, Franco e Coradine (2012).

Figura 5.4: Diagrama UML do Falibras e os seus estilos arquiteturais



Fonte: Elaborada pela autora.

5.4 Execução do Processo Proposto

5.4.1 Participantes

O público participante das atividades para avaliar o referido processo foi composto por 16 (dezesseis) alunos da UFAL - Campus Arapiraca, onde 8 (oito) desses alunos conheciam o sistema Falibras e 8 (oito) não tiveram qualquer interação com ele. Foi aplicado o *pair programming* para realizar as atividades de monitoração.

Através do *pair programming* o objetivo foi permitir a interação entre os alunos para realizarem as atividades de monitoração do sistema. Desse modo, as Duplas ficaram dispostas de modo que 4 (quatro) conheciam o Projeto Falibras e 4 (quatro) não tinham conhecimento prévio sobre ele, ver Tabela 5.1. Essa escolha teve o intuito de não influenciar o experimento apenas com o conhecimento prévio do sistema para teste. Ainda, foi aplicado o **delineamento em quadrado latino** (KEEDWELL; DÉNES, 2015), de

modo que foi planejada uma disposição específica de recursos que foram disponibilizados para as Duplas.

Além disso, ao fim das atividades foi disponibilizado um *questionário de avaliação da monitoração do Falibras*. O questionário foi semiestruturado, onde foi divido em duas partes: 3 questões fechadas e o espaço para comentários e sugestões, que foi uma questão opcional. Essa configuração foi considerada adequada, pois há aspectos específicos da monitoração que é importante saber quanto à visão dos participantes sobre o experimento e foi fundamental considerar uma parte aberta no questionário para que os mesmos expressassem suas opiniões.

5.4.2 Recursos disponibilizados para as duplas

Os recursos disponibilizados foram o *diagrama de componentes da arquitetura do Falibras* onde estão explicitados os estilos arquiteturais que o sistema possui e o *diagrama de atividades do processo de monitoração* ArMoni.

No delineamento em quadrado latino, ver Tabela 5.1, foi considerada a seguinte organização: Duas Duplas realizaram a atividade com o auxílio do diagrama de componentes do Falibras e o diagrama de atividades do processo de monitoração; duas Duplas utilizaram apenas o diagrama de atividades do processo ArMoni, sem detalhes da arquitetura do Falibras; duas Duplas tiveram acesso apenas ao diagrama de componentes do Falibras; e duas Duplas não tiveram nenhum recurso para realizar a atividade.

Outro ponto importante é que em cada uma das duas Duplas dispostas quanto aos recursos, em uma os dois integrantes da Dupla apresentavam conhecimento prévio sobre o Falibras, e na outra Dupla os integrantes não tiveram qualquer contato com o código-fonte do referido sistema.

Tabela 5.1: Delineamento do quadrado latino

Dupla	**ArMoni**	**Arquitetura**	**Sem recurso**	**Conhecimento prévio Falibras**
1	X	X		X
2	X	X		
3	X			X
4	X			
5		X		X
6		X		
7			X	X
8			X	

Fonte: Elaborada pela autora.

5.4.3 Ambiente de Execução

Os *logs* foram obtidos através da monitoração do Falibras que está instalado em um ambiente que possui ao todo 2 máquinas virtuais (um servidor *Web* com distribuição de carga e 1 nó de tradução) e 2 máquinas reais (1 nó de tradução e 1 máquina cliente). As máquinas virtuais ficam hospedadas em um *cluster* localizado no NTI (Núcleo de Tecnologia da Informação) da UFAL - Campus Arapiraca.

A *máquinas virtuais* possuem a seguinte configuração:

Servidor Web com distribuição de carga entre os nós de tradução:

- SO: Debian (modo texto);
- 512 MB de RAM;
- 2 núcleos de processamento;
- Máquina virtual.

Nó de tradução 1:

- SO: Debian (modo texto);
- 1GB de RAM;
- 1 núcleo de processamento;
- Máquina virtual.

Configuração das *máquinas reais*:
Nó de tradução 2:

- SO: Ubuntu 16.04 (modo gráfico);
- 4GB de RAM;
- 2 núcleos de processamento (Intel Core2Duo);
- Máquina real.

Máquina Cliente:

- SO: Ubuntu 16.04 (modo gráfico);
- 4GB de RAM;
- 2 núcleos de processamento (Intel Core2Duo);

- Máquina real.

Ainda, na máquina cliente foi utilizado:

- *Java da Oracle, versão "1.8.0_111"*;
- *Java(TM) SE Runtime Environment (build 1.8.0_111-b14)*;
- *Java HotSpot(TM) 64-Bit Server VM (build 25.111-b14, mixed mode)*;
- *Eclipse Java EE IDE for Web Developers. Versão: Neon Release (4.6.0). Build id: 20160613-1800.*

O experimento avaliativo realizado com os alunos foi realizado no Laboratório de Ensino em Ciência da Computação (LECC) da UFAL - Campus Arapiraca e para isso foram instaladas e configuradas oito máquinas virtuais para os alunos realizarem o teste de monitoração. Para evitar interferências externas, optou-se por executar os testes na intranet da UFAL. Caso fosse utilizada a Internet a latência da rede poderia variar e até mesmo interferir no resultado do cálculo de *overhead*.

5.4.4 Etapas do GQM

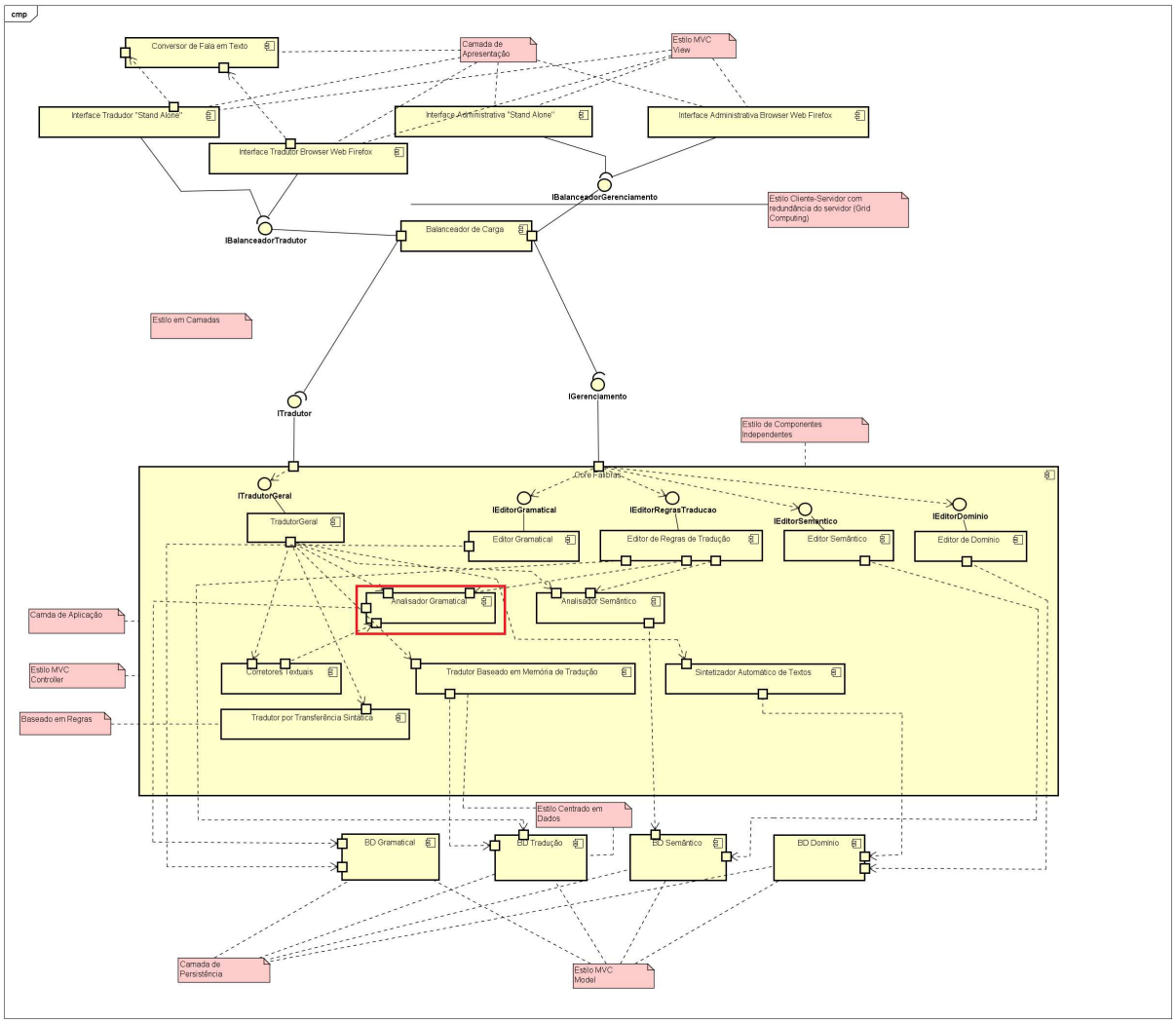
O GQM foi dividido em duas etapas: *anotação das classes para monitoração da função de tradução do Falibras* e *monitoração para solucionar uma falha injetada no código do sistema*.

Na primeira primeira etapa, os alunos tiveram acesso ao código do Falibras e acesso ao *QuAM Framework* para fazer as anotações que ele julgaram necessárias para monitorar tal função. As Duplas 1 e 2 utilizaram o processo ArMoni e a arquitetura de componentes do Falibras para auxiliar nessa atividade; as Duplas 3 e 4 tiveram acesso apenas ao processo de monitoração; as Duplas 5 e 6 utilizaram apenas a arquitetura do sistema; e as Duplas 7 e 8 não tiveram recursos auxiliares. Desse modo, uma das variáveis consideradas nesse etapa foi o *tempo de anotação das classes* do Falibras.

Na segunda etapa foi injetada uma falha no código do sistema para afetar a execução da funcionalidade de tradução, mais especificamente no *módulo de análise morfossintática*, ver Figura 5.5. O componente "TradutorGeral" não foi alterado, mas foi afetado indiretamente, por utilizar o componente alterado: "AnalisadorGramatical". Dessa forma, apenas um componente foi alterado. A monitoração foi voltada para analisar o requisito de qualidade de eficiência do sistema.

A falha foi implantada no sistema após as anotações que os alunos fizeram com o intuito de evitar que os mesmos a identificasse enquanto definiam os pontos de monitoração. O código inserido teve o objetivo de tornar a execução mais lenta, através da inserção do trecho de código apresentado na Listagem 5.1.

Figura 5.5: Componente onde foi injetada a falha



Fonte: Elaborada pela autora.

Código 5.1: Trecho de código para injetar uma falha no Falibras

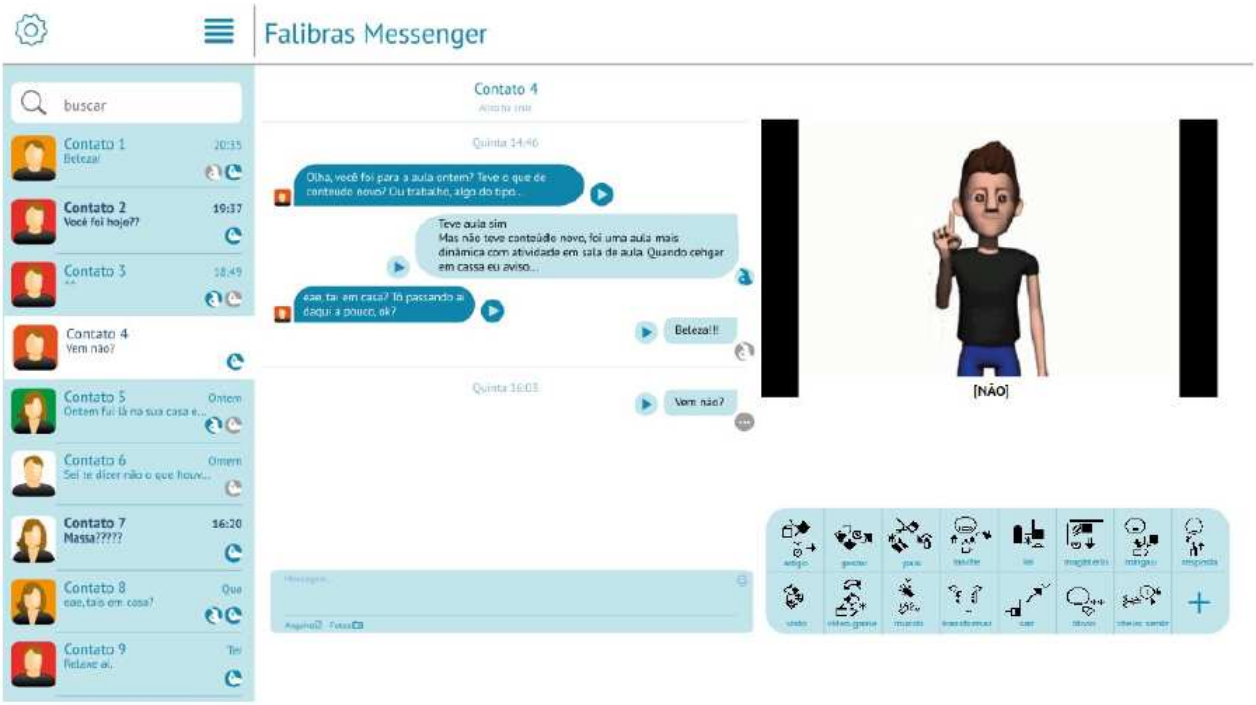
```
for (int x=1; x<5000; x++){
    long y = (12*x)/2;
}
```

Fonte: Elaborado pela autora.

Depois de implantadas as falhas, o sistema foi colocado em funcionamento e recebeu mensagens aleatórias de cenários de tradução no contexto de conversas utilizando o aplicativo *Telegram* integrado ao Falibras, denominado *Falibras Messenger*, ver Figura 5.6. Segundo Silva, Brito e Barbosa (2015), o "Falibras Messenger é um tradutor adaptado a um comunicador de mensagens instantâneas integrado com a rede social Telegram". O objetivo desse aplicativo é possibilitar que ouvintes e surdos possuam se comunicar por meio de uma interface adaptada, com isso promover o aprendizado de uma nova língua

(SILVA; BRITO; BARBOSA, 2015).

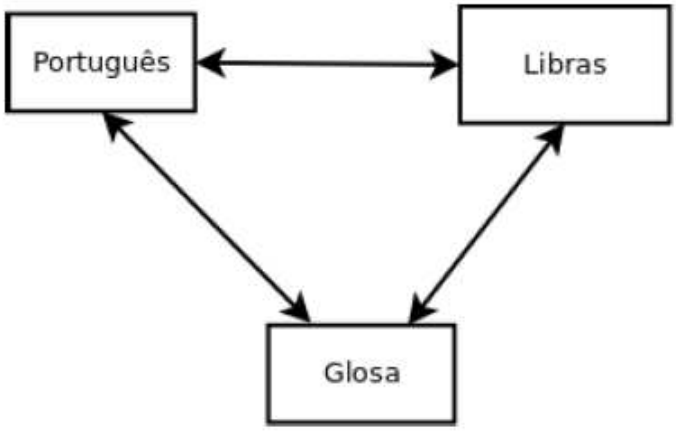
Figura 5.6: Tela principal do Falibras Messenger



Fonte: Silva, Brito e Barbosa (2015).

Na Figura 5.7 é apresentado um resumo das combinações de tradução que o sistema realiza. Podem ser intercaladas traduções entre Língua portuguesa, Libras e Glosa. As glosas referem-se a anotações que dão sentido a uma palavra. Elas fazem o intermédio entre Língua portuguesa e Libras para auxiliar no processo de aquisição da Língua portuguesa pelo surdo ou da Libras pelo ouvinte (SILVA; BRITO; BARBOSA, 2015).

Figura 5.7: Possibilidade de traduções do Falibras Messenger



Fonte: Silva, Brito e Barbosa (2015).

As mensagens utilizadas no experimento variaram entre mensagens curtas, médias e longas. Ao fim da monitoração foi observado se as Duplas conseguiram atingir o objetivo

dessa segunda etapa que foi definido como identificar onde estava a falha no sistema, isto é, qual o módulo problemático em termos de desempenho.

5.5 Forma de Obtenção de Dados

Os dados foram obtidos a partir dos *logs* de monitoração do Falibras após a falha ser "injetada". Os *logs* foram analisados pelas Duplas com o objetivo de auxiliar na identificação da falha. Foi estabelecido o período de 24 horas para que o sistema funcionasse com a falha e gerasse os *logs* para analisar posteriormente.

Foram consideradas as seguintes variáveis para análise: *tempo para anotar (min), quantidade de dúvidas, classes anotadas, overhead identificado (%), classes totais, tempo melhor caso (milissegundos), tempo pior caso (milissegundos), conseguiu o objetivo, tempo de análise (min), total de traduções.*

5.6 Análise dos Resultados

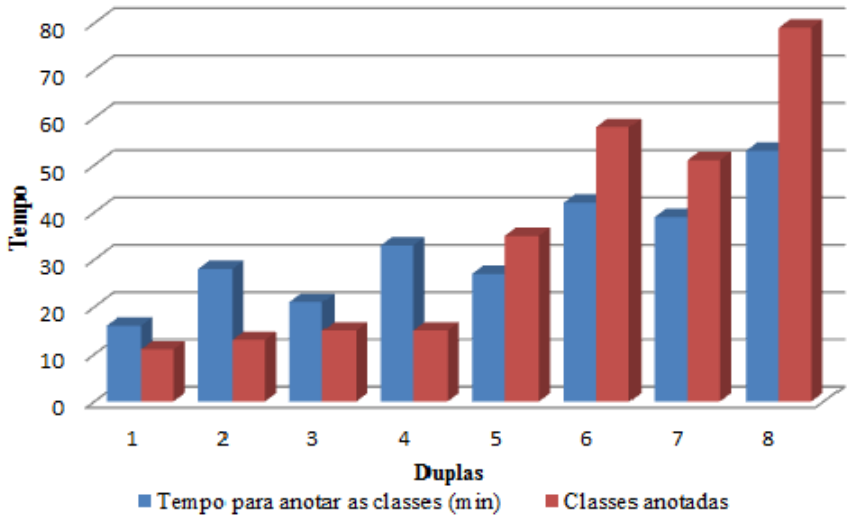
Para realizar a análise dos dados, foram construídos gráficos com base nos dados obtidos a fim de compreender como as Duplas se desenvolveram ao longo das duas etapas do experimento avaliativo.

A Figura 5.8 mostra o tempo que cada Dupla precisou para fazer as anotações das classes que consideraram importantes para realizar a monitoração do Falibras. Os dados das variáveis de *tempo para anotar as classes* e *classes anotadas* estão dispostos por Dupla para facilitar compreensão de como cada uma realizou o experimento. Pode-se notar que as Duplas 2, 4, 6 e 8 demandaram mais tempo para decidir quais classes deveriam ser monitoradas, em relação as Duplas que tinham o mesmo recurso e que conheciam o Falibras.

Em especial, a Dupla 8 que não possuía nenhum recurso, aparentemente, teve mais dificuldade em fazer as anotações no Falibras e fez anotações em um número maior de classes, onde selecionou 79 das 105 classes. Monitorar muitas classes não é uma boa estratégia, pois pode interferir no funcionamento do sistema e torná-lo demasiadamente lento. Além de gerar um maior volume de dados de *log*, o que pode dificultar a análise desses dados.

Nesse sentido, pode-se considerar que um(a) desenvolvedor(a) que possui pouca experiência ou não tem qualquer recurso para auxiliar a monitoração do sistema apresenta mais dificuldades para executar essa atividade. Na Dupla 1 foi percebido que o uso conjunto do processo ArMoni e o conhecimento dos componentes da arquitetura levou-a a desempenhar em menor tempo a atividade solicitada, além disso o conhecimento prévio sobre o Falibras, provavelmente, contribuiu para que ela tivesse o melhor resultado dentre

Figura 5.8: Tempo de anotação das classes realizado pelas Duplas



Fonte: Elaborada pela autora.

as demais.

Na Figura 5.9 pode-se observar o desempenho geral das Duplas gradativamente. Nota-se que quanto menos recursos que auxiliem as atividades de monitoração, mais dificuldade as Duplas aparentam ter. Isso é percebido quanto ao tempo de anotação demandado, bem como consideraram necessário monitorar mais classes. Provavelmente a maior escolha de classes está relacionada com a expectativa de cobrir mais código do sistema e descobrir mais rapidamente a falha. Contudo, essa decisão prejudica a análise do *log* como será visto mais adiante.

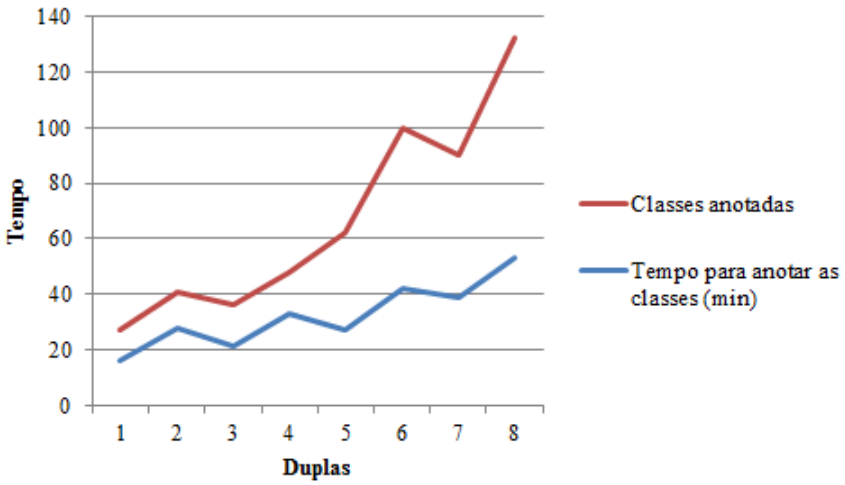
Desse modo, pode-se entender que realizar a monitoração de um sistema pode ser potencializada através de atividades que direcionem os alvos de monitoração, bem como o acesso à arquitetura permite que os alunos que a analisaram reflitam sobre os componentes que o referido sistema possui. Além disso, a compreensão sobre os estilos arquiteturais é um ponto chave para analisar o sistema.

Durante a configuração da monitoração, as Duplas apresentaram dúvidas quanto ao código do sistema, como o processo proposto funcionava, sobre o diagrama do Falibras e como o *framework* podia ser utilizado. Por isso, também é interessante o registro da quantidade de dúvidas, ver Tabela 5.2, que elas manifestaram durante a primeira etapa.

Na Figura 5.10 pode-se visualizar a quantidade de traduções que o Falibras realizou, enquanto estava sendo monitorado. Considerando que o tempo de monitoração do sistema foi de 24 horas, a variação no número de traduções realizadas nesse período de tempo foi considerada baixa e não beneficiou nenhuma das Duplas de voluntários.

Na segunda etapa, a falha foi injetada no sistema e o mesmo passou a ser monitorado. Após o período de monitoramento, através dos *logs* os alunos puderam analisar os dados dessas classes para buscar solucionar o problema. As classes escolhidas para monitoração

Figura 5.9: Tempo de anotação do Falibras para a monitoração



Fonte: Elaborada pela autora.

Tabela 5.2: Quantidade de dúvidas na definição da monitoração

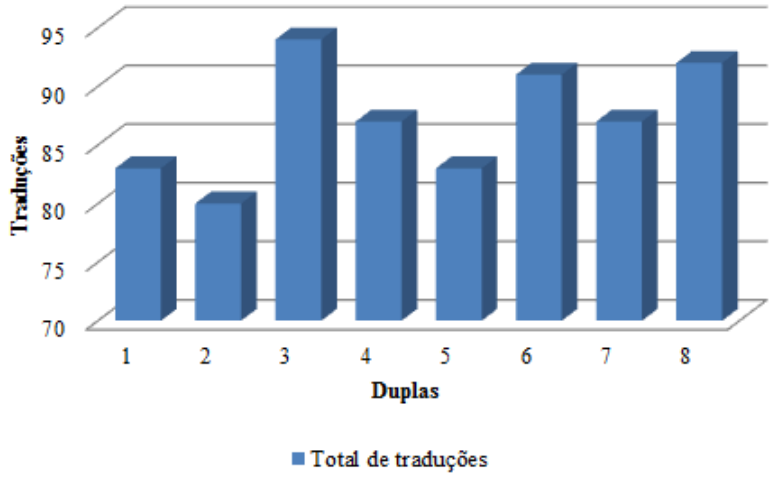
Dupla	Dúvidas
1	16
2	29
3	23
4	31
5	19
6	27
7	07
8	13

Fonte: Elaborada pela autora.

na primeira etapa permaneceram as mesmas e através do funcionamento delas é que os *logs* foram gerados. Conforme as Figuras 5.11 e 5.12, percebe-se que o tempo de análise variou entre as Duplas, contudo é visível que aquelas que fizeram uso de algum recurso ou de ambos para auxiliar a monitoração levaram menor tempo para identificar onde a falha estava localizada em relação a Dupla com mesmo recurso e sem conhecimento sobre o Falibras.

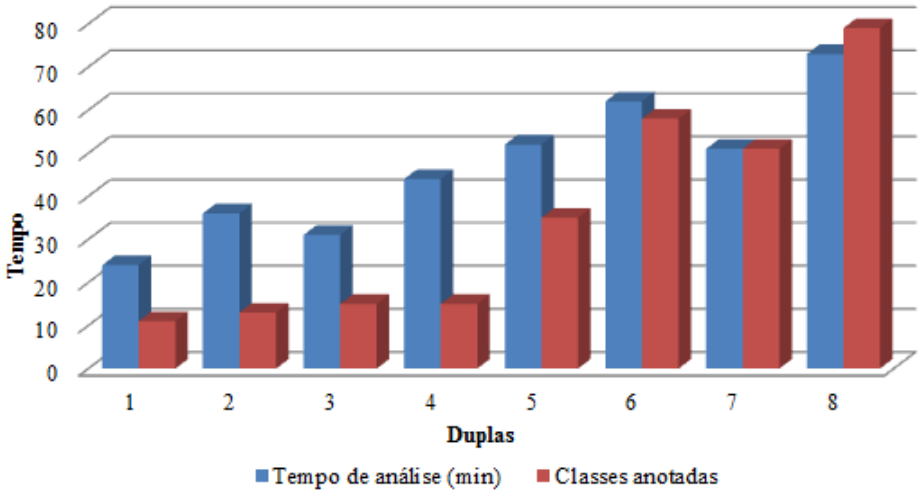
Sobre as Duplas 1, 3, 5 e 7, que conheciam o Falibras, observou-se que o tempo empregado pelas 3 primeiras foi gradualmente maior e percebe-se que os recursos utilizados por elas contribuiram para que seus tempos de análise fossem menores que os tempos empregados pelas Duplas 2, 4, 6 e 8, que não possuíam conhecimento sobre o sistema. Considerando as Duplas de acordo com os recursos que utilizaram, nota-se que a Dupla 1 em relação a Dupla 2 apresentou melhor desempenho quanto ao tempo. Ambas usaram o processo proposto e o diagrama de componentes do Falibras. O conhecimento prévio sobre o sistema, provavelmente ajudou a fazer a análise mais rapidamente. Ainda, é interessante

Figura 5.10: Quantidade de traduções durante a monitoração



Fonte: Elaborada pela autora.

Figura 5.11: Tempo de identificação das falhas pelas Duplas e classes anotadas



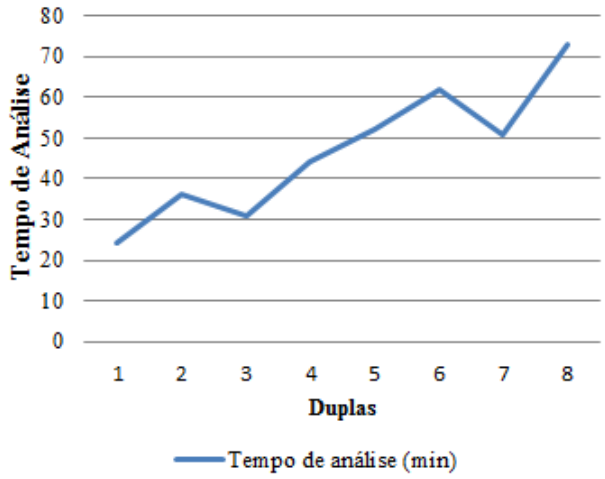
Fonte: Elaborada pela autora.

notar que essa situação se repete no restante das Duplas (3 e 4, 5 e 6).

Além disso, é interessante ressaltar o benefício que o processo proporcionou para as Duplas. Por exemplo, ao comparar os resultados das Duplas 4 e 5. A Dupla 4 seguindo os passos pragmáticos do processo conseguiu um melhor aproveitamento do que a Dupla 5, que além de já conhecer previamente o sistema, teve acesso ao documento de arquitetura. Esse é um fato importante quanto a viabilidade do ArMoni, indicando que pode de fato auxiliar a monitoração mesmo quando não se conhece os detalhes internos do sistema.

Sobre as Duplas que não tinham nenhum recurso, o tempo da Dupla 7, que possuía conhecimento sobre o sistema, foi menor que o da Dupla 8, que não conhecia, onde essa característica pode ter sido decisiva para a diferença nos resultados. As duas Duplas

Figura 5.12: Tempo de identificação das falhas pelas Duplas

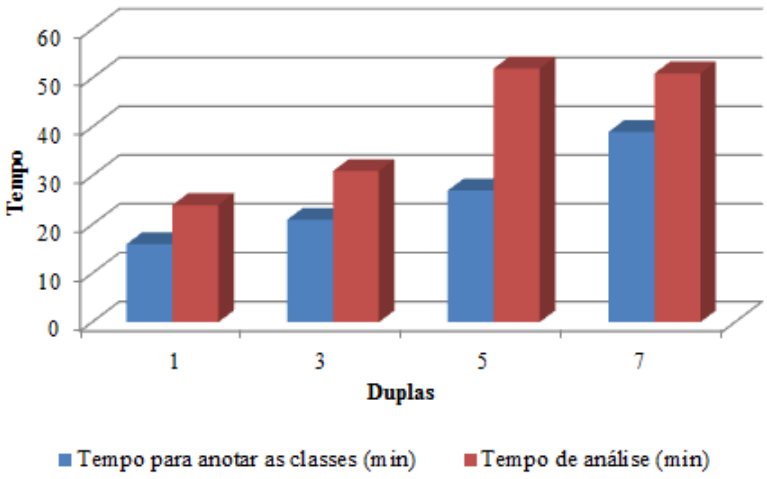


Fonte: Elaborada pela autora.

consideraram um número grande classes para monitoração. De modo geral, as demais Duplas que não conheciam sobre o Falibras e usaram os recursos disponibilizados tiveram resultados melhores que as Duplas que estavam fazendo a atividade sem qualquer auxílio.

Considerando que houve dois grupos, onde um possuía conhecimento sobre o Falibras e outro não, fica claro que em relação a cada um deles, tanto no *tempo de anotação das classes* quanto no *tempo de análise*, as Duplas que tiveram acesso aos recursos obtiveram melhores resultados. Esse aspecto pode ser visualizado nas Figuras 5.13 e 5.14.

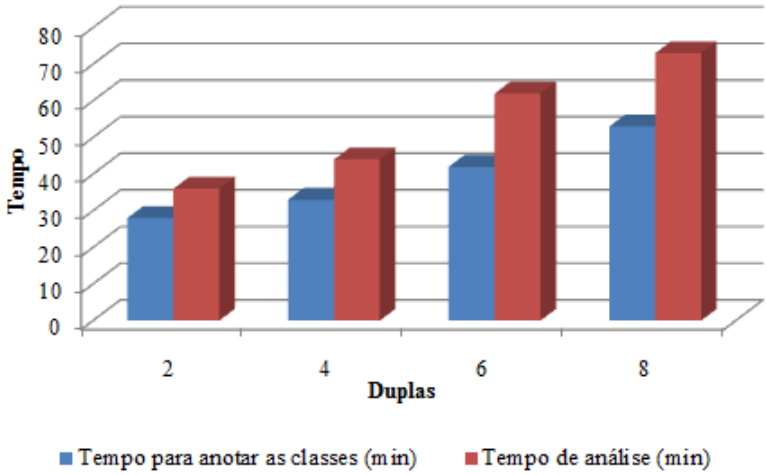
Figura 5.13: Grupo com conhecimento prévio sobre o Falibras



Fonte: Elaborada pela autora.

Ao fim da segunda etapa, apenas uma das Duplas, Dupla 8, não conseguiu identificar onde estavam as falhas no sistema, ver Tabela 5.3. Esse resultado pode ser atribuído ao

Figura 5.14: Grupo sem conhecimento prévio sobre o Falibras



Fonte: Elaborada pela autora.

fato dela ser a Dupla com as condições mais difíceis dentro do estudo, pois não teve acesso a nenhum recurso para auxiliar a monitoração, além de não conhecer o funcionamento interno do sistema. Esses aspectos são bastante relevantes e em relação às demais Duplas apontam para a importância de utilizar instrumentos que auxiliem na monitoração. Pode-se afirmar isso, pois mesmo as Duplas que não conheciam o sistema, mas se basearam no processo e/ou no diagrama de componentes onde os estilos arquiteturais estavam explícitos, conseguiram atingir o objetivo estabelecido.

Nas Figuras 5.13 e 5.14 nota-se que as Duplas que utilizaram apenas o ArMoni tiveram melhores resultados que aquelas que só analisaram a arquitetura ou que estavam sem os recursos. Outro aspecto que fica claro é a união positiva para a monitoração que o processo traz ao ser aliado à análise da arquitetura com os estilos arquiteturais.

Tabela 5.3: Duplas que identificaram as falhas no Falibras

Dupla	**Objetivo**
1	Sucesso
2	Sucesso
3	Sucesso
4	Sucesso
5	Sucesso
6	Sucesso
7	Sucesso
8	Não conseguiu

Fonte: Elaborada pela autora.

Na Figura 5.15 e na Tabela 5.5 pode-se visualizar o *overhead* da execução da funcionalidade de tradução do Falibras na monitoração realizada por cada uma das Duplas. Foi

observado que quanto mais classes os alunos anotaram, maiores foram os valores. Logo, o Falibras teve seu funcionamento mais sobrecarregado quanto maior o número de pontos de monitoração. Esse fato é importante, pois quanto mais criteriosa for a monitoração, menos comprometido fica o sistema.

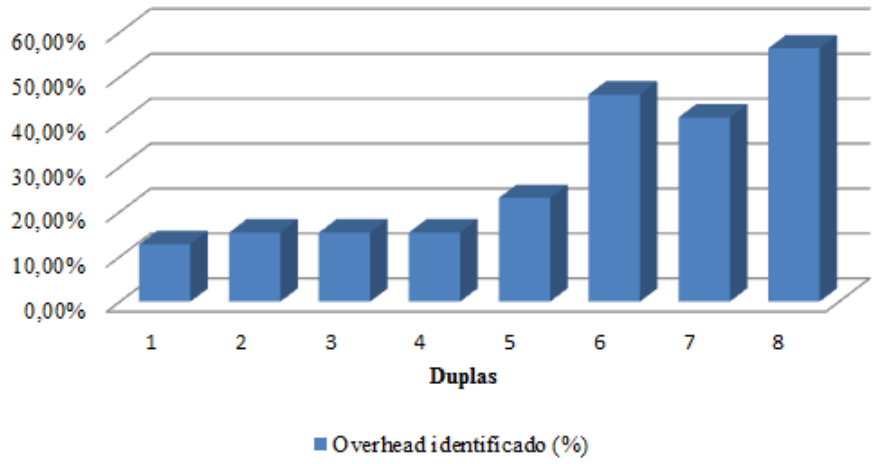
O cálculo para determinar o *overhead* do sistema foi com base em dois cenários de utilização da função de tradução do Falibras. O primeiro considera o sistema sem nenhum alvo de monitoração, de modo que ele não tem seu funcionamento alterado. O segundo refere-se ao sistema ter todas as suas classes monitoradas na Tabela 5.4

Tabela 5.4: Melhor caso e pior caso de funcionamento do Falibras

Melhor caso (ms)	**Pior caso (ms)**
39	67

Fonte: Elaborada pela autora.

Figura 5.15: *Overhead* na monitoração do Falibras



Fonte: Elaborada pela autora.

Além disso, foi disponibilizado um formulário *online* denominado *Questionário de Avaliação do Sistema Falibras*, ver apêndice A, para obter uma visão qualitativa sobre a experiência dos alunos no experimento executado. Foram elencadas as seguintes perguntas:

- Como você classifica o esforço para anotação das classes?
- Como você classifica o esforço para avaliação do *log*?
- Como você classifica o esforço para identificação de módulos problemáticos?

Tabela 5.5: *Overhead* na monitoraço do Falibras

Dupla	Overhead	Classes anotadas
1	12,82%	11
2	15,38%	13
3	15,38%	15
4	15,38%	15
5	23,07%	35
6	46,15%	58
7	41,02%	51
8	56,41%	79

Fonte: Elaborada pela autora.

Para definir as alternativas de resposta foi considerado o nível de complexidade que eles julgaram pertinentes para cada pergunta:

1. Complexidade baixa; eu utilizaria novamente.
2. Complexidade moderada; eu utilizaria novamente.
3. Difícil; não sei se é viável.
4. Inviável para utilização prática.

Na Figura 5.16 pode-se visualizar as respostas dadas pelos alunos para classificar o esforço para fazer anotação das classes. A maioria do alunos considerou que a complexidade para realizar essa atividade é baixa. Ainda, quatro alunos consideraram que a complexidade foi moderada e um considerou difícil. O aluno que considerou a anotação difícil era integrante da Dupla 8. A Tabela 5.6 mostra quantos alunos escolheram cada nível para avaliar essa atividade.

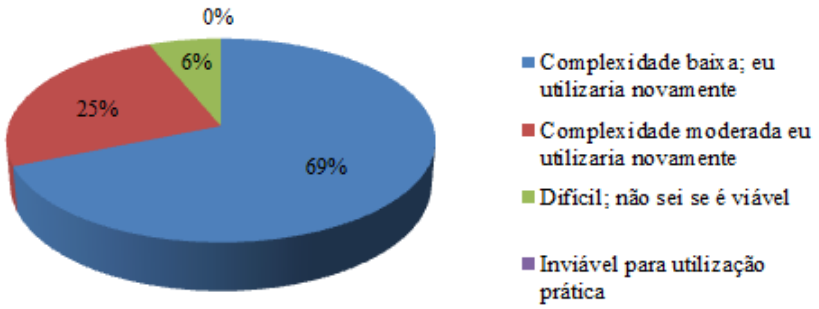
Na Tabela 5.7 está organizada a frequência das Duplas quanto os níveis de esforço para a atividade de anotação. Cada um dos integrantes das Duplas respondeu aos questionários *online* e isso foi importante para ter uma visão geral sobre como cada indivíduo avaliou o experimento.

Tabela 5.6: Respostas sobre o esforço para anotação das classes

Avaliação	Quantidade de respostas
Complexidade Baixa	11
Complexidade Moderada	04
Difícil	01
Inviável	00

Fonte: Elaborada pela autora.

Figura 5.16: Classificação do esforço para anotação das classes



Fonte: Elaborada pela autora.

Tabela 5.7: Frequência das respostas das duplas sobre a anotação das classes

Respostas	**Duplas**
Complexidade Baixa	1,1,2,3,3,4,4,5,6,6,7
Complexidade Moderada	2,5,7,8
Difícil	8
Inviável	

Fonte: Elaborada pela autora.

Na Figura 5.17 pode-se visualizar como os alunos classificaram o esforço para avaliar o *log*. A Tabela 5.8 mostra quantos alunos escolheram cada nível para avaliar essa atividade. A Tabela 5.9 apresenta os níveis de esforço considerados por cada integrante das Duplas.

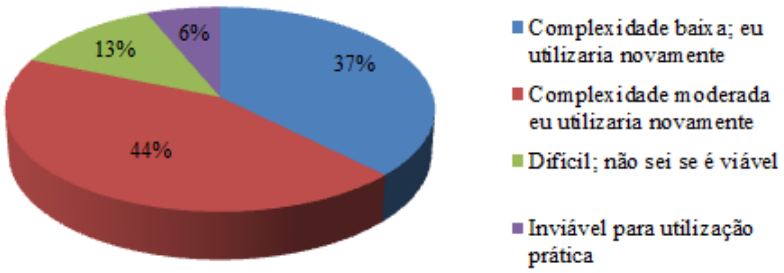
Tabela 5.8: Respostas sobre o esforço para avaliação do *log*

Avaliação	**Quantidade de respostas**
Complexidade Baixa	06
Complexidade Moderada	07
Difícil	02
Inviável	01

Fonte: Elaborada pela autora.

Foi percebido que os alunos apresentaram mais dificuldades quanto à análise dos *logs*. Nos comentários realizados nos questionários, alguns deles citaram a dificuldade de analisar os dados dos *logs* por serem volumosos, não conseguir identificar os dados que são úteis para análise da monitoração e, por isso, consideraram difícil filtrá-los. Dessa forma, apesar da análise de *logs* ser uma atividade mais complexa, considera-se que essa se torna mais

Figura 5.17: Classificação do esforço para avaliação do *log*



Fonte: Elaborada pela autora.

Tabela 5.9: Frequência das respostas das duplas sobre a avaliação do *log*

Respostas	**Duplas**
Complexidade Baixa	1,1,2,2,3,4
Complexidade Moderada	3,4,5,5,6,7,7
Difícil	6,8
Inviável	8

Fonte: Elaborada pela autora.

difícil para as Duplas que não utilizaram o processo, principalmente, quando é necessário monitorar mais de uma funcionalidade do sistema.

Com o processo, a monitoração é realizada para cada funcionalidade, ao invés de todas de uma só vez. Dessa forma, aliado a uma seleção mais criteriosa das classes a serem monitoradas, o volume de dados gerado tende a ser menor e mais contextualizado por funcionalidade.

Ademais, foi observado que os comentários com reclamações quanto ao grande volume de dados estiveram presentes nas respostas das Duplas que anotaram muitas classes que não eram estratégicas para a monitoração, consequentemente gerando dados sem relevância para identificar qual o módulo problemático. Aqueles que utilizaram o processo geraram dados mais específicos e pontuais. Como visto na Tabela 5.5, as Duplas 1, 2, 3 e 4 fizeram menos anotações e seguiram o processo para guiar as escolhas das classes. As Duplas 5, 6, 7 e 8 anotaram um número maior de classes e consideraram que os *logs* eram volumosos e com muitos dados considerados irrelevantes.

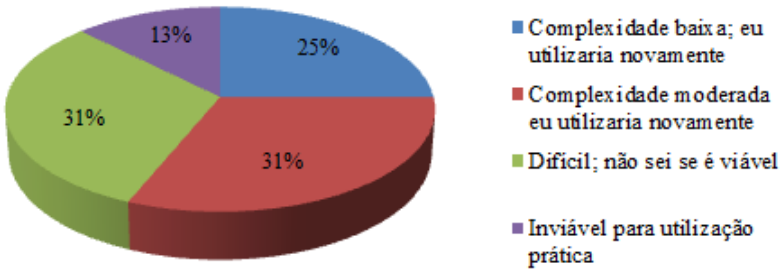
Para auxiliar a análise dos dados dos *logs* foram considerados os seguintes aspectos:

- Agrupar por componente;

- Agrupar os dados por intervalos de tempo;
- Calcular a média dos valores;
- Comparar o histórico de cada componente.

A Figura 5.18 refere-se ao esforço que os alunos fizeram para identificar os módulos problemáticos. A Tabela 5.10 mostra quantos alunos escolheram cada nível para avaliar essa atividade. Na Tabela 5.11 vemos a frequência das respostas dos integrantes das Duplas quanto a identificação dos módulos problemáticos.

Figura 5.18: Classificação do esforço para identificação de módulos problemáticos



Fonte: Elaborada pela autora.

Tabela 5.10: Respostas sobre o esforço para identificação de módulos problemáticos

Avaliação	**Quantidade de respostas**
Complexidade Baixa	04
Complexidade Moderada	05
Difícil	05
Inviável	02

Fonte: Elaborada pela autora.

Sobre os comentários e sugestões colocados pelos alunos, a Dupla 1 comentou que o experimento tornou mais clara a importância da arquitetura e ajudou a compreender melhor os detalhes do sistema. Também, recomendou a utilização do processo, pois considerou a utilização dele simples e pragmática. Essas observações, evidenciam como a união entre teoria e prática é de grande relevância, pois a compreensão da importância da arquitetura de software, dos estilos arquiteturais e do ArMoni para auxiliar a monitoração

Tabela 5.11: Frequência das respostas sobre a identificação de módulos problemáticos

Respostas	**Duplas**
Complexidade Baixa	1,1,2,3
Complexidade Moderada	2,3,4,4,5
Difícil	5,6,6,7,7
Inviável	8,8

Fonte: Elaborada pela autora.

traz uma nova visão para aluno sobre como entender o sistema, bem como reforça a contribuição que essa dissertação pretende trazer para o(a) desenvolvedor(a) que utilizar o processo proposto.

A Dupla 2 considerou que apesar da monitoração ter sido trabalhosa, foi relativamente fácil seguir as atividades propostas no ArMoni. Descreveu a atividade como simples e intuitiva e sugeriu que os dados pudessem ser migrados para alguma ferramenta que possibilite a análise dos dados. A Dupla 3 apesar de já ter tido experiências anteriores com o sistema Falibas, afirmou que aprendeu muito sobre o sistema e como poder monitorá-lo. Considerou que o processo ajudou bastante e ressaltou a necessidade de conhecer o software para realizar a monitoração.

A Dupla 4 afirmou que fazer a anotação das classes com o *framework* e a análise do *log* foram simples, mas identificar o módulo problemático exigiu maior capacidade analítica. Apesar de considerar o Falibras complexo, ressaltaram que a estruturação do mesmo em pacotes facilitou a atividade. Além disso, afirmou que o processo ajudou significativamente, onde o mesmo foi utilizado para estabelecer uma lista de tarefas. A Dupla 5 considerou o *QuAM framework* prático, mas apontou a dificuldade em analisar os dados do *log*. A Dupla 6 também considerou a análise do *log* difícil, pois havia muitos dados e não sabiam discernir quais eram mais relevantes.

A Dupla 7 notou que o *framework* gera o *log* facilmente, mas evidenciou que é difícil saber onde fazer as anotações. A Dupla conhecia o Falibras e considera que desconhecer o sistema teria tornado a atividade ainda mais difícil. Além da dificuldade em fazer as anotações, considerou difícil fazer a análise do *log*. Pelo fato de terem anotado muitas classes o *framework* gerou um volume grande de dados e a Dupla não sabia como selecionar os que eram importantes, por isso acreditaram ser quase inviável fazer essa análise, principalmente se o tempo de monitoração fosse maior que 24 horas. Ainda, sugeriram a automatização da análise do *log*. Por fim, a Dupla 8, considerou o experimento muito abstrato e que analisar o *log* não era possível. Afirmou que era complicado, com muitos dados irrelevantes e difíceis de filtrar. Consequentemente, não conseguiram identificar onde estava a falha no Falibras.

Assim, a análise dos dados, dos comentários e sugestões foi muito enriquecedora, pois

foi possível entender aspectos práticos e também subjetivos quanto ao desenvolvimento do experimento. Durante a avaliação, foram percebidos alguns ajustes necessários para o aperfeiçoamento do ArMoni. Esses ajustes foram considerados pertinentes dados alguns questionamentos realizados durante o experimento. Essas alterações no processo foram positivas e evidenciam a contribuição que a interação com os voluntários da pesquisa proporcionou para o aperfeiçoamento do ArMoni.

Capítulo 6

Conclusões e Trabalhos Futuros

Com o objetivo de propor um processo de monitoração de requisitos de qualidade dirigido por estilos arquiteturais, esta dissertação propôs um conjunto de atividades para compor o referido processo e abordou sobre as principais características de estilos arquiteturais comumente utilizados.

Com base nos objetivos definidos para o presente trabalho, foi elaborado o processo de monitoração ArMoni onde através do mesmo foram definidas diretrizes para determinar com mais critério os pontos de monitoração dentro de um sistema. A utilização do processo também contribui para direcionar a análise de *logs* gerados por ferramentas ou *frameworks* de monitoração.

A questão de pesquisa (QP) levantada para desenvolver essa pesquisa foi: Como monitorar e analisar requisitos de qualidade em um sistema de software? A partir desta foram estabelecidas subquestões para direcionar o processo de pesquisa. Sobre a QP1, foi observada a ISO/IEC 9126-1, onde a mesma proporcionou a base para determinar os requisitos de qualidade que seriam observados nos estudos analisados na revisão da literatura. Sobre a QP2, pode-se notar a escassez de estudos que tratam especificamente da monitoração de requisitos de qualidade. De modo que, teve-se como foco no estudo um *framework* de monitoração desenvolvido com base no trabalho de Silva (2015). Por último, sobre a QP3, uma solução genérica para identificar aspectos de monitoração foi proposta. Esta se trata do processo ArMoni que é composto por um conjunto de atividades que visa direcionar as atividades do(a) desenvolvedor(a) para realizar a monitoração.

Ainda, o estudo sobre os estilos arquiteturais buscou entender como esses estão diretamente ligados com os requisitos de qualidade, onde percebeu-se que os aspectos estruturais do software são fortemente relevantes para garantir tais requisitos. Conhecer a disposição dos componentes e conectores dentro do software contribui para o(a) desenvolvedor(a) de software direcionar seus esforços para definir a monitoração.

Esta dissertação também traz contribuições para o grupo de pesquisa na UFAL que trabalha com Engenharia de Software e vem desenvolvendo pesquisas voltadas para a

medição e monitoração de requisitos de qualidade.

Para avaliar o processo de monitoração proposto, denominado *ArMoni*, foi utilizado o *QuAM Framework*, Lima (2016). Com isso, foi realizado um experimento avaliativo onde foi possível que alunos do curso de Ciência da Computação - Campus Arapiraca monitorassem um sistema real, denominado *Falibras*. A aplicação prática do processo foi importante, pois permitiu perceber que o *ArMoni* pode auxiliar e guiar a monitoração. É válido salientar que o embora o *QuAM Framework* tenha sido utilizado para validar o processo ArMoni, tal processo não é dependente do referido *framework*. Desse maneira, o mesmo pode ser aplicado em outros contextos para direcionar atividades de monitoração.

Esta dissertação também contribuiu para validar o *QuaM Framework*, uma vez que esse pôde ser utilizado por alunos de Ciência da Computação para monitorar o referido sistema. Com isso, surgiram algumas observações pertinentes sobre aperfeiçoamentos para esse *framework*.

Uma das limitações dentro da pesquisa foi referente à monitoração intrusiva, contudo a mesma não se constitui um aspecto grave para analisar os requisitos de qualidade uma vez que o processo ArMoni não depende diretamente do meio de monitoração do sistema, mas apenas do *log*. Dessa maneira, o processo pode ser utilizado para sistemas que apresentem diferentes estilos arquiteturais, onde foram apresentados nesta dissertação alguns dos principais estilos.

A monitoração realizada com base no QuAM apresenta a limitação de ser apropriada apenas para sistemas implementados em Java, pois tal *framework* foi adaptado de uma ferramenta escrita na citada linguagem através de reúso de software. Todavia, o mesmo foi importante para compreender como as atividades do ArMoni podem direcionar a monitoração.

Além disso, o presente trabalho não tem o intuito de esgotar a discussão sobre o tema em questão, mas promover reflexões que permitam a realização de estudos posteriores sobre a monitoração de requisitos de qualidade. Trabalhos futuros podem ampliar os resultados observados nesse estudo. Possíveis trabalhos evolvem:

1. Aplicação do *ArMoni* em um conjunto maior de participantes a fim de corroborar a viabilidade deste processo e trazer novas contribuições sobre como os participantes avaliam tal processo e como ele contribui para monitorar outros sistemas.

2. Com base nos comentários dos alunos que participaram da pesquisa, é necessário construir uma ferramenta que apóie o processo proposto e automatize algumas atividades dele. De acordo com a opinião deles, as automações mais necessárias dizem respeito à análise e visualização dos dados, incluindo a possibilidade de filtragem automática, dado o grande volume de dados que pode ser gerado nos arquivos de *log*.

3. Executar o experimento avaliativo presente neste trabalho sem intrusão no código. Em outras palavras, monitorar as interfaces dos componentes e/ou conectores de modo a não adentrar em trechos específicos do código-fonte do Falibras.

4. Converter o *QuAM Framework* para serviços de monitoramento não intrusivo. Essa mudança contribui para que a monitoração interfira menos no funcionamento do sistema a ser monitorado.

Além disso, um aspecto diferencial do processo ArMoni é que ele pode apoiar desenvolvedores(as) que utilizam metodologias ágeis no desenvolvimento de software. Essa característica pode ser notada na simplicidade e praticidade que o processo propõe nas atividades que o compõe. Esse fato põe em evidência a atenção para a preocupação arquitetural dentro da metodologia ágil. Ademais, vale ressaltar que a aplicabilidade do ArMoni no apoio ao desenvolvimento da arquitetura, consistência e satisfação dos requisitos de qualidade e em TDD (*Test Drive Development*).

Assim, conforme o que foi discutido nesta dissertação, considera-se que o processo proposto contribui para a área de Engenharia de Software, pois visa garantir que os requisitos de qualidade desejados para o software sejam alcançados. Desse modo, tem o intuito de refletir como esses requisitos estão endereçados desde as fases iniciais de desenvolvimento de software até a fase de implantação e evolução do mesmo, onde há poucos mecanismos de garantia de que os requisitos de qualidade estão sendo efetivamente atendidos.

Referências Bibliográficas

ALEBRAHIM, A.; HATEBUR, D.; HEISEL, M. A method to derive software architectures from quality requirements. In: THU, T. D.; LEUNG, K. (Ed.). *Proceedings of the 18th Asia-Pacific Software Engineering Conference (APSEC)*. 18th Asia Pacific Software Engineering Conference (APSEC): IEEE, 2011. p. 322–330.

AMELLER, D. et al. Non-functional requirements in architectural decision making. *IEEE Software*, v. 30, n. 2, p. 61–67, March 2013.

AMELLER, D. et al. How do software architects consider non-functional requirements: An exploratory study. In: *20th IEEE International Requirements Engineering Conference (RE)*. Chicago, IL, USA: IEEE, 2012. p. 41–50.

BABAR, M. A.; GORTON, I.; JEFFERY, R. Capturing and using software architecture knowledge for architecture-based software development. In: *Fifth International Conference on Quality Software (QSIC'05)*. [S.l.: s.n.], 2005. p. 169–176.

Designing software architectures to achieve quality attribute requirements, v. 152, n. 4. IEE Proceedings - Software: IET, 2005. 153–165 p.

BAJPAI, V.; GORTHI, R. P. On non-functional requirements: A survey. In: *Electrical, Electronics and Computer Science (SCEECS), 2012 IEEE Students' Conference on*. [S.l.: s.n.], 2012. p. 1–4.

BARBACCI, M. et al. *Quality Attributes*. Pittsburgh, PA, 1995.

BASS, L.; CLEMENTS, P.; KAZMAN, R. *Software Architecture in Practice*. 2. ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003.

BASS, L.; KLEIN, M.; BACHMANN, F. Quality attribute design primitives and the attribute driven design method. In: *Revised Papers from the 4th International Workshop on Software Product-Family Engineering*. London, UK: Springer-Verlag, 2002. p. 169–186.

BOEHM, B. Architecture-based quality attribute synergies and conflicts. In: *2015 IEEE/ACM 2nd International Workshop on Software Architecture and Metrics*. [S.l.: s.n.], 2015. p. 29–34.

BRITO, P. H. d. S.; GUERRA, P. A. d. C.; RUBIRA, C. M. F. *Estudo sobre Estilos Arquiteturais para Sistemas de Software Baseados em Componentes*. Universidade Estadual de Campinas. Instituto de Computação, 2007.

BRITO, P. H. da S.; FRANCO, N.; CORADINE, L. C. Falibras: uma ferramenta flexível para promover acessibilidade de pessoas surdas. *Nuevas Ideas en Informática Educativa. TISE. Chile*, 2012.

CANESSANE, R. A.; SRINIVASAN, S. A framework for analysing the system quality. In: *2013 International Conference on Circuits, Power and Computing Technologies (ICCPCT)*. [S.l.: s.n.], 2013. p. 1111–1115.

CARRIERE, S. J.; KAZMAN, R.; WOODS, S. G. Assessing and maintaining architectural quality. In: *Proceedings of the Third European Conference on Software Maintenance and Reengineering (Cat. No. PR00090)*. [S.l.: s.n.], 1999. p. 22–30.

CHAVAN, P. U.; MURUGAN, M.; CHAVAN, P. P. A review on software architecture styles with layered robotic software architecture. In: *2015 International Conference on Computing Communication Control and Automation*. [S.l.: s.n.], 2015. p. 827–831.

CHEESMAN, J.; DANIELS, J. *UML Components: A Simple Process for Specifying Component-based Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2000.

CHEN, X. et al. A replicated experiment on architecture pattern recommendation based on quality requirements. In: *2014 IEEE 5th International Conference on Software Engineering and Service Science*. [S.l.: s.n.], 2014. p. 32–36.

CHUNG, L.; LEITE, J. C. S. d. P. On non-functional requirements in software engineering. In: BORGIDA, A. T. et al. (Ed.). *Conceptual Modeling: Foundations and Applications*. Berlin, Heidelberg: Springer-Verlag, 2009. p. 363–379.

CHUNG, L.; NIXON, B.; YU, E. Using non-functional requirements to systematically select among alternatives in architectural design. In: *Proc. 1st Int. Workshop on Architectures for Software Systems*. [S.l.: s.n.], 1994. p. 31–43.

CHUNG, L.; NIXON, B.; YU, E. Using quality requirements to sistematically develop quality software. In: . Fourth International Conference on Software Quality. McLean, VA, U.S.A: [s.n.], 1994.

CHUNG, L. et al. *Non-functional requirements in software engineering*. New York: Kluwer Academic Publishers, 2000.

CONDORI-FERNANDEZ, N.; LAGO, P. Can we know upfront how to prioritize quality requirements? In: *2015 IEEE Fifth International Workshop on Empirical Requirements Engineering (EmpiRE)*. [S.l.: s.n.], 2015. p. 33–40.

CYSNEIROS, L. M.; LEITE, J. C. S. do P. Nonfunctional requirements: from elicitation to conceptual models. *IEEE Transactions on Software Engineering*, v. 30, n. 5, p. 328–350, May 2004.

DABBAGH, M.; LEE, S. P. A consistent approach for prioritizing system quality attributes. In: *14th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)*. Honolulu, HI, USA: IEEE, 2013. p. 317–322.

DHARAM, R.; SHIVA, S. G. A framework for development of runtime monitors. In: *2012 International Conference on Computer Information Science (ICCIS)*. [S.l.: s.n.], 2012. v. 2, p. 953–957.

DWIVEDI, A. K.; RATH, S. K. Selecting and formalizing an architectural style: A comparative study. In: . Seventh International Conference on Contemporary Computing (IC3): IEEE, 2014.

FERNANDEZ, E. B.; ORTEGA-ARJONA, J. L. The secure pipes and filters pattern. *20th International Workshop on Database and Expert Systems Application*, p. 181–185, 2009.

FRANCO, R. J. *FlexMonitorWS: uma solução para monitoração de Serviços Web com foco em qualidade*. Dissertação (Mestrado) — Universidade Estadual de Campinas. Instituto de Computação., Campinas, São Paulo, Brasil, 2014.

GARLAN, D. et al. *Documenting Software Architectures: Views and Beyond*. 2nd. ed. Boston, MA, USA: Addison-Wesley Professional, 2010.

GARLAN, D.; SHAW, M. *An Introduction to Software Architecture*. Pittsburgh, PA, USA, 1994.

GAYARD, L. A.; RUBIRA, C. M. F.; GUERRA, P. A. de C. *COSMOS*: a COmponent System MOdel for Software Architectures*. Universidade Estadual de Campinas. Insituto de Computação., 2008.

GLINZ, M. On non-functional requirements. In: *15th IEEE International Requirements Engineering Conference*. New Delhi, India: IEEE, 2007. p. 21–26.

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

HEWARD, G. et al. Assessing the performance impact of service monitoring. In: *2010 21st Australian Software Engineering Conference*. [S.l.: s.n.], 2010. p. 192–201.

IQBAL, M. Nfr modeling approaches. In: *2011 First ACIS International Symposium on Software and Network Engineering*. [S.l.: s.n.], 2011. p. 109–114.

ISO/IEC. *ISO/IEC/IEEE 9000:2000. Quality Management Systems - Fundamentals and Vocabulary*. [S.l.]: ISO/IEC/IEEE, 2000.

ISO/IEC. *ISO/IEC 9126-1. Engenharia de software - Qualidade de produto*. Rio de Janeiro: ISO/IEC, 2003.

ISO/IEC. *ISO/IEC/IEEE 29148:2011 - Systems and software engineering - Life cycle processes - Requirements engineering*. [S.l.]: ISO/IEC/IEEE, 2011.

JAYATHILAKE, D. A software monitoring framework for quality verification. In: *2012 Ninth International Conference on Computer Science and Software Engineering (JCSSE)*. [S.l.: s.n.], 2012. p. 311–316.

JR., M. C. da S.; GUERRA, P. A. de C.; RUBIRA, C. M. F. A java component model for evolving software systems. In: *18th IEEE International Conference on Automated Software Engineering (ASE 2003)*. Montreal, Canada: IEEE, 2003. p. 327–330.

KAZMAN, R.; BASS, L. *Toward deriving software architectures from quality attributes*. Carnegie Mellon University. Pittsburgh, Pennsylvania, 1994.

KEEDWELL, A.; DÉNES, J. *Latin Squares and Their Applications*. [S.l.]: Elsevier Science, 2015.

KHATTER, K.; KALIA, A. Integration of non-functional requirements in model-driven architecture. In: *Fifth International Conference on Advances in Recent Technologies in Communication and Computing (ARTCom 2013)*. [S.l.: s.n.], 2013. p. 359–364.

KICZALES, G. et al. Aspect-oriented programming. In: *European Conference on Object-Oriented Programming (ECOOP)*. Finlândia: Springer-Verlag, 1997.

KIM, J. S.; GARLAN, D. Analyzing architectural styles with alloy. *Workshop on the Role of Software Architecture for Testing and Analysis*, Jul. 2006.

KIM, J. S.; GARLAN, D. Analyzing architectural styles. *The Journal of Systems and Software*, p. 1216–1235, Fev. 2010.

KLEIN, M.; KAZMAN, R. *Attribute-Based Architectural Styles*. Pittsburgh, PA, 1999.

KOSCIANSKI, A.; SOARES, M. dos S. *Qualidade de Software: aprenda as metodologias e técnicas mais modernas para o desenvolvimento de software*. 2. ed. São Paulo: Novatec Editora, 2007.

KOZIOLEK, A. Architecture-driven quality requirements prioritization. In: *First IEEE International Workshop on the Twin Peaks of Requirements and Architecture (TwinPeaks)*. [S.l.: s.n.], 2012. p. 15–19.

LADDAD, R. *AspectJ in Action*. 2. ed. USA: Manning, 2010.

LAGERSTEDT, R. Using automated tests for communicating and verifying non-functional requirements. In: *2014 IEEE 1st International Workshop on Requirements Engineering and Testing (RET)*. [S.l.: s.n.], 2014. p. 26–28.

LEE, P. A.; ANDERSON, T. *Fault Tolerance: Principles and Practice*. 2nd. ed. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1990.

LIMA, I. C. A. de. Quam framework: solução extensível para monitoramento de software orientado a aspectos e objetos. *Universidade Federal de Alagoas*, 2016.

LIU, Y. et al. An approach to integrating non-functional requirements into uml design models based on nfr-specific patterns. In: *2012 12th International Conference on Quality Software*. [S.l.: s.n.], 2012. p. 132–135.

LUNDBERG, L. et al. Quality attributes in software architecture design. In: *Proceedings Of The Iasted 3rd International Conference On Software Engineering And Applications*. [S.l.: s.n.], 1999. p. 353–362.

ME, G.; CALERO, C.; LAGO, P. Architectural patterns and quality attributes interaction. In: *2016 Qualitative Reasoning about Software Architectures (QRASA)*. [S.l.: s.n.], 2016. p. 27–36.

OUHBI, S. et al. Software quality requirements: A systematic mapping study. In: *2013 20th Asia-Pacific Software Engineering Conference (APSEC)*. [S.l.: s.n.], 2013. v. 1, p. 231–238.

PETROV, P.; BUY, U. A systemic methodology for software architecture analysis and design. In: *2011 Eighth International Conference on Information Technology: New Generations*. [S.l.: s.n.], 2011. p. 196–200.

SAADATMAND, M.; CICCHETTI, A.; SJöDIN, M. Toward model-based trade-off analysis of non-functional requirements. In: *2012 38th Euromicro Conference on Software Engineering and Advanced Applications*. [S.l.: s.n.], 2012. p. 142–149.

SHARAFI, S. M.; GHAZVINI, G. A.; EMADI, S. An analytical model for performance evaluation of software architectural styles. In: *2010 2nd International Conference on Software Technology and Engineering*. [S.l.: s.n.], 2010. v. 1, p. V1–394–V1–398.

SHAW, M.; GARLAN, D. *Software Architecture: Perspectives on an Emerging Discipline*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1996.

SILVA, A.; ALBUQUERQUE, P. P. e A.; BARROSO, J. A survey about the situation of the elicitation of non-functional requirements. In: *2016 11th Iberian Conference on Information Systems and Technologies (CISTI)*. [S.l.: s.n.], 2016. p. 1–6.

SILVA, A. A. *Monitoração de Requisitos de Qualidade Baseada na Arquitetura de Software*. Dissertação (Mestrado) — Universidade Federal de Alagoas. Instituto de Computação., Maceió, Alagoas, Brasil, 2015.

SILVA, B. R. F. S. B.; BRITO, P. H. S.; BARBOSA, A. A. Tradutor português-libras adaptado a um comunicador de mensagens instantâneas. *Nuevas Ideas en Informática Educativa. TISE. Chile.*, p. 371–378, 2015.

SILVA, I. C. L. et al. A tool for trade-off resolution on architecture-centered software development. In: *The 26th International Conference on Software Engineering and Knowledge Engineering*. Hyatt Regency, Vancouver, BC, Canada: [s.n.], 2014. p. 35–38.

SILVA, I. C. L. et al. A decision-making tool to support architectural designs based on quality attributes. In: *Proceedings of the 30th Annual ACM Symposium on Applied Computing*. Proceedings of the 30th Annual ACM Symposium on Applied Computing: ACM, 2015. p. 1457–1463.

SILVA, J. P. F. da; BRITO, P. H. da S. Otimização de desempenho e escalabilidade do sistema falibras-web com o uso de grid computing. 2015.

SOARES, L. S. et al. Patterns selection for software architecture: An approach based on quality attribute parameters. In: *Proceedings of the 9th Latin-American Conference on Pattern Languages of Programming*. New York, NY, USA: ACM, 2012. p. 10:1–10:13.

SOMMERVILLE, I. *Engenharia de software*. São Paulo: Pearson Prentice Hall, 2003.

STAA, A. V. *Programação Modular*. Rio de Janeiro: Campus, 2000.

STIGER, P.; GAMBLE, R. Blackboard systems formalized within a software architectural style. *IEEE International Conference on Systems, Man, and Cybernetics*, Out. 1997.

SUNDMARK, D.; MOLLER, A.; NOLIN, M. Monitored software components - a novel software engineering approach. In: *11th Asia-Pacific Software Engineering Conference*. [S.l.: s.n.], 2004. p. 624–631.

THANHOFER-PILISCH, J. et al. Event capture and compare for runtime monitoring of systems of systems. In: *Proceedings of the 1st International Workshop on Variability and Complexity in Software Design*. New York, NY, USA: ACM, 2016. (VACE '16), p. 1–4.

UM, T. et al. A quality attributes evaluation method for an agile approach. In: *2011 First ACIS/JNU International Conference on Computers, Networks, Systems and Industrial Engineering*. [S.l.: s.n.], 2011. p. 460–461.

WANG, J.; SONG, Y.-T.; CHUNG, L. From software architecture to design patterns: a case study of an nfr approach. In: *Sixth International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing and First ACIS International Workshop on Self-Assembling Wireless Network*. [S.l.: s.n.], 2005. p. 170–177.

WANG, Q. et al. An online monitoring approach for web service requirements. *IEEE Transactions on Services Computing*, v. 2, n. 4, p. 338–351, Oct 2009.

WANGENHEIM, C. G. von. *Utilização do GQM no Desenvolvimento de Software*. Universidade do Vale do Rio dos Sinos. Centro de Ciências Exatas e Tecnológicas. Instituto de Informática. Laboratório de Qualidade de Software., 2000.

XAVIER, J. R.; WERNER, C. M. L.; TRAVASSOS, G. H. Uma abordagem para a seleção de padrões arquiteturais baseada em características de qualidade. *XVI Simpósio Brasileiro de Engenharia de Software*, p. 52–67, 2002.

YUE, D. et al. Based on soa architecture and component software reuse architecture research. *The 2nd IEEE International Conference on Information Management and Engineering (ICIME)*, 2010.

ZAYARAZ, G.; THAMBIDURAI, P. Software architecture selection framework based on quality attributes. In: *2005 Annual IEEE India Conference - Indicon*. [S.l.: s.n.], 2005. p. 167–170.

Apêndice A

Questionário de Avaliação da Monitoração do Falibras

O presente apêndice apresenta o questionário de avaliação de monitoração do Falibras que foi aplicado com os alunos de Ciência da Computação da UFAL Campus Arapiraca que participaram do experimento avaliativo referente a utilização do ArMoni e do QuAM *Framework* para analisar a monitoração do sistema Falibras.

- **Questionário de Avaliação da Monitoração do Falibras**

  Q1. Qual o número da dupla?

  Q2. Como você classifica o esforço para anotação das classes?

  - *Complexidade baixa; eu utilizaria novamente*
  - *Complexidade moderada; eu utilizaria novamente*
  - *Difícil; não sei se é viável*
  - *Inviável para utilização prática*

  Q3. Como você classifica o esforço para avaliação do log?

  - *Complexidade baixa; eu utilizaria novamente*
  - *Complexidade moderada; eu utilizaria novamente*
  - *Difícil; não sei se é viável*
  - *Inviável para utilização prática*

  Q4. Como você classifica o esforço para identificação de módulos problemáticos?

  - *Complexidade baixa; eu utilizaria novamente*
  - *Complexidade moderada; eu utilizaria novamente*
  - *Difícil; não sei se é viável*
  - *Inviável para utilização prática*

  Comentários e Sugestões – *Espaço para os alunos opinarem sobre a atividade*