UNIVERSIDADE FEDERAL DE GOIÁS
INSTITUTO DE INFORMÁTICA

GUILHERME SILVA MARQUES

Projeto Arquitetural de Serviços de Representação e Agregação de Informações de Contexto Baseadas em Ontologias

Goiânia
2016

sistema de bibliotecas ufg

UFG

TERMO DE CIÊNCIA E DE AUTORIZAÇÃO PARA DISPONIBILIZAR AS TESES E DISSERTAÇÕES ELETRÔNICAS NA BIBLIOTECA DIGITAL DA UFG

Na qualidade de titular dos direitos de autor, autorizo a Universidade Federal de Goiás (UFG) a disponibilizar, gratuitamente, por meio da Biblioteca Digital de Teses e Dissertações (BDTD/UFG), regulamentada pela Resolução CEPEC nº 832/2007, sem ressarcimento dos direitos autorais, de acordo com a Lei nº 9610/98, o documento conforme permissões assinaladas abaixo, para fins de leitura, impressão e/ou *download*, a título de divulgação da produção científica brasileira, a partir desta data.

1. Identificação do material bibliográfico: **[X] Dissertação** **[ ] Tese**

2. Identificação da Tese ou Dissertação

Nome completo do autor: Guilherme Silva Marques

Título do trabalho: Projeto Arquitetural de Serviços de Representação e Agregação de Informações de Contexto Baseadas em Ontologias

3. Informações de acesso ao documento:

Concorda com a liberação total do documento [X] SIM [ ] NÃO[1]

Havendo concordância com a disponibilização eletrônica, torna-se imprescindível o envio do(s) arquivo(s) em formato digital PDF da tese ou dissertação.

Guilherme Silva Marques

Assinatura do (a) autor (a)

Data: 21 / 09 / 2016

[1] Neste caso o documento será embargado por até um ano a partir da data de defesa. A extensão deste prazo suscita justificativa junto à coordenação do curso. Os dados do documento não serão disponibilizados durante o período de embargo.

GUILHERME SILVA MARQUES

Projeto Arquitetural de Serviços de Representação e Agregação de Informações de Contexto Baseadas em Ontologias

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação do Instituto de Informática da Universidade Federal de Goiás, como requisito parcial para obtenção do título de Mestre em Programa de Pós-Graduação em Ciência da Computação.

Área de concentração: Ciência da Computação.

Orientador: Prof. Dr. Renato de Freitas Bulcão Neto

Goiânia
2016

Ficha de identificação da obra elaborada pelo autor, através do
Programa de Geração Automática do Sistema de Bibliotecas da UFG.

Silva Marques, Guilherme
Projeto Arquitetural de Serviços de Representação e Agregação de Informações de Contexto Baseadas em Ontologias [manuscrito] / Guilherme Silva Marques. - 2016.
100 f.

Orientador: Prof. Dr. Renato de Freitas Bulcão Neto.
Dissertação (Mestrado) - Universidade Federal de Goiás, Instituto de Informática (INF), Programa de Pós-Graduação em Ciência da Computação, Goiânia, 2016.
Bibliografia.

1. Projeto arquitetural. 2. Sistemas sensíveis a contexto. 3. Padrões de projeto. 4. Avaliação. 5. Semântica. I. Bulcão Neto, Renato de Freitas, orient. II. Título.

CDU 004

MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DE GOIÁS
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

UFG

ATA Nº 16/2016

ATA DA SESSÃO DE JULGAMENTO DA DISSERTAÇÃO DE MESTRADO DE GUILHERME SILVA MARQUES

Aos cinco dias do mês de setembro de dois mil e dezesseis, às catorze horas, na sala 150 do Instituto de Informática da Universidade Federal de Goiás, Campus Samambaia, reuniu-se a banca examinadora designada na forma regimental pela Coordenação do Curso para julgar a dissertação de mestrado intitulada **"Projeto Arquitetural de Serviços de Representação e Agregação de Informações de Contexto Baseadas em Ontologias"**, apresentada pelo aluno Guilherme Silva Marques como parte dos requisitos necessários à obtenção do grau de Mestre em Ciência da Computação, área de concentração Ciência da Computação. A banca examinadora foi presidida pelo orientador do trabalho de dissertação, Professor Doutor Renato de Freitas Bulcão Neto (INF/UFG), tendo como membros os Professores Doutores Cássio Leonardo Rodrigues (INF/UFG) e Iwens Gervásio Sene Júnior (INF/UFG). Aberta a sessão, o candidato expôs seu trabalho. Em seguida, o aluno foi arguido pelos membros da banca e:

(X) tendo demonstrado suficiência de conhecimento e capacidade de sistematização do tema de sua dissertação, a banca concluiu pela **aprovação** do candidato, sem restrições.

( ) tendo demonstrado suficiência de conhecimento e capacidade de sistematização do tema de sua dissertação, a banca concluiu pela **aprovação** do candidato, condicionado a satisfazer as exigências listadas na Folha de Modificação de Dissertação de Mestrado anexa à presente ata, no prazo máximo de 60 dias, a contar da presente data, ficando o professor-orientador responsável por atestar o cumprimento dessas exigências.

( ) não tendo demonstrado suficiência de conhecimento e capacidade de sistematização do tema de sua dissertação, a banca concluiu pela **reprovação** do candidato.

Os trabalhos foram encerrados às 17 horas. Nos termos do Regulamento Geral dos Cursos de Pós-Graduação desta Universidade, lavrou-se a presente ata que, lida e julgada conforme, segue assinada pelos membros da banca examinadora.

Prof. Dr. Renato de Freitas Bulcão Neto Renato Bulcão Neto

Prof. Dr. Cássio Leonardo Rodrigues

Prof. Dr. Iwens Gervásio Sene Júnior

Todos os direitos reservados. É proibida a reprodução total ou parcial do trabalho sem autorização da universidade, do autor e do orientador(a).

Guilherme Silva Marques

Graduou–se em Sistemas de Informação na UFG - Universidade Federal de Goiás. Durante a graduação realizou estágio no Centro de Seleção da UFG, onde hoje é Analista de Tecnologia da Informação. Atua como empreendedor e co-fundador da plataforma Meu Plano de Negócios. No Mestrado, na UFG, desenvolve trabalho teórico em avaliação de projeto arquitetural para sistemas sensíveis a contexto.

Dedico este trabalho à minha família, de modo especial a minha noiva e aos meus pais e ao meu irmão, que me ensinam a cada dia buscar forças para continuar estudando e me dedicando para alcançar meus objetivos e estar aqui.

Agradecimentos

A Deus, nosso criador.

A meus pais, Luis Carlos e Maria do Carmo, por me darem o dom da vida e estarem junto comigo em todos os momentos.

A minha noiva, Dra. Karen Cristina Mello, por me ensinar o amor e me fortificar em todos os momentos de angústia e fraqueza.

A meu irmão, Gustavo Marques, que esteve sempre ao meu lado.

A toda minha família que sempre acreditou em mim. Minhas Tias, Tios, Primos e Primas. Madrinhas e Padrinhos. Faço uma homenagem especial ao meu Tio e Padrinho de batismo Olímpio Cândido, que não está entre nós, mas que sempre me incentivou a estudar e foi (é) uma pessoa muito especial na minha vida.

Ao meu orientador, Prof. Dr. Renato Bulcão, que está ao meu lado nessa caminhada desde a graduação, passando pelo TCC e agora no Mestrado. Trabalhamos juntos, e me incentivou a todo momento chegar até aqui.

Aos amigos de pesquisa, que compartilham momentos de alegrias e angústias. Mas que sempre tiveram uma mão amiga ao precisar: Márco Vinícius Sena, Ernesto Veiga, Bruno Nunes, Guilherme Maranhão e Heyde Francielle.

Aos amigos pessoais por acreditarem em meu pontencial e me incentivaram, de alguma forma, a concluir este Mestrado: Cristiane Ribeiro, Guilherme Nogueira, João Paulo Nascimento e Márcio Vinicius Sena.

Aos colegas de sala: Prof. Dr. Iwens Sene, Bruna Michelly, Cristiane Ribeiro, David Sue, Ronaldo Santos, Jacson Rodrigues, Eduardo Humberto e Mário Pyscoia. Que sempre me fortificaram e me apoiaram nos momentos de ausência.

Aos colegas de trabalho, dos demais departamentos, que sempre compreenderam meus momentos de ausência em momentos de muito trabalho, mas que era necessário. Pessoal da recepção, pedagógicas, logística, secretaria.

Resumo

Contexto é o conjunto de informações que pode ser utilizado para caracterizar a situação de uma entidade (pessoa, lugar ou objeto relevante para a interação usuário-aplicação) [10]. Perera et al. [36] propõem um ciclo de vida para a informação de contexto composto por 4 etapas: aquisição, modelagem, raciocínio e disseminação. A etapa de modelagem de contexto consiste em representar o mundo real na computação. Segundo Bettini et. al. [2] "uma boa modelagem de informação de contexto reduz a complexidade das aplicações sensíveis a contexto". Um Sistema Sensível ao Contexto Semântico (SSCS) é um sistema computacional que utiliza ontologias que, segundo Gruber [15], é "uma especificação explícita de uma conceitualização", e consistem em uma das formas de modelar as informações de contexto. Como resultado de uma revisão sistemática feita pelo autor deste trabalho evidenciou que na literatura o foco dos trabalhos tem sido na modelagem da informação de contexto com menor enfoque no uso de projeto arquitetural. Com base nisso, o principal problema endereçado neste trabalho é a criação e avaliação dos projetos arquiteturais que trabalhem em conjunto na modelagem do ciclo de vida de contexto em SSCS. Objetiva-se elaborar uma solução arquitetural para sistemas sensíveis ao contexto semântico baseado em ontologias. Propôs-se neste trabalho o projeto arquitetural para o software de representação ontológica de contexto e para o software de agregação ontológica de contexto. O primeiro representa as informações coletadas através de sensores e o segundo agrega as informações representadas para um mesmo indivíduo ou entidade. Realizou-se, assim, a avaliação dos projetos arquiteturais propostos e explana-se sobre os resultados obtidos, comparando-os com outros projetos existentes na literatura. Com a obtenção dos resultados propostos, espera-se contribuir com a literatura, apresentando projetos arquiteturais e avaliações destes, auxiliando, assim, o desenvolvimento de SSCS no que se refere à modelagem do ciclo de vida de contexto.

Palavras–chave

Projeto arquitetural; Sistemas sensíveis a contexto; Padrões de projeto; Avaliação; Semântica; Ontologia.

Abstract

Dey et al. [10] define context as the information set that can be used to characterize the status of an entity (e.g. person, place or object relevant to the user-application interaction). Perera et al. [36] propose a context information life cycle in 4 phases: acquisition, modeling, reasoning and dissemination. Context modeling consists in representing the real world in computing and, according to Bettini et al. [2], a good information context modeling mitigates the complexity of context-aware applications. In the context of this work, a systematic review demonstrated that the literature has focused on context information modeling, but with less emphasis on the architectural project. The main problem addressed on this work is the creation and evaluation of architectural projects that work together on the context modeling phase of a Semantic Context-Aware System (SCAS), which uses ontologies as main information modeling technique. This work proposes an architectural solution to two relevant services of the ontology-based context modeling phase: the former represents context information collected through sensors, and the latter aggregates context information represented by the former for a same individual or entity. Both architectural designs were built using well-known architectural styles and design patterns, as well as were evaluated against architectural designs of similar projects. As a result, the main contributions include the architectural designs of ontology-based services for context representation and aggregation as references for the development of SCAS in terms of the context modeling phase.

Keywords

Architectural design; Context-aware systems; Design patterns; Evaluation; Semantics; Ontology.

Sumário

Lista de Figuras **10**

Lista de Tabelas **11**

1 Introdução **12**
- 1.1 Problema 14
- 1.2 Objetivos 14
- 1.3 Metodologia 14
- 1.4 Estrutura da Dissertação 16

2 Fundamentação Teórica **17**
- 2.1 Projeto Arquitetural de Software 17
- 2.2 Avaliação de Projeto Arquitetural 22
  - 2.2.1 Técnicas e Métricas 22
  - 2.2.2 Ferramentas 24
- 2.3 Hermes - Sistema de Gerenciamento de Contexto 25
- 2.4 Considerações Finais 27

3 Projeto Arquitetural para SSCS **28**
- 3.1 Componente de Representação Ontológica de Contexto (CROC) 28
  - 3.1.1 Requisitos funcionais e não funcionais 29
  - 3.1.2 Arquitetura do componente 34
- 3.2 Componente de Agregação Semântica de Contexto (CASC) 45
  - 3.2.1 Requisitos funcionais e não funcionais 45
  - 3.2.2 Arquitetura do componente 47
- 3.3 Considerações finais 53

4 Avaliação da Arquitetura do CROC/CASC **56**
- 4.1 Materiais e métodos 56
- 4.2 Resultados e discussões 57
  - 4.2.1 Experimento 57
  - 4.2.2 Discusão 58
- 4.3 Considerações Finais 61

5 Trabalhos Relacionados **63**
- 5.1 Revisão Sistemática 63
- 5.2 Resultados 64
- 5.3 Considerações Finais 65

6 Conclusões e Trabalhos Futuros **66**
6.1 Contribuições 66
6.2 Trabalhos Futuros 67
6.3 Considerações Finais 68

Referências Bibliográficas **69**

A Relatório da revisão sistemática **74**
A.1 Fase de Planejamento 75
A.2 Fase de Execução 76
A.3 Fase de Seleção 76
A.4 Fase de Extração 77
A.4.1 Dados da fase de extração 77
A.4.2 Análise da fase de extração - Respondendo às questões de pesquisa 79
A.5 Conclusão 85

B Descrição dos diagramas de sequência **87**
B.1 CROC - Diagrama de sequência 87
B.2 CASC - Diagrama de sequência 93

Lista de Figuras

1.1 Ciclo de vida de contexto tradicional [36] 13

2.1 Visão geral da arquitetura de *Hermes* [29] 26

3.1 Arquitetura em Camadas — CROC 35
3.2 Diagrama de componentes — CROC 37
3.3 Diagrama de classe — CROC 39
3.4 Diagrama de classe - com implementação — CROC 44
3.5 Arquitetura em Camadas — CASC 47
3.6 Diagrama de componentes — CASC 48
3.7 Diagrama de classe — CASC 50
3.8 Diagrama de classe — com implementação — CASC 52

A.1 Protocolo da revisão sistemática 74
A.2 Quantitativo de artigos por etapa 75
A.3 Quantitativo de artigos - extração 78
A.4 Ocorrência de padrões de projeto 83

B.1 Diagrama de sequência da coleta de dados — CROC 88
B.2 Diagrama de sequência da coleta de dados (interno) — CROC 89
B.3 Diagrama de sequência Gerenciador de Notificações — Assinatura 91
B.4 Diagrama de sequência — CASC 94

Lista de Tabelas

2.1 Comparativo de uso e benefícios - Design Patterns Criacionais 19
2.2 Comparativo de uso e benefícios — Design Patterns Estruturais 20
2.3 Comparativo de uso e benefícios — Design Patterns Comportamentais 21

3.1 Correlação entre os requisitos, camada da arquitetura e padrões de projeto 54
3.2 Correlação entre os padrões de projeto e os problemas resolvidos 55

4.1 Valores da avaliação - CROC e HW 58
4.2 Valores da avaliação - CASC e HA 60

A.1 Quantidade de artigos excluídos por critério 77
A.2 Quantidade final de artigos incluídos por base de dados 77
A.3 Quantidade de artigos excluídos por critério, após a leitura por completo 78
A.4 Quantidade final de artigos incluídos por base de dados, após a etapa de seleção 79
A.5 Estilos Arquiteturais 80
A.6 Estilos Arquiteturais - ocorrência nos artigos 81

CAPÍTULO 1

Introdução

Na década de 90, o cientista Mark Weiser [48] escreveu um trabalho em que projetava como seria a "computação para o século 21", que nomeou de computação ubíqua. Computação ubíqua é mais do que apenas poder levar os dispositivos para aeroportos, praias, etc é estar conectado em todos os lugares, de forma transparente. A computação sensível a contexto é uma subárea da computação ubíqua.

Dey et al. [10] descrevem a computação sensível a contexto como sendo a utilização de informações cotidianas do ambiente (contexto), sendo estas necessárias e suficientes para fazer prognósticos, planejar e executar ações em tempo real ou no futuro, com o minímo de interferência do usuário. Dey et al. [10] definem contexto como toda informação que pode ser utilizada para caracterizar a situação de uma entidade. Uma entidade é uma pessoa, lugar ou objeto que é considerado relevante para a interação entre um usuário e uma aplicação, incluindo o usuário e as próprias aplicações.

Um exemplo de aplicação sensível a contexto pode ser encontrado em Ongenae et. al. [34], que demonstram o uso e o desenvolvimento de uma aplicação para auxiliar enfermeiras no cuidado com pacientes internados em um hospital. Nessa aplicação, as informações de contexto e a relação entre essas informações estão em um modelo, que associa os dados dos pacientes (localização, perfil e fatores de risco) com as regras definidas nesse modelo. Com base nas inferências realizadas sobre esses dados, as enfermeiras são notificadas por meio da aplicação e das ações a serem tomadas para cada paciente.

Perera et al. [36] propõem um ciclo de vida para a informação de contexto, composto por 4 etapas: aquisição, modelagem, raciocínio e disseminação de contexto, como pode ser observado na Figura 1.1. O trabalho ora apresentado se enquadra na etapa de *modelagem de contexto*.

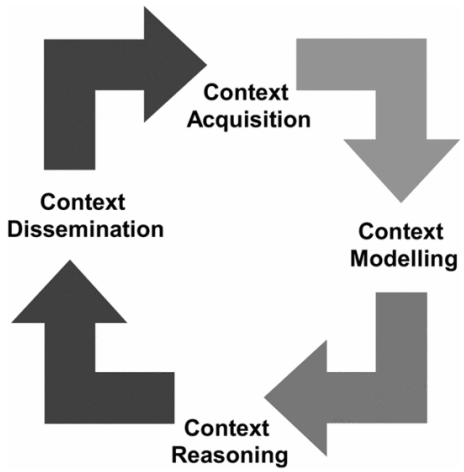


Figura 1.1: *Ciclo de vida de contexto tradicional [36]*

Para Perera et al. [36], a modelagem de contexto pode ser estática ou dinâmica. A estática é aplicada a modelos que possuem os conjuntos de informações que serão *coletados* e *armazenados* previamente definidos. Por outro lado, a modelagem dinâmica é aplicada a modelos nos quais as informações são coletadas em tempo de execução e, assim, sofrem *frequentes alterações*, sejam elas por questões temporais, mudanças de ambiente ou outras alterações de contexto. Bettini et. al. [2] afirmam que "uma boa modelagem de informação de contexto reduz a complexidade das aplicações sensíveis ao contexto".

Uma das formas mais utilizadas para modelar as informações de contexto é por meio de uma ontologia, que, segundo Gruber [15], é "uma especificação explicíta de uma conceitualização". O uso de ontologias traz benefícios quanto à interoperabilidade, reúso, estruturação, etc. Segundo Noy et. al [33], algumas das razões para usar ontologias são: compartilhar o entendimento comum das informações entre as pessoas ou *agentes de software*, realizar suposição de domínios explícitos, realizar a separação do conhecimento de domínio do conhecimento operacional e analisar o conhecimento de domínio.

Sistema Sensível ao Contexto Semântico (SSCS) é um sistema computacional que utiliza ontologias para modelar informações de contexto. Em um projeto arquitetural de SSCS, a modelagem de informações de contexto é essencial para a definição da arquitetura e comportoramento do SSCS.

1.1 Problema

O autor deste trabalho realizou uma revisão sistemática, descrita no Apêndice A, mediante o qual analisa estudos relacionados a padrões arquiteturais utilizados em SSCS, bem como levantou as características da aplicação de padrões de projeto e estilos arquiteturais de SSCS.

Uma das inferências resultantes desta revisão é que o foco dos trabalhos tem sido na modelagem da informação de contexto, semelhante ao descrito em Shams e Kamran [42] e Sánchez et al. [40], com menor enfoque no uso de projeto arquitetural.

Os projetos de SSCS são, em particular, desenvolvidos para resolver problemas de nichos específicos, dificultando assim a reutilização do sistema em situações (problemas) semelhantes. Um dos resultados obtidos na revisão sistemática é que apenas 13% dos artigos analisados tratam da reusabilidade de código e, mesmo assim, estes artigos abrangem somente o reúso de partes do sistema, como a descoberta de serviços, motor de regras, motor de coleção de dados, filtros, agregadores e outros.

Há-se uma dificuldade em obter de trabalhos da literatura o projeto e software desenvolvido para realizar-se uma análise. Com uma análise dos projetos desenvolvidos seria possível estabelecer: o que está sendo usado; quais os problemas; possíveis melhorias; e, poder comparar os projetos para discussões.

Assim, o principal problema endereçado neste trabalho é a carência de soluções arquiteturais de SSCS na etapa de modelagem. Para isso utiliza-se de soluções voltadas para reúso e consequentemente técnicas que facilitem o desenvolvimento de SSCS.

1.2 Objetivos

Este trabalho visa elaborar uma solução arquitetural para sistemas sensíveis ao contexto semântico baseado em ontologias. São objetivos específicos deste trabalho:

1. avaliar a aplicabilidade das soluções de projeto macro e micro arquiteturais existentes na literatura;
2. identificar soluções de projeto arquitetural, envolvendo estilos e padrões de projeto para SSCS; e
3. construir e avaliar uma solução de projeto arquitetural para SSCS.

1.3 Metodologia

Esta seção apresenta as atividades realizadas e a metodologia utilizada para alcançar os objetivos deste trabalho.

1. **Revisão bibliográfica**: realizada durante todo o trabalho de pesquisa. No início do processo de pesquisa, foi feita uma Revisão Sistemática, está em completo no Apêndice A. O intuíto dessa revisão sistemática foi levantar as características da aplicação de padrões de projeto e estilos arquiteturais na área de Sistemas Sensíveis ao Contexto Semântico. As informações obtidas como resultado dessa revisão sistemática serviu de embasamento para as decisões da presente pesquisa. Nela foi identificado os gargalos que esta pesquisa visa explorar.
2. **Definição de serviços necessários à proposta**: com a realização da revisão sistemática definiram-se os serviços que seriam explorados nesta pesquisa: representação ontológica e agregação semântica de informações de contexto, ambas incluídas na etapa de modelagem do ciclo de vida de contexto [36].
3. **Definição dos requisitos e projeto arquitetural dos componentes**: com o embasamento na literatura, definiram-se os requisitos e o projeto arquitetural dos componentes CROC (Componente de Representação Ontológica de Contexto) e CASC (Componente de Agregação Semântica de Contexto).
4. **Definição da abordagem e de ferramentas necessárias**: metodologias, padrões de projeto e ferramentas utilizadas neste trabalho.

    - Adota-se nesta pesquisa que as informações de contexto estão modeladas por ontologias para que sejam utilizadas nos serviços de CROC e CASC. Este tipo de modelagem é formal e padronizado, permitindo a aplicação de diversas técnicas de raciocínio e a interoperabilidade entre diferentes sistemas.
    - Utiliza-se, nesta pesquisa, padrões de projeto difundidos na literatura e estilos arquiteturais que auxiliem o desenvolvimento de aplicações SSCS.
    - Para elaboração do trabalho aqui apresentado, utilizou-se de softwares que auxiliem o desenvolvimento e representação dos sistemas aqui propostos. São ferramentas que contribuem para a explanação e validação dos projetos apresentados.

5. **Projetar e validar o serviço de representação e agregação ontológica de contexto**:

    - projetaram-se os componentes de representação e agregação ontológica de contexto com o embasamento da literatura (com os resultados obtidos da revisão sistemática),
    - realizou-se uma validação do projeto por meio de requisitos e decisões de projetos, definidos na pesquisa.

6. **Definição dos projetos utilizados para comparação**: definiu-se a infraestrutura de Hermes [29] como forma de comparar os projetos desenvolvidos com outros projetos da literatura. Hermes possui duas estruturas que possuem objetivos semelhantes

aos propostos neste projeto (CROC e CASC), são elas: *Hermes Widget* (HW) e *Hermes Aggregator* (HA). Em que HW está relacionado com a aquisição e representação de dados e HA com a agregação de dados representados.

7. **Avaliação comparativa dos projetos de CROC e CASC com projetos equivalentes**: após o desenvolvimento do CROC e CASC, buscou-se na literatura outros dois componentes equivalentes em objetivo final do software para realizar uma avaliação comparativa entre o que há na literatura e o que é proposto neste estudo.

    - utilizou-se dos projetos *Hermes Widget* (HW) e *Hermes Aggregator* (HA) para fins comparativos com os projetos ora apresentados e propostos neste trabalho.
    - definiram-se as ferramentas que auxiliariam na obtenção das métricas a serem utilizadas para avaliação.
    - obtiveram-se os códigos-fonte dos componentes externos quem seriam avaliados.
    - submeteram-se ambos os projetos às mesmas condições nas ferramentas de avaliação e, com isso, lograram-se os valores das métricas para a avaliação comparativa entre os projetos ora apresentados.

1.4 Estrutura da Dissertação

Esta dissertação contém 6 capítulos, incluindo esta introdução. O Capítulo 2 discorre sobre a Fundamentação Teórica do trabalho, que apoia os assuntos abordados. O Capítulo 3 traz a descrição do Projeto Arquitetural realizado para atingir os objetivos aqui propostos. Apresentaram-se os detalhes do projeto e as características necessárias ao alcance dos requisitos e das decisões de projeto. No Capítulo 4 expõe-se sobre a Avaliação realizada nos projetos arquiteturais propostos no Capítulo 3 e nos projetos externos utilizados como comparativos. São descritos os detalhes da avaliação, como esta foi realizada, como as métricas são obtidas e qual o objetivo de cada métrica para avaliar os projetos. O Capítulo 5 é composto pelos trabalhos relacionados que embasaram esta dissertação e que são norteadores dos projetos aqui apresentados. O Capítulo 6 consiste nas conclusões finais do trabalho, elencando as limitações do trabalho, deixando como sugestão trabalhos futuros que possam contribuir e agregar informações. No Apêndice A explana-se como foi realizada a revisão sistemática para apoiar o discorrer deste trabalho. E, por fim, no Apêndice B são exibidos os diagramas de sequência dos projetos aqui propostos, nos quais é possível observar detalhes do funcionamento dos componentes CROC e CASC.

CAPÍTULO 2

Fundamentação Teórica

Este capítulo descreve conceitos, padrões e estilos relacionados a projeto arquitetural e uma infraestrutura de gerenciamento de contexto que servem de embasamento para o desenvolvimento da proposta de elaborar uma solução arquitetural para sistemas sensíveis a contexto (componentes de representação e agregação ontológica de contexto), solução focada na modelagem do ciclo de vida do contexto. Foram considerados os pressupostos apresentados em SWEBOK v3.0 [35], que trata de estilos arquiteturais, em Gamma et. al. [13], que trata de padrões de projeto, em Maranhão et. al. [29], que trata de uma infraestrutura para apoio ao desenvolvimento de aplicações sensíveis ao contexto semântico, denominado Hermes, além de Pressman [38], que aborda a avaliação de projeto arquitetural.

2.1 Projeto Arquitetural de Software

O processo de projetar a arquitetura de software envolve o estabelecimento de um conjunto de ferramentas e decisões que identificam os principais componentes de um sistema e as comunicações entre eles [43].

Bass et al. [27] elenca três vantagens de projetar e documentar explicitamente uma arquitetura de software:

1. **Comunicação entre os interessados** — a arquitetura é uma apresentação em alto nível do sistema e pode ser usada para reforçar a discussão entre os diferentes interessados.
2. **Análise de sistema** — tomar a arquitetura do sistema explícita em um estágio inicial de desenvolvimento do sistema requer alguma análise, essa que tem efeito sobre a possibilidade de o sistema atender a requisitos críticos.
3. **Reúso em larga escala** — um modelo de arquitetura de sistema é uma descrição compacta e administrável de como um sistema está organizado e de como os componentes operam entre si. Muitas vezes, arquitetura de sistema é a mesma para

sistemas com requisitos similares e, assim, pode apoiar o reúso de software em larga escala.

SWEBOK v3.0 [35] define estilos arquiteturais como “uma especialização de elementos e tipos de relações, juntamente com um conjunto de restrições sobre a forma como eles podem ser usados”.

Os estilos arquiteturais são divididos em categorias: estruturas gerais, sistemas distribuídos, sistemas iterativos, sistemas adaptáveis e outros. O estilo em camadas, presente na categoria de estruturas gerais, é o que mais se destaca.

Padrões de projeto (ou *design patterns*, em inglês), são definidos em *SWEBOK v3.0* [35] como “uma solução comum a problemas comuns dentro de um dado contexto”. Esses padrões são agrupados em 3 categorias: criacionais, estruturais e comportamentais, os quais, segundo Gamma et al. [13] e SWEBOK v3.0 [35], totalizam 23 padrões.

A categoria de padrões criacionais relaciona os padrões que são utilizados para realizar a criação de objetos: *Abstract Factory*, *Builder*, *Factory Method*, *Prototype* e *Singleton*. Na Tabela 2.1 estão relacionados esses padrões seguindo sua usabilidade e suas vantagens.

A categoria padrões estruturais agrupam os padrões que são associados a classes e objetos: *Adapter*, *Bridge*, *Composite*, *Decorator*, *Facade*, *Flyweight* e *Proxy*. Na Tabela 2.2 visualiza-se o comparativo entre o uso e benefícios desses padrões de projeto.

E, por fim, a categoria padrões comportamentais apresenta os padrões que tratam das interações e divisões de responsabilidades entre as classes ou objetos: *Chain of Responsibility*, *Command*, *Interpreter*, *Iterator*, *Mediator*, *Memento*, *Observer*, *State*, *Strategy*, *Template Method* e *Visitor*, em que se demonstra na Tabela 2.3 o comparativo entre o uso e benefícios desses padrões de projeto.

Tabela 2.1: *Comparativo de uso e benefícios - Design Patterns Criacionais*

Design Patterns Criacionais	**Quando usa-los**	**Benefícios**
Abstract Factory	Fornecer uma interface para criação de famílias de objetos relacionados ou dependentes sem especificar suas classes concretas.	Isola as classes concretas
Builder	Separar a construção de um objeto complexo da sua representação, de modo que o mesmo processo de construção possa criar diferentes representações.	Isola o código para construção e representação
Factory Method	Definir uma interface para criar um objeto, mas deixar as subclasses decidirem que classe instanciar. O Factory Method permite adiar a instanciação para subclasses.	Fornece ganchos para subclasses Conecta hierarquias de classes paralelas
Prototype	Especificar os tipos de objetos a serem criados, usando uma instância-protótipo e criar novos objetos pela cópia desse protótipo.	Acrescenta e remove produtos em tempo de execução. Especifica novos objetos pela variação de valores. Especifica novos objetos pela variação da estrutura Reduz o número de subclasses
Singleton	Garantir que uma classe tenha somente uma instância e fornecer um ponto global de acesso para a classe	Acesso controlado à instância única Espaço de nomes reduzido Mais flexível do que operações de classe

Tabela 2.2: *Comparativo de uso e benefícios — Design Patterns Estruturais*

Design Patterns Estruturais	**Quando usá-los**	**Benefícios**
Adapter	Converter a interface de uma classe em outra interface, esperada pelos clientes. O Adapter permite que classes com interfaces incompatíveis trabalhem em conjunto — o que de outra forma, seria impossível.	Utilização de adaptadores de dois sentidos para fornecer transparência.
Bridge	Desacoplar uma abstração da sua implementação, de modo que as duas possam variar independentemente.	Desacopla a interface da implementação Melhora a extensibilidade Ocultação de detalhes de implementação dos clientes.
Composite	Compor objetos em estruturas de árvore para representarem hierarquias partes-todo. Composite permite aos clientes tratarem de maneira uniforme objetos individuais e composições de objetos.	Torna o cliente simples Torna mais fácil acrescentar novas espécies de componentes Pode tornar o objeto excessivamente genérico.
Decorator	Dinamicamente, agregar responsabilidades adicionais a um objeto. Os Decorators fornecem uma alternativa flexível ao uso de subclasses para extensão de funcionalidades.	Maior flexibilidade do que a herança estática Evita classes sobrecarregadas de características na parte superior da hierarquia.
Façade	Fornecer uma interface unificada para um conjunto de interfaces em um subsistema. Façade define uma interface de nível mais alto que torna o subsistema mais fácil de ser usado.	Isola os clientes dos componentes do subsistema Promove um acoplamento fraco entre subsistema e seus clientes.
Flyweight	Usar compartilhamento para suportar eficientemente grandes quantidades de objetos de granularidade fina.	Maior economia de armazenagem
Proxy	Fornece um substituto (surrogate) ou marcador da localização de outro objeto para controlar o acesso a esse objeto.	Diminui a contagem de referências contadas Abordagem copy-on-write, pode reduzir o custo computacional da cópia de objetos muito pesados.

Tabela 2.3: *Comparativo de uso e benefícios — Design Patterns Comportamentais*

Design Patterns Comportamentais	**Quando usa-los**	**Benefícios**
Chain of Responsability	Evitar o acoplamento do remetente de uma solicitação ao seu receptor, ao dar a mais de um objeto a oportunidade de trata a solicitação. Encadear os objetos receptores passando a solicitação ao longo da cadeia até que um objeto a trate	Acoplamento reduzido Flexibilidade adicional na atribuição de responsabilidades a objetos
Command	Encapsular uma solicitação como um objeto, desta forma permitindo parametrizar clientes com diferentes solicitações, enfileirar ou fazer o registro (log) de solicitações e suportar operações que podem ser desfeitas	Desacoplamento do objeto que invoca a operação, daquele que sabe como executá-la Montar comandos para formar comandos composto
Interpreter	Dada uma linguagem, definir uma representação para a sua gramática juntamente com um interpretador que usa a representação para interpretar sentenças dessa linguagem	É fácil de mudar e estender a gramática Implementar a gramática também é fácil
Iterator	Fornecer um meio de acessar, sequencialmente, os elementos de um objeto agregado sem expor a sua representação subjacente	Suporta variações no percurso de um agregado Iteradores simplificam a interface do agregado
Mediator	Definir um objeto que encapsula a forma como um conjunto de objetos interage. O Mediator promove o acoplamento fraco ao evitar que os objetos se refiram uns aos outros explicitamente e permite variar suas interações independentemente	Limita o uso de subclasses Desacopla colegas Simplifica o protocolo dos objetos
Memento	Sem violar o encapsulamento, capturar e externalizar um estado interno de um objeto, de maneira que o objeto possa ser restaurado para esse estado mais tarde	Preservação das fronteiras de encapsulamento Definição de interfaces mínimas e amplas
Observer	Definir uma dependência um-para-muitos entre objetos, de maneira que quando um objeto muda de estado todos os seus dependentes são notificados e atualizados automaticamente	Acoplamento abstrato entre Subject e Observer Suporte para comunicação do tipo broadcast
State	Permite a um objeto alterar seu comportamento quando o seu estado interno muda. O objeto parecerá ter mudado sua classe	Torna explícitas as transições de estado Objetos states podem ser compartilhados
Strategy	Definir uma família de algoritmos, encapsular cada uma delas e torná-las intercambiáveis. Strategy permite que o algoritmo varie independentemente dos clientes que o utilizam	Uma alternativa ao uso de subclasses Estratégias eliminam comandos condicionais da linguagem de programação
Template Method	Definir o esqueleto de um algoritmo em uma operação, postergando alguns passos para as subclasses. Template Method permite que subclasses redefinam certos passos de um algoritmo sem mudar a estrutura do mesmo	Técnica fundamental para reutilização de código Meios para fatoração dos comportamentos comuns
Visitor	Representar uma operação a ser executada nos elementos de uma estrutura de objetos. Visitor permite definir uma nova operação sem mudar as classes dos elementos sobre os quais opera	Torna fácil a adição de novas operações Um visitante reúne operações relacionadas e separa as operações não-relacionadas

2.2 Avaliação de Projeto Arquitetural

Nesta subseção, são abordadas técnicas e métricas que podem ser utilizadas para validar, verificar e analisar um projeto arquitetural de sistemas computacionais, em especial de SSCS. Em SWEBOK v3.0 [35], são apresentadas técnicas para analisar e avaliar a qualidade de um *software*. Dentre os atributos utilizados para analisar a qualidade, estão a manutenabilidade, portabilidade, testabilidade, usabilidade, correção, robustez, etc.

Há diferença entre atributos de qualidade em tempo de execução (desempenho, segurança, disponibilidade, funcionabilidade, usabilidade), não em tempo de execução (modificabilidade, portabilidade, reusabilidade, testabilidade) e qualidades intrínsecas à arquitetura (integridade conceitual, corretude, completeza).

Em SWEBOK v3.0 [35] são apresentadas, também medidas para avaliar ou quantificar os aspectos analisados de um software, como tamanho, estrutura e qualidade. Essas métricas são classificadas em duas categorias: baseadas em função (código estruturado) e baseadas em orientação a objetos. As primeiras referem-se à análise de decomposição funcional e representação geral com uso de diagramas. Já as segundas dizem respeito ao mensuramento por meio de diagramas de classe (nos quais várias medições podem ser realizadas) e medições de propriedades internas de cada classe.

2.2.1 Técnicas e Métricas

As métricas orientadas a objetos apresentadas em Pressman [38] estão listadas abaixo:

- Métricas de projeto em nível de componente
  - métricas de coesão
  - métricas de acoplamento
  - métricas de complexidade
- Métricas orientadas a operação
  - tamanho médio de operação
  - complexidade de operação
  - número médio de parâmetros por operação

Neste trabalho, são utilizadas as métricas de projeto em **nível de componente**, pois o deste estudo é a avaliação do projeto arquitetural. Não há interesse, neste momento, em avaliar o projeto em nível de classe, pois este é de alto nível de detalhamento. Sendo assim, as métricas utilizadas incluem aquelas ligadas à *coesão*, ao *acoplamento* e à *complexidade*.

Segundo Pressman [38], métricas de projeto em nível de componente estão focadas, em "*três Cs*", coesão de módulo, acoplamento e complexidade, e são consideradas uma "caixa de vidro',' pois requer conhecimento do funcionamento interno do módulo que está sendo analisado e, para tanto, é necessário que o código fonte esteja disponível. Pressman [38] afirma ainda que "é possível computar medidas da independência funcional – acoplamento e coesão – de um componente e usá-las para avaliar a qualidade de um projeto".

Pressman [38] cita itens que podem auxiliar as métricas de coesão e que são verificados a fundo em Bieman [3], que propôs o conceito de que "coesão funcional é baseada em procedimentos de saída". Cada "objeto" de saída (parâmetro de saída, variáveis globais modificadas, ou arquivos) representa um componente de um procedimento funcional. A seguir, discorre-se sobre métricas definidas em Bieman [3].

- *Program Slice* (fatias de programa) é um método de redução de código. Busca-se a redução do programa em pedaços menores para a criação de um grafo. Com o grafo, é possível "caminhar para frente" ou "caminhar para trás" a partir de uma variável de interesse e verificar quantos nós são afetados por essa variável. Faz-se então uma proporção entre os nós existentes e os nós que são de interesse.
- *Data slices* (fatias de dados) é utilizada para identificar as modificações em um programa e mensurar suas implicações. As alterações de interesse são de inclusão ou exclusão de código, ou alteração de contexto de uma variável. Alterações de operadores, por exemplo, não são consideradas modificações.
- *Slice Abstractions* é a análise de coesão funcional de Bieman [3] e é desenvolvida utilizando-se a abstração de modelos de procedimentos baseados em fatias de dados.

Há, ainda, as métricas de acoplamento eferente (Ce) e aferente (Ca), instabilidade (I) e *distância da sequência principal* (D). Além dessas, existem métricas que mensuram a falta de coesão (LCOM) e métodos ponderados por classe (WMC).

Ca — refere-se ao número de outros pacotes que dependem de classes dentro do pacote, indicando a responsabilidade deste.

Ce — consiste no número de outros pacotes dos quais as classes no pacote dependem, indicando independência do pacote.

I — trata-se da proporção de acoplamento eferente (Ce) de acoplamento total (Ce + Ce), tal que I = Ce / (Ce + Ca). Indica a capacidade de resistência do pacote em mudar. Essa métrica possui o intervalo de 0 a 1, onde 0 indica um pacote completamente estável e 1, um pacote completamente instável.

D — esta métrica é um indicador de equilíbrio do pacote entre abstração e estabilidade. O intervalo para esta métrica é de 0 a 1, com D = 0, revelando que um pacote

coincide com a sequência principal e D = 1 e que um pacote está longe da sequência principal, podendo ser completamente abstrato e estável ou totalmente concreto e instável.

LCOM — indica a falta de coesão de uma classe. Um valor elevado de LCOM geralmente aponta uma classe pouco coesa.

WMC — alude aosmétodos ponderados por classes, evidência o quão difícil foi o desenvolvimento de determinada classe em relação as demais no projeto. E, a métrica utilizada para definir o custo de desenvolvimento do projeto.

2.2.2 Ferramentas

Para obtenção dos valores necessários à avaliação, com base na definição das métricas, utilizou-se das ferramentas *JDepend* [1] e *Source Code Metrics* [2]. Com o auxílio dessas ferramentas, é possível automatizar a obtenção dos números para análise da qualidade de projetos arquiteturais.

JDepend

A ferramenta *JDepend* foi utilizada como um *plugin* na IDE de desenvolvimento *Eclipse* [3]. O objetivo dessa ferramenta é medir a qualidade de um projeto concernente a sua extensibilidade, reutilização e facilidade de manutenção. As métricas, por si só, não definem se um projeto é ruim ou bom. Geram informações para serem utilizadas por projetistas, que com elas são capazes de identificar falhas e melhorias no projeto em análise.

Com o auxílio da ferramenta, é possível automatizar a quantificação dos valores que permitem inferir a qualidade do projeto quanto aos graus de extensibilidade, reutilização e facilidade de manutenção. Essas qualidades são influenciadas pelas dependências existentes entre os pacotes da implementação.

Esse *plugin* separa os resultados em:

- *Dependencies* — são resultados gerais do código em análise. Permite visualizar globalmente o comportamento do projeto em análise comparativa.
- *Depends Upon — efferent dependencies* — são dependências que estão acima da classe ou pacote atual. Exemplo: todos dependem do *Object*.
- *Used by — afferent dependencies* — são quais classes ou pacotes usam a classe ou pacote atual.

[1] JDepend: `http://clarkware.com/software/JDepend.html`
[2] Source Code Metrics: `http://plugins.netbeans.org/plugin/42970/sourcecodemetrics`
[3] `https://eclipse.org/`

Artigos refereciam a utilização dessa ferramenta para quantificar a qualidade de suas aplicações [9, 22, 12].

Source Code Metrics

O *plugin Source Code Metrics* na IDE de desenvolvimento *NetBeans* [4] quantifica as métricas de LCOM e WMC, as quais permitem analisar a falta de coesão existente no projeto e estimar qual de suas partes gera mais custos em seu desenvolvimento.

A métrica LCOM foi criada por Chidamber et al. [8]. Posteriormente Henderson-Sellers [17] propôs uma fórmula diferente de calcular a falta de coesão, baseado-se no que já havia sido proposto.

Chidamber et al. [8] apresentam alguns pontos de vistas, divulgados em seu trabalho:

- a coesão de métodos dentro de uma classe é desejável, uma vez que promove o encapsulamento.
- a falta de coesão implica classes que podem ser divididas em duas ou mais subclasses.
- com baixa coesão, aumenta-se a complexidade, podendo elevar, assim, o risco de erros durante o processo de desenvolvimento.

Essas afirmações ajudam a compreender o que pode ser extraído de informação, quando há os valores da métrica LCOM. Com base nesses valores, pode-se verificar em que o projeto pode ser melhorado, ajudando a identificar os pontos de falha na arquitetura.

2.3 Hermes - Sistema de Gerenciamento de Contexto

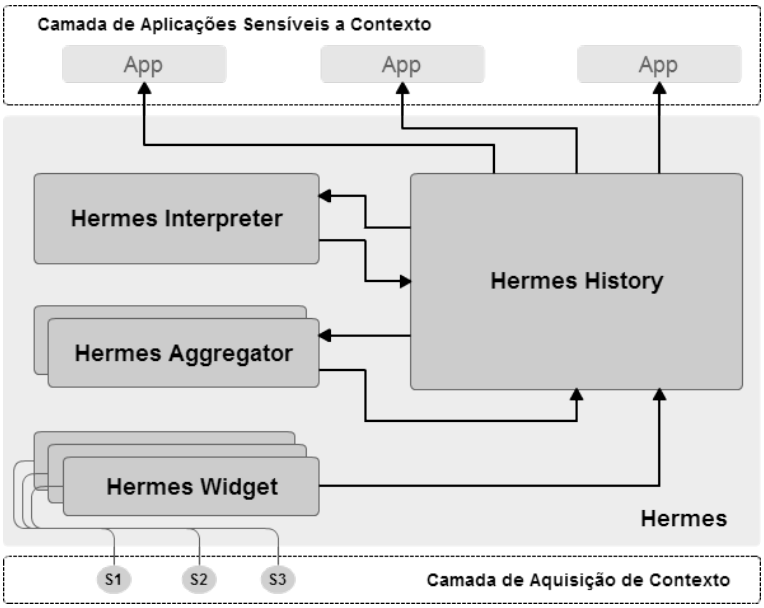
Maranhão et. al [29] demonstram o desenvolvimento da infraestrutura Hermes, que trata de sistemas sensíveis ao contexto semântico. Essa infraestrutura (Hermes) foi projetada em componentes dispostos em camadas, a fim de promover a integração e reusabilidade dos sistemas.

Dentre os componentes da infraestrutura Hermes, estão o *Hermes Widget*, que realiza a aquisição lógica dos dados; o *Hermes Aggregator*, que realiza a fusão dos dados do *Hermes Widget*; e, por fim, o *Hermes Interpreter*, que realiza a interpretação semântica das informações obtidas nos demais componentes em suas diferentes camadas. A Figura 2.1 apresenta a arquitetura da infraestrutura *Hermes*, na qual é nítida a possibilidade de utilização de vários componentes, idênticos ou não, em um conjunto, trazendo

[4] `https://netbeans.org/`

como vantagem a possibilidade de adequação do sistema diante o surgimento de novas necessidades.

Figura 2.1: *Visão geral da arquitetura de Hermes [29]*



O componente *Hermes Widget* (HW) é responsável pela abstração do serviço de representação ontológica do contexto, sendo fundamental dentro da infraestrutura de Hermes ou de qualquer outro sistema gerenciador de contexto. Seu objetivo é oferecer informações de entrada necessárias à realização dos demais serviços. Utiliza de ontologia para a representação das informações.

Dentre seus requisitos estão: permitir a especificação de um sensor e do processo de sensoriamento e representação ontológica das informações de contexto, utilizando-se da arquitetura em camadas para sua construção. As camadas do componente são: aquisição, representação, configuração, comunicação e notificação.

O componente *Hermes Aggregator* (HA) é responsável por reunir/agregar em, um único modelo, o contexto de diferentes fontes (diversos HW), relacionado com uma mesma entidade/indivíduo. Com este componente, aumenta-se a expressividade das informações a serem processadas por outros componentes da infratestrutura, como o raciocínio e a disseminação de contexto.

Os principais requisitos deste componente são: permitir a especificação de uma entidade, agregar contexto de diferentes provedores sobre uma mesma entidade e reduzir a complexidade no desenvolvimento de aplicações sensíveis a contexto. Utilizou-se da arquitetura em camadas, quais sejam: comunicação, configuração, notificação e agregação.

O componente *Hermes History* (HH) [45] é responsável por gerenciar o armazenamento e recuperação de informações. Este trabalha em conjunto com o HW e HA

estabelecendo uma interface entre eles e armazenando as informações adquiridas e representadas pelo HW e armazenando as informações agrupadas pelo HA.

Hermes irá atuar nas quatro etapas do ciclo de vida de contexto, com o objetivo de facilitar o desenvolvimento de SSCS, para isso utiliza, também, de estilos arquiteturais e de padrões de projeto nos serviços associados à fase de modelagem, que inclui os *Hermes Widgets*, *Hermes History*, *Hermes Aggregator* e outros serviços futuros.

2.4 Considerações Finais

Neste capítulo, tratou-se de projeto arquitetural de software, com o objetivo de esclarecer como o projeto dos componentes CROC e CASC foram construídos e são apresentados no Capítulo 3. Tratou-se também das vantagens e os benefícios do projeto arquitetural no desenvolvimento de sistemas.

Por definição, os padrões de projeto buscam resolver problemas comuns, com soluções comuns em um dado ambiente. Este trabalho aborda como utilizar os padrões de projeto já existentes para sistemas sensíveis ao contexto semântico.

As métricas de projeto no *nível de componente* (que são métricas não em tempo de execução), Seção 2.2, são utilizadas neste trabalho, pois o foco é a avaliação do projeto arquitetural. No atual momento, o foco não é avaliar o projeto em nível de classe, uma vez que não se deseja chegar a este nível de detalhamento. Dessa forma, as métricas utilizadas estão relacionadas à coesão, ao acoplamento e à complexidade.

Para quantificar as métricas utilizadas na avaliação do projeto arquitetural de SSCS, recorreu-se a ferramentas que automatizem a coleta das métricas. Os *plugins JDepend* e *Source Code Metrics* têm o objetivo de quantificar os valores das métricas, para que se possa analisá-los e para o projetista poder fazer uma análise comparativa.

CAPÍTULO 3

Projeto Arquitetural para SSCS

Neste capítulo, serão apresentados os resultados obtidos do projeto arquitetural de dois componentes que atuam na fase de *modelagem do ciclo de vida do contexto* [36], o Componente de Representação Ontológica de Contexto (CROC) e o Componente de Agregação Semântica de Contexto (CASC).

O primeiro trata de como as informações são colhidas de sistemas gerenciadores de sensores e como elas são representadas, e o segundo versa sobre como as informações representadas podem ser agrupadas em torno de uma entidade/indivíduo com a finalidade de gerar uma informação completa para utilização nas próximas etapas do ciclo de vida do contexto.

O presente capítulo tem por objetivo demonstrar os detalhes do projeto arquitetural para os componentes anteriormente descritos. Serão demonstrados os diagramas de classe e de componentes. Será apresentada também a correlação existente entre cada fluxo e ação do componente com seu respectivo requisito (funcional e não funcional), a fim de avaliar se todos os requisitos foram atendidos no projeto.

3.1 Componente de Representação Ontológica de Contexto (CROC)

O CROC é um componente que atua na etapa de *modelagem de contexto* do ciclo de vida do contexto, apresentado por Perera et al. [36]. A sua responsabilidade é coletar as informações que foram adquiridas por sensores. O componente aqui descrito não atua na primeira etapa do ciclo de vida do contexto (aquisição de contexto).

Os dados são adquiridos e gerenciados por um gerenciador de sensores, o qual realiza todo o interfaceamento entre a aquisição dos dados (sensores) e a aplicação. Após os dados serem adquiridos, eles precisam ser coletados e modelados. O CROC não realiza a aquisição do contexto, por não realizar tarefas como limpeza dos dados, gerenciamento de sensores ativos e outras tarefas inerentes a um gerenciador de sensores. O CROC se preocupa com a coleta dos dados, para realizar a modelagem destes, que, para isso, utiliza

uma ontologia como modelo. Com os dados modelados, é possível disponibiliza-los para serem utilizados em outros componentes, como o componente de agregação semântica de contexto 3.2.

Ontologia é uma das formas de realizar a modelagem de informações de contexto. Segundo Gruber [15], ontologia é "uma especificação explicíta de uma conceitualização". De acordo com Noy et al. [33], as razões para uso de ontologia são: compartilhar o entendimento comum das informações entre as pessoas ou *agentes de software*, realizar suposição de domínios explícitos, separação do conhecimento de domínio do conhecimento operacional e analisar o conhecimento de domínio. O que nos traz como benefícios à interoperabilidade, reúso e estruturação. Segundo Bettini [2], "uma boa modelagem de informação de contexto reduz a complexidade das aplicações sensíveis a contexto".

Para delimitar as funcionalidades do Componente CROC, o mesmo não realiza determinadas ações e há limitações, como as descritas:

- não adquire dados/informações diretamente de sensores;
- implementa um padrão de intercâmbio de informações;
- é independente de domínio de aplicação; (o CROC que coleta informação de temperatura ambiente em uma UTI é o mesmo para colher a temperatura ambiente em uma residência);
- é especifica para um determinado tipo de informação de contexto; (CROC para temperatura corpórea; CROC para temperatura ambiente).

3.1.1 Requisitos funcionais e não funcionais

Os utilizadores do componente CROC são identificados como os demais componentes de uma infraestrutura de software que abrange as 4 etapas do ciclo de vida de contexto, quais sejam: aquisição, modelagem, raciocínio e disseminação de contexto [36]. Para atender a esses utilizadores, são descritos os requisitos funcionais e não funcionais.

Neste estudo, para a criação do projeto arquitetural do CROC, foram definidos os requisitos funcionais e não funcionais, os quais são descritos neste trabalho. Os requisitos, em sua maioria, são mencionados por Dey et al. [10] e Perera et al. [36]. Outros autores também citam tais requisitos que serão mencionados em suas respectivas propostas, requisitos funcionais e não funcionais.

1. **Coleta de informações** — Segundo Dey et al. [10] os "*widgets* são usados para realizar a separação entre como o contexto é adquirido e como o contexto é usado". Dessa forma, as informações são adquiridas por um componente externo que realiza o gerenciamento dos sensores, e o componente de representação colhe as informações que foram adquiridas por meio dos sensores. Então, faz-se necessária

uma funcionalidade para colher as informações a serem modeladas. Este requisito é citado por Medaglia [32] e Khan et al. [23], ambos ratificam a necessidade da coleção de dados.

2. **Representação ontológica do modelo de contexto** — O componente deve ser capaz de receber como modelo de contexto (modelo do mundo real) uma ontologia [15], que descreve como o mundo real pode ser interpretado por um componente de software, em computação sensível a contexto. Essa funcionalidade irá representar as informações colhidas, pela da ontologia (que contém as regras e representações). Com essa funcionalidade, o componente é capaz de prover informações de alto nível, com uma maior expressividade semântica do que os dados brutos coletados pelos sensores.
3. **Especificação de contexto** — o componente deve ser apto a receber como configuração uma ontologia que especifique o contexto a ser utilizado. Dessa forma, as informações adquiridas na fase de aquisição de contexto são enviadas para o CROC a fim de que sejam modeladas de acordo com seu nicho específico. O objetivo desse requisito é que o sistema possa ser capaz de modelar as informações de forma específica ao contexto para o qual está configurado. Isso implica redução da complexidade de implementação e implantação, facilita o reúso de software e aumenta a manutenibilidade do componente.
4. **Separação de interesses e manipulação de contexto** — O componente deve ser desenvolvido de forma que seja eficaz para separar as informações que foram colhidas e as informações que foram modeladas de acordo com a ontologia. Desse modo, as informações geradas não serão genéricas, mas sim específicas do ambiente em que se encontram.
5. **Comunicação distribuída e transparente** — O componente deve ser eficiente de realizar uma comunicação entre os demais componentes integrantes do ciclo de vida, e outros sistemas de software, de forma transparente ao usuário, sendo assim, não haverá preocupação em direcionar ou realizar alguma intervenção no componente para que as informações sejam intercambiadas. Para isso, deve-se definir um padrão para o intercâmbio das informações (específico de cada implementação). Segundo Jaroucheh [21], o protocolo de comunicação deve estar adequado às necessidades da aplicação e prover diferentes plataformas de comunicação. Em Celikkan [6] há a preocupação de um componente de comunicação.
6. **Armazenamento e histórico de contexto** — Para Perera et al. [36], componente deve ser adequado a armazenar todas informações que o mesmo realizar sua representação. As informações armazenadas devem ser de fácil acesso (entende-se como de fácil acesso aquela informação que pode ser recuperada a qualquer momento). O componente deve ser capacitado a permitir que sejam realizadas

consultas em sua base de dados [10], com o objetivo de recuperar informações históricas (ou a informação mais recente). Então, essa funcionalidade é responsável por guardar as informações colhidas e oferecer um serviço de recuperação de informações. Esse requisito confirma-se seu uso em Gaultier et al. [14] e Jaroucheh et al. [21].

7. **Notificação** — O componente deve oferecer uma interface para a assinatura e o cancelamento de tópicos, o que fornecerá as informações solicitadas a um usuário ou sistema de software [10]. Para receber estas informações, é necessário ter uma chave de acesso, que será gerada pelo mantenedor do sistema e informará quais as informações e o período dos quais está chave recuperará as informações, para garantir a segurança, privacidade e integridade dos dados manipulados. Celikkan [6] e Hong et al. [19] sugerem a necessidade de um serviço de notificação em sistemas sensíveis ao contexto, a fim de informar os usuários e outros sistemas de ações ocorridas.
8. **Prover informações internas(configuração)** — De acordo com Dey et al. [10], essa funcionalidade tem como propriedades o envio de informações do próprio componente para quem as requisita. São informações de configuração que podem ser publicas, como: nome/descrição da ontologia e tipo de notificações. Do mesmo modo, essa funcionalidade provê informações internas, que são utilizadas pelos demais componentes, como: caminho da ontologia, informações de configuração do histórico e especificidades da notificação. Deve também prover facilidade em seu uso, com parâmetros de alto nível, segundo Gaultier et al. [14].
9. **Interoperabilidade** — Esse componente deve estar apto a realizar as intercomunicações, interna e externa, de forma transparente e padronizada, com o intuito de poder utilizar-se de um mesmo componente em um ou mais ambientes de software, pois as informações produzidas por este componente serão facilmente intercambiadas. Celikkan [6] cita que esse requisito é listado como um dos primeiros quando se trata de aplicações inteligentes. Assim o objetivo desse requisito não funcional é garantir que, ao progredir com o sistema e acoplar novos componentes, seja utilizado um padrão para intercâmbio de informações, e dessa forma não será necessário produzir especificidades para cada conexão com outras partes do sistema global. O componente é um receptor de dados, em grande parte, brutos, que distribui dados detalhados e com mais informação aos demais componentes do sistema.
10. **Expressividade** — Este componente deve ser adequado a suportar fortes modelagens de contexto, a fim de prover uma alta expressividade nas informações colhidas. Para tanto, o recomendado é a utilização de ontologias para garantir a expressividade nas informações adquiridas pelo componente. Ontologias possuem vantagens como fortes validações, interoperabilidade e expressividade Gruber [15]. O uso de

ontologias é visto também em Khan et al. [23, 24].

11. **Extensibilidade** — O componente deve ser arquitetado de forma a assegurar o máximo de extensibilidade. Isto irá garantir que as próximas alterações e atualizações a serem implementadas causem os menores efeitos possíveis dentro da arquitetura do componente. Para Celikkan et al. [6], este é um dos princípios de projeto que uma plataforma deve seguir. O sistema deve ser extensível quanto à capacidade de receber novas informações de contexto e quanto a transferir as informações para outros sistemas de software requisitantes, bem como a receber novas formas de modelagem do contexto. Esse requisito implica em manutenibilidade, pois um software extensível também deve prover manutenção facilitada.
12. **Reusabilidade** Como demonstrado na Revisão Sistemática [30], a reutilização de componentes foi pouco referenciada. O propósito é que a reusabilidade seja um componente totalmente reusável, haja vista que esse componente é usado por diversas vezes em um mesmo sistema e pode ser utilizado, da mesma forma, em sistemas diferentes, pois a informação a ser coletada é a mesma. Dessa forma é possível a reutilização do componente em outros sistemas de software que necessitem realizar o mesmo tipo de coleta. Faz-se, então, a necessária construção de um componente que seja completamente reusável. [28]
13. **Flexibilidade** — A representação ontológica de contexto, por se tratar de um componente ligado à fase de modelagem do ciclo de vida de contexto, deve ser flexível quanto à variação desse contexto. Isso é assim entendido: para alterar o contexto e necessário apenas a alteração da ontologia nas configurações, pois o contexto pode sofrer mudanças com frequência e, sendo assim, a forma de representação também sofrerá alterações, ou seja, a ontologia utilizada poderá sofrer atualizações de acordo com a evolução natural do contexto. [7]
14. **Manutenibilidade** — O CROC deve ser manutenível, em decorrência de ser um componente que poderá ser modificado com frequência, o que implica em melhorias na manutenabilidade para promover o reúso [28]. Por estar ligado diretamente com as informações do contexto onde são adquiridas, essas sofrem frequentes alterações, seja na forma como são adquiridas ou na forma como são repassadas ao CROC. Essas alterações ocorrem devido ao fato da busca constante por formas melhores e precisas nas informações que se deseja modelar. Este requisito deve seguir os preceitos definidos na ISO/IEC 9126, (que define as características de qualidade que todo software deve ter).
15. **Escalabilidade** — Este componente deve ser capaz de suportar o aumento do número de informações que são neles trafegadas. As informações que são recebidas pelo componente podem sofrer um aumento rápido e grande, em decorrência das ações realizadas no ambiente que influencia diretamente no processamento

[21] [19] [7]. Desta forma, o componente deve estar preparado para receber essa quantidade de informações e desenvolve as ações necessárias, como: a) limitar a quantidade de processamento; b) adequar a sua capacidade as informações que são processadas; e c) emitir alerta para que seja adicionado mais recursos.

16. **Suporte a ambientes heterogêneos** — Este componente deve ser capaz de suportar sua implantação em ambientes heterogêneos de software. Entende-se isso, como: a) heterogeneidade quanto à plataforma; e b) armazenamento em diferentes bases de dados. Essas definições podem estar contidas no módulo de configuração do componente, haja vista que há muitos sistemas em desenvolvimento e com essa diversidade eles devem suportar essa heterogeneidade [49].
17. **Suporte a implementações alternativas** — Este requisito não funcional diz respeito a capacidade de o componente poder ser implementado ou implantando em diferentes linguagens de codificação Dey et al. [10]. Sendo assim, o componente deve facilitar a integração da arquitetura dentro de aplicações existentes e facilitar a manutenção em seus mais diversos ambientes.
18. **Interface comum de consulta** — O componente deve oferecer uma interface de consulta que seja simples e objetiva [16], com o intuito de prover informações públicas a componentes externos e a informações internas. Com isso, os componentes externos obterão as informações necessárias para estar cientes de quais informações poderão consumir esse componente.
19. **Disponibilidade Contínua e constante de modelagem de contexto** — O componente deve ter a capacidade de realizar a aquisição de novos dados e informações de contexto, de forma constante e sem intervenção do usuário e, ao mesmo tempo, realizar o processamento das informações que são colhidas.
20. **Segurança** — E o componente deve seguir três pilares de segurança, apresentados em Makris et al. [28], que são: integridade, avaliabilidade e confidencialidade. Integridade faz referência aos componentes físicos que integram o software. Avaliabilidade alude aos acessos que são permitidos e executados quando a entidade é legítima. Confidencialidade é a proteção das informações, que não podem ser acessadas por pessoas/entidades não autorizadas a recuperar este tipo de informação. Dessa forma, a segurança deve ser implementada nas camadas de coleta de informações, serviço de notificação e recuperação de informações. Para atender a este requisito, é recomendada a implantação da segurança no componente central de comunicação. Segundo Khan [23] é prudente implantar e garantir a segurança na comunicação.
21. **Privacidade** — Este componente deve prover a privacidade no que tange a seus dados, tanto na comunicação quanto no armazenamento. Para atender a esse requisito, é necessário implantar diretivas quanto ao acesso e recuperação das informações,

todas em identificação e devidas permissões ao usuário. Em atendimento a esse requisito, deve-se implementar um módulo de privacidade nas camadas de coleta e a recuperação de informações [21, 19].

Com os requisitos aqui citados, o componente é capaz de atender aos princípios de: extensibilidade, manutenibilidade, escalabilidade, interoperabilidade e flexibilidade, princípios que corroboram para seu desenvolvimento. Do ponto de vista arquitetural, é necessário ser desenvolvido para atender a esses requisitos de código que possibilite o desenvolvimento de novos CROCs. Dessa forma é preciso que seja um código que permita de forma simples a adição de novos componentes; seja fácil de manter, pois, tratando-se de contexto, as informações podem mudar com frequência; é necessário que seja flexível para permitir que novas linguagens, sistemas e arquiteturas interajam com a aplicação; desenvolvimento em camadas e com componentes independentes mas que facilitem a intercomunicação.

3.1.2 Arquitetura do componente

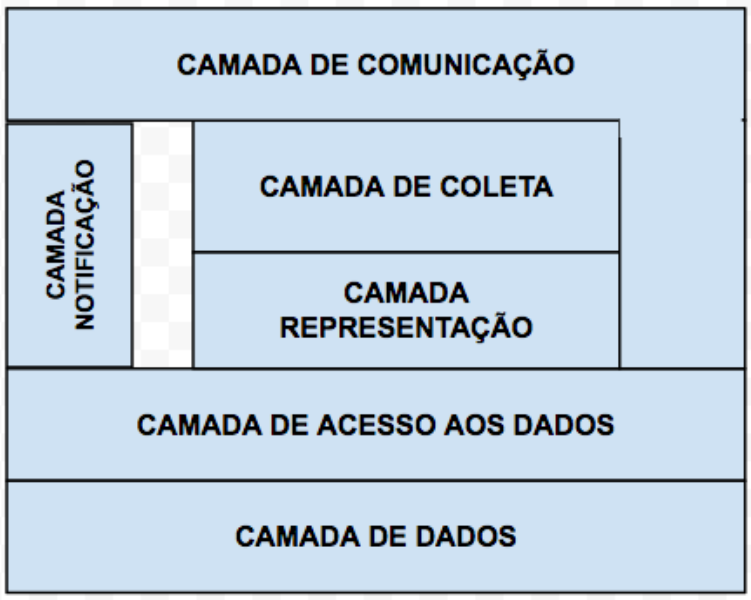
Realizou-se uma Revisão Sistemática, disponível no Apêndice A [30], com o objetivo de levantar as características da aplicação de padrões de projeto e estilos arquiteturais na área de sistemas sensíveis ao contexto semântico. Como fruto desse trabalho, obteveram-se resultados que permitiram guiar esta dissertação. Um dos resultados foi a utilização da arquitetura em camadas para o desenvolvimento dos componentes aqui descritos (3.1 e 3.2).

Arquitetura em camadas apresenta-se na Figura 3.1, onde é possível visualizar, graficamente, como estão dispostas as camadas que compõem o componente CROC, sendo essa é composta por: camada de comunicação, coleta de dados, representação de contexto, acesso ao dados, dados e notificação. A seguir será explanada a funcionalidade de cada uma.

A primeira camada a de comunicação, é responsável por gerenciar todas as comunicações e instanciar os serviços oferecidos e realizados pelo componente, que comunica com agentes internos e externos. Oferece o serviço de configuração e inicia os serviços de coleta de dados, representação de contexto e histórico. Essa camada relacionada com os requisitos funcionais de comunicação distribuída e transparente e provê informações internas (configuração).

A camada de coleta realiza a coleta dos dados inerentes ao componente. É responsável por reunir os dados que são adquiridos por um gerenciador de sensores e fornecer esses dados para a próxima camada. Provê informações para a camada de representação de contexto e consome os serviços oferecidos pela camada de comunicação

Figura 3.1: *Arquitetura em Camadas — CROC*



(serviço de configuração e é instanciada pela camada de comunicação). Está relacionada com o requisito funcional de coleta de dados.

A camada de representação é responsável por realizar a representação ontológica dos dados coletados, tendo como base uma ontologia específica do domínio em que se trata. É nessa camada que os dados coletados são transformados em dados usáveis. Os dados usáveis serão utilizados por todos os demais componentes que integram a infraestrutura de software para gerar informações semânticas. Essa camada é instanciada pela camada de comunicação e recebe informações da camada de coleta de dados, gerando assim, insumos para a camada de acesso aos dados. Foi concebida com bae nos seguintes requisitos: representação ontológica do modelo de contexto; especificação de contexto; e separação de interesses e manipulação de contexto.

A camada de acesso aos dados é uma camada simbólica, faz a ligação entre os dados representados, camada superior, com a camada imediatamente abaixo, camada de dados. O objetivo dessa camada é estabelecer uma interface de abstração do armazenamento dos dados. Sendo assim, quando for necessário realizar modificações na forma de armazenamento dos dados, não haverá alterações na forma como os dados são tratados até chegarem à camada de armazenamento, camada de dados.

A camada de dados realiza o armazenamento dos dados que foram representados. Esse armazenamento pode ocorrer de diversas formas, dependendo da especificidade e da implementação de cada projeto. A camada de dados é instanciada pela camada de comunicação, e recebe os dados advindos da camada de representação de contexto por meio da camada simbólica de acesso aos dados. É responsável por prover dados a camada de notificação. Sua composição surgiu dos requisitos: armazenamento e histórico

de contexto; e comunicação distribuída e transparente.

A camada de notificação tem a função de obter dados armazenados e enviar uma notificação a quem a solicitou. É responsável também por gerenciar as assinaturas para publicação das informações. Dessa maneira, a camada de notificação recebe um informativo da camada de dados, informando que há um novo dado. Logo após, uma verificação é realizada, a fim de obter a lista de assinantes que desejam receber essa informação em forma de notificação. A informação é então enviada aos que compõem a lista de assinantes, processo este realizado por meio da camada de comunicação. A camada de notificação é uma camada que possui ligação com a camada de acesso aos dados (consome dados) e com a camada de comunicação (utiliza o serviço para notificar os assinantes).

Diagrama de componentes

Usa-se o diagrama de componentes para apresentar como as classes estão organizadas em um software [4], indicando em cada componente quais classes estão contidas, facilitando assim o entendimento do software. Os componentes aqui representados correspondem a cada camada da arquitetura do sistema apresentado na Figura 3.1.

Esse diagrama é utilizado para exemplificar ações de como as classes estão arquitetadas e organizadas, apresentando uma visão geral do funcionamento do software, a fim de que, quando apresentados os diagramas de classes, seja de fácil visualização a responsabilidade de cada um.

Na Figura 3.2, são representados os componentes inerentes ao CROC, onde suas funcionalidades estão projetadas de acordo com as camadas da arquitetura, que agrupam os requisitos funcionais que direcionaram a criação de cada componente. Com isso objetiva-se ter um projeto modular e com maior reusabilidade.

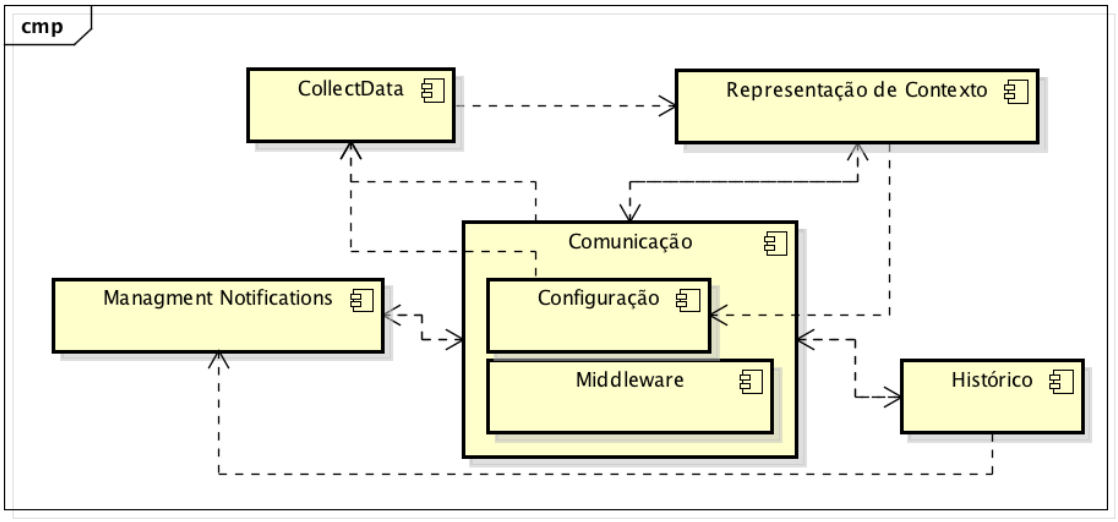
No diagrama, Figura 3.2, primeiramente é possível visualizar como os dados são adquiridos pelos sensores (por intermédio do *middleware*), que está contido no componente de comunicação. A configuração também está contida no componente de comunicação.

O componente de coleta de dados é instanciado por meio da comunicação. Dessa maneira, já recebe as informações de configuração necessárias para coletar os dados adquiridos pelo *middleware*.

A próxima ação é a representação de contexto, que tem ligação com a comunicação (iniciar o serviço e enviar dados para o histórico) e com a configuração.

O histórico por meio da comunicação recebe os dados representados e envia para a notificação. Na arquitetura do sistema, esse componente está representado pelas camadas de acesso aos dados e de dados.

Figura 3.2: *Diagrama de componentes — CROC*



A notificação realiza a disseminação das informações, para isso é instanciada mediante o componente de histórico. Para que a informação possa ser enviada (publicada), é utilizado o componente central de comunicação para realizar tais procedimentos.

A decisão de projeto de criar esses componentes originou-se nos requisitos do sistema. A separação da coleta e da representação deve-se às ações independentes que as mesmas devem executar e atender ao requisito não funcional de reusabilidade (RNF4).

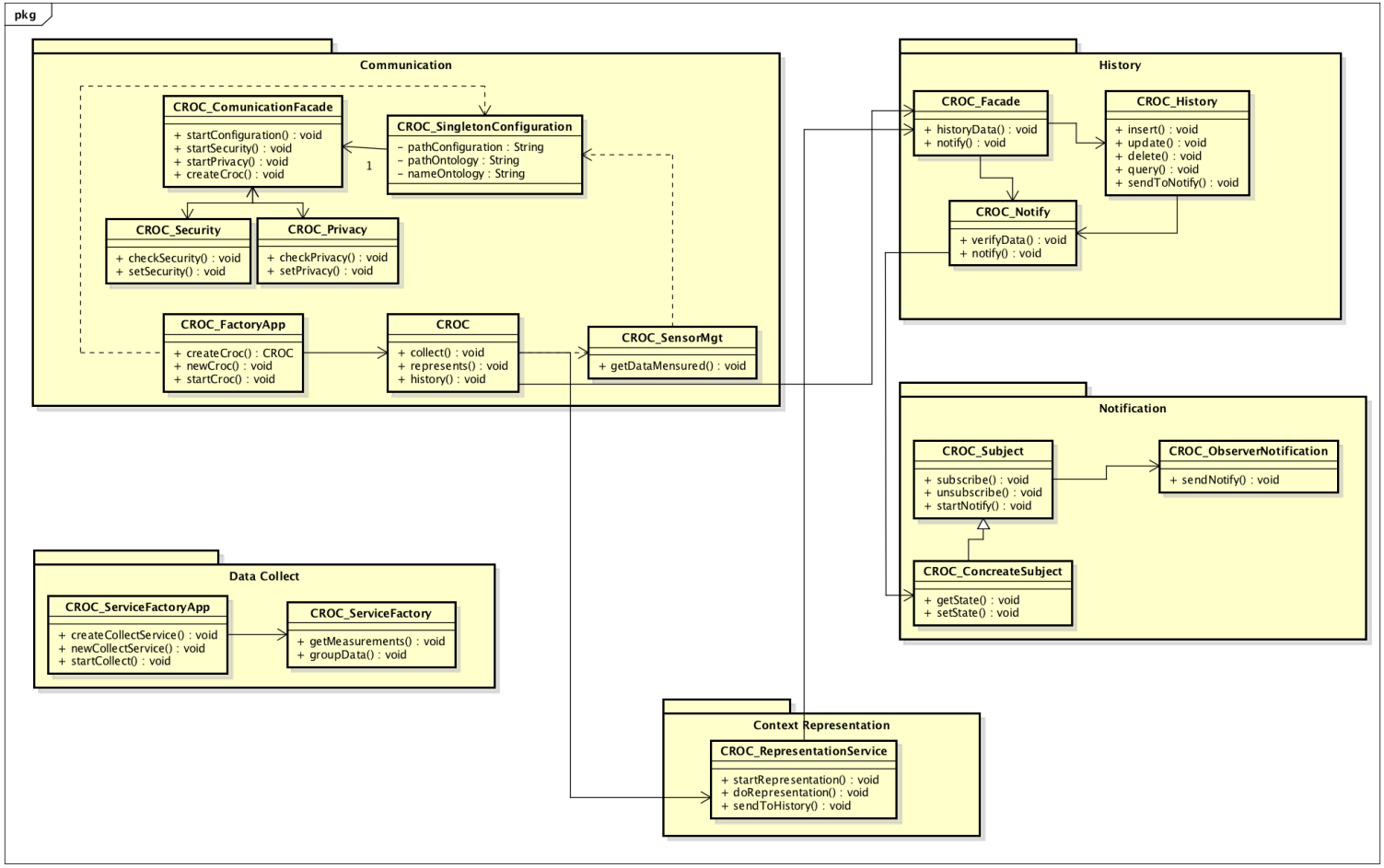
A comunicação é um componente central dentro da aplicação, devido ao conjunto de requisitos de manter um componente que gerencie toda a parte de comunicação distribuída e transparente (RF5) e que atenda aos requisitos de privacidade (RNF13) como também de segurança (RNF12). Como é apresentado no diagrama de componentes, boa parte das ações são originárias da comunicação.

Histórico (RF6) e notificação (RF7) são componentes independentes, fazem parte da aplicação e são gerenciados por ações transpassadas pelo componente de comunicação. Isso ocorre para atender aos requisitos de RNF12 e RN13. Para que uma informação possa ser armazenada ou enviada a assinantes, a aplicação deve garantir que a comunicação esteja segura, e que as informações em trânsito sejam privativas de quem as recebem, evitando que entidades não autorizadas recebam informações confidenciais.

Diagrama de classes

O CROC está divido em 5 componentes, que possuem suas funções e com atividades bem claras e definidas. São eles: comunicação, coleta de dados, representação de contexto, histórico e notificação. A seguir, apresenta-se a função de cada um deles e, na Figura 3.3, estão representadas, graficamente, as classes do componente e como estas estão relacionadas.

Figura 3.3: *Diagrama de classe — CROC*



- **Comunicação** é um componente que agrega outros três: *access management*, *security* e *privacy*. Devido à criticidade do sistema, esses três componentes são inerentes ao componente de comunicação por este ser central e gerenciar as interligações. Dessa forma, tomou-se a decisão de projeto de utilizar um componente central de comunicação que realizasse todo o controle de comunicação (interna e externa), a fim de garantir a segurança necessária dos dados e das informações transitantes. Garantir a privacidade necessária ao sistema, pois os dados são de uso pessoal, realizar o controle de quais solicitações podem ser atendidas, evita que usuários (entidades) não autorizados tenham acesso a qualquer tipo de informação. Sendo assim, o componente de comunicação é um ponto crítico e central na arquitetura desse componente, pois recebe solicitações dos demais.
- **Configuração** (contido no componente de comunicação) possui todas as informações necessárias para a definição dos parâmetros e dados de configuração a serem utilizados em todos os demais componentes. É neste módulo que se definem informações, como: o caminho do arquivo de ontologia a ser utilizado; parâmetros de conexão com banco de dados; dados de acesso ao sistema.
- **Coleta de dados** é responsável pelo agrupamento (coleta) das informações advindas do Gerenciado de Sensores (não tratado nesse componente), para que essas informações possam ser utilizadas pelo componente de Representação de Contexto.
- **Representação de contexto** é responsável por estabelecer um elo de comunicação com o componente de comunicação para obter acesso ao componente de Configuração e verificar qual ontologia e quais parâmetros de configuração serão utilizados para a requisição atual, a fim de que a informação seja representada conforme o estabelecido na ontologia. Após representada, a informação será armazenada. O responsável por armazenar as informações é o componente Histórico.
- **Histórico** é composto por dois módulos: *Storage* e *Query*. *Storage* é responsável por realizar o armazenamento das informações depois de representadas no componente de Representação de Contexto. *Query* é responsável por realizar as consultas na base de dados e retornar ao solicitante o resultado dessas consultas para utilização, como, por exemplo, no componente de *Management Notifications*.
- **Notificação** realiza o gerenciamento das solicitações de notificação (assinatura) e a entrega das informações. Quando um novo usuário solicita ser notificado, é esse componente que possui o papel de armazenar e verificar as solicitações, bem como manter a validade dos dados cadastrados. Quando alguma entidade, já cadastrada, solicita ser notificada de tempos em tempos, esse componente executa rotinas periódicas com a finalidade de atender aos parâmetros cadastrados. A entrega da informação para um agente interno ou externo ao componente passará, necessariamente, pelo componente de Comunicação, a fim de que este possa manter

a privacidade das informações que estão sendo entregues e garantir que essa informação seja entregue com segurança ao requisitante.

Decisões de projeto

Projetar software orientado a objetos é difícil, mas projetar software *reutilizável* orientado a objetos é ainda mais complicado, segundo Gamma [13]. Esses dizeres exemplificam um dos motivos da utilização e do uso de padrões de projeto (DP) para sistemas sensíveis a contexto semântico. Os DP tem por objetivo usar de experiências passadas para criar padrões que resolvam problemas parecidos de uma mesma maneira.

Para a elaboração do projeto arquitetural de sistemas sensíveis a contexto semântico procurou-se usar o máximo de padrões de projetos possíveis, a fim de tornar um projeto com características de reusabilidade e de fácil entendimento para implementações futuras. Além disso utilizou-se a arquitetura em camadas para nortear o desenvolvimento do componente.

Arquitetura em camadas como demonstrado na Revisão Sistemática de [30] e [46] é a arquitetura de software mais utilizada nos softwares desse contexto. Sendo assim a recomendação é que se siga esse modelo arquitetural para o desenvolvimento do componente. Dessa forma, o componente terá uma melhor separação de código, as alterações realizadas em determinadas camadas não irão afetar as demais e facilitarão os testes e o reuso. Uma possível desvantagem está em ter uma maior complexidade e tempo no desenvolvimento, contudo quando, se trata de software complexo e de grande porte, essa desvantagem não se torna um empecilho Sommerville [43], que com o uso dos padrões de projeto procura-se mitigar essas desvantagens.

No componente CROC utilizou-se os padrões de projeto *Factory Method*, *Singleton*, *Facade*, *Template Method* e *Observer*, oriundos de Gamma [13]. Esses padrões em conjunto têm por objetivo beneficiar o projeto quanto a questões de manutenibilidade, reúso e facilidade de implementação.

O padrão de projeto *Factory Method* tem por intenção definir uma interface para criar um objeto, mas deixar as subclasses decidirem que classe instanciar. O *Factory Method* permite adiar a instanciação para subclasses.

O *Factory Method* é utilizado nos componentes de comunicação e coleta de dados, pois há um padrão de criação da ação a ser realizada.

No componente de comunicação, é criado um método que irá definir qual o tipo do CROC que está sendo implementado. O projeto é o mesmo independente do tipo de CROC a ser criado, seja de temperatura, pressão sanguínea ou de umidade. Então utilizou-se desse padrão de projeto, para que na criação de um novo CROC a alteração de projeto ocorresse apenas nas classes de criação concreta do objeto, não sendo necessária a intervenção nas demais classes que compõem o componente.

Os métodos que implementam as ações do componente serão as mesma, independentemente do tipo de componente criado. Ações como criação, início, coleta e representação do CROC serão sempre as mesmas. Esse é o mesmo motivo da utilização desse padrão de projeto no componente de coleta de dados.

No componente de coleta de dados, há a necessidade de alteração nas classes que criam o serviço de coleta concreto e no tipo de serviço de coleta a ser executado, pois a forma como a coleta é realizada e a forma como os dados são adquiridos é a mesma independentemente do tipo de CROC implementado.

A implementação desse padrão de projeto nos componentes de comunicação e de coleta de dados pode ser visualizado no retângulo tracejado da Figura 3.3, em seus respectivos pacotes.

O padrão de projeto *Singleton* tem a intenção de garantir que uma classe tenha somente uma instância e lhe forneça um ponto global de acesso.

O *Singleton* está sendo utilizado no componente de comunicação, em específico na classe de configuração. Essa requer unicidade no projeto, haja vista que configuração é um requisito do projeto exigido por ser um ponto único e chave no projeto para que toda e qualquer alteração de configuração seja realizada em um mesmo lugar, evitando assim erros de usuário.

A classe de configuração é implementada apenas no componente de comunicação e, após o início da aplicação, seus valores passam a ser acessíveis pelas demais partes do projeto, garantindo assim que todas as demais partes do sistema que necessitem acessar dados de configuração utilizem-se dos mesmo dados. Dessa forma, a aplicação executará todas as ações de um mesmo propósito.

O padrão de projeto *Facade* tem por intenção fornecer uma interface unificada para um conjunto de interfaces em um subsistema. *Facade* define uma interface de nível mais alto, tornando o subsistema mais fácil de ser usado.

Facade é utilizado no projeto arquitetural do CROC em dois componentes: comunicação e histórico. A decisão de utilizar esse padrão de projeto nestes componentes justifica-se pelo fato de o componente de comunicação realizar ligações com agentes externos à aplicação, e o componente de histórico implementar variadas funcionalidades.

A fim de fornecer uma interface unificada para acesso de aplicações externas, o padrão facade resolveu esse problema em que foi implementado na comunicação. Como já explicado, estas diversas ações, tais como: verificar privacidade e segurança, iniciar os demais serviços. Devido a esse acúmulo de funções, o facade fornecerá uma interface simplificada e unificada para a criação do CROC, que com sua criação iniciará os serviços internos inerentes à aplicação.

No histórico, há demasiadas ações que são de sua responsabilidade. A ideia de utilizar o padrão facade nesse componente foi de facilitar o acesso às ações que são de

responsabilidade do histórico. Eximindo a responsabilidade de quem chama a ação de saber como a mesma está implementada, ou detalhes internos de seu funcionamento.

O padrão de projeto Template Method tem a intenção de definir o esqueleto de um algoritmo em uma operação, postergando alguns passos para as subclasses. Template Method permite que subclasses redefinam certos passos de um algoritmo, sem mudar a estrutura deste.

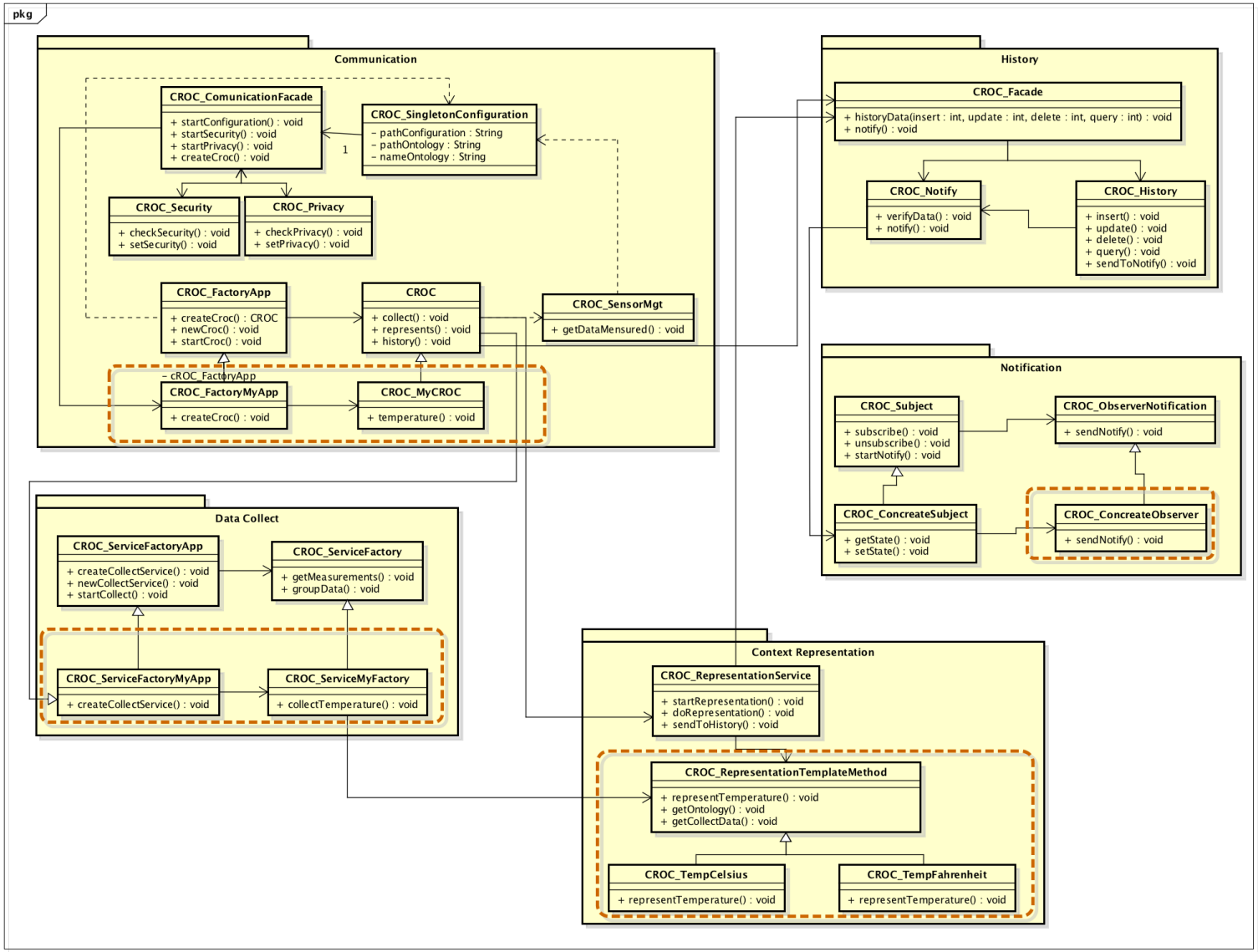
A opção pelo padrão que foi utilizado no componente de representação de contexto tem como objetivo que esse padrão auxilie variadas formas que podem ocorrer na representação do contexto mediante uma ontologia. Um exemplo é com a representação da temperatura, que pode ser realizada com uso da escala *Celsius* ou *Fahrenheit*. O esqueleto do algoritmo é o mesmo para definição da ontologia, dados coletados e a representação, contudo a implementação de cada algoritmo se torna diferente, então, posterga se a representação de fato para as subclasses de representação na escala celsius e fahrenheit. Sendo assim não há necessidade de implementação das ações que são as mesmas para as duas formas de representação.

O padrão de projeto *Observer* define uma dependência um-para-muitos entre objetos, de maneira que, quando um objeto muda de estado, todos os seus dependentes são notificados e atualizados automaticamente.

Utilizou-se o *observer* no componente de notificação. Com o objetivo de a notificação gerenciar uma lista de assinantes que desejem receber informações de determinadas ações executadas na aplicação. O padrão observer cria uma dependência de um-para-muitos, ou seja, uma notificação poderá ter muitos assinantes.

No componente de notificação, com o uso do padrão observer, há um gerenciamento das assinaturas e remoção de assinatura, identificação do estado de uma ação e a disseminação desse novo estado para os assinantes. Portanto, reduz-se a complexidade da aplicação em executar o requisito de notificação.

Figura 3.4: *Diagrama de classe - com implementação — CROC*



3.2 Componente de Agregação Semântica de Contexto (CASC)

O componente de agregação semântica de contexto — CASC — enquadra-se na fase de modelagem de contexto, diferentemente de Perera et al. [36] que trata esse componente na fase de raciocínio de contexto. O CASC não está incluso na fase de raciocínio de Perera et al. [36], pois este não implementa diferentes técnicas de fusão de dados. Realiza-se uma junção de várias informações de contextos providas do CROC, as quais devem ser limpas e melhoradas por meio do componente de Qualidade de Contexto (não tratado neste trabalho).

O componente CASC tem por objetivo realizar a junção, agregação, de múltiplos dados colhidos através de sensores e que passaram pela representação das informações de contexto por intermédio do CROC, o que proverá informações mais precisas, completas e de uma entidade (objeto, pessoa, etc). O CASC agrupa as informações (várias informações de uma mesma entidade), dessa forma poderá realizar-se o raciocínio das informações de contexto e completar mais uma fase do ciclo de vida do contexto, porém, antes de as informações serem agrupadas pelo CASC estas devem ser filtradas com o componente de QoC (Qualidade de Contexto).

Arquiteturalmente, esse componente atende a requisitos, como: configuração; armazenamento; comunicação distribuída e transparente [10]; notificação; separação de contexto e manipulação de contexto [10]; disponibilidade constante [10].

3.2.1 Requisitos funcionais e não funcionais

O projeto do Componente CASC é delimitado pelas seguintes funcionalidades:

- **Agregar informações** — reunir as informações coletadas, referente a um indivíduo, e agrupá-las para que possam ser utilizada por outro componente. No CROC as informações estavam "soltas", no agregador as informações são reunidas e fornecem a informação de um indivíduo (pessoa ou objeto), como: João localizado na sala X às 14 horas. Informações de **Quem** (João), **GPS** (localização) e **horário** (tempo).

O componente CASC compartilha das mesmas funcionalidades descritas no componente CROC 3.1. Exclui as funcionalidades de Coleta de Dados e Representação de Contexto, as quais são exclusivas do CROC e adiciona-se ao CASC a funcionalidade exclusiva de agregação de informações. Os dois componentes atuam em conjunto, na fase de modelagem de contexto do ciclo de vida do contexto, para prover as informações adquiridas (fase anterior). A próxima fase do ciclo de vida que é a de raciocínio de contexto tem como característica utilizar as informações geradas pelo componente de agregação.

Para fechar o escopo do componente de agregação, há algumas restrições que o mesmo possui em seu projeto quanto a sua função dentro do ciclo de vida de contexto. As restrições são descritas, como:

- não realiza tratamento nos dados obtidos e/ou gerados;
- implementa um padrão de intercâmbio de informações;
- independente de domínio de aplicação; (o agregador é o mesmo para diferentes tipos de aplicação, não sendo necessárias alterações impactantes em sua arquitetura);

Com o escopo fechado e suas limitações definidas, o CASC possui a dependência do CROC. O componente de agregação de contexto depende diretamente do CROC para obter as informações que serão agregadas a partir das informações colhidas independentemente.

Para a criação do projeto arquitetural do CASC, foram definidos os requisitos funcionais e não funcionais, os quais são descritos nesse trabalho. Alguns dos requisitos aqui descritos são os mesmos utilizados no CROC 3.1, sendo assim são descritos apenas os requisitos específicos aos componente em descrição.

Os requisitos funcionais semelhantes são: Separação de interesses e manipulação de contexto; comunicação distribuída transparente; armazenamento e consulta de informações de contexto.

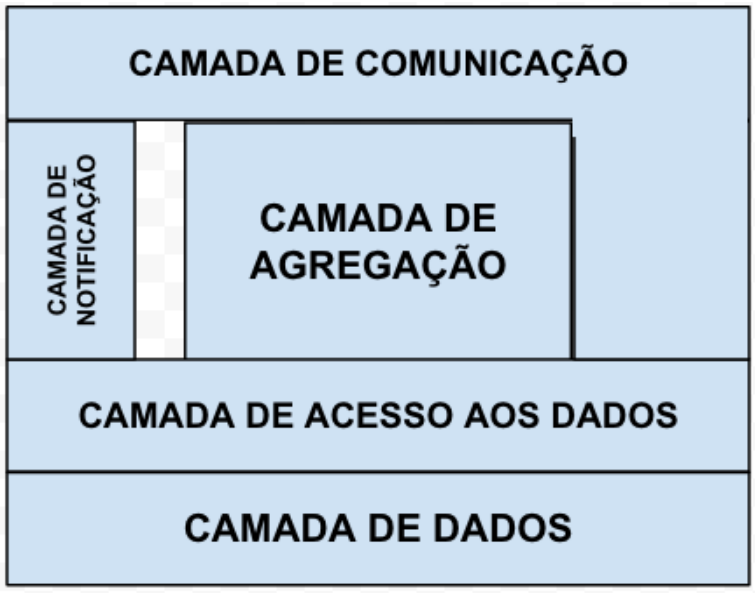
Os requisitos não funcionais semelhantes são: Interoperabilidade, Extensibilidade, Arquitetura em camadas, Reutilização de componentes, Manutenibilidade, Escalabilidade, Suporte a ambientes heterogêneos, Suporte a implementações alternativas, Notificação, Segurança e a Privacidade.

1. **Agregação de contexto** — O componente deve ser capaz de agrupar informações diversas de uma mesma entidade (pessoa ou objeto). Informações diversas entende-se como toda e qualquer informação específica de uma entidade adquirida por um CROC, como temperatura, localização, identificação e outras. A informação agrupada refere-se a um conjunto das informações específicas relacionadas a uma entidade, como "A temperatura do Vinho Terralis é de 23^{o}C na Adega Dom Juan". Nessa frase, tem-se informação de: temperatura (23 ^{o}C), localização (Adega Dom Juan) e identificação (Vinho Terralis). Então, a informação foi agregada, gerando uma frase significativa.
2. **Disponibilidade Contínua e constante de agregação** — O componente deve ter a capacidade de realizar a agregação de novos dados e informações de contexto de forma constante e sem intervenção do usuário, fornecendo as informações agrupadas aos interessados.

3.2.2 Arquitetura do componente

De acordo com a revisão sistemática realizada por [30], contida no Apêndice A, foi identificado que a arquitetura de software utilizada e recomendada para esse modelo de sistema computacional é a utilização da Arquitetura em Camadas. A Figura 3.5 representa, graficamente, como estão organizadas as camadas no componente CASC.

Figura 3.5: *Arquitetura em Camadas — CASC*



A camada de comunicação, acesso aos dados, dados e notificação exercem a mesma função das camadas descritas na Seção 3.1, havendo alteração na camada de agregação.

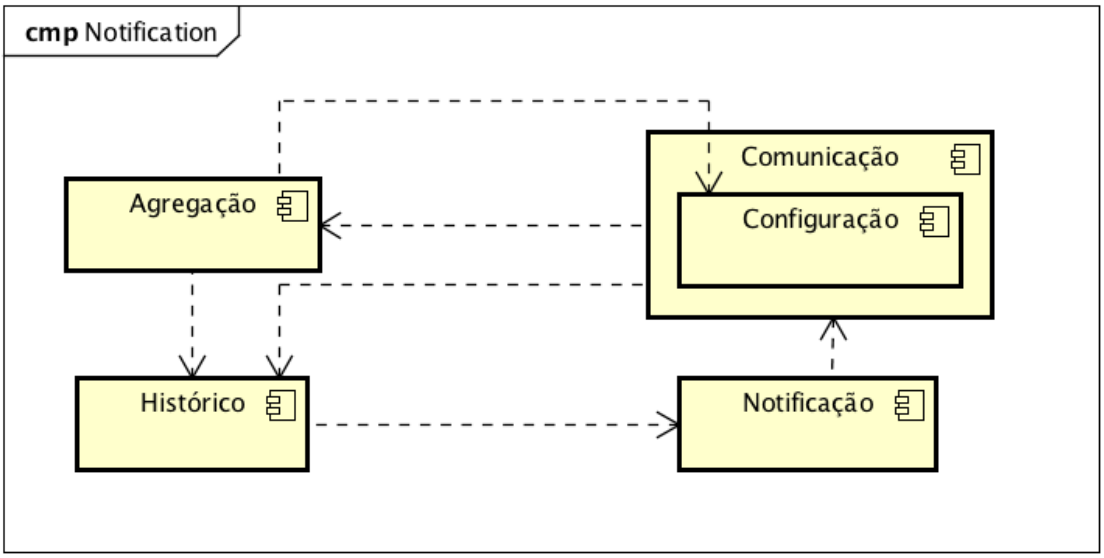
A camada de agregação é a responsável por realizar o agrupamento, dos dados que foram coletados pelos diversos CROC, gerando assim uma nova informação. Essa nova informação possui características de uma mesma entidade, como, por exemplo a temperatura, pressão arterial e saturação de oxigênio de um mesmo indivíduo. Essas informações são necessárias e úteis para a próxima etapa do ciclo de vida do contexto: raciocínio.

Diagrama de componentes

As definições e os motivos do uso do diagrama de componentes são apresentadas na subseção 3.1.2. Na Figura 3.6, apresenta-se o diagrama de componentes do CASC 3.2.

O funcionamento geral do componente compreende-se pela ação de iniciar através da comunicação, que carrega as informações de configuração. A comunicação é responsável por instanciar o componente de agregação (ponto central da aplicação), que, por sua vez, armazena os dados processados no histórico, que instância a notificação. Notificação realiza a publicação das mensagens por meio da comunicação.

Figura 3.6: *Diagrama de componentes — CASC*



A comunicação é o ponto central do componente CASC. Partindo dela são instanciados os demais serviços do componente, e por ela passam as requisições que devem ser enviadas aos componentes externos e também aos componentes internos. Sendo assim, a comunicação possui ligação com agregação (inicia o serviço de agrupar as informações), configuração (carrega para a aplicação as informações de configuração), histórico (busca informações já armazenadas) e notificação (envio das mensagens públicas).

A agregação é responsável por realizar a operação fim do componente: agrupar informações de contexto. Para tanto, esse componente recebe da comunicação a ação de iniciar o serviço de agregar e para que este seja executado com êxito, ele é carregado das informações de configuração. Após o agrupamento ser realizado, a mesma envia os novos dados para o histórico.

A descrição e as ações realizadas pelos componentes histórico, pela configuração e notificação, são detalhadas na subseção 3.1.2, onde há também um maior detalhamento do funcionamento do componente de comunicação.

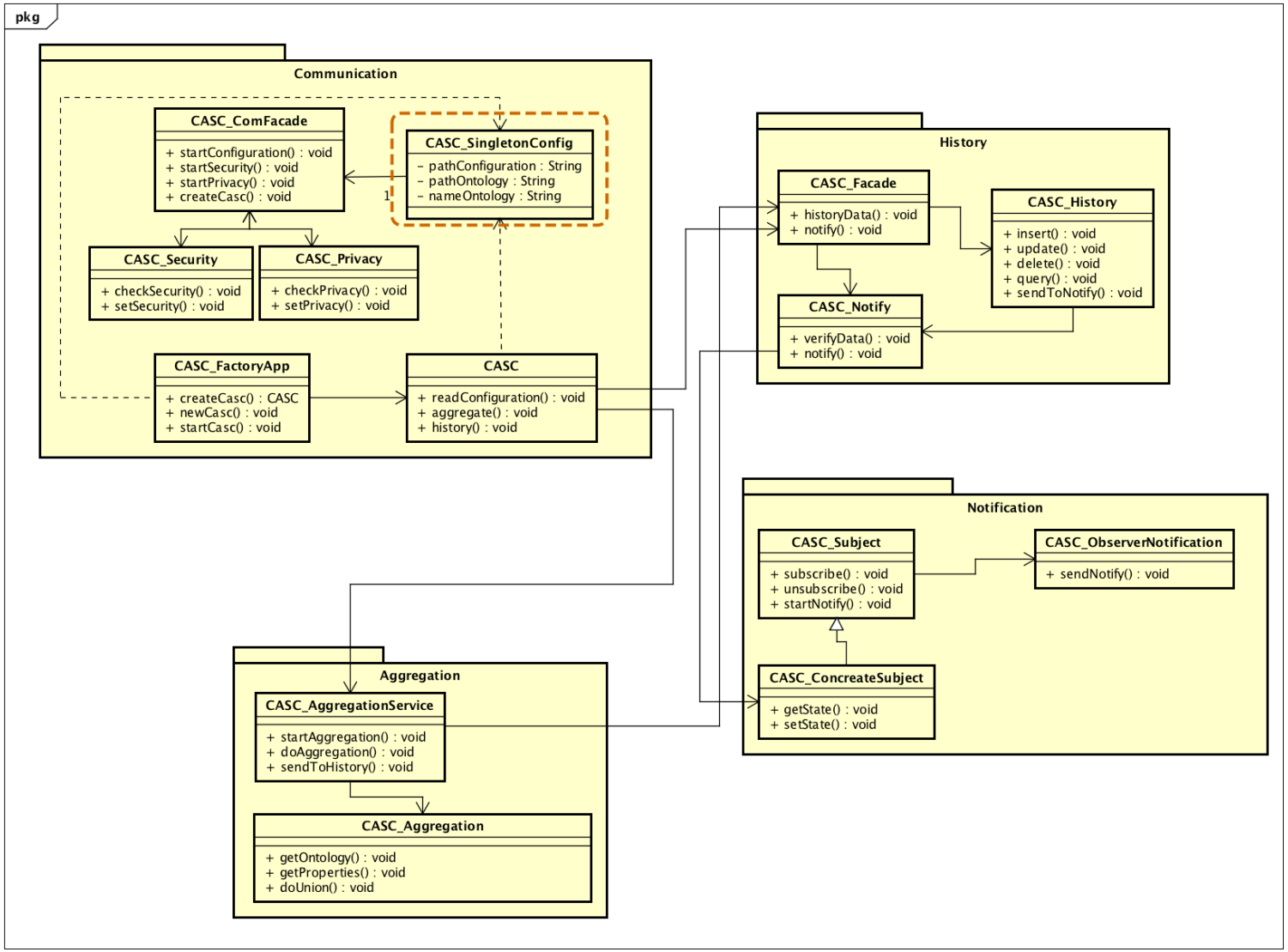
Diagrama de classes

O CASC está divido em 4 componentes, que possuem suas funções e atividades claramente definidas. São eles: comunicação, agregação, histórico e notificação. A seguir será explanada a função de cada um desses componentes, e, na Figura 3.7, estão representadas, graficamente, as classes e como estão relacionadas.

As classes de comunicação, histórico e notificação são as mesmas apresentadas na Seção 3.1. Neste componente — o diagrama de classes — há a adição da classe de

agregação (que realiza as funções inerentes ao componente) e retirada das classes de coleta de dados e representação de contexto.

Figura 3.7: *Diagrama de classe — CASC*



- **Agregação** é o componente principal do CASC. No componente de agregação é realizada, de fato, a união dos dados que foram representados pelos diversos CROCs da infraestrutura. Para a realização dessa união, o componente, como um todo, utiliza-se de uma ontologia que descreve como as informações deverão estar dispostas, para que possam ser unidas. Para essa tarefa, o componente faz uma requisição para buscar a ontologia e suas propriedades (colhidas pelos CROC).

Decisões de projeto

Os padrões de projeto utilizado nos componentes de comunicação, histórico e notificação são os mesmos descritos na seção que trata do componente CROC 3.1.2. Dessa forma, serão apresentados os padrões de projeto utilizados no componente de agregação, que é específico do componente CASC.

O padrão de projeto utilizado no componente de agregação é o *singleton*, que foi descrito em 3.1.2. A decisão de utilizar este padrão nesse componente se deu em razão de que a agregação deva ocorrer de forma única dentro do componente.

A unicidade na agregação relaciona-se com o fato de que os dados gerados serão utilizados por outras aplicações (a fim de completar o ciclo de vida de contexto). É necessário que a agregação seja um *singleton* para evitar que as informações advindas dos CROCs gerem inconsistência na agregação.

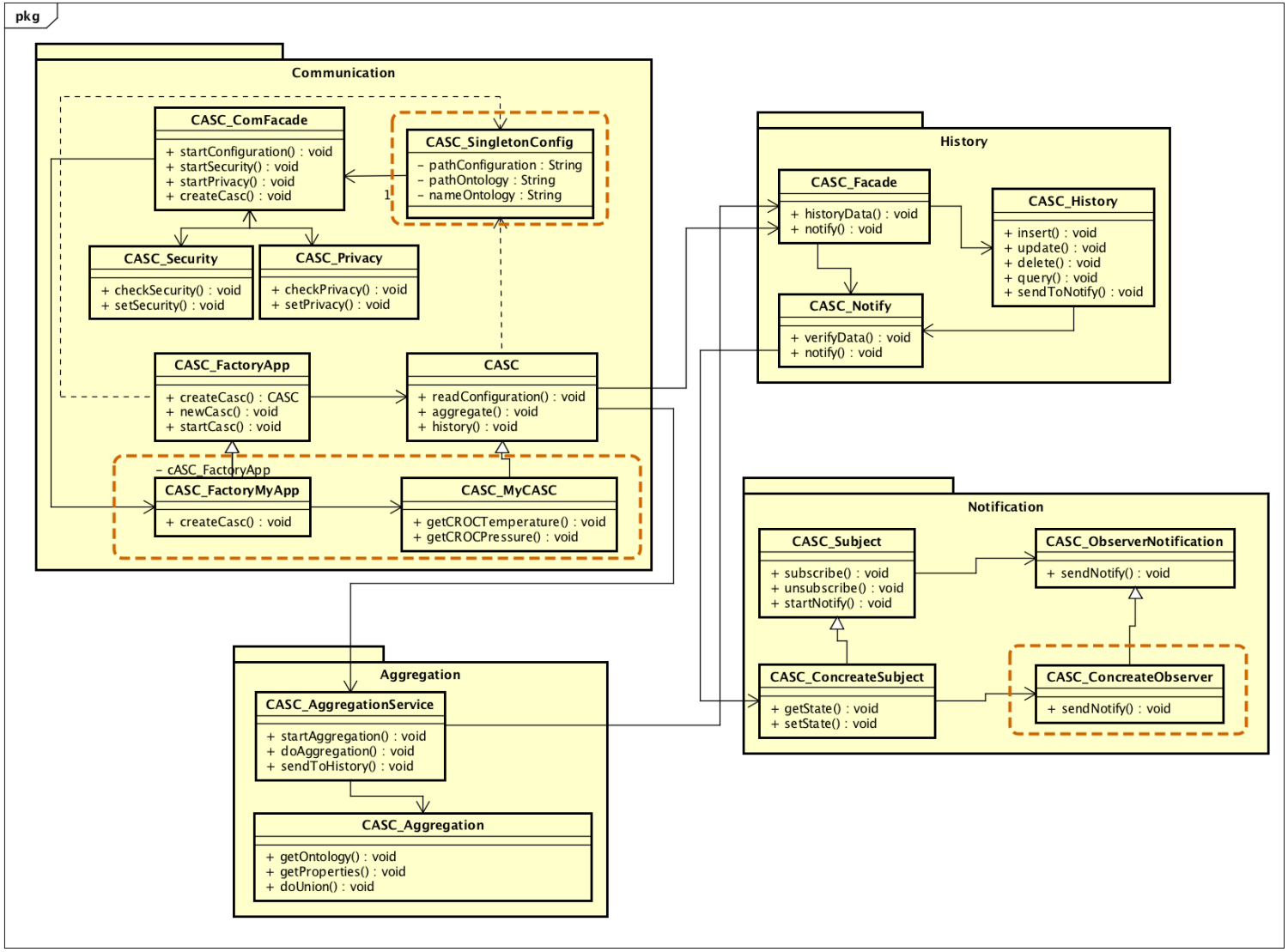


Figura 3.8: *Diagrama de classe — com implementação — CASC*

3.3 Considerações finais

Para sintetizar o exposto neste capítulo, a Tabela 3.1, contendo a correlação existente entre os componentes, requisitos, camada da arquitetura, classes e padrões de projeto.

Na Tabela 3.2 é possível visualizar a correlação entre os padrões de projetos utilizados no CROC e CASC, o problema que eles resolveram e em quais componentes internos foram implementados.

Por definição, os padrões de projeto buscam resolver problemas comuns, com soluções comuns em um dado ambiente. Sendo assim, este trabalho aborda a aplicação dos padrões de projeto já existentes para sistemas sensíveis ao contexto semântico. Nesse Capítulo 3, foi apresentado o projeto arquitetural dos componentes CROC e CASC [3.2 e 3.1] com a implantação dos padrões de projeto, contidos nas Figuras 3.4 e 3.8.

O incentivo para o desenvolvimento desse projeto arquitetural partiu da Revisão Sistemática [30]. O objetivo proposto foi de completar as lacunas obtidas como resultado. São elas: manutenibilidade, reúso de software e aplicação de padrões de projeto.

Os requisitos funcionais, coleta de dados, representação de contexto e agregação, são os principais dentro dos componentes. Os demais requisitos são comuns a ambos os componentes e realizam funções auxiliares.

Para apresentar o projeto arquitetural dos componentes, utilizou-se dos diagramas de componentes, classes e de sequência. Com essas figuras, é possível ter uma melhor visualização e entendimento de como os componentes estão arquitetados.

Foram apresentados os requisitos funcionais e não funcionais dos componentes, para que se tenha embasamento de como os componentes foram construídos e entendimento das decisões de projeto.

O próximo capítulo versa sobre as métricas e os resultados obtidos no desenvolvimento do projeto arquitetural dos componentes CROC e CASC. Será apresentada também, uma análise comparativa do projeto com outro sistema desenvolvido com o mesmo intuito.

Tabela 3.1: *Correlação entre os requisitos, camada da arquitetura e padrões de projeto*

Comp.	**Requisito**	**Camada**	**Classe**	**Padrões de projeto**
CROC	Comunicação distribuída e transparente	Comunicação	*Communication*	*Factory Method*, *Singleton*, *Facade*
	Prover informações internas (configuração)			
	Coleta de dados	Coleta	*Data Collect*	*Factory Method*
	Representação ontológica do modelo de contexto	Representação	*Context Representation*	*Template Method*
	Especificação de contexto			
	Separação de interesses e manipulação de contexto			
	Armazenamento e histórico de contexto	Acesso aos dados e dados	*History*	*Facade*
	Comunicação distribuída e transparente			
	Notificação	Notificação	*Notification*	*Observer*
CASC	Comunicação distribuída e transparente	Comunicação	*Communication*	*Factory Method*, *Singleton*, Facade
	Prover informações internas (configuração)			*Factory Method*
	Agregador	Agregação	*Aggregation*	*Singleton*
	Armazenamento e histórico de contexto	Acesso aos dados e dados	*History*	*Facade*
	Comunicação distribuída e transparente			
	Notificação	Notificação	*Notification*	*Observer*

Tabela 3.2: *Correlação entre os padrões de projeto e os problemas resolvidos*

Design Pattern	**Problema Resolvido**	**Componente do Projeto**
Factory Method	Criação de novas classes sem afetar a estrutura geral do sistema. No caso do CROC permite a criação, por exemplo, para temperatura e umidade, sem que seja necessário alterar a estrutura geral, alterando-se, então, apenas as classes concretas.	Comunicação e coleta de dados.
Singleton	Permite a criação de apenas uma classe, única, de configuração. Sendo esta o único ponto global de acesso aos dados de configuração.	Comunicação (configuração) Agregação de contexto
Facade	Simplifica o acesso externo ao componente. É responsável por abstrair as interfaces de acesso externo ao sistema. Está implementado na comunicação.	Comunicação e histórico.
Template Method	Implementado na representação e agregação das informações, soluciona o problema de alteração da ontologia utilizada. Em que é possível alterar apenas a ontologia e não a codificação de outras classes.	Representação de contexto
Observer	Soluciona o problema de quando há uma notificação e que esta deve ser enviada para inúmeras outras pessoas. Exemplificando a situação de um-para-muitos.	Notificação

CAPÍTULO 4

Avaliação da Arquitetura do CROC/CASC

Neste capítulo é apresentado como se realizou a avaliação do projeto arquitetural descrito no Capítulo 3, no qual discorreu-se sobre os componentes de representação ontológica e a agregação ontológica de contexto. A avaliação baseia-se principalmente nos métodos descritos por Martin [31].

O objetivo do capítulo é avaliar como os projetos foram construídos e propostos, como as classes e pacotes estão arquitetados e avaliar o projeto orientado a objetos. Não se objetivou examinar a qualidade do código ou a codificação do componente, que não são o foco deste trabalho, mas sim a arquitetura de cada componente.

Com as métricas descritas por Martin [31] e dos pressupostos expostos por Sommerville [43], foi possível comparar os projetos aqui propostos com projetos executados por outros autores.

Objetiva-se com essa avaliação obter insumos para estabelecer parâmetros comparativos, a fim de avaliar, realmente, os pontos positivos e negativos obtidos com o projeto arquitetural e estabelecer comparativos com outros projetos arquiteturais existentes na literatura. Como objetivos secundários, espera-se criar parâmetros que possam ser utilizados para avaliar projetos que se baseiem neste ou que tenham o mesmo cerne.

4.1 Materiais e métodos

Nesta seção, explana-se sobre as ferramentas que auxiliaram na obtenção das métricas, sobre os necessários para a avaliação e sobre métodos utilizados pelas ferramentas que resultam nas métricas.

Para a avaliação aqui em análise são utilizados os projetos CROC e CASC, Hermes Widget (HW) [44] e Hermes Aggregator (HA) [44]. A comparação é feita entre o CROC e o HW, e entre o CASC e o HA, por terem objetivos semelhantes. Os primeiros, com o objetivo de representar semanticamente, coletar e adquirir dados, e os outros dois, com o objetivo de agrupar dados representados semanticamente.

Para avaliar os projetos propostos, valeu-se de métricas já difundidas na literatura e empregadas por vários autores [31, 37, 5, 1, 11, 20]. São elas:

- *Efferent Coupling* (Ce)
- *Afferent Coupling* (Ca)
- *Instability* (I)
- *Distance from Main Sequence* (D)
- *Lack of cohesion* (LCOM)
- *Weighted Methods per Class* (WMC)

Com as métricas aqui descritas, obteveram-se dados necessários e suficientes para comparar os projetos propostos na avaliação. As métricas, por si só, não geram informações suficientes para afirmar que um projeto está certo ou errado, ou que precise de correções em determinados componentes, mas, sim, embasam o projetista a fim de que este possa realizar comparações e identificar possíveis pontos de falha, já que é de responsabilidade do projetista de software a análise dos valores obtidos com as métricas e análise de seu projeto para determinar pontos de melhorias.

4.2 Resultados e discussões

Nesta seção, são explicitados o passo-a-passo do experimento e os resultados obtidos. O objetivo é demonstrar como a avaliação foi realizada e uma base para que outros projetos possam recorrer ao mesmo procedimento de avaliação para obter dados comparativos.

4.2.1 Experimento

São ferramentas utilizadas para realizar o experimento:

- *Astah Professional*
- *JDepend* (*plugin*)
- *Source Code Metrics* (*plugin*)
- Eclipse (IDE de desenvolvimento)
- Netbeans (IDE de desenvolvimento)

Os componentes CROC e CASC foram codificados com base no projeto desenvolvido no *Astah Professional*. Sendo assim, primeiramente desenvolveu-se todo o projeto arquitetural dos componentes (diagrama de classes). Em seguida realizou-se a validação, no nível de projeto, necessária para o prosseguimento do projeto. Por fim, exportou-se o projeto (diagrama de classes) em forma de código Java, para que este pudesse ser importado em uma IDE de desenvolvimento para avaliação dos plugins.

A fim de que a avaliação pudesse ser realizada de forma satisfatória, foi necessária a importação da codificação nas IDEs. A partir desse ponto, com as IDEs configuradas

com os devidos *plugins* e o código organizado em seus respectivos pacotes e arquivos, executou-se a análise do código.

Como resultado da análise do código, cada plugin retornou um conjunto de dados tabulados, os quais são organizados e referenciados de acordo com as métricas mensuradas por cada plugin.

Os componentes HW e HA da infraestrutura Hermes de [44], são de terceiros (ou seja, não desenvolvidos nesse trabalho). Realizou-se a importação dos projetos obtidos com o desenvolvedor para as IDEs de desenvolvimento e aferiram-se as métricas do CROC e CASC, para que se tenham os mesmos parâmetros de comparação.

É importante e necessário ressaltar que os componentes CROC e CASC não foram implementados por completo, utilizou-se para análise apenas as classes e métodos. Diferentemente dos componentes HW e HA, que foram implementados por completo. O foco deste trabalho não é a implementação e pleno funcionamento dos componentes, mas, sim, a proposição de uma arquitetura de software, de padrões, de projeto e recomendações de desenvolvimento para aplicações que tenham o mesmo objetivo de cobrir a fase de modelagem de contexto do ciclo de vida de contexto proposto por Perera et al. [36].

4.2.2 Discusão

Como resultados, obteve-se um conjunto de dados que permitiu a realização de análises e comparações com o projeto arquitetural, para embasar as conclusões deste trabalho. Os resultados são apresentados de forma detalhada por cada comparativo e posteriormente são apresentadas as conclusões finais obtidas no geral.

Comparação entre CROC e HW

Ao realizar o experimentou, foram obtidos os resultados expostos na Tabela 4.1, e é com base nos valores colhidos e contidos nessa tabela que são apresentadas as justificativas e afirmações a respeito de cada projeto aqui avaliado.

Tabela 4.1: *Valores da avaliação - CROC e HW*

	Ca	Ce	I	D	
CROC	6	6	0,43	0,56	*Dependencies*
HW	57	37	0,49	0,52	
CROC	6	3	0,29	0,71	*Depends upon*
HW	56	32	0,43	0,59	
CROC	5	6	0,54	0,46	*Used by*
HW	58	42	0,49	0,52	

Com base na análise primária dos dados evidenciados na Tabela 4.1, é possível inferir algumas informações, que serão discutidas. Para tanto, a análise das métricas de aferente (Ca), eferente (Ce), instabilidade (I) e distância da sequência principal (D) são divididas em três categorias: *dependencies*, *depends upon* e *used by*. Em que Ca e Ce são utilizadas como forma de cálculo para as métricas de I e D, desta forma estes valores (Ca e Ce), por si só, não serão analisados.

A categoria *depends upon* constitui as dependências que estão vinculados a uma classe ou pacote atual e as classes que estão um nível anterior (exemplo: todos dependem de *Object*). A categoria *used by* refere-se a quais classes ou pacotes usam a classe ou pacote atual. Exemplo: *Object* é usado por determinadas classes e pacotes. *Dependencies* é uma avaliação, de forma geral, que o *plugin* retorna ao usuário. Permitindo uma análise genérica do comportamento do projeto.

Em relação a *Dependencies* os componentes se diferenciaram pouco em relação a (I) e (D) pouco. O CROC com (I): 0,43 e 0,56 de (D), e CASC com (I): 0,49 e 0,52 de (D). É conclusivo, então, que ambos os componentes estão com características parecidas quando se analisa as métricas de *Dependencies*, remetendo ao fato de que ambos componentes estão com mesmo objetivo. As diferenças marcantes estão em *Depends upon*.

Analisa-se que, em relação ao HW em *depends upon*, a estabilidade do CROC é maior (CROC: 0,29 e HW 0,43). Isto ocorre pelo fato de que no projeto do CROC há um componente central (que é o de comunicação) e, com base neste, derivam-se todas as demais funções internas. Resultando, então, em que por se ter um componente central de comunicação (para que as demais funções do sistema sejam instanciadas) o projeto esteja estável e naturalmente se distancia da sequência principal do software(CROC: 0,71 e HW: 0,59), para atender às requisições de seus componentes internos.

Quando se trata da categoria *used by*, a diferença de valores é mínima, sugerindo a equiparação no desenvolvimento de ambos os projetos. Justifica-se um maior valor dessa métrica (em relação ao HW), em comparação com a *depends upon* em razão de sua arquitetura ser mais modular, gerando um maior número de requisições de entrada.

Ambos os projetos dispõem de suas diretrizes e decisões de projetos que os tornam diferentes um do outro. Neste trabalho, o ponto que se aborda é a necessidade e importância da utilização de padrões de projeto, de criar um projeto reusável e manutenível. Usou-se no módulo de configuração do CROC o padrão de projeto *singleton*, para se obter um único ponto no sistema configurável, o que acarreta melhorias na manutenção, pois haverá um único ponto de alteração.

Por meio da revisão sistemática, notou-se que um componente central seria o mais adequado ao tipo de projeto em análise. Isso decorre do fato de que, na existência de um componente central, espera-se que a parte de gerenciamento fique concentrada,

tendo, então, como consequência melhorias de manutenabilidade, reusabilidade e criação de novos componentes (objetivo do software de representação semântica).

Com essas decisões de projeto, confirma-se esse ganho de melhorias em comparação ao projeto do HW nas métricas de *depends upon*, em que o CROC é mais estável do que o HW. A centralização do componente de comunicação, que gerencia as demais ações do sistema, corroboram para essa estabilidade. Decorrentemente a origem das ações concentra em um único lugar.

Quanto à coesão, foi constatado por meio das métricas aferidas pelo *Source Code Metrics* que o sistema de HW está mais coeso do que o CROC. Em que valorou-se 0,36 para o HW e 0,76 para o CROC. Isso também pode ser corroborado pelo fato de que o CROC não foi desenvolvido em sua completude, utilizou-se, sim, do código com as interfaces para análise. Em detrimento de que não foi possível analisar-se apenas o projeto do HW.

Sobre a métrica de WMC, que mede a complexidade de cada método (no sentido do desenvolvimento), o CROC em seu componente de comunicação obteve 17 e o HW obteve 21 pontos. Reafirmando, o fato, de que, com a implementação e uso de padrões de projeto conseguiu-se diminuir a complexidade no desenvolvimento na ordem de 21%. No geral não é possível afirmar que um ou outro componente é mais ou menos complexo de se desenvolver, devido ao fato dos códigos analisados estarem em diferentes estágios de desenvolvimento.

Comparação entre CASC e HA

A Tabela 4.2 apresenta os valores obtidos na avaliação dos componentes CASC e HA mediante a ferramenta *JDepend*. Por meio dos dados são apresentadas as comparações possíveis entre os dois projetos e as vantagens e desvantagens das decisões de projeto.

Tabela 4.2: *Valores da avaliação - CASC e HA*

	Ca	Ce	I	D	
CASC	7	7	0,52	0,48	*Dependencies*
HA	14	22	0,61	0,38	
CASC	7	7	0,52	0,48	*Depends upon*
HA	18	17	0,41	0,59	
CASC	7	7	0,52	0,48	*Used by*
HA	7	20	0,76	0,24	

Em *dependencies* o projeto do CASC demonstrou-se mais estável (CASC: 0,52 e HA: 0,61) e com uma maior distância da sequência principal (CASC: 0,48 e HA: 0,38).

Isso remete ao fato de que a centralização do componente de comunicação no projeto do CASC obteve como resultado maior intercomunicação entre os demais componentes diretamente com a comunicação e uma menor interligação entre os componentes. Em outros termos, há ligações que ramificam direto da comunicação e não há necessidade de excesso de ligações entre os componentes.

Quando se analisa os valores alcançados na categoria *depends upon*, verifica-se que o HA demonstrou-se mais estável (HA:0,52 e CASC: 0,41) e o CASC mais próximo da sequência principal (HA: 0,48 e CASC: 0,59). Neste quesito, essa maior instabilidade do CASC era de se esperar, haja vista que essa análise diz respeito às dependências (*depends upon*) que há no software. Essa dependência justifica-se pelo fato de que o CASC propõe como decisão de projeto um componente central. Com isso é possível que todos os demais componentes (*used by*) tenham uma dependência com o componente central, elevando, então, o valor de instabilidade. Isso não pode ser considerado como uma desvantagem em relação ao HA, por se tratar de uma decisão de projeto que visa à manutenabilidade e reusabilidade do componente.

Os resultados para *used by* expõem que o CASC está mais estável, e que o HA aproxima-se mais da sequência principal do software. A estabilidade, nesse ponto, é fundamentada, mais uma vez, pelas decisões de projeto em que se tem um componente central, e por serem apresentados, padrões de projeto como *facade* e *observer*, os quais implicam menor interligação entre os componentes do sistema.

Quanto à coesão, tanto o HA quanto o CASC estão coesos. Em que o HA mediu-se 0,44 e o CASC 0,78, esta diferença é confirmada pelo fato da análise de diferentes códigos (o HA em código completo e CASC apenas o projeto). Analisa-se nesta métrica é a capacidade do software em atender os requisitos não funcionais de manutenabilidade e de reusabilidade, desta forma os valores mensurados são considerados eficientes para o atendimento destes requisitos não funcionais.

A complexidade de desenvolvimento do CASC é 36 e do HA 74, de forma geral, esta métrica é utilizada quando se deseja comparar o esforço necessário para desenvolver um software em comparação com outro. Desta maneira o CASC é menos dispendioso de desenvolver do que o HA. Em relação ao componente de comunicação o CASC consome 44% do esforço de desenvolvimento, enquanto o HA consome 32%. Esta diferença de 12% é corroborada pelo fato do CASC ter um componente de comunicação mais complexo e com maiores funções do que o componente de comunicação do HA.

4.3 Considerações Finais

Neste capítulo tratou-se dos resultados e conclusões que puderam ser produzidas com base nos projetos arquiteturais aqui analisados.

Para a realização da avaliação, recorreu-se às métricas propostas por Martin [31] e Sommerville [43], além de outras métricas já difundidas na literatura, como *efferent coupling*, *afferent coupling* e *instability*.

Os projetos aqui avaliados foram o CROC, HW, CASC e HA, os quais atuam na fase de modelagem de contexto do ciclo de vida proposto por Perera et al. [36].

Para a automatização das avaliações e obtenção de valores necessários para análise, usou-se de ferramentas como: *Astah Professional*, *JDepend*, *Source Code Metrics*, *Eclipse* e *Netbeans*.

Os resultados demonstram diferenças em cada projeto, bem como permitiu inferir algumas informações a respeito do projeto arquitetural de cada componente, haja vista que a implementação não foi concluída. Por conseguinte, a avaliação decorreu apenas no âmbito do projeto. Dessa forma a análise foi feita no código das interfaces e métodos, em que estes não foram implementados por completo.

CAPÍTULO 5

Trabalhos Relacionados

Este capítulo apresenta os trabalhos relacionados com projeto arquitetural de SSCS modelados por ontologias. Serão exibidos quais trabalhos têm relação com os projetos propostos, e quais os trabalhos que serviram de embasamento teórico para ratificar as necessidades apresentadas.

5.1 Revisão Sistemática

Para o embasamento e início desta dissertação, realizou-se uma Revisão Sistemática (RS), a qual está na íntegra no Apêndice A. A RS foi realizada com foco em levantar as características da aplicação de padrões de projeto e estilos arquiteturais na área de SSCS. Com base em seus resultados, foi possível estabelecer os objetivos e anseios esperados nessa dissertação.

Uma revisão sistemática é de grande importância para que se possa ter uma síntese do que está em discussão na literatura para determinado assunto. Neste intuíto realizou-se neste trabalho uma revisão sistemática para elucidar as discussões dos últimos 5 anos sobre o tema proposto.

Para a realização da RS utilizou-se de um protocolo, o qual é demonstrado na Figura A.1, abrangendo as fases de planejamento, execução, seleção extração e conclusão.

Na fase de planejamento definiu-se as *strings* de busca, questões de pesquisa e as fontes de pesquisa. Na próxima fase, de execução, buscou se os artigos para serem utilizados na próxima fase, que selecionou os artigos relevantes por meio da leitura do título e resumo. A extração procedeu-se com a leitura na íntegra de todos os trabalhos (artigos) selecionados anteriormente. E, por fim, a última fase, de conclusão, elaborou-se um relatório contendo os resultados obtidos.

A execução da RS compreendeu os anos de 2009 a 2014, e foi realizada nas principais bibliotecas digitais da área de computação (ACM Digital Library[1], IEEE[2],

[1] dl.acm.org
[2] ieeexplore.ieee.org

Scopus[3] e SPRINGER[4]). Retornando 893 artigos, dos quais foram filtrados de acordo com o protocolo de execução da RS, restou-se então 101 artigos para serem analisados por completo.

5.2 Resultados

Nesta seção serão apresentados os resultados considerados mais relevantes para a execução da dissertação. A questão de pesquisa *Quais são os estilos arquiteturais aplicados no projeto arquitetural de SSCS?*, ocorreu em 38 artigos. E nos ajudou a identificar os estilos arquiteturais que foram aplicados em SSCS, onde temos como base que estilos arquiteturais são uma especialização de elementos e tipos relacionados, juntos com um conjunto de restrições de como eles podem ser utilizado [35].

Como resultado, esta questão de pesquisa retornou os dados tabulados na Tabela A.6, onde se constata que o estilo arquitetural mais utilizado pelos autores é *camadas*.

A questão de pesquisa *Quais requisitos não funcionais influem no projeto arquitetural de SSCS?*, foi concebida a partir do conceito estabelecido em Sommerville [43]: *requisitos que não estão relacionados com os serviços específicos oferecidos pelo sistema a seus usuários*, com o objetivo de levantar quais os principais requisitos não funcionais que os autores consideravam em seus sistemas.

Através desta questão de pesquisa foi possível verificar a necessidade em projetar sistema que continham os requisitos não funcionais de: usabilidade, privacidade e manutenabilidade. Pois através da revisão sistemática foi constado a preocupação constante dos autores com estes requisitos, e que os mesmos influência na forma de projetar e implementar os sistemas propostos nesta dissertação.

A revisão sistemática buscou, também, elucidar quais são os padrões de projetos mais utilizados pela literatura, e que tipos de problemas são resolvidos com a aplicação desses padrões. Sendo assim os padrões de projeto mais utilizados na literatura são *observer, prototype e decorator* [13].

Os padrões de projeto citados são utilizados em sistemas de tempo real, replicação de objetos já criados no sistema e adicionar informações em tempo de execução.

Observer padrão de projeto utilizado para atualizar instâncias de sistemas de tempo real, ou seja, manter o sistema atualizado com relação ao que ocorre em seu ambiente. Utilizou-se este padrão para a construção de uma aplicação consciente de contexto em diferentes domínios, pois assim a aplicação *se mantém atualizada* do contexto, independente do domínio da aplicação, conforme expresso por [18].

[3] `https://www.scopus.com/`
[4] `http://link.springer.com/`

Prototype é um padrão de projeto utilizado para a criação de novos objetos a partir de um modelo original. Nos SSCS este padrão de projeto auxilia a criação de novas instâncias semelhantes para objetos diferentes.

Decorator é um padrão de projeto utilizado em SSCS para interceptar um objeto, em tempo de execução, e adicionar alguma informação relevante para o objeto e que não poderia ser adicionada anteriormente, conforme é demonstrado em [39] e [25]. Sendo assim, é uma forma de atualização do comportamento do objeto em tempo de execução.

Além, dos citados, há o padrão arquitetural denominado MVCC proposto por Shams et al. [42]. MVCC (*Model-View-Controller-Context*) é uma solução arquitetural voltada para *frameworks* com foco em desenvolvimento rápido de sistemas e na reusabilidade de código.

Concluiu-se que os padrões de projeto mais utilizados são o *observer* e o *prototype*, e que os sistemas são desenvolvidos em camadas, dessa forma entende-se que para desenvolver um SSCS deve-se seguir esta linha de desenvolvimento. Os requisitos não funcionais mais utilizados são *usabilidade*, *segurança*, *privacidade* e *manutenabilidade*. Conclui-se também que, pelo fato de existirem diversas iniciativas para produzir conteúdos para SSCS, não há um empenho em unir as pesquisas realizadas, fazendo com que estas ideias e iniciativas fiquem esparsas no ambiente de pesquisa.

O quantitativo completo da ocorrência dos padrões de projeto nos artigos extraídos na revisão sistemática pode ser visualizado na Figura A.4.

5.3 Considerações Finais

Nestes capítulo apresentamos os trabalhos que estão relacionados diretamente com o proposto nesta dissertação. Exibimos também de onde surgiram as principais ideias para a realização desse trabalho, que foram nos estudos realizado pela revisão sistemática. Demonstrando assim, o embasamento que obtivemos na literatura. A proposta do trabalho em utilizar padrões de projeto, demonstrar os estilos arquiteturais recomendados para SSCS foram embasados nos relatórios obtidos e apresentados tanto no Apêndice A, quanto neste capítulo.

CAPÍTULO 6

Conclusões e Trabalhos Futuros

Este capítulo apresenta as conclusões inerentes a este trabalho de desenvolvimento dos componentes de representação ontológica de contexto (CROC) e do componente de agregação semântica de contexto (CASC), quanto ao projeto arquitetural no tocante às decisões de projeto e dos requisitos necessários a cada um.

Primeiramente estabeleceu-se um par comparativo com os componentes, para avaliar os componentes aqui propostos. O comparativo ocorre com outros dois componentes propostos na literatura: *Hermes Widget* e *Hermes Aggregator*. Os pares de comparação são o CROC e HW e o CASC e HA. Com base nesses comparativos, desenvolveu-se a avaliação descrita no Capítulo 4, e com os resultados chegou-se às conclusões aqui apresentadas.

Esses componentes, que atuam na fase de modelagem do ciclo de vida do contexto [36], corroboram o atendimento de um problema identificado na literatura: a carência de soluções arquiteturais de SSCS na etapa de modelagem, contribuindo com a melhoria no desenvolvimento de sistemas que atenda a essa fase.

6.1 Contribuições

As principais contribuições deste trabalho podem ser assim destacadas:

1. **Solução arquitetural para o serviço de representação ontológica de contexto** — CROC possui a responsabilidade de representar por meio de ontologias informações de contexto coletadas de gerenciadores de sensores. Conforme seu projeto arquitetural, CROC visa à manutenabilidade e reusabilidade de código, aplicando padrões de projeto e decisões de projeto que busquem melhorias identificadas na revisão sistemática. CROC utiliza-se de padrões de projeto e de decisões de projeto como componente central de comunicação e arquitetura em camadas.
2. **Solução arquitetural para o serviço de agregação ontológica de contexto** — CASC possui a responsabilidade de agregar as informações representadas por dois ou mais CROC, por meio de ontologias que explicitem esse agrupamento.

Conforme seu projeto arquitetural, CASC visa à manutenabilidade (para gerenciar as ontologias), aplicando-se para isso padrões e decisões de projeto identificados na revisão sistemática deste trabalho. Utilizou-se de padrões de projeto para garantir a unicidade da agregação de informações de contexto e da arquitetura em camadas.

3. **Avaliação das soluções arquiteturais** — segundo os resultados obtidos com as métricas de acoplamento aferente e eferente, instabilidade, distância da sequência principal, falta de coesão e métodos ponderados por classe concluiu-se que os projetos deste trabalho se sobressaem de forma geral quanto aos projetos de terceiros ora analisados e comparados. O ponto de destaque do presente trabalho é a utilização dos padrões de projeto na solução arquitetural do software.
4. **Revisão Sistemática** — foram obtidos os padrões de projeto incluídos na literatura e mais utilizados pela literatura (*observer* e *prototype*) e a arquietura do software (em camadas), o que serviu de base para as soluções arquiteturais propostas neste trabalho.
5. **Coleta dos requisitos** — por meio de uma revisão sistemática e da leitura de artigos integrantes da literatura, foi possível estabelecer os requisitos necessários para o desenvolvimento dos projetos de coleta e representação ontológica de contexto e do projeto de agregação ontológica das informações de contexto.

As contribuições esperadas neste trabalho relacionam-se a constatação da necesidade de identificar e propor a estrutura para desenvolvimento de aplicações que atendam à fase de modelagem do ciclo de vida de contexto, identificando e propondo o uso de padrões de projeto, estilos arquiteturais e especificações de projeto que corroborem com o problema proposto. Sendo assim, os componentes CROC e CASC atenderam ao esperado e tornam-se referências para o desenvolvimento de serviços de representação e agregação de contexto modelado por ontologias.

6.2 Trabalhos Futuros

Os trabalhos futuros a serem realizados no projeto em análise são:

1. solução arquitetural para outros serviços de SSCS da etapa de modelagem de contexto (ex.: componente de qualidade de contexto *Hermes Quality* [41]);
2. solução arquitetural para serviços de outras fases do ciclo de vida (ex: raciocínio e disseminação de contexto);
3. métodos de avaliação de soluções arquiteturais que combinem as métricas de Martin [31] com ferramentas e análise de resultados;
4. implementação dos componentes. Dessa maneira é possível aferir testes com métricas que retornem possíveis melhorias e/ou erros não previstos no projeto;

5. criação de vários CROC para que possa ser verificada a real implantação do requisito de independência de domínio, o qual implicaria, também, verificar o pleno funcionamento do CASC com a agregação de várias informações de contexto;
6. criação de uma linha de produto de software e/ou *frameworks* para o desenvolvimento de CROC e CASC.

6.3 Considerações Finais

Esta dissertação apresentou o caminho a ser percorrido para a elaboração de componentes de software que atendam à coleta, representação e agregação de informações de contexto, para sistemas sensíveis ao contexto semântico. Com isso, o trabalho contribui para o desenvolvimento de novas aplicações e apresenta como resultado a importância do uso de padrões de projeto. O uso destes padrões estabelece uma perspectiva de desenvolvimento de sistemas que foquem em reusabilidade e manutenabilidade.

Com o presente trabalho propõe-se a submissão de artigos para disseminar os resultados aqui obtidos e compartilhar com a comunidade científica o caminho a ser seguido para a criação de novas aplicações na área de computação ubíqua.

Referências Bibliográficas

[1] ANQUETIL, N.; LAVAL, J. **Legacy software restructuring: Analyzing a concrete case**. In: *Software Maintenance and Reengineering (CSMR), 2011 15th European Conference on*, p. 279–286, March 2011.

[2] BETTINI, C.; BRDICZKA, O.; HENRICKSEN, K.; INDULSKA, J.; NICKLAS, D.; RANGANATHAN, A.; RIBONI, D. **A survey of context modelling and reasoning techniques**. *Pervasive and Mobile Computing*, 6(2):161 – 180, 2010.

[3] BIEMAN, J.; OTT, L. **Measuring functional cohesion**. *Software Engineering, IEEE Transactions on*, 20(8):644–657, Aug 1994.

[4] BOOCH, G.; JACOBSON, I.; RUMBAUGH, J. **UML Guia do Usuário**. Campus, 2 edition, 2006.

[5] CAPILUPPI, A.; BOLDYREFF, C. **Coupling patterns in the effective reuse of open source software**. In: *Emerging Trends in FLOSS Research and Development, 2007. FLOSS '07. First International Workshop on*, p. 9–9, May 2007.

[6] CELIKKAN, U.; KURTEL, K. **A platform for context-aware application development: Pcad**. In: *Computer Science and Information Systems (FedCSIS), 2015 Federated Conference on*, p. 1481–1488, Sept 2015.

[7] CHARLES, P. J.; KUMAR, S. B. R. **Design of a secure architecture for context-aware web services using access control mechanism**. In: *Contemporary Computing and Informatics (IC3I), 2014 International Conference on*, p. 780–784, Nov 2014.

[8] CHIDAMBER, S. R.; KEMERER, C. F. **A metrics suite for object oriented design**. *IEEE Transactions on Software Engineering*, 20(6):476–493, Jun 1994.

[9] DEISSENBOECK, F.; JUERGENS, E.; HUMMEL, B.; WAGNER, S.; Y. PARAREDA, B. M.; PIZKA, M. **Tool support for continuous quality control**. *IEEE Software*, 25(5):60–67, Sept 2008.

[10] DEY, A. K.; ABOWD, G. D.; SALBER, D. **A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications**. *Human-computer interaction*, 16(2):97–166, 2001.

[11] FENTON, N. **Software measurement: a necessary scientific basis**. *IEEE Transactions on Software Engineering*, 20(3):199–206, Mar 1994.

[12] FREY, S.; VAN HOORN, A.; JUNG, R.; KIEL, B.; HASSELBRING, W. **Mamba: Model-based software analysis utilizing omg's smm**. In: *Proceedings of the 14. Workshop Software-Reengineering (WSR '12)*, p. 37–38, May 2012. Also appeared in Softwaretechnik-Trends 32(2) (May 2012) 49-50.

[13] GAMMA, E.; HELM, R.; JOHNSON, R.; VLISSIDES, J. **Design Patterns - Elements of Reusable Object-Oriented Software**. Addison-Wesley, 1994.

[14] GAULTIER, B.; RAYANA, R. B.; BONNIN, J.-M. **Context Management systems applied to mobility**. *Intelligent Transport Systems Telecommunications,(ITST),2009 9th International Conference on*, p. 330–335, 2009.

[15] GRUBER, T. R. **A translation approach to portable ontology specifications**. *Knowl. Acquis.*, 5(2):199–220, June 1993.

[16] GUO, B.; ZHANG, D.; SUN, L.; YU, Z.; ZHOU, X. **icross: toward a scalable infrastructure for cross-domain context management**. *Personal and Ubiquitous Computing*, 17(3):591–602, 2013.

[17] HENDERSON-SELLERS, B. **Object-oriented Metrics: Measures of Complexity**. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

[18] HERVAS, R.; BRAVO, J. **COIVA : context-aware and ontology-powered information visualization architecture**. (September 2010):403–426, 2011.

[19] HONG, J. I.; LANDAY, J. A. **An architecture for privacy-sensitive ubiquitous computing**. In: *Proceedings of the 2Nd International Conference on Mobile Systems, Applications, and Services*, MobiSys '04, p. 177–189, New York, NY, USA, 2004. ACM.

[20] JACQUET, J. P.; ABRAN, A. **From software metrics to software measurement methods: a process model**. In: *Software Engineering Standards Symposium and Forum, 1997. Emerging International Standards. ISESS 97., Third IEEE International*, p. 128–135, Jun 1997.

[21] JAROUCHEH, Z.; LIU, X.; SMITH, S. **An approach to domain-based scalable context management architecture in pervasive environments**. *Personal and Ubiquitous Computing*, 16(6):741–755, 2012.

[22] KAUFMANN, R.; JANZEN, D. **Implications of test-driven development: A pilot study**. In: *Companion of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '03, p. 298–299, New York, NY, USA, 2003. ACM.

[23] KHAN, M. F. F.; SAKAMURA, K. **Context-aware access control for clinical information systems**. *2012 International Conference on Innovations in Information Technology, IIT 2012*, p. 123–128, 2012.

[24] KHAN, M. F. F.; SAKAMURA, K. **Fine-grained access control to medical records in digital healthcare enterprises**. In: *Networks, Computers and Communications (ISNCC), 2015 International Symposium on*, p. 1–6, May 2015.

[25] KIM, J. E.; BOULOS, G.; YACKOVICH, J.; BARTH, T.; BECKEL, C.; MOSSE, D. **Seamless Integration of Heterogeneous Devices and Access Control in Smart Homes**. *2012 Eighth International Conference on Intelligent Environments*, p. 206–213, 2012.

[26] LARMAN, C. **Utilizando UML e padrões - Uma introdução à análise e ao projeto orientados**, volume 3. Bookman, 2008.

[27] LEN BASS, PAUL C. CLEMENTS, R. K. **Software Architecture in Practice (2nd Edition)**, volume 2. Addison-Wesley Professional, 2003.

[28] MAKRIS, P.; SKOUTAS, D. N.; SKIANIS, C. **A survey on context-aware mobile and wireless networking: On networking and computing environments' integration**. *IEEE Communications Surveys Tutorials*, 15(1):362–386, First 2013.

[29] MARANHÃO, G. M.; SENE JÚNIOR, I. G.; BULCÃO NETO, R. F. **Anatomia de um interpretador de contexto semântico com suporte à notificação de eventos em tempo real**. In: *XX Simpósio Brasileiro de Sistemas Multimídia e Web*, João Pessoa-PB, Brasil, 2014.

[30] MARQUES, G. S.; BULCÃO, R. D. F. **Projeto arquitetural de sistemas sensíveis ao contexto semântico: uma revisão sistemática**. Technical report, UFG, 2015.

[31] MARTIN, R. C. **Agile Software Development, Principles, Patterns, and Practices**. Prentice Hall, 2 edition, October 2002.

[32] MEDAGLIA, C. M.; SERBANATI, A. **The Internet of Things**. p. 389–395, 2010.

[33] NOY, N. F.; MCGUINNESS, D. L. **Ontology development 101: A guide to creating your first ontology**. Technical report, 2001.

[34] ONGENAE, F.; CLAEYS, M.; KERCKHOVE, W.; DUPONT, T.; VERHOEVE, P.; TURCK, F. **A self-learning nurse call system femke**. *Computers in Biology and Medicine*, p. 110–123, 2014.

[35] P. BOURQUE AND R.E. FAIRLEY. **Guide to the Software Engineering Body of Knowledge**. 2014.

[36] PERERA, C.; ZASLAVSKY, A.; CHRISTEN, P.; GEORGAKOPOULOS, D. **Context aware computing for the internet of things: A survey**. *IEEE Communications Surveys Tutorials*, 16(1):414–454, 2014.

[37] POSHYVANYK, D.; MARCUS, A. **The conceptual coupling metrics for object-oriented systems**. In: *2006 22nd IEEE International Conference on Software Maintenance*, p. 469–478, Sept 2006.

[38] PRESSMAN, R. S. **Engenharia de Software**, volume 7. ?, 2011.

[39] SALVANESCHI, G.; GHEZZI, C.; PRADELLA, M. **Context-oriented programming: A software engineering perspective**. *Journal of Systems and Software*, 85(8):1801–1817, Aug. 2012.

[40] SÁNCHEZ, P.; MOREIRA, A.; FUENTES, L.; ARAÚJO, J. A.; MAGNO, J. **Model-driven development for early aspects**. *Information and Software Technology*, 52(3):249–273, Mar. 2010.

[41] SENA, M. V. O. **Uma solução para qualidade de contexto baseada em ontologia e lógica nebulosa com aplicação em monitoramento de sinais vitais em uti**. Master's thesis, Universidade Federal de Goiás, 2016.

[42] SHAMS, H.; ZAMANIFAR, K. **MVCC: An Architectural Pattern for Developing Context-aware Frameworks**. *Procedia Computer Science*, 34:344–351, 2014.

[43] SOMMERVILLE, I. **ENGENHARIA DE SOFTWARE**, volume 9. PEARSON BRASIL, 2011.

[44] VEIGA, E. F.; MARANHAO, G. M.; NETO, R. F. B. **Development and scalability evaluation of an ontology-based context representation service**. *IEEE Latin America Transactions*, 14(3):1372–1379, March 2016.

[45] VEIGA, E. F. **Serviços de representação, agregação e histórico de contexto ontológico em apoio ao desenvolvimento de aplicações**. Master's thesis, Universidade Federal de Goiás, 2016.

[46] VEIGA, E. F.; BULCÃO, R. D. F. **Uma revisão sistemática sobre a etapa de modelagem do ciclo de vida do contexto baseado em ontologias**. Technical report, UFG, 2015.

[47] VESCOUKIS, V.; DOULAMIS, N.; KARAGIORGOU, S. **A service oriented architecture for decision support systems in environmental crisis management**. *Future Generation Computer Systems*, 28(3):593–604, Mar. 2012.

[48] WEISER, M. **The computer for the 21st century**. *Scientific American*, 265(3):66–75, 1991.

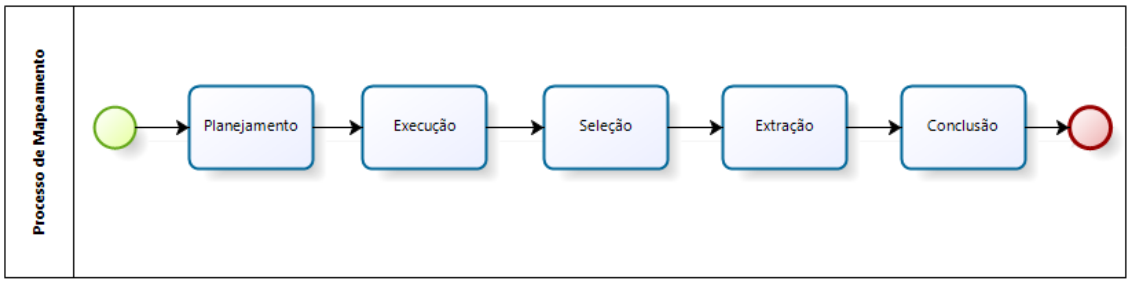
[49] ZAPPIA, I.; PARLANTI, D.; PAGANELLI, F. **Lisep: A lightweight and extensible tool for complex event processing**. In: *Services Computing (SCC), 2011 IEEE International Conference on*, p. 701–708, July 2011.

APÊNDICE A

Relatório da revisão sistemática

Neste apêndice, é apresentada a revisão sistemática da literatura mediante a qual se analisou o estado da arte de sistemas sensíveis ao contexto semântico (SSCS). O principal foco foi levantar as características da aplicação de padrões de projeto e estilos arquiteturais na área de SSCS. A Figura A.1 apresenta o protocolo utilizado para a realização da revisão sistemática, e logo abaixo é feita a decrição de cada etapa.

Figura A.1: *Protocolo da revisão sistemática*



Planejamento: esta etapa consistiu na definição do protocolo que seria utilizado e que nortearia as etapas seguintes. Dentre as etapas, estão (a) definição da *string* de busca; (b) definição das questões de pesquisa; (c) fontes de pesquisa.

Execução (ou Coleta): esta etapa diz respeito à (1) execução das *strings* de busca em cada uma das máquinas de busca selecionadas (bibliotecas digitais que permitem busca com expressões) e (2) à construção de uma base de dados em planilha eletrônica, contendo os artigos coletados durante a busca.

Seleção: esta etapa constou da seleção dos artigos da base de dados e de leitura, do título e resumo de cada artigo, tomando-se como parâmetros os critérios de inclusão e exclusão definidos na fase de Planejamento. Os artigos, resultantes desta fase, foram lidos por completo, pois têm relação direta com o foco do estudo.

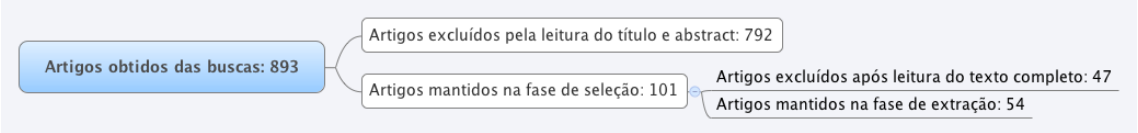
Extração: esta etapa consiste em proceder a leitura do texto, na íntegra, de cada trabalho, obtendo-se, desse modo, informações mais detalhadas sobre o trabalho. Com essa atividade, foi possível decidir se o artigo permanece na base de dados ou se é

excluído por não satisfazer os critérios estabelecidos. Além dos critérios, ALUNO, ORIENTADOR e/ou CO-ORIENTADOR, excluiu-se, caso necessário, trabalhos que não atendiam aos objetivos da pesquisa.

Conclusão/Sumarização: nesta etapa, ALUNO, ORIENTADOR e/ou CO-ORIENTADOR foram analisados e elaborados relatórios e gráficos com base nos resultados encontrados, individualmente pelas máquinas de busca, pelos status de cada artigo e demais atributos de cada destes.

Valendo-se do protocolo apresentado, uma revisão sistemática foi conduzida. Está detalhada nas sessões seguintes deste trabalho. A figura A.2 apresenta alguns resultados quantitativos de artigos filtrados ao longo da execução das etapas do protocolo.

Figura A.2: *Quantitativo de artigos por etapa*



A.1 Fase de Planejamento

Existem diferentes estruturas, estilos e padrões de projeto que são aplicados ao Projeto Arquitetural de Sistemas Sensíveis ao Contexto Semântico (SSCS). Sendo assim, o objetivo da revisão sistemática aqui realizada foi identificar e analisar estudos na área de arquitetura e estrutura de software aplicados a SSCS. Questões de pesquisa abrangeram perguntas com as quais se pudesse extrair o máximo de informação relevante dos artigos lidos, para o tema de pesquisa. Foram formuladas também questões de pesquisa que ajudariam o autor deste trabalho a prosseguir a revisão sistemática. Essas questões são enumeradas a seguir:

1. Quais os estilos arquiteturais aplicados no projeto arquitetural de SSCS?
2. Que tipos de artefatos são produzidos no projeto arquitetural de SSCS?
3. Quais requisitos não funcionais influem no projeto arquitetural de SSCS?
4. Quais padrões internacionais são utilizados para a descrição do projeto arquitetural de SSCS?
5. Quais padrões de projeto “criacionais” são utilizados no projeto microarquitetural de SSCS?
6. Quais padrões de projeto estruturais são utilizados no projeto microarquitetural de SSCS?

7. Quais padrões de projeto comportamentais são utilizados no projeto microarquitetural de SSCS?
8. Que tipos de problemas são resolvidos pela adoção de padrões de projeto "criacionais"?
9. Quais são os tipos de problemas que são resolvidos pela adoção de padrões de projeto estruturais?
10. Quais são os tipos de problemas que são resolvidos pela adoção de padrões de projeto comportamentais?
11. Há interesse em explorar linhas de produtos de software para SSCS?
12. Quais componentes de SSCS são passíveis de reúso e/ou customização para o projeto de linhas de produtos de software?
13. Quais os desafios associados ao projeto arquitetural de SSCS?
14. Quais são as questões em aberto sobre projeto arquitetural de SSCS?
15. Quais ferramentas são utilizadas no projeto arquitetural de SSCS?

O próximo passo executado após o levantamento das questões foi a elaboração de uma *string* de busca para utilização nas bases de dados selecionadas. Levando em conta os objetivos da revisão e as questões de pesquisa levantadas, o autor concluíu a *string* de busca que seria utilizada, *(("design pattern" OR "architectural design") AND ("context-aware" AND semantic))*, pois, com base no foco de estudo e nas questões de pesquisa levantadas, o autor chegou à conclusão de que artigos que tivessem as expressões *design pattern* ou *architectural design* poderiam trazer conteúdo relevante para soluções ao problema abordado. Todos os artigos deveriam estar relacionados à área de computação sensível ao contexto e semântica.

A.2 Fase de Execução

Definiu-se como filtros de busca as publicações dos anos de 2009 a 2014 até o mês de março nos quais a *string de busca* seria aplicada nos metadados e texto completo dos trabalhos. Na primeira análise, a pesquisa apenas no título e resumo não retornou um número suficiente de artigos. Dessa forma os últimos 5 anos foram definidos como parâmetro porque o autor buscou o estado recente das pesquisas.

A.3 Fase de Seleção

Após as buscas nas bases de dados terem sido realizadas, a quantidade de artigos recuperados foi de 893. Destes 792 foram excluídos por algum dos critérios de exclusão. O principal motivo de exclusão de um artigo foi *Não tratam estudos de Projeto*

Arquitetural, como pode ser visto na Tabela A.1, pois os artigos tratavam de semântica e computação sensível ao contexto, porém na área de redes de computadores e dispositivos físicos(hardware). Outros 180 artigos foram excluídos pelo critério de *Não relacionados à Computação Sensível ao Contexto*. Mais 47 artigos foram rejeitados em face do critério de *Não se enquadram nos Tipos de Estudo*, pois não adequavam à área de computação sensível ao contexto, mesmo contendo a expressão da *string* de busca em seu texto. As Tabelas A.2 e A.4 contêm as principais fontes de pesquisa que continham artigos relevantes, sendo a *ACM Digital Library* a que retornou a maior quantidade de artigos.

Tabela A.1: *Quantidade de artigos excluídos por critério*

Critérios	Quantidade de Artigos
Não relacionados à Computação Sensível ao Contexto	180
Duplicados	169
Não tratam estudos de Projeto Arquitetural	317
Não exploram a semântica de informações de contexto	79
Não se enquadram nos Tipos de Estudos	47
TOTAL	792

Tabela A.2: *Quantidade final de artigos incluídos por base de dados*

Biblioteca	Quantidade de Artigos
ACM Digital Library	34
IEEE	26
Scopus	19
SPRINGER	17
Science Direct	5
TOTAL	101

A.4 Fase de Extração

Nesta seção, serão apresentados os resultados obtidos na etapa de extração do protocolo de revisão sistemática contido na figura A.1.

A.4.1 Dados da fase de extração

Após a etapa de seleção, restaram 101 artigos que foram lidos e analisados na íntegra. Como resultado dessa leitura e análise, mais 47 artigos foram excluídos, perante os critérios constantes na Tabela A.3, restando apenas 54.

Tabela A.3: *Quantidade de artigos excluídos por critério, após a leitura por completo*

Critérios	Quantidade de Artigos
Texto completo indisponível para acesso gratuito na Web, ou por meio do portal CAPES	27
Não relacionados à Computação Sensível ao Contexto	2
Não tratam estudos de Projeto Arquitetural	3
Não exploram a semântica de informações de contexto	5
Não se enquadram nos Tipos de Estudos	10
TOTAL	47

Na Figura A.3, é exibido um gráfico que mostra a quantidade de artigos incluídos na etapa de extração, agrupados por ano de publicação. Analisando-se o gráfico pode-se perceber que de 2009 a 2012 houve um crescimento nas publicações, em 2012 registrou-se o ápice deste crescimento. A partir de 2013, a quantidade de artigos publicados na área começou a decrescer, porém com uma queda menor em relação aos anos anteriores. Sendo assim, a tendência é que a quantidade de publicações se mantenha em um nível estável.

Mesmo havendo um decréscimo, no triênio 2012-2014 registrou-se um crescimento de 70% na quantidade de artigos publicados e incluídos, em comparação com o triênio 2009-2011. Vale ressaltar a revisão sistemática aqui apresentada levou em consideração apenas artigos publicados até março de 2014. Sendo assim, o número de publicações nesse ano ainda sofreu um aumento.

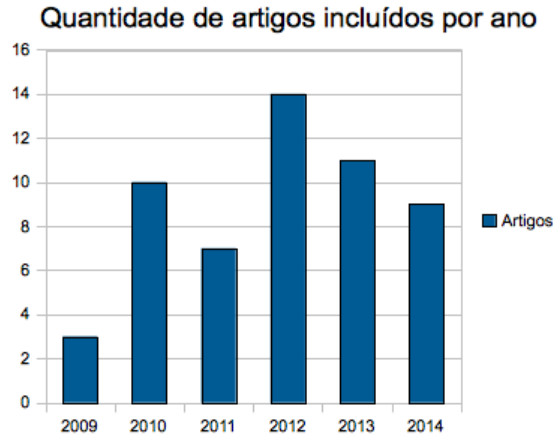


Figura A.3: *Quantitativo de artigos - extração*

Após uma triagem da leitura completa dos artigos, a proporção destes, por base de dados, foi alterada, passando *ACM Digital Library* e *IEEE* a somarem mais de 75% do total de artigos incluídos.

Assim encerra-se a fase de análise quantitativa (número de artigos incluídos neste

Tabela A.4: *Quantidade final de artigos incluídos por base de dados, após a etapa de seleção*

Biblioteca	Quantidade de Artigos
ACM Digital Library	22
IEEE	19
Scopus	2
SPRINGER	2
Science Direct	2
TOTAL	47

estudo) na realização dessa revisão sistemática para SSCS. Nas próximas seções, discorre-se sobre as informações obtidas nos artigos selecionados. Essas informações são utilizadas para responder às questões enumeradas na fase de planejamento deste trabalho.

A.4.2 Análise da fase de extração - Respondendo às questões de pesquisa

Neste tópico, são apresentadas as respostas às questões de pesquisa após a análise dos artigos selecionados. Nas subsessões seguintes, cada questão será analisada individualmente, com exceção das questões de 5 a 10, que são examinadas em uma única subseção, por se tratar de questões relacionadas a padrões de projetos.

Questão de pesquisa número 1 — *Quais são os estilos arquiteturais aplicados no projeto arquitetural de SSCS?*

Em 38 dos artigos encontrados foi possível extrair informações para responder esta questão. A referência utilizada para analise consiste nas especificações contidas em [35], em que, na seção 3.2. [35], os estilos arquiteturais são definidos como: *"uma especialização de elementos e tipos relacionados, juntos com um conjunto de restrições de como eles podem ser utilizados"*. Esses elementos são divididos em quatro categorias principais: Estruturas Gerais, Sistemas Distribuídos, Sistemas Interativos, Sistemas Adaptáveis e Outros. A Tabela A.5 exibe os exemplos de [35] estilos arquiteturais e suas respectivas classificações nas referidas categorias.

Mediante as informações obtidas da Tabela A.6, infere-se que, dos 38 artigos em que se extraíram informações, apenas 24 continham informações relevantes no tocante a estilos arquiteturais. Nesta análise percebe-se que o estilo arquitetural mais utilizado é o em camadas, pois trata-se de uma arquitetura bem difundida e bastante usada nos demais sistemas (não sensíveis ao contexto, como afirmado em [43]).

Tabela A.5: *Estilos Arquiteturais*

	Genereal Structure	Distributed System	Interactive Systems	Adaptabel Systems	Others
Layers					
Pipes and filters					
Blackboard					
Cliente-server					
Three-tiers					
Broker					
MVC					
Presentation-abstraction-control					
Microkernel					
Reflection					
Batch					
Interpreters					
Process control					
Rule-based					

Questão de pesquisa número 2 — *Que tipos de artefatos são produzidos no projeto arquitetural de SSCS?*

Esta questão de pesquisa objetiva a identificação dos artefatos produzidos para representar, de forma visual, as características dos sistemas, como por exemplo: gráficos, fluxogramas, diagramas, etc.

Artefatos são modelos de documentos, roteiros (rotinas), normas, diagramas, etc. que auxiliam na explanação e entendimento do funcionamento do software e definem suas características. Artefatos podem ser detalhados ou superficiais e agregam informações relevantes aos interessados no sistema.

Com base nas informações coletadas, deduz-se: *"não há uma preocupação com o uso de artefatos descritos em algum tipo de padrão. Pouco se vê de UML, definições e padrões da Object Management Group (OMG), ou algum padrão da International Organization for Standardization (ISO)."*

Tabela A.6: *Estilos Arquiteturais - ocorrência nos artigos*

	Genereal Structure	Distributed System	Interactive Systems	Adaptabel Systems	Others
Pipes and filters	1				
MVCC			1		
Broker		1			
MVC			2		
Publish-subscribe			1		
WS		2			
SOA	4				
Camadas(lay	12				
REST		1			
TOTAL	**17**	**4**	**4**	**0**	**0**

Questão de pesquisa número 3 — *Quais requisitos não funcionais influem no projeto arquitetural de SSCS?*

Para [43] o conceito de requisitos não funcionais compreende *"requisitos que não estão diretamente relacionados com os serviços específicos oferecidos pelo sistema a seus usuários."*. Estes requisitos podem afetar o desenvolvimento do sistema, uma vez que podem conter necessidades que impliquem alteração da estrutura e arquitetura do sistema. Ainda segundo [43], os requisitos não funcionais são divididos em três categorias: 1) requisitos de produto; 2) requisitos organizacionais; e 3) requisitos externos.

O requisito referenciado com maior frequência, dentre os identificados nesta revisão sistemática, é a **usabilidade (referenciado em 10 artigos)**. Os requisitos não funcionais identificados abrangem vários aspectos do sistema, desde a interação com o usuário (usabilidade e privacidade), a proteção dos dados do sistema (privacidade) e o esforço necessário para garantir o pleno funcionamento do sistema (manutenibilidade). Ao todo, 8 artigos desta revisão sistemática mencionam a manutenção do sistema como um ponto importante, para Vescoukis [47] "manutenibilidade é tipicamente um conceito primário em estilo arquitetural".

Questão de pesquisa número 4 — *Quais padrões internacionais são utilizados para a descrição do projeto arquitetural de SSCS?*

Esta questão não pôde ser respondida com a revisão sistemática, pois os artigos colhidos não trataram deste assunto.

Questão de pesquisa de número 5 a 10

Dentre os artigos incluídos na etapa de extração da revisão sistemática, 48% referenciam o uso de algum padrão de projeto expresso em Gamma et al. [13]. Os padrões de projeto objetivam criar modelos de generalização de soluções, evitando assim o redesenho de uma solução para cada novo processo de desenvolvimento de sistemas que tratem o mesmo problema. Como pelo menos metade dos artigos colhidos menciona padrões de projeto, deduz-se que há uma lacuna nos estudos de desenvolvimento dos SSCS.

Os padrões de projeto *Observer*, *Prototype* e *Decorator*, descritos em Gamma et al. [13] foram os mais utilizados nos artigos analisados. Na área de SSCS, estes padrões são utilizados para:

- **Observer** — padrão de projeto aplicado para atualizar instâncias de sistemas de tempo real, ou seja, para manter o sistema atualizado em relação ao que ocorre em seu ambiente. Este padrão foi utilizado para a construção de uma aplicação consciente de contexto em diferentes domínios, pois assim a aplicação *se mantém atualizada* do contexto, independente do domínio da aplicação, conforme expresso por [18].
- **Prototype** — é um padrão de projeto utilizado para a criação de novos objetos com base em um modelo original. Nos SSCS este padrão de projeto auxilia a criação de novas instâncias semelhantes para objetos diferentes.
- **Decorator** — padrão de projeto utilizado em SSCS para interceptar um objeto, em tempo de execução, e adicionar alguma informação relevante para o objeto a qual não poderia ser adicionada anteriormente, conforme é demonstrado em [39] e [25]. Sendo assim, é uma forma de atualização do comportamento do objeto em tempo de execução.

Outro fato analisado, após este trabalho de revisão sistemática, foi o de artigos não fazerem referências ao uso de mais de um padrão de projeto em um mesmo sistema ou software.

Questão de pesquisa número 11 — *Há interesse em explorar linhas de produtos de software para SSCS?*

Esta questão de pesquisa visa investigar se há adoção de linhas de produto de software (conjunto de sistemas que têm funcionalidades em comum para satisfazer a necessidade de um nicho de mercado) para SSCS. Desta forma, quatro artigos atenderam ao solicitado, inferindo-se, assim, que a literatura não possui muito interesse neste assunto, neste momento.

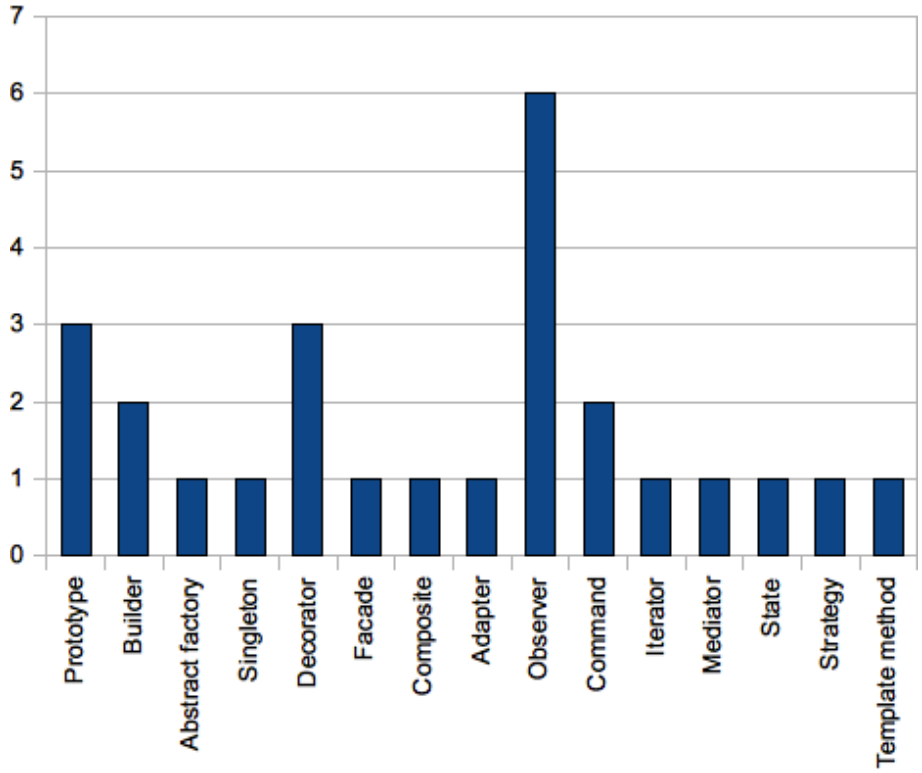


Figura A.4: *Ocorrência de padrões de projeto*

Questão de pesquisa número 12 — *Que componentes de SSCS são passíveis de reúso e/ou customização para o projeto de linhas de produtos de software?*

Em sete artigos, foi possível extrair informações para responder a esta questão de pesquisa. Esta questão objetiva identificar quais são os artigos que apresentam algum componente passível de reúso no desenvolvimento de novas aplicações. O reúso contribui para o desenvolvimento rápido, uma vez que não houve necessidade implementar todos os módulos do início, de sistemas que possuem características em comum. A utilização de componentes reusáveis é uma maneira de manter os sistemas SSCS interconectados quando necessário. A reutilização facilita a intercomunicação de sistemas e propaga uma melhoria na entrega dos serviços.

Os componentes que podem ser passíveis de reúso citados nos artigos e suas respectivas funções são:

- **Kernel** — padrão microkernel; gerencia os componentes;
- **System service** — descoberta de dispositivos; comunica com componentes remotos; permiti o uso de componentes locais por dispositivos remotos; monitora o estado dos componentes;
- **Context Middleware** — responsável por monitorar, detectar e analisar as mudanças de contexto;

- **Adaptation middleware** — é responsável por mudar o sistema em decorrência das mudanças de contexto, como as detectadas pelo *Context Middleware*;
- **Data collection engine** — integra as fontes de informação;
- **Rule engine** — um motor de regras do tipo *if-then-else;*
- **GUI engine** — interface de interação com o usuário; componentes de usuário;
- **Resource agent** — monitora as consultas de contexto;
- **Registration and Directory Service** — repositório primário para todos recursos descobertos;
- **Context Service** — avalia as informações de contexto;
- **Discovery Service** — identifica os recursos e cria uma lista dos recursos disponíveis;
- **Filter** — redução de ruído nas informações colhidas;
- **Aggregate** — agrega as informações de contexto, a fim de produzir novas informações.

Questão de pesquisa número 13 — *Quais os desafios associados ao projeto arquitetural de SSCS?*

Em 10 artigos foi possível extrair informações relevantes para responder a este questionamento. Com esta questão de competência, buscou-se identificar quais foram as maiores dificuldades e desafios enfrentados pelo desenvolvedores, ao criar suas aplicações de contexto semântico.

Os desafios encontrados nos artigos são:

- o tempo de latência no envio das informações; processa-las; e envia-las;
- integração com *middlewares*;
- utilização do tempo, pois as informações são vinculadas no tempo de excução, em que acontecem.

Questão de pesquisa número 14 — *Quais são as questões em aberto sobre projeto arquitetural de SSCS?*

Não foi possível responder a esta questão, dada a possibilidade de extrair dados suficientes para análise dos artigos.

Questão de pesquisa número 15 — *Quais são as ferramentas utilizadas no projeto arquitetural de SSCS?*

Em 10 artigos, extraíram-se informações para responder a esta questão de pesquisa. Desta forma foi possível identificar quais ferramentas os desenvolvedores

utilizaram ao fazer seus projetos e sistemas. Com isto, há um arcabouço de ferramentas que novos desenvolvedores poderão conhecer e passar a usar em seus projetos, com o intuito de melhorar os sistemas desenvolvidos.

As ferramentas identificadas são listadas:

- uso do Jboss JBPM para desenvolver fluxos de trabalho;
- uso de OWL, Jena e Pellet, SWRL;
- Protégé para OWL ontologia; spark; XMPP/Jingle; JMF;
- *Jesss inference engine for selecting appropriate learning items*;
- YAWL; BPEL;
- protégé que é utilizado para modelar a ontologia. As triplas RDF são armazenadas em AllegroGraph. Emprega-se *prolog* e *lisp* como *reasoner*, com a vantagem de serem configurados pelo Java. Utilizar-se também a ferramenta CPN.

Os dados obtidos permitem afirmar que as ferramentas e os arcabouços mais utilizados são o **JENA**, **OWL**, **SWRL**, **Protégé**.

A.5 Conclusão

Pode-se concluir que os padrões de projetos mais recorrentes são: *observer e prototype* e que os sistemas são desenvolvidos em camadas. Infere-se que não há um padrão de artefatos produzidos para expressar os SSCS. Os requisitos não funcionais mais utilizados são: usabilidade, segurança, privacidade e manutenibilidade. Não foi constatado interesse em desenvolver sistemas como linhas de produto de software para atender a determinadas demandas de usuários. Alguns desafios devem ser enfrentados e repensados, como o uso de SSCS de forma corriqueira e habitual no dia a dia das pessoas. Encontra-se disponível um arcabouço de ferramentas que podem ser utilizadas em projetos futuros, facilitando, assim, o desenvolvimento das novas aplicações.

Pelo fato de existirem diversas iniciativas para produzir conteúdos para SSCS, pode-se afirmar que não há um empenho em unir informações, fazendo com que estas ideias e iniciativas esparsas no ambiente de pesquisa. Torna-se necessário então fazer um agrupamento deste conteúdo para melhor aplicá-lo no desenvolvimento de novos sistemas e serviços voltados à computação sensível ao contexto.

Os benefícios esperados com a sugestão de soluções de projeto arquitetural para SSCS estão relacionados a:

- proposta de novos estilos arquiteturais e padrões de projeto para SSCS;
- adequação de estilos arquiteturais e padrões de projeto existentes para SSCS;

- macro e micro arquitetural: demonstração de uma arquitetura, alto nível, de como os componentes devem ser arquitetados para prover uma melhor qualidade no SSCS desenvolvido.

Desta maneira, será possível obter SSCS que se comuniquem, possam ser integrados, reutilizados e sejam manuteníveis, aumentando assim a utilização desta tecnologia no dia a dia das pessoas.

APÊNDICE B

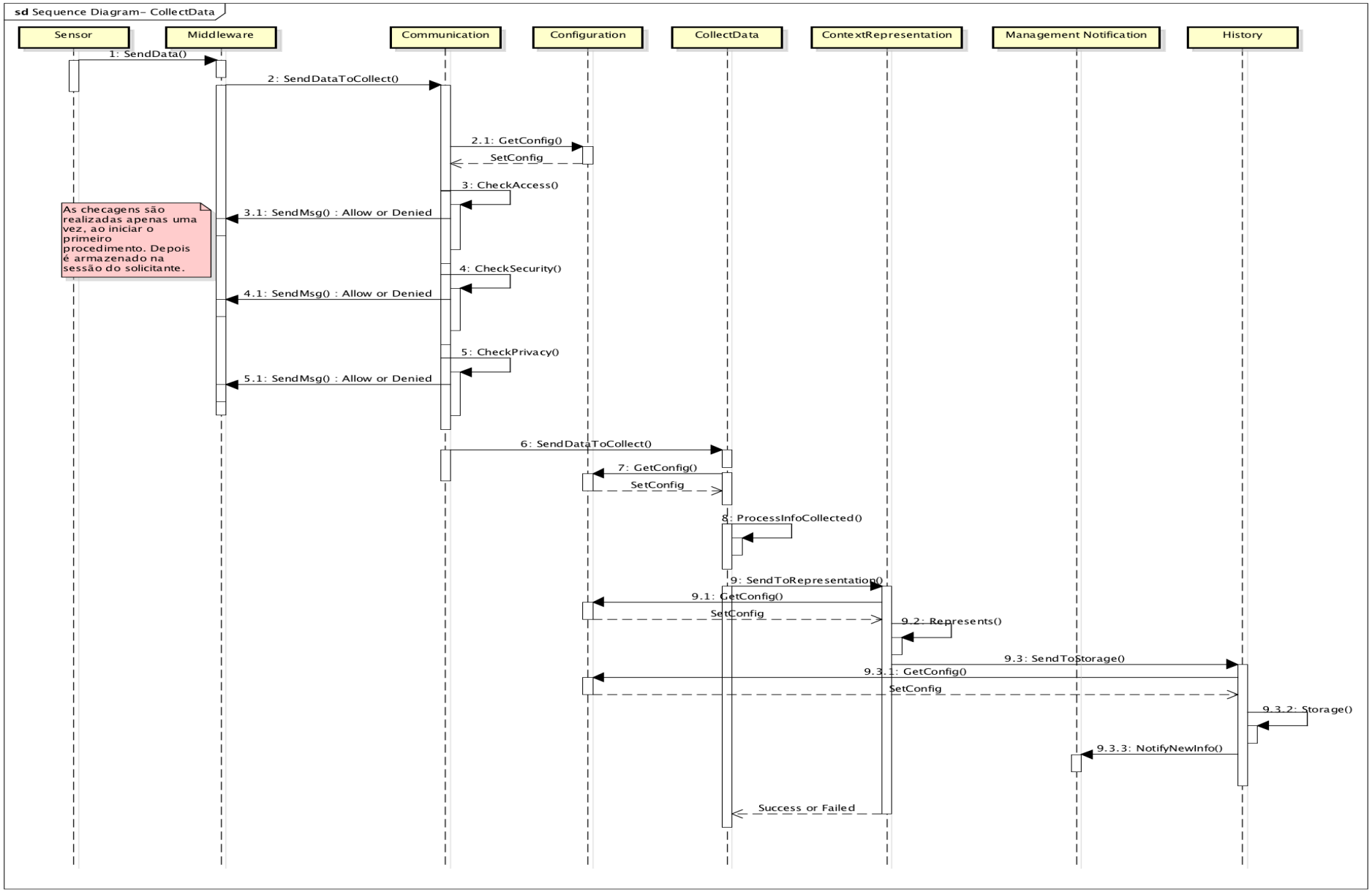
Descrição dos diagramas de sequência

Diagrama de sequência é um diagrama usado em UML, representando a sequência de processos em um programa de computador Larman [26]. Esse diagrama tem o objetivo de demonstrar o fluxo de mensagens em um sistema.

B.1 CROC - Diagrama de sequência

Com a visualização das mensagens de um sistema, é possível estabelecer o fluxo das informações, sendo assim, é possível identificar como os dados e informações trafegam no sistema e analisar a completude do software. A Figura B.1 apresenta o diagrama de sequência do componente em análise (CROC).

Figura B.1: *Diagrama de sequência da coleta de dados — CROC*

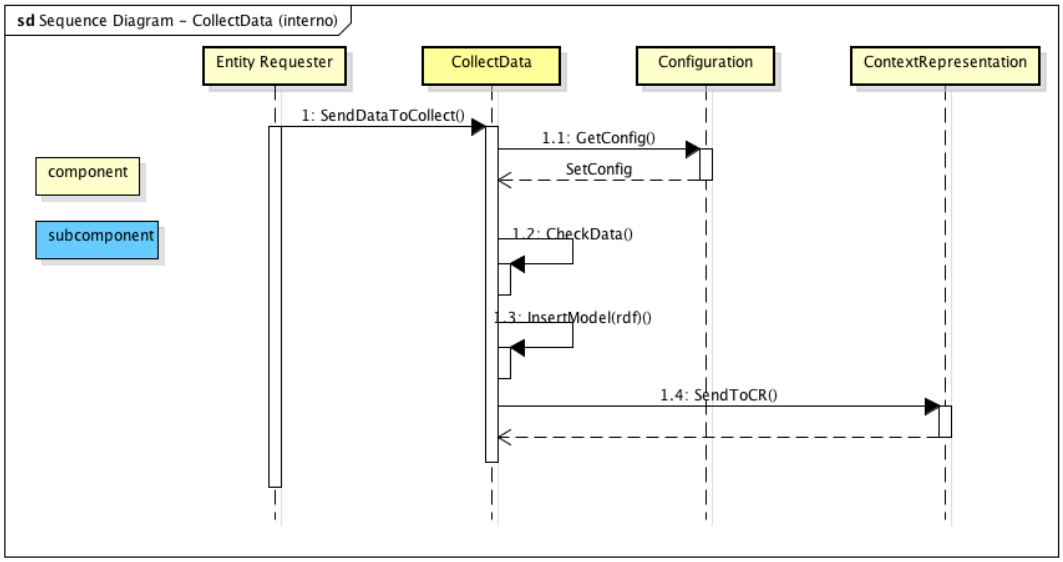


O diagrama de sequência do coletor de dados contém as seguintes entidades:

- ***Sensor***: realiza a aquisição dos dados dos sensores. Não faz parte do componente.
- **Middleware**: realiza o interfaceamento do sensor com o componente de coleta de dados e, também, não faz parte do componente.
- ***Communication***: representa a camada de comunicação da arquitetura.
- ***Configuration***: está contido na camada de comunicação e é responsável por gerir a configuração do componente.
- ***CollectData***: representa a camada de coleta, na arquitetura, responsável pela coleta dos dados adquiridos.
- ***ContextRepresentation***: demonstra a camada de representação na arquitetura, a qual é responsável por representar os dados coletados, utilizando uma ontologia de domínio.
- ***Management Notification***: representa a camada de notificação na arquitetura, e é responsável por enviar novas informações por meio de notificações a quem necessita recebê-las.
- ***History***: representa a camada de acesso aos dados e camada de dados, na arquitetura, responsável por gerir o armazenamento de informações representadas. É responsável, também, por instanciar a camada de notificação quando há novas informações armazenadas.

Para cada entidade, ou *linha do tempo*, há um fluxo de informações (mensagens) que trafegam entre os módulos. Na Figura B.2, o diagrama exibe a sequência realizada no processo de coleta de dados.

Figura B.2: *Diagrama de sequência da coleta de dados (interno) — CROC*



A parte que envolve os sensores e *middleware* não é tratada pelo CROC. O sensor faz a aquisição e envia a informação a um *middleware*, que faz todo o gerenciamento de sensores. A informação é, então, recebida no CROC por meio do subcomponente de comunicação.

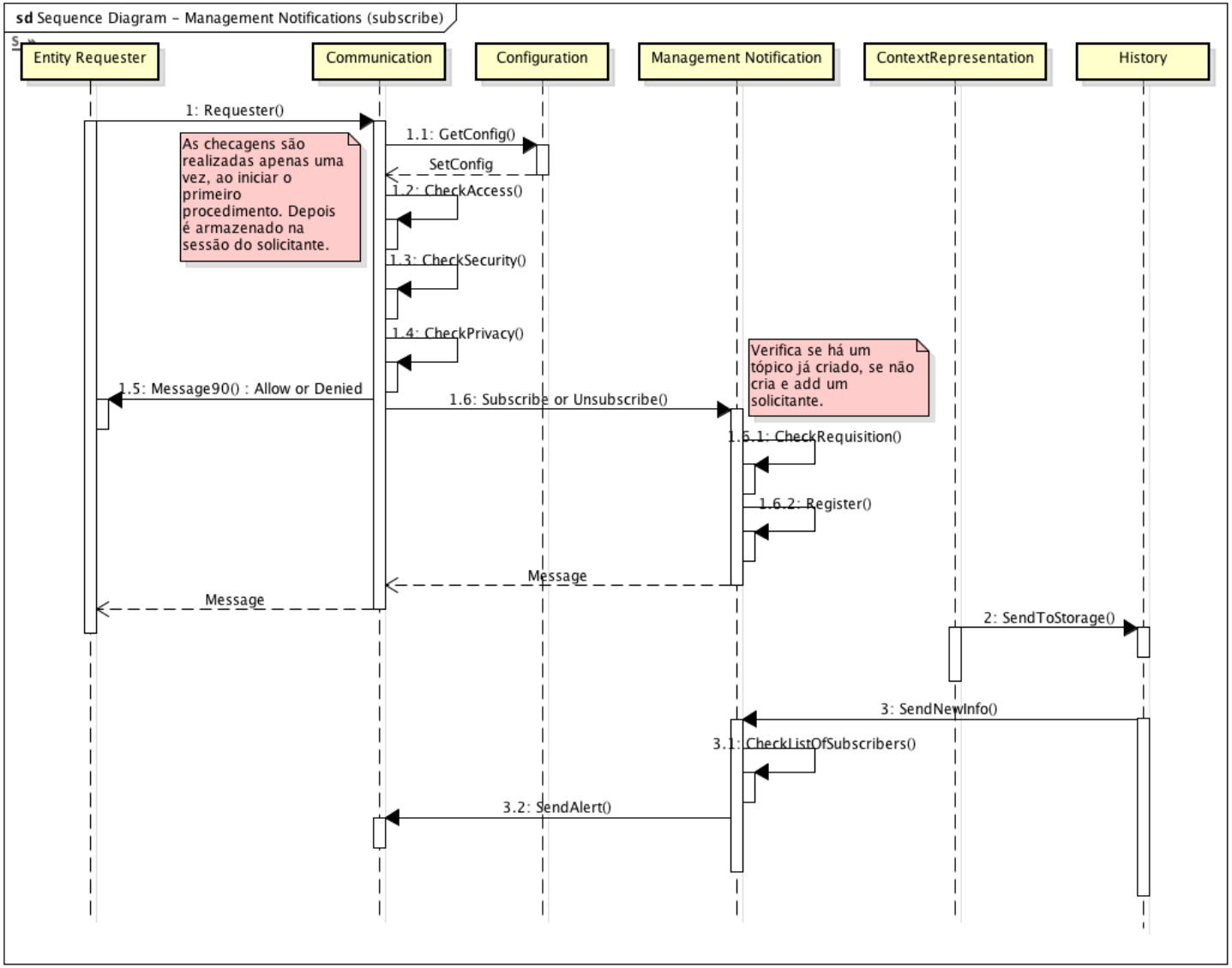
O subcomponente de comunicação recebe os dados para serem coletados (enviados ao *collect data*). Para prosseguir com a requisição, o componente de comunicação realiza as seguintes ações:

1. busca, no módulo de configuração, seus parâmetros de execução (2);
2. com o modulo de acesso, checa se a requisição possui as devidas permissões para fazer a ação solicitada (3);
3. com o módulo de segurança, valida a segurança das informações trafegadas, examina se não há vulnerabilidades e/ou algo que possa impactar a segurança do sistema (5);
4. verifica, pelo módulo de privacidade, se as informações trafegantes estão de acordo com as políticas de privacidade e podem ser distribuídas ao requisitante. Após efetuar todos esses procedimentos, a solicitação é enviada ao subcomponente de coleta de dados (6).

O componente de coleta de dados (*CollectData*) faz uma requisição ao Módulo de Configuração para obter os parâmetros necessários a sua execução (*path, type, permissions*, etc). Feito isso, o componente executa o processamento das informações colhidas (7), realizando uma consistência das informações, para que o próximo passo possa ser executado. Após todo esse processo, envia as informações coletadas para o SubComponente de Representação de Contexto.

O subcomponente de Representação de Contexto recebe as informações coletadas e realiza a representação dessas informações com uso de ontologia. A informação representada é enviada ao componente de histórico que realiza o armazenamento dessa informação. Para proceder ao armazenamento, o componente de histórico executa uma requisição ao componente de configuração para obter os parâmetros de conexão com a base de dados. O subcomponente realiza então o armazenamento e retorna à mensagem de sucesso ou falha para a Representação de Contexto (RC). A RC notifica o componente de Gerenciamento de Notificações, para que possa verificar se há uma nova informação e, em caso positivo, enviar um alerta a algum solicitante (este procedimento está representado no Sequence Diagram - Management Notifications (subscribe) Figura B.3). Após a realização de todos esses procedimentos, o coletor de dados é informado se as informações foram utilizadas com sucesso ou se ocorreu alguma falha.

Figura B.3: *Diagrama de sequência Gerenciador de Notificações — Assinatura*



O Diagrama de sequência do gerenciador de notificações (assinar) é composto pelas seguintes entidades:

- ***Entity Requester***
- ***Communication***
- ***Configuration***
- ***Management Notification***
- ***Context Representation***
- ***History***

Uma entidade requisitante faz uma solicitação para assinatura ou cancelamento de um alerta (tópico). Esta requisição é enviada para o Gerenciador de Notificações através do componente Communication.

Para prosseguir com a requisição, o componente Communication realiza as seguintes ações: 1) busca no componente de configuração seus parâmetros de execução; 2) mediante o módulo de acesso, verifica se a requisição possui as devidas permissões para fazer a ação solicitada; 3) pelo módulo de segurança, valida a segurança das informações trafegadas, se há vulnerabilidades e/ou algo que possa impactar a segurança do sistema; 4) valendo do módulo de privacidade, verifica se as informações trafegadas estão de acordo com as políticas de privacidade e podem ser distribuídas ao requisitante. Após realizar todos esses procedimentos, a solicitação é enviada ao componente de Gerenciamento de Notificações, para que seja realizada uma inscrição, no componente, ou cancelamento de uma inscrição já realizada.

O componente Management Notification (MN) recebe a solicitação de inscrição ou cancelamento e verifica qual é a ação requisitada (*CheckRequisition*). Depois de a ação do solicitante ser executada, ela é registrada e, neste momento, são armazenados e consolidados os tópicos assinados, a frequência de envio de informação, etc (*Register*).

Outra ação realizada pelo componente de Gerenciador de Notificações é a de notificar quando há novas informações e quem possui assinatura deste Gerenciador, para que receba esse tipo de informação. Para tanto, quando uma informação é coletada e processada pelo componente de *ContextRepresentation* e após o armazenamento da informação no módulo *History*, este componente informa ao MN que há uma nova informação no sistema.

O MN faz então uma varredura em sua lista de assinantes com a finalidade de verificar se algum deles tem interesse em receber essa nova informação. Identificando um assinante que deseja recebê-la, ela é enviada por intermédio de um alerta ao Componente de Comunicação que realiza a entrega para a entidade requisitante.

B.2 CASC - Diagrama de sequência

Para exemplificar o funcionamento do componente CASC é apresentado na Figura B.4 o diagrama de sequência. Este diagrama é composto pelas seguintes entidades Comunicação, Configuração, Agregação, Histórico e Notificação.

As entidades são descritas:

- **Comunicação** é responsável por gerir e iniciar as demais instâncias do componente. O componente é iniciado na comunicação, e sua primeira ação é carregar as configurações. Para isso, é realizada uma requisição ao módulo de configuração (1). As próximas ações são: verificar segurança (2); verificar privacidade (3); e recebimento dos dados do componente CROC. De posse dos dados coletados, é realizada a ação de enviar ao módulo de agregação (5).
- **Configuração** é responsável por gerir as informações necessárias para o pleno funcionamento do componente. Guarda informações, como: caminho da ontologia; detalhes da ontologia e informações de armazenamento.
- **Agregação** é responsável por buscar a ontologia a ser utilizada como modelo, para isso, faz uma requisição à Configuração (6). O próximo passo é a realização da agregação de fato (7). Nesse instante, as informações coletadas no CROC são agrupadas de acordo com o modelo ontológico. E, por fim, as informações agregadas são enviadas para o histórico (8).
- **Histórico** é responsável por gerir novas informações e guardar as informações processadas. Com o recebimento das informações agregadas (8), o histórico realiza o armazenamento (8.1) e envia os novos dados para a notificação (9).
- **Notificação** é responsável por enviar as novas informações agregadas a quem solicitou ser notificado. De posse das novas informações, é realizada a notificação (9.1).

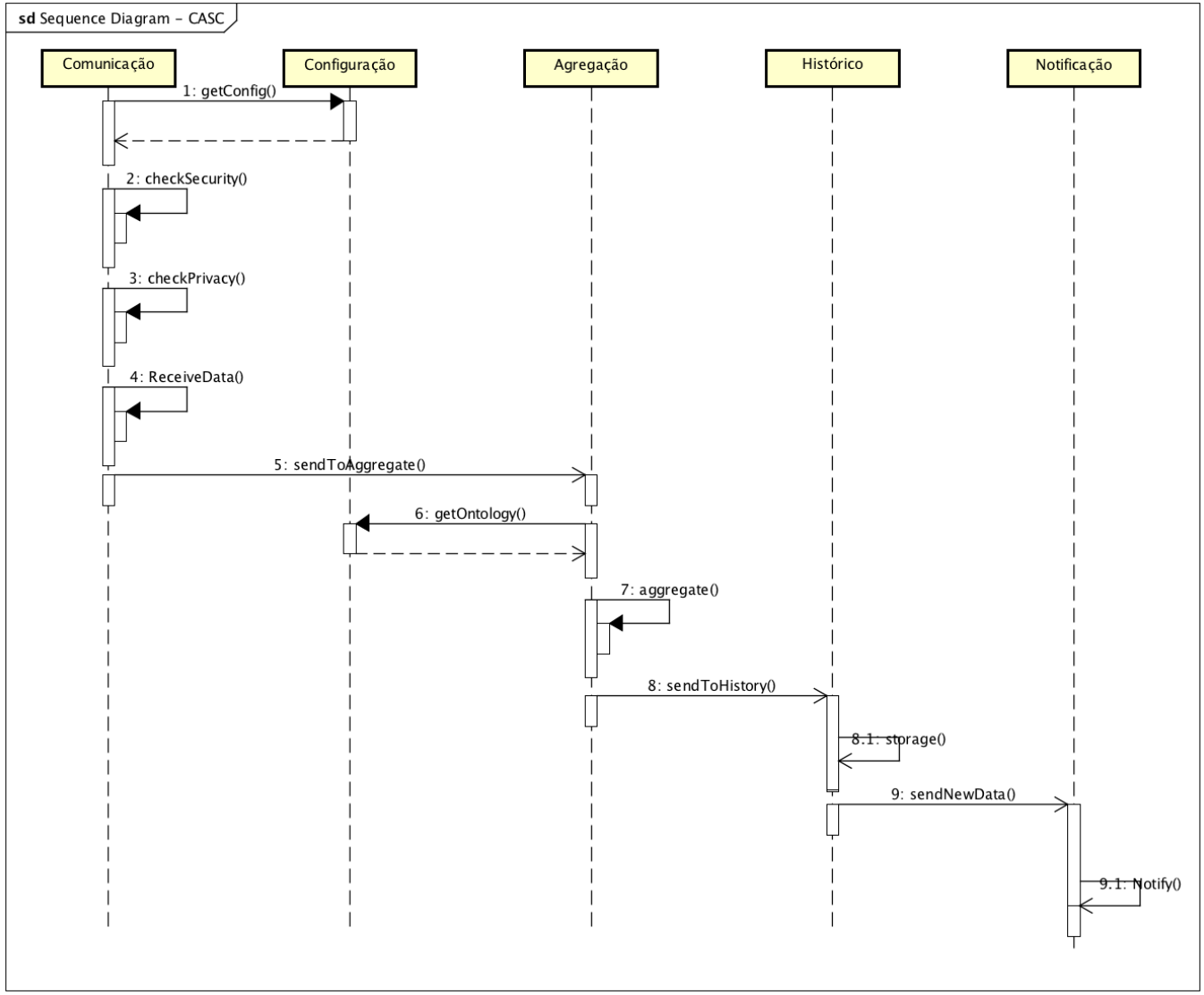



Figura B.4: *Diagrama de sequência — CASC*