

HIGOR BORJAILLE SONEGUETTI

SMAAS: SITUATION MANAGEMENT AS A SERVICE

VITÓRIA - ES

2016

HIGOR BORJAILLE SONEGUETTI

SMAAS: SITUATION MANAGEMENT AS A SERVICE

Monografia apresentada ao Curso de Ciência da Computação, Centro Tecnológico, Universidade Federal do Espírito Santo, como requisito parcial para a obtenção do título de Bacharel em Ciência da Computação.

Orientador: Prof^a. Dr^a. Patrícia Dockhorn Costa

VITÓRIA - ES

2016

**UNIVERSIDADE FEDERAL DO ESPÍRITO SANTO
CENTRO TÉCNOLÓGICO
DEPARTAMENTO DE INFORMÁTICA
COLEGIADO DO CURSO DE CIÊNCIA DA COMPUTAÇÃO**

HIGOR BORJAILLE SONEGUETTI

SMAAS: SITUATION MANAGEMENT AS A SERVICE

COMISSÃO EXAMINADORA

Prof^a. Dr^a. Patrícia Dockhorn Costa

Orientadora

Isaac Simões Araújo Pereira

Co-Orientador

Prof. Dr. José Gonçalves

Examinador

Sérgio Teixeira

Examinador

Dedico este trabalho a minha mãe Nagine, que sempre acreditou em mim e não importa a distância que algum dia nos separe, o nosso elo mãe e filho jamais será quebrado.

AGRADECIMENTOS

Ao grande Isaac, pois graças a sua organização tornou meu desafio de lidar com um tempo possível viável, sempre acreditando em mim e me apoiando em todos os passos que eu dei desde o dia em que nos conhecemos. Uma pessoa que eu tenho um imenso prazer em chamar de amigo.

A minha orientadora Prof^a. Dr^a. Patrícia Dockhorn Costa que aceitou me orientar e teve muita paciência comigo em relação a escrita e também sempre me dando suporte em organizar as minhas idéias, me encorajando em momentos de dificuldades.

A minha amiga Jordana que desde que nos conhecemos me tomou como filho e durante todos os anos de curso me mostrou como me organizar e sempre me fazendo melhorar em vários aspectos pessoais e profissionais.

Aos demais professores do Departamento de Informática, que sempre me proporcionaram uma ótima recepção e resolução de dúvidas e também pela paciência em tentar entender minha mente confusa.

“Tudo que existe ou existiu, começou com um sonho. “
Lavagirl.

SUMÁRIO

LISTA DE ILUSTRAÇÕES	VIII
LISTA DE CÓDIGOS UTILIZADOS.....	IX
RESUMO.....	X
ABSTRACT.....	XI
1 INTRODUÇÃO	13
1.1 OBJETIVOS	15
1.2 METODOLOGIA.....	15
1.3 ESTRUTURA	16
2 FUNDAMENTAÇÃO TEÓRICA.....	18
2.1 SITUATION AWARENESS	18
2.1.1 Contexto	18
2.1.2 Sensibilidade ao Contexto.....	18
2.1.3 Situações.....	20
2.1.3.1 Técnicas de detecção de Situações	20
2.2 DROOLS	22
2.2.1 Sistemas Baseados em Regras	22
2.2.1.1 Base de Conhecimento	23
2.2.1.2 Base de Regras	23
2.2.1.3 Máquina de Inferência.....	23
2.2.2 Drools Rule Language (DRL).....	24
2.2.3 Truth Maintenance System	25
2.3 PLATAFORMA SCENE.....	25
2.3.1 Modelo de Detecção de Situações.....	26
2.3.1.1 Modelo estrutural da máquina de situações	26
2.3.1.2 Comportamento de uma situação	27
2.3.1.3 Relações Temporais entre Situações	28
2.3.2 Implementação da plataforma SCENE	29
2.3.2.1 Arquitetura Interna	29
2.3.3 Situação de Relação Formal	30
2.3.4 Situação de Situações.....	31
2.3.5 Considerações	32
2.4 TECNOLOGIAS, CONCEITOS E PADRÕES UTILIZADOS.....	32

2.4.1	SaaS.....	32
2.4.2	PaaS.....	33
2.4.3	RESTful API	33
2.4.4	JavaScript Object Notation (JSON).....	34
2.4.5	JSON Schema.....	34
3	NOVA PLATAFORMA SCENE.....	37
3.1	MODELAGEM DE DOMÍNIO E SUAS SITUAÇÕES	37
3.2	GERÊNCIAMENTO DE CONTEXTO.....	38
3.2.1	Usabilidade.....	39
3.3	REQUISITOS	40
3.4	ATUALIZAÇÃO DA VERSÃO DO JBOSS DROOLS	41
3.4.1	Knowledge is Everything (KIE).....	41
3.4.2	Convenção e configuração tomadas.....	42
3.5	ARQUITETURA PROPOSTA.....	43
3.5.1	Modelo.....	43
3.5.2	Regras de SCENE	44
3.5.3	Aplicação SCENE	45
3.5.4	Gerenciamento de Situações	47
3.6	SCENE DRL.....	50
3.7	EXECUÇÃO E ASPECTOS COMPORTAMENTAIS	53
3.8	CONSIDERAÇÕES.....	54
4	SITUATION MANAGEMENT AS A SERVICE (SMAAS)	56
4.1	VISÃO GERAL	57
4.1.1	Funcionalidades	57
4.1.2	Clientes do Serviço SCENE	58
4.2	INTERAÇÃO DE CLIENTES COM O SERVIÇO	58
4.2.1	Inserindo aplicação	59
4.2.2	Inserir, atualizar e excluir contexto de uma Aplicação Cliente	60
4.2.3	Coletando informações do Serviço	61
4.2.4	Assinatura para receber informações do Serviço	63
4.3	ARQUITETURA INTERNA DO SERVICO	64
4.3.1	Play Framework	64
4.3.2	Estrutura e Implementação	65
4.3.2.1	Inserindo Aplicação	66

4.3.2.2	Gerenciando Dados de uma Aplicação.....	68
4.3.2.3	Gerenciando Assinaturas	69
5	ESTUDO DE CASO	73
5.1	MODELO	75
5.2	REGRAS DAS SITUAÇÕES	76
6	CONCLUSÕES	80
6.1	TRABALHOS FUTUROS	81
	REFERÊNCIAS.....	82
	APÊNDICE 1 - REQUISIÇÕES	85

LISTA DE ILUSTRAÇÕES

FIGURA 2.1 – APLICAÇÃO SENSÍVEL AO CONTEXTO (LIEBERMAN, 2000)	19
FIGURA 2.2 - TÉCNICAS BASEADAS EM ESPECIFICAÇÃO (YE, ET AL. 2011)	21
FIGURA 2.3 - MODELO ESTRUTURAL DA MÁQUINA DE SITUAÇÕES.....	27
FIGURA 2.4 - INSTÂNCIAS DE SITUAÇÃO NO TEMPO (PEREIRA, 2013)	28
FIGURA 2.5 - CORRELAÇÃO DE EVENTOS SITUACIONAIS (PEREIRA, 2013).....	29
FIGURA 2.6 - BASE DE REGRAS DE SCENE (PEREIRA, 2013)	30
FIGURA 2.7 - MODELO DA SITUAÇÃO DE RELAÇÃO FORMAL SITUATIONWITHINRANGE (COSTA, 2007)	31
FIGURA 2.8 - MODELO DA SITUAÇÃO DE SITUAÇÕES SITUATIONSWITCH (COSTA, 2007).....	31
FIGURA 3.1 - ARQUITETURA DA APLICAÇÃO	38
FIGURA 3.2 - ARQUITETURA KIE (DRL, 2016)	42
FIGURA 3.3 - MODELO DA NOVA ARQUITETURA DE SCENE.....	43
FIGURA 3.4 - ESTRUTURA DA APLICAÇÃO SCENE.....	46
FIGURA 3.5 - SCENE WORKING FLOW	53
FIGURA 4.1 - COMUNICAÇÃO CLIENTE-SERVIÇO	56
FIGURA 4.2 - INTERAÇÃO COM O SERVIÇO	59
FIGURA 4.3 - ARQUITETURA DO SERVIÇO SCENE	65
FIGURA 5.1 - MODELO DO ESTUDO DE CASO.....	75

LISTA DE CÓDIGOS UTILIZADOS

CÓDIGO 2.1 – REGRA EM DROOLS	23
CÓDIGO 2.2 - ESTRUTURA DE UMA REGRA DROOLS.....	24
CÓDIGO 2.3 - REGRA DA SITUAÇÃO SITUATIONWITHINRANGE.....	31
CÓDIGO 2.4 - REGRA DA SITUAÇÃO SITUATIONSWITCH	32
CÓDIGO 2.5 - EXEMPLO DE JSON.....	34
CÓDIGO 2.6 - EXEMPLO DE JSON SCHEMA	35
CÓDIGO 3.1 - EXEMPLO DE GERENCIAMENTO DE CONTEXTO USANDO SCENE.....	39
CÓDIGO 3.2 - USO DA KIESESSION	42
CÓDIGO 3.3 - REGRAS DE AVALIAÇÃO DE SITUAÇÕES	45
CÓDIGO 3.4 - CONSTRUTOR DA CLASSE SCENEAPPLICATION	46
CÓDIGO 3.5 - FUNÇÃO QUE INSERE O METAMODELO NA SESSÃO DE DROOLS	46
CÓDIGO 3.6 - SITUATIONCAST FUNÇÃO PUT.....	47
CÓDIGO 3.7 - FUNÇÃO PARA DETECÇÃO DE UMA SITUAÇÃO	48
CÓDIGO 3.8 - FUNÇÃO QUE CRIA E ATIVA A SITUAÇÃO.....	49
CÓDIGO 3.9 - FUNÇÃO QUE DESATIVA A SITUAÇÃO	50
CÓDIGO 3.10 - ESTRUTURA DRL VOLTADA À SCENE	51
CÓDIGO 3.11 – EXEMPLIFICAÇÃO DA ESTRUTURA DO CÓDIGO 3.10	52
CÓDIGO 4.1 - COMUNICAÇÃO ENTRE O PLAY FRAMEWORK E SCENE-CORE	66
CÓDIGO 4.2 - VALIDAÇÃO DO APLICAÇÃO RECEBIDA POR JSON	67
CÓDIGO 4.3 - VALIDAÇÃO DE DADOS DRL	67
CÓDIGO 4.4 - INSERÇÃO DE APLICAÇÃO	68
CÓDIGO 4.5 - VALIDAÇÃO DOS CAMPOS NECESSÁRIOS PARA GERENCIAMENTO DE DADOS	69
CÓDIGO 4.6 - DECLARAÇÃO DA CLASSE SUBSCRIPTION.....	70
CÓDIGO 4.7 - REGRA DE ASSINATURA	70
CÓDIGO 4.8 - REQUISIÇÃO PARA SERVIÇO DE ORIGEM.....	71
CÓDIGO 4.9 - QUERY PARA LOCALIZAÇÃO DO OBJETO SUBSCRIPTION	71
CÓDIGO 5.1 - REGRA DE DETECÇÃO DE AUMENTO DE TEMPERATURA.....	76
CÓDIGO 5.2 - REGRA DE DETECÇÃO DE AUMENTO DE DISTÂNCIA.....	77
CÓDIGO 5.3 - REGRA DE SITUAÇÃO DE SITUAÇÕES PARA DETECÇÃO DE AUMENTO DE TEMPERATURA E DISTÂNCIA AO MESMO TEMPO.....	78

RESUMO

SMAAS: SITUATION MANAGEMENT AS A SERVICE

Este trabalho tem por objetivo projetar e implementar um serviço de apoio ao desenvolvimento de aplicações sensíveis ao contexto, baseando-se em SCENE, uma ferramenta para especificação e identificação de situações por meio de regras. Ele considera as experiências e sugestões de trabalhos acadêmicos anteriores, relativos ao uso de SCENE na construção de cenários de sensibilidade ao contexto, reunindo os requisitos arquiteturais para uma solução que, por fim, desobriga o comprometimento do desenvolvedor com (i) plataforma, (ii) linguagem de programação, ou (iii) da compreensão dos mecanismos internos do SCENE, no processo de desenvolvimento de aplicações sensíveis a situações.

Palavras-chave: Contexto, Sensibilidade ao contexto, Detecção de Situações, JBoss Drools, SCENE, PaaS (Platform as a Service).

ABSTRACT

SMAAS: SITUATION MANAGEMENT AS A SERVICE

This work aims to design and implement a service to support the development of context-aware applications, based on SCENE, a framework for rule-based situation detection. It considers the experience and suggestions from previous academic work related to the use of SCENE in the design of context-aware scenarios, gathering the architectural requirements for a solution which releases the developer's from commit to (i) a certain platform (ii) a certain programming language, or (iii) to understand the inner mechanisms of SCENE in the development process of situation-aware applications.

Key words: Context, Context-awareness, Situation detection, JBoss Drools, SCENE, PaaS (Platform as a Service).

1 INTRODUÇÃO

1 INTRODUÇÃO

A crescente popularização dos dispositivos móveis impulsionou significativamente a busca por técnicas de desenvolvimento de aplicações sensíveis a contexto. Tais aplicações são capazes de basear-se no contexto de entidades que sejam relevantes para o usuário a fim de fornecer de forma autônoma serviços de interesse. Alguns exemplos desse tipo de aplicação podem ser encontrados em (Pichler, 2014) e (Rendon, 2016).

Com a evolução da tecnologia, sensores estão cada vez mais avançados e suas informações contextuais vêm ficando cada vez mais complexas, como por exemplo, a descrição do padrão do movimento de uma pessoa, da qual pode-se inferir que forma de locomoção a mesma está utilizando. Sendo assim, sensores são grandes fornecedores de informações que podem ser utilizadas por aplicações sensíveis a contexto. A esse conjunto complexo de informações podemos dar o nome de *situação*. Segundo (Costa, 2007), uma situação é um conceito composto cujos constituintes podem ser uma combinação de entidades e suas informações contextuais. Assim, situações são um estado particular de coisas que são de interesse para aplicações, objetivando oferecer soluções que permitam analisar, correlacionar e coordenar interações entre pessoas, informações, tecnologias (Jakobson, 2014). Aplicações estas que para fazerem o uso adequado de situações, precisam do devido suporte.

Considerando esse cenário, diversos trabalhos têm sido desenvolvidos. Trabalhos como (COSTA, ALMEIDA, et al., 2006), (COSTA, 2007) e (COSTA, ALMEIDA, et al., 2007) avançam no sentido de propor suporte conceitual e arquitetural para a detecção de situações e tornaram-se o alicerce para a construção de uma ferramenta acadêmica que implementa estes conceitos, chamada plataforma SCENE.

SCENE, como desenvolvido em (Pereira, 2013), é uma plataforma que auxilia o desenvolvimento de aplicações sensíveis a situação, oferecendo ferramentas para a especificação de tipos de situações e um ambiente de execução para controle de seu ciclo de vida, envolvendo a identificação de situação (identificação de tipos de situação), composição de padrões de situação complexos e, por fim, sua desativação. SCENE foi construído sobre o *JBoss Drools* (Drools,

2016), uma máquina de inferência baseada em regras, cujo módulo integrado de processamento de eventos complexos foi elevado para suportar nativamente sensibilidade a situações baseadas em regras.

Desde o seu desenvolvimento, a plataforma SCENE foi empregada em diversos trabalhos, como em (Costa, 2016) e (Moreira, 2015), e principalmente em cenários de saúde pública. A primeira versão de SCENE obteve bom reconhecimento da comunidade acadêmica, como pode ser observado pelo número de publicações geradas em torno deste tema. Contudo, como em qualquer tecnologia, SCENE precisa avançar. Graças à seus usuários e, naturalmente, de reflexões de seus próprios desenvolvedores, foi possível obter feedback reportando as maiores dificuldades no uso da plataforma, que giram em torno das seguintes funcionalidades: (i) modelar o domínio e suas situações; (ii) popular e refletir alterações externas na *working memory*; e (iii) notificar interessados sobre situações identificadas. Além disso, a plataforma não considerava aspectos de distribuição, como por exemplo, a coleta remota de informações de contexto por dispositivos móveis.

No entanto, com o visível crescimento da população de dispositivos móveis, houve uma exigência de uma forma diferenciada para o gerenciamento de contexto que levasse em consideração esse fator. Portanto, faz-se necessário o desenvolvimento de novas formas de inserção, atualização e exclusão de informações de contexto observadas remotamente. Além disto, dada a heterogeneidade tecnológica das plataformas móveis, deseja-se que as funcionalidades propostas sejam independentes de plataforma. Em adição, ao longo dos anos novos requisitos foram identificados e a plataforma SCENE precisa evoluir para incorporar novas funções que atendam a esses requisitos. Assim, foi feita uma reformulação da plataforma SCENE para dar suporte a essas novas funcionalidades e requisitos.

Após a reformulação da plataforma SCENE, em conjunto com o estudo de novas tecnologias que dão suporte a vários tipos de evolução, uma questão de pesquisa foi levantada: “Como SCENE seria capaz de interagir com outros sistemas sem que suas funcionalidades fossem comprometidas?”. Visando responder a essa pergunta acredita-se que a criação de uma nova arquitetura para a plataforma e a disponibilização de suas funcionalidades na forma de serviços a serem acessados

por aplicações de terceiros possa auxiliar os pesquisadores e estudantes a detectar eventos situacionais quase imperceptíveis a um ser humano em seus respectivos domínios de pesquisa e possa continuar a evolução da pesquisa de sensibilidade a contextos e situações. Para validar o funcionamento da nova arquitetura da plataforma SCENE, foi utilizado um estudo de caso referente ao domínio de monitoramento e controle de temperatura de frascos de Botox. Cabe dizer que o conhecimento do autor deste trabalho sobre o tema sensibilidade a contexto, adquirido ao longo de um projeto de Iniciação Científica, reforçou a motivação para desenvolver a evolução de uma ferramenta já amplamente utilizada neste domínio.

1.1 OBJETIVOS

O principal objetivo deste trabalho é apresentar a evolução da plataforma SCENE apresentada em (Pereira, 2013), de forma que seja considerada uma plataforma como Serviço (Stankov, 2010). Para isso, propõe-se uma implementação do modelo de detecção de situações que seja independente de plataforma e, particularizando este modelo de forma que consista numa categoria de serviços de computação em nuvem, que permita aos clientes desenvolver, executar e gerenciar aplicações sem a complexidade de criar e manter a infraestrutura normalmente associada ao desenvolvimento e lançamento de uma aplicação.

1.2 METODOLOGIA

A abordagem utilizada neste trabalho consistiu em:

- (i) Revisão Bibliográfica: O trabalho teve início com uma revisão bibliográfica acerca da área de sensibilidade a situações e da plataforma SCENE;
- (ii) Levantamento e Análise de Requisitos: Nesta etapa foi realizado o levantamento e análise dos requisitos visando estabelecer a gerência de situações como um serviço, além da proposta de refatorações necessárias à plataforma SCENE;
- (iii) Implementação e Testes da Ferramenta: Nesta etapa foi elaborado o projeto da ferramenta, além da implementação de um serviço de identificação de situações utilizando a nova versão de SCENE, atendendo aos requisitos

levantados em (ii). Foi também apresentado um estudo de caso que demonstra as facilidades oferecidas pela plataforma desenvolvida.

- (iv) Elaboração da Monografia: Nesta etapa foi realizada a escrita desta monografia.

1.3 ESTRUTURA

Neste capítulo foi apresentada uma introdução, contendo uma breve descrição do contexto em que o trabalho está inserido e a motivação que levou ao seu desenvolvimento. Além deste capítulo, a monografia possui outros cinco, a saber:

Capítulo 2 – Fundamentação Teórica: Apresenta os principais conceitos referentes à área de sensibilidade a situações. Também são apresentados conceitos gerais sobre sistemas baseados em regras e um aprofundamento nas plataformas *JBoss Drools* e *SCENE*.

Capítulo 3 – Plataforma SCENE: Apresenta uma introdução sobre a plataforma SCENE e uma nova formulação da plataforma, seus requisitos, arquitetura e mudanças em relação à plataforma antiga.

Capítulo 4 – Situation Management as Service: Apresenta as novas funcionalidades da plataforma SCENE, contribuições deste trabalho para a pesquisa de sensibilidade a situações.

Capítulo 5 – Estudo de Caso: Apresenta o estudo de caso utilizado para realização de testes na ferramenta.

Capítulo 6 – Considerações Finais: Apresenta as considerações finais do trabalho, contribuições e experiência adquirida em seu desenvolvimento e também são identificados alguns trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo os principais conceitos e tecnologias utilizadas no desenvolvimento do trabalho. A seção 2.1 apresenta o conceito de situação, sensibilidade ao contexto e técnicas de detecção de situações. A seção 2.2 apresenta o conceito de sistemas baseados em regras com foco no Drools que foi a máquina de inferência de regras adotada no trabalho. A seção 2.3 apresenta a plataforma SCENE que serviu base para a proposta que, na prática, é uma evolução do SCENE proposto por Pereira (Pereira, 2013), por fim, a seção 2.4 apresenta tecnologias, conceitos e padrões utilizados na proposta.

2.1 SITUATION AWARENESS

2.1.1 Contexto

Quando ocorre um diálogo entre as pessoas, o uso de informações situacionais (contexto), ocorre de forma transparente e implícita, aumentando a qualidade do diálogo. Entretanto, o uso de métodos computacionais ainda não são capazes de fazer o máximo aproveitamento do contexto em um diálogo entre uma pessoa e um computador. Recolher informações contextuais através de meios automatizados pode facilitar o processamento destas informações. Nesse caso, o designer pode decidir quais informações são relevantes e a melhor forma de utilização, buscando um melhor entendimento do que é contexto.

De acordo com (Dey, 1999), “Contexto é qualquer informação que pode ser usada 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 próprio usuário e as próprias aplicações”.

2.1.2 Sensibilidade ao Contexto

Uma categorização de tipos de contexto ajuda os desenvolvedores a descobrir as partes mais prováveis do contexto que serão úteis em suas aplicações. As aplicações sensíveis ao contexto examinam o que o usuário está fazendo através

de suas entidades e usam essas informações para determinar porque a situação está ocorrendo.

De acordo com (Dey, 1999), um sistema é sensível ao contexto se usa o contexto para fornecer informações e / ou serviços relevantes ao usuário. A relevância da informação depende da tarefa.

A Figura 2.1 (Lieberman, 2000) ilustra como a informação contextual pode ser utilizada para prover informações ou serviços para uma aplicação sensível ao contexto. Uma aplicação pode, por exemplo, utilizar a localização do usuário para que seja possível identificar a distância que ele está de um ponto inicial prefixado.

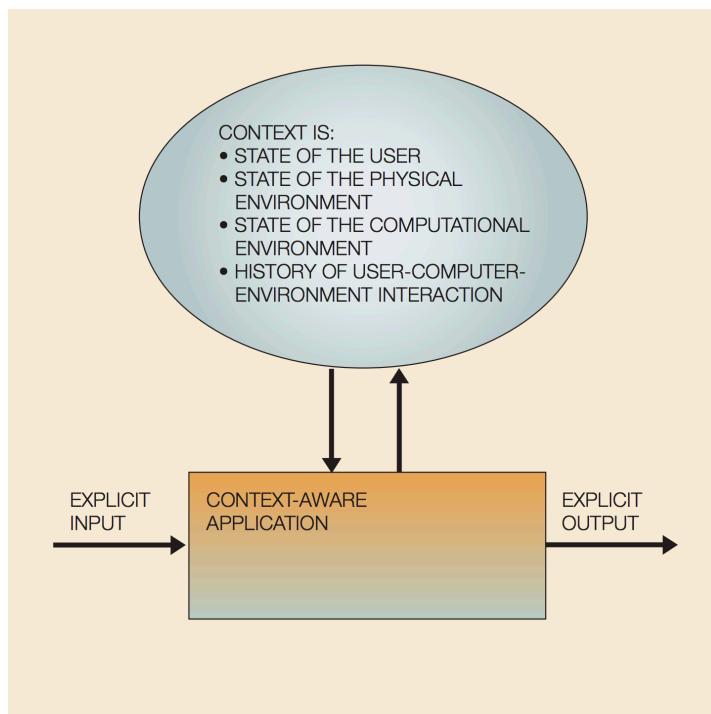


FIGURA 2.1 – APLICAÇÃO SENSÍVEL AO CONTEXTO (LIEBERMAN, 2000)

Com o propósito auxiliar na definição de aplicações sensíveis ao contexto, (Dey, 1999) apresentou uma categorização de acordo com as características das aplicações sensíveis ao contexto. Uma aplicação que recupera informações automaticamente, com base no contexto disponível, é classificada como reconfiguração automática de contexto, sendo uma técnica a nível de sistema que cria uma vinculação automática a um recurso disponível com base no contexto atual, ou seja, se o contexto está mudando rapidamente, pode ser uma distração para o usuário ou a adaptação a cada mudança pode ser impraticável.

Essa categorização de aplicações sensíveis ao contexto nos fornece dois principais benefícios. A primeiro especifica os tipos de aplicações para os quais a arquitetura deve fornecer suporte. O segundo mostra os tipos de recursos que devem ser planejados ou definidos antes de desenvolver qualquer aplicação sensível ao contexto.

2.1.3 Situações

Uma situação pode ser definida como uma interpretação semântica externa de dados, ou seja, atribuem algum significado aos mesmos. Segundo a perspectiva de aplicações, temos uma atribuição de significado aos dados tendo base em estruturas e relações dentro do mesmo e entre diferentes tipos de dados, portanto, o simples estabelecimento de uma situação já é, por si só, uma informação determinante para afetar o comportamento de uma aplicação.

Segundo (Ye, 2011) uma situação pode ser definida ao coletar contextos relevantes, descobrindo correlações significativas entre eles, e rotulando-os com um nome descriptivo. O nome descriptivo pode ser chamado de descrição de uma situação, que na prática é a forma como uma pessoa define um estado de coisas na realidade. Uma expressão lógica de predicados de contexto correlacionados é definida como uma especificação lógica de uma situação. Por exemplo, se a temperatura de uma pessoa está acima de 37 graus, então é atribuída uma situação de febre para essa pessoa.

As aplicações sensíveis a situação (*situation-aware*) têm a capacidade de estimar uma situação atual do usuário e reagir adequadamente, podendo interagir e aprender a partir do comportamento e ações, adaptando-se ao contexto situacional corrente do usuário.

2.1.3.1 Técnicas de detecção de Situações

Entre os fatores que dificultam a identificação de situações sendo possível destacar a imprecisão, incompletude e expressividade na descrição de relações e manipulação de situações. O estudo realizado em (YE et al, 2011) discute técnicas comumente aplicadas na detecção de situações na área da computação pervasiva,

à luz dos fatores citados previamente. As abordagens são classificadas entre dois supertipos: as técnicas baseadas em especificação e as técnicas baseadas em aprendizado.

As técnicas baseadas em especificação conforme ilustra a figura 2.2 se baseia na construção de modelos de situações com um conhecimento do especialista que pode racionalizar sobre as entradas de dados provenientes de sensores, interpretando-as. (YE et al, 2011) destaca o uso de Lógicas Formais, Lógicas Espaço-temporais, Teoria de Situações e Ontologias, como exemplos deste tipo de abordagem. Estendendo estas soluções, são citadas: a aplicação de Lógica Fuzzy e da Teoria de Evidência de Dempster-Shafer, como formas de lidar com determinados níveis de imprecisão, incompletude e inconsistência dos dados capturados via sensores.

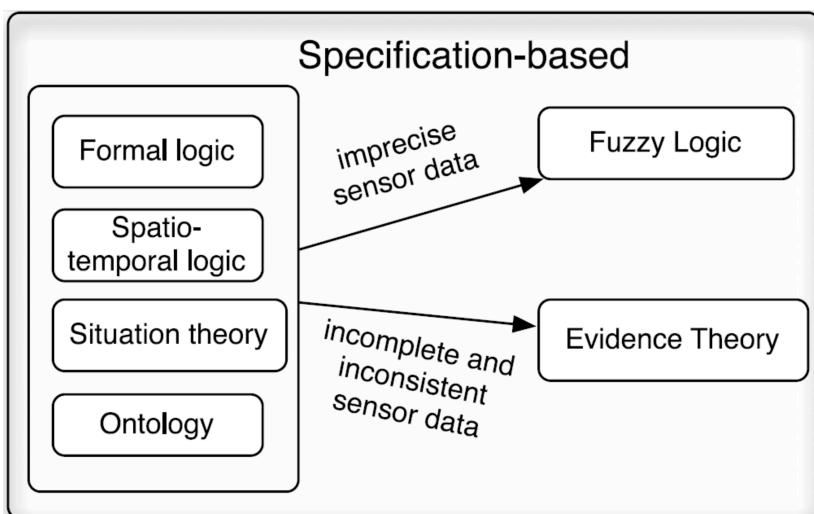


FIGURA 2.2 - TÉCNICAS BASEADAS EM ESPECIFICAÇÃO (YE, ET AL. 2011)

Como contraponto, temos as técnicas baseadas em aprendizado, motivadas por questões relacionadas à escalabilidade, complexidade e inconsistência (em alto grau) de sensores, que, como afirma (YE et al, 2011), “tornam as técnicas baseadas em especificação, dependentes de um a priori expert knowledge, impraticáveis”. Destacam-se, neste escopo, a aplicação de Redes Bayesianas, Hidden Markov Models (HMMs), ou Modelos Ocultos de Markov, Redes Neurais. Tais técnicas aplicam aprendizado de máquina para alcançar um refinamento no processo de

identificação, porém, exigem uma grande quantidade de treinamento e parametrização.

2.2 DROOLS

2.2.1 Sistemas Baseados em Regras

De modo geral, uma regra é uma diretriz que aponta um curso usual, costumeiro, ou generalizado de ação ou comportamento sob certas condições. Ao construirmos uma aplicação, estamos invariavelmente aplicando regras ao caracterizarmos seu comportamento, decidindo uma sequência determinística de ações caso determinadas condições sejam satisfeitas.

Em abordagens ditas tradicionais, i. e., abordagem estruturada e afins, esta caracterização se dá em meio à estrutura da aplicação, denotadas por declarações *if-else* da própria linguagem de programação. Tem-se então, arquitetura e comportamento fatalmente compartilhando as mesmas estruturas de código.

Um exemplo clássico de um sistema baseado em regras é um sistema especializado de domínio específico que usa regras para fazer deduções ou escolhas. Por exemplo, um sistema especialista pode ajudar um médico a escolher o diagnóstico correto com base em um conjunto de sintomas ou selecionar movimentos táticos para jogar um jogo.

A programação baseada em regras tenta derivar instruções de execução a partir de um conjunto inicial de dados e regras. Este é um método mais indireto do que o empregado por uma linguagem de programação imperativa, que lista os passos de execução seqüencialmente.

Sistemas Baseados em Regras (SBR) propõem uma separação clara entre estrutura da aplicação e suas regras de negócio, sendo composto por uma base de conhecimento (seção 2.2.1.1), base de regras (seção 2.2.1.2) e uma máquina de inferência (seção 2.2.1.3) focando no último aspecto, de maneira que a manutenibilidade das regras é uma vantagem imediata desta abordagem. Uma das vantagens dessa arquitetura é oferecer uma clara separação entre o conhecimento do domínio e o controle dos dados envolvidos no mesmo.

2.2.1.1 Base de Conhecimento

A Base de Conhecimento representa um espaço de memória usado para armazenar a coleção de fatos que serão utilizados pelas regras. A *working memory* é usada pelo mecanismo de inferência para obter fatos e combiná-los com as regras. Os fatos podem ser adicionados à *working memory* aplicando algumas regras. Por exemplo, supondo uma aplicação que mede a velocidade de carros e os caracteriza em grupos de risco, um fato pode ser adicionado se um carro ultrapassou uma velocidade de x km/h. O Código 2.1 mostra uma representação de uma regra em DRL (seção 2.2.2) cujo o RHS (seção 2.2.2) adiciona um fato.

```

01. rule "RiskGroup"
02.   when
03.     car: Car(speed > 100)
04.   then
05.     $car.setGroup("risk");
06. end

```

CÓDIGO 2.1 – REGRA EM DROOLS

2.2.1.2 Base de Regras

A base de regras contém todas as definições de regras aplicáveis, representando, dessa forma, todo o comportamento da aplicação. A base define de que forma o conhecimento armazenado na *working memory* deverá ser usado.

Uma regra é formada por uma tupla de precondição e consequência (se-então). A primeira, também conhecida como *Left Hand Side* (abreviadamente LHS), define as premissas impostas aos fatos para que a mesma seja satisfeita. A última, a *Right Hand Side* (RHS), é o conjunto de ações a serem executadas caso as precondições sejam atendidas. O Código 2.1 define uma regra em Drools, *rule* como o início da definição de uma regra, *when* como o LHS e o *then* como RHS.

2.2.1.3 Máquina de Inferência

A Máquina de Inferência decide quais regras são satisfeitas pelos fatos, e executa a regra com a maior prioridade. Existem dois tipos de inferência:

encadeamento progressivo e encadeamento regressivo. Encadeamento progressivo (*forward chaining*) é um raciocínio de fatos para a conclusão, enquanto encadeamento regressivo (*backward chaining*) é um raciocínio de hipótese para os fatos que suportam esta hipótese. Se um mecanismo de inferência executa encadeamento progressivo ou encadeamento regressivo depende inteiramente do projeto que por sua vez depende do tipo de problema.

Encadeamento progressivo é mais adequado para o prognóstico, monitoramento e controle. O encadeamento regressivo é geralmente utilizado para problemas de diagnóstico. O mecanismo de inferência opera em ciclos, executando um grupo de tarefas até que determinados critérios façam parar a execução. As tarefas a serem feitas repetidamente podem conter alguma resolução de conflito, a serem consideradas antes da verificação de parada. Múltiplas regras podem ser ativadas e colocadas na agenda durante um ciclo.

2.2.2 Drools Rule Language (DRL)

O Drools tem uma linguagem de regras "nativa" (DRL, 2016), denominada *Drools Rule Language* (DRL). Esse formato é muito leve em termos de pontuação e suporta linguagens específicas de domínios naturais através de "expansores" que permitem que a linguagem se altere para o seu domínio de problema.

O Código 2.2 apresenta a estrutura básica de uma regra na linguagem DRL.

```

01. rule "name"
02. attributes
03.   when
04.     LHS
05.   then
06.     RHS
07. end
  
```

CÓDIGO 2.2 - ESTRUTURA DE UMA REGRA DROOLS

A pontuação não é necessária, mesmo as aspas duplas para "name" são opcionais, assim como as novas linhas. Os atributos são sugestões simples (sempre opcionais) sobre como a regra deve se comportar. LHS é a parte condicional da

regra, que segue uma certa sintaxe que será abordada mais adiante. O RHS é basicamente um bloco que permite que o código semântico específico do dialeto seja executado.

O *Left Hand Side* (LHS) é um nome comum para a parte condicional da regra. Consiste em zero ou mais elementos condicionais. Se o LHS estiver vazio, ele será considerado como um elemento de condição que é sempre verdadeiro e será ativado uma vez, quando uma nova sessão *working memory* é criada.

2.2.3 Truth Maintenance System

É um componente do Drools responsável pelo controle de inserções lógicas, que consiste em atrelar a existência do fato lógico à validade das condições da regra que o produziu, desencadeando o recolhimento automático do fato. Este conceito é particularmente interessante, pois, se aproxima da abordagem para detecção do fim de uma situação, isto é, quando as invariantes da situação não são mais satisfeitas pelos componentes de uma instância. Por exemplo, para que uma pessoa esteja com febre, sua temperatura tem que ser maior ou igual a 37°C. O fato que caracteriza essa situação deve estar na *working memory*. Portanto, se o fato é atualizado com a informação de que a temperatura da pessoa é 36°C, significa que a condição que caracteriza a situação febre não é mais válida. Portanto, o fato sobre a febre da pessoa é automaticamente recolhido da *working memory*.

2.3 PLATAFORMA SCENE

SCENE é uma plataforma de apoio à especificação, detecção e gerência de tipos de situação, baseada no modelo de situações orientado a regras e composição de eventos, desenvolvido por Pereira (Pereira, 2013) com base no suporte conceitual e arquitetural sobre situações fornecido em trabalhos como (COSTA, ALMEIDA, et al., 2006), (COSTA, 2007) e (COSTA, MIELKE, et al., 2012). Este modelo representa uma especialização do modelo abstrato de sistemas baseados em regras clássicos. E, de forma análoga, no plano de realização, SCENE representa a especialização de uma plataforma de regras.

2.3.1 Modelo de Detecção de Situações

Segundo (ANAGNOSTOPOULOS, 2007) pela perspectiva ontológica, situações podem ser modeladas como conjuntos de conceitos agregados de diversas ontologias. Essas ontologias também são chamadas de fragmentos de contexto, pois cada um deles representa um fragmento das informações situacionais do usuário. Assim, uma situação é um conjunto de pedaços locais de contexto e cada pedaço local de contexto é um conjunto de conceitos. Há partes locais do contexto que descrevem, tempo, localização, etc.

Entende-se como tipo de situação um padrão que caracteriza um estado de interesse dentro do escopo de um determinado domínio, atuando sobre propriedades inerentes a fragmentos de contexto entre entidades deste universo. Cada entidade é caracterizada pelo papel que desempenham na situação. Portanto, febre pode ser caracterizada como um tipo de situação que define o estado ao qual a pessoa está com uma temperatura igual ou superior a 37°C. Dessa forma, a pessoa nessas condições está em estado febril. Diante disso, “febril” caracteriza um papel situacional do tipo de situação “febre”.

2.3.1.1 Modelo estrutural da máquina de situações

A Figura 2.3 apresenta uma visão geral do modelo estrutural da máquina de situações para a base de conhecimento, ou *working memory*, da plataforma de situações, de forma que, todo elemento incluído na base de conhecimento é, ou torna-se, invariavelmente um fato, ou, como o modelo define, a classe *FactType* provida por *JBoss Drools*. Ser um objeto do tipo *FactType* denota que a máquina de inferência está ciente de sua existência e o considera no processo de casamento de padrões de regras.

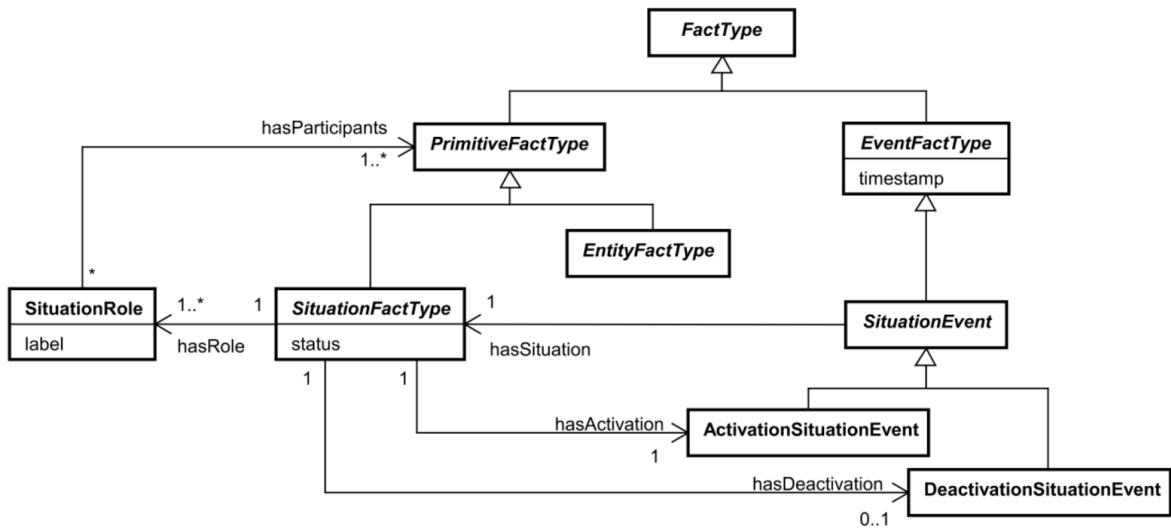


FIGURA 2.3 - MODELO ESTRUTURAL DA MÁQUINA DE SITUAÇÕES

Através da hierarquia do *FactType* podemos observar a classe *EventFactType*, que agrupa métodos de correlação temporal, e os tipos restantes, classificados como *PrimitiveFactTypes*. Essa distinção é importante para expressar corretamente quais tipos de fatos podem participar de um tipo de situação, sendo representado por um *SituationFactType*, cujos papéis situacionais são descritos através de *SituationRoles*, implicando em que *EntityFactTypes* ou o próprio *SituationFactType* desempenham um papel na situação. A temporalidade não é uma propriedade de *SituationFactType*, mas sim, de seus derivados eventos, como na ativação um fato sobre a classe *ActivationSituationEvent* é gerado e relacionado com a instância do *SituationFactType* e o mesmo ocorre com a desativação, na qual um fato sobre a classe *DeactivationSituationEvent* é gerado e relacionado a uma instância do *SituationFactType*.

2.3.1.2 Comportamento de uma situação

O comportamento de uma situação está inteiramente ligado à gerência de seu ciclo de vida, que consiste da identificação de um tipo de situação, instanciação e manutenção até a sua desativação, respectivamente. A instância de uma situação é a evidência do estabelecimento de um tipo de situação para um conjunto de indivíduos de um domínio específico, dentro de um intervalo de tempo contínuo. Os

extremos deste intervalo são o instante de ativação, quando a situação passa a ser válida, e a desativação, quando os requisitos que a tornam válida deixam de existir.

A figura 2.4 apresenta um gráfico de visão geral da instanciação de diferentes situações ao longo do tempo. As instâncias de situações são apresentadas no eixo X e o eixo Y representa diferentes combinações ou os momentos de ativação e desativação das situações em um determinado período. As coordenadas identificam o estado do domínio ou momento em que a situação é ou não satisfeita em um determinado instante de tempo. Simplificando, suponha, em um domínio limitado, no qual existe apenas um único atributo de temperatura relacionado somente a um único indivíduo, por exemplo, um tipo de situação **febre** é caracterizada quando sua temperatura atinge um valor maior ou igual à 37°C (área cinza), tornando este indivíduo **febril**. Durante o intervalo retratado na figura 2.4, ocorrem três instâncias de **febre**.

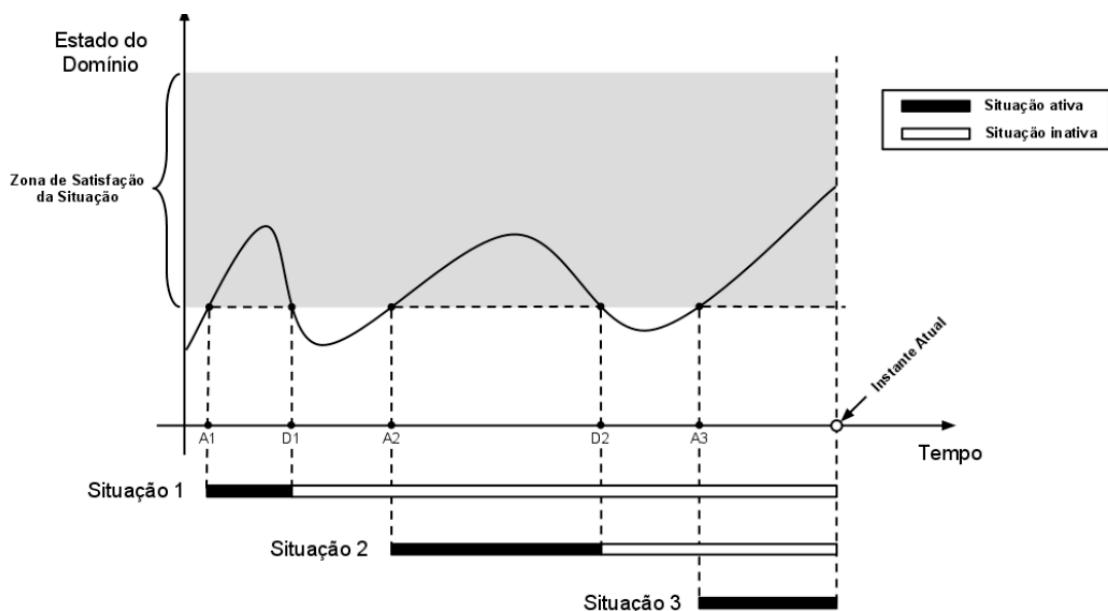


FIGURA 2.4 - INSTÂNCIAS DE SITUAÇÃO NO TEMPO (PEREIRA, 2013)

2.3.1.3 Relações Temporais entre Situações

As relações temporais entre situações derivam da abstração da álgebra temporal entre eventos. Analogamente a elas, têm-se treze operadores temporais sobre situações. No entanto, devido à dinâmica dos *SituationFactTypes*, ora ativos,

ora inativos, as correlações temporais sobre os mesmos, tem sua semântica afetada, dado que sua aplicabilidade é dependente do momento da avaliação. Por exemplo, supondo um par de situações ativas, A e B, respectivamente, avaliar se A ocorreu depois de B não é razoável, pois B nunca foi encerrada. Para os casos não aplicáveis, toda avaliação resulta em falso. A figura 2.5 apresenta treze tipos de relações temporais de situação, considerando o estado entre elas.

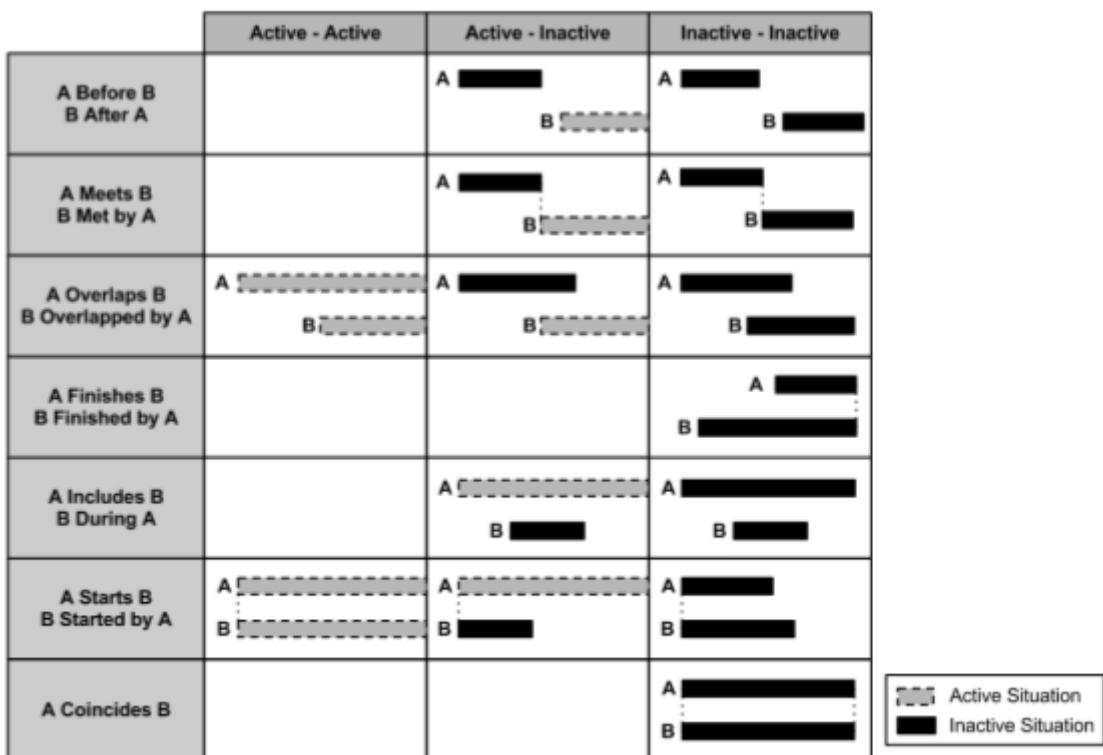


FIGURA 2.5 - CORRELAÇÃO DE EVENTOS SITUACIONAIS (PEREIRA, 2013)

2.3.2 Implementação da plataforma SCENE

A plataforma SCENE implementa um modelo de execução de detecção de situações e estendendo a plataforma *JBoss Drools* na forma de um módulo de gerência de situações para suporte ao desenvolvimento de aplicações sensíveis ao contexto.

2.3.2.1 Arquitetura Interna

A figura 2.6 apresenta uma visão geral da arquitetura interna de SCENE. A camada acessível pelo usuário “User Layer” contém as regras que compõem uma

situação. A camada inferior que não é acessível ao usuário contém as regras que dão suporte ao gerenciamento do ciclo de vida de uma situação conforme apresentado na seção 2.3.1.1.

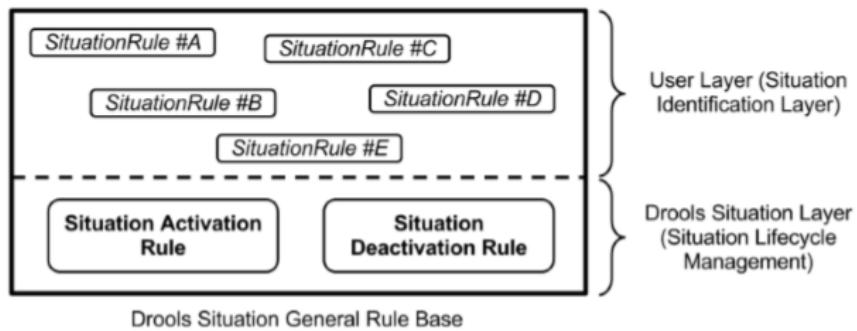


FIGURA 2.6 - BASE DE REGRAS DE SCENE (PEREIRA, 2013)

O ciclo de vida da situação é descrito por sua criação, ativação e desativação. A desativação pode não acontecer. Todas as situações desativadas são preservadas na *working memory* e podem compor uma situação complexa.

A situação permanece ativa enquanto as suposições tomadas que a tornaram ativa são ainda válidas, mas mesmo que a situação seja desativada ela permanecerá na *working memory* diferente de um fato inserido que é removido da *working memory* quando suas suposições se tornam falsas, pois a regra que cria a situação é interna a SCENE e não pela regra composta pelo usuário para a detecção da situação.

Nas próximas duas seções serão introduzidos os tipos simples e compostos de situações, com exemplos de cenários propostos por (COSTA, 2007).

2.3.3 Situação de Relação Formal

O cenário proposto por (COSTA, 2007) para representar as situações de relação formal, foi o de proximidade entre entidades de um espaço.

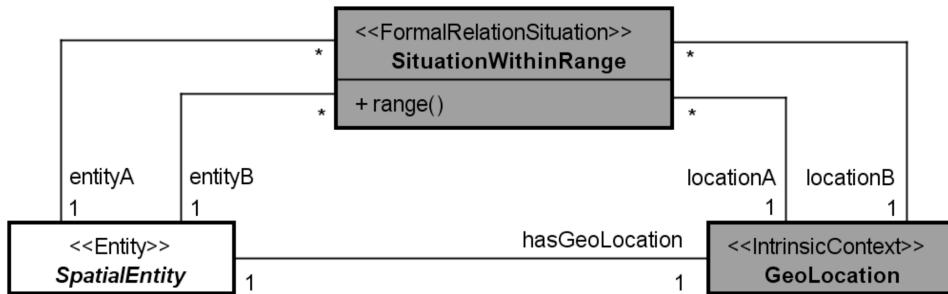


FIGURA 2.7 - MODELO DA SITUAÇÃO DE RELAÇÃO FORMAL
SITUATIONWITHINRANGE (COSTA, 2007)

No Código 2.3 têm-se a declaração da *SituationConnected* por meio de uma regra de situação.

```

01. rule "SituationWithinRange"
02. when
03.     $entityA: SpatialEntity($locationA: location)
04.     $entityB: SpatialEntity($locationB: location)
05.     eval($locationA.value.distance($locationB) < range)
06. then
07.     SituationHelper.situationDetected(drools);
08. end
  
```

CÓDIGO 2.3 - REGRA DA SITUAÇÃO SITUATIONWITHINRANGE

2.3.4 Situação de Situações

Na Figura 2.8 tem se o modelo de uma situação *SituationSwitch*, a qual se estabelece quando há um determinado dispositivo abandona uma rede WLAN e se associa a uma rede *Bluetooth*.

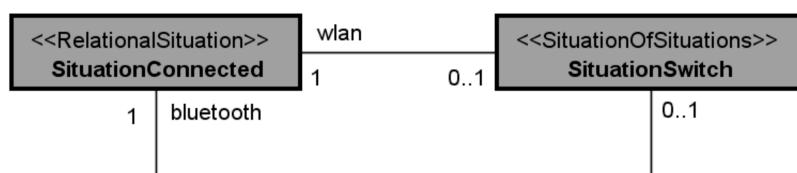


FIGURA 2.8 - MODELO DA SITUAÇÃO DE SITUAÇÕES SITUATIONSWITCH (COSTA, 2007)

No Código 2.4 têm-se a declaração da *SituationSwitch* por meio de uma regra de situação.

```

01. rule "SituationSwitch"
02.   when
03.     $wlan: SituationConnected($device: device, network.type==WLAN)
04.     $bluetooth: SituationConnected(device==$device,
05.                                         network.type==BLUETOOTH, this metby[ls] $wlan)
06.   then
07.     SituationHelper.situationDetected(drools);
08. end

```

CÓDIGO 2.4 - REGRA DA SITUAÇÃO SITUATIONSWITCH

2.3.5 Considerações

Essa seção apresentou os principais conceitos alicerces da plataforma SCENE. Outros conceitos como interação com o cliente, execução e implementação serão abordados no Capítulo 3. Essa decisão metodológica foi tomada com o objetivo de facilitar a apresentação da nova proposta da plataforma SCENE, pois esses aspectos estão diretamente relacionados com as melhorias propostas para a nova plataforma. Será mais didático e vai evitar a repetição de conceitos ao apresentarmos essas questões na seção 3.

2.4 TECNOLOGIAS, CONCEITOS E PADRÕES UTILIZADOS

2.4.1 SaaS

Software como um Serviço (SaaS) em tradução direta do Inglês, (Deyo, 2008) refere-se essencialmente a softwares que estão hospedados em servidores e são fornecidos como serviço. Alguns usos iniciais para SaaS incluíam ofertas de gerenciamento de relacionamento com clientes, sistemas de gerenciamento de conteúdo, videoconferência e sistemas de comunicação por e-mail. Os aplicativos SaaS são fornecidos pela web, o que significa que podem ser acessados a partir de qualquer computador sem qualquer software especial instalado. Na verdade, muitos aplicativos são projetados para executar através de um navegador da Web padrão. Quando as atualizações de um aplicativo SaaS precisam ser instaladas, elas simplesmente são instaladas no servidor, o que imediatamente garante que todos os usuários estão executando a versão mais recente, ao contrário dos aplicativos de software tradicionais. O Dropbox (referencia) é uma exemplo de SaaS.

2.4.2 PaaS

Segundo (Stankov, 2010) *Platform-as-a-Service* ou "plataforma como um serviço" geralmente se refere a plataformas de entrega de software baseadas na Internet e para as quais fornecedores de software independentes ou desenvolvedores de aplicativos podem criar aplicativos multiclientes, baseados na web, hospedados na infraestrutura do provedor e são oferecidos como um serviço aos clientes. A principal premissa da PaaS é fornecer aos desenvolvedores de software e aos fornecedores um ambiente integrado para desenvolvimento, hospedagem, entrega, colaboração e suporte para seus aplicativos de software sob demanda. Como outras plataformas de software, PaaS pretende ser uma fundação para um amplo e interdependente ecossistema de usuários e empresas. Ele pode suportar tarefas de edição de código para implantação, tempo de execução e gerenciamento.

2.4.3 RESTful API

REST (*Representational State Transfer*, 2016), em português, *Transferência de Estado Representacional*, define um conjunto de princípios arquitetônicos através dos quais se pode criar serviços web que se concentram nos recursos de um sistema, incluindo como os estados de recursos são endereçados e transferidos por HTTP por uma ampla gama de clientes escritos em diferentes idiomas. Se medido pelo número de serviços da Web que o utilizam, o REST surgiu nos últimos anos apenas como um modelo predominante de design de serviços da Web.

De fato, o REST teve um impacto significativo na Web que, na sua maioria, o projeto de interface baseado em SOAP (*Simple Object Access Protocol*, 2016) e WSDL (*Web Services Description Languages*, 2016) foram deslocados, porque REST é um estilo consideravelmente mais simples de se usar. Ele ignora os detalhes da implementação de componente e a sintaxe de protocolo com o objetivo de focar nos papéis dos mesmos, nas restrições sobre sua interação com outros componentes e na sua interpretação de elementos de dados significantes.

2.4.4 JavaScript Object Notation (JSON)

JSON (*Javascript Object Notation*, 2016) é um modelo para armazenamento e transmissão de informações em formato texto. Apesar de muito simples, tem sido bastante utilizado por aplicações Web devido a sua capacidade de estruturar informações de uma forma bem mais compacta do que a conseguida pelo modelo XML, tornando mais rápido o parsing dessas informações.

No Código 2.1 está descrito um exemplo de JSON, onde “número” é um valor double e “palavra” representa uma string.

```
01. {  
02.     "numero": 2.5,  
03.     "palavra": "string"  
04. }
```

CÓDIGO 2.5 - EXEMPLO DE JSON

2.4.5 JSON Schema

O JSON Schema (*Javascript Object Notation Schema*, 2016) especifica um formato baseado em JSON para definir a estrutura dos dados para validação, documentação e controle de interação. Um esquema JSON fornece o formato dos dados JSON necessários para um determinado aplicativo e como esses dados podem ser modificados. E quando um formato de dados é possível se obter metadados sobre o que os campos significam e quais entradas válidas para os mesmos. JSON Schema é uma especificação para padronizar como responder a essas perguntas para dados JSON. O exemplo de JSON do código 2.3 é baseado no schema do código 2.2.

```
01.  {
02.    "type": "object",
03.    "properties": {
04.      "numero": {
05.        "type": "number"
06.      },
07.      "palavra": {
08.        "type": "string"
09.      },
10.      "required": [
11.        "object"
12.      ]
13.    }
14.  },
15.  "required": [
16.    "numero",
17.    "palavra"
18.  ]
19. }
20. }
```

CÓDIGO 2.6 - EXEMPLO DE JSON SCHEMA

3 NOVA PLATAFORMA SCENE

3 NOVA PLATAFORMA SCENE

Desde o seu desenvolvimento, a plataforma SCENE foi empregada em trabalhos como em (Costa, 2016) e (Moreira, 2015), principalmente em cenários de saúde pública. A primeira versão de SCENE obteve bom reconhecimento da comunidade acadêmica, como pode ser observado pelo número de publicações geradas em torno deste tema. Contudo, como em qualquer tecnologia, SCENE precisa avançar. Graças as informações providas por seus usuários SCENE (e, naturalmente, de reflexões de seus próprios desenvolvedores), foi possível obter feedback reportando as maiores dificuldades no uso da plataforma, que giram em torno das seguintes funcionalidades: (i) modelar domínio e suas situações; (ii) popular e refletir alterações externas na *working memory*; e (iii) notificar interessados sobre situações identificadas.

3.1 MODELAGEM DE DOMÍNIO E SUAS SITUAÇÕES

Na plataforma SCENE as regras de situação dependem de entidades definidas em Java. Apesar do aspecto comportamental de um tipo de situação (*SituationType*) ser declarado via DRL, seu aspecto estrutural precisa ser declarado explicitamente como uma classe Java, fazendo com que o cliente tenha que gerenciar a atualização das classes em Java e o versionamento do modelo manualmente. Esta restrição pode gerar problemas, pois um ser humano está sujeito a cometer vários erros. Os próximos parágrafos explicam a experiência do ponto de vista do cliente ao utilizar SCENE como sua ferramenta de detecção e gerenciamento de situações.

SCENE foi usado em um cenário para análise de inconsistências em modelos formais de domínio, definidos via *OLED* (Guerson, 2015) e utilizando o *Alloy* (Jackson, 2002) para gerar instâncias do domínio no tempo. Situações em SCENE descreviam possíveis inconsistências sobre o modelo e possivelmente as detectava sobre as instâncias geradas, por exemplo, supondo que uma classe Pessoa esteja em um domínio no qual seus atributos sejam um booleano indicando o estado de vida ou morte desta Pessoa. *Alloy*, de acordo somente com o modelo, não teria discernimento para gerar dados não errôneos, portanto uma pessoa

poderia estar morta em um determinado período de tempo e viva em um período posterior a sua morte. Como claramente o modelo está inconsistente devido a este fato, SCENE foi utilizado para adicionar restrições temporais aos testes de consistência de modelo. A figura 3.1 apresenta uma visão geral simplificada da arquitetura da proposta, utilizando Alloy para geração de instâncias no tempo e SCENE como um restritor temporal.

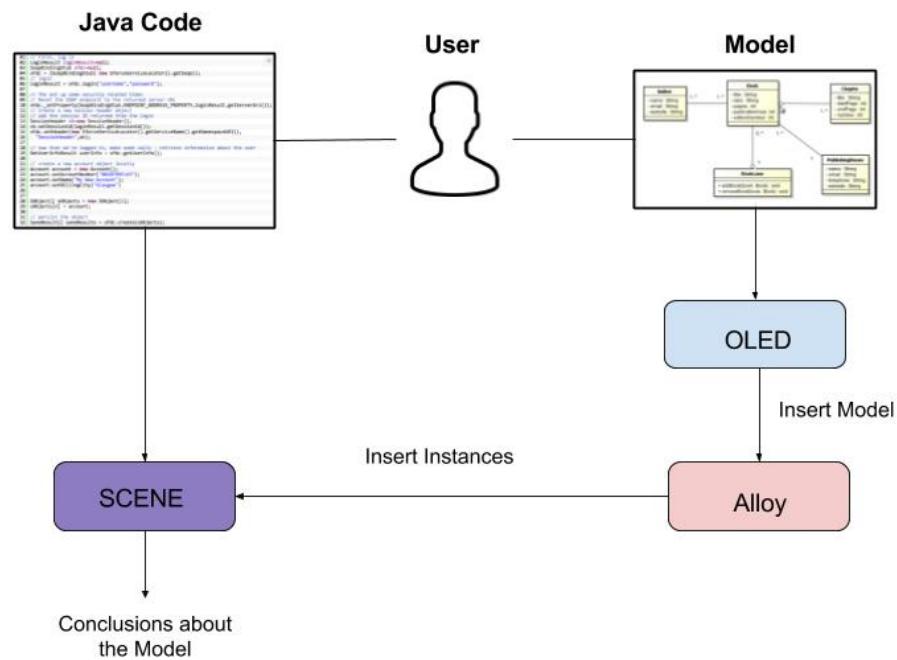


FIGURA 3.1 - ARQUITETURA DA APLICAÇÃO

A necessidade de especificar classes Java para cada situação e para as entidades do modelo exigiu um passo extra no processo de desenvolvimento da aplicação, ou seja, o cliente da aplicação tem que manter implementações de classes Java correspondentes às suas respectivas declarações dos modelos em OLED, exigindo uma revisão constante quanto a alterações do modelo. Dessa forma, a necessidade de alterações podem ocasionar em uma versão descontínua do modelo para com as classes Java exigidas.

3.2 GERÊNCIAMENTO DE CONTEXTO

O *scene-core* (Pereira, 2013) também não oferece nenhuma ferramenta de auxílio à inserção de contexto para a engine, portanto, é de inteira responsabilidade do desenvolvedor da aplicação sensível ao contexto gerenciar a inserção,

atualização e remoção de entidades e contextos da engine do SCENE manualmente. O cliente SCENE deve dominar algumas habilidades em Drools para que possa ser apto na gerência de informações trocadas entre a JVM (Lindholm, 2014) e a *working memory* de *Drools*. Uma destas habilidades se trata do uso da classe FactHandle que faz um mapeamento dos objetos na JVM para a *working memory* do *Drools*. Portanto os objetos não são manipulados diretamente. Cada modificação de um objeto na JVM seu respectivo *FactHandle* também necessita de uma atualização. O código 3.1 apresenta uma descrição de como o gerenciamento de contexto deve ser feito na plataforma SCENE.

```

01. // Loading the KnowledgeBase
02. KnowledgeBase kBase = readKnowledgeBase();
03. StatefulKnowledgeSession kSession = kBase.newStatefulKnowledgeSession();
04.
05. Object p1 = new Object();
06. FactHandle fh1 = kSession.insert(p1);
07. p1 = new Object();
08. kSession.update(fh1, p1);

```

CÓDIGO 3.1 - EXEMPLO DE GERENCIAMENTO DE CONTEXTO USANDO SCENE

Uma *KnowledgeBase* que contém as regras conforme discutido na seção 2.2.1.2 e que utiliza a máquina de inferência apresentada na seção 2.2.1.3 é carregada para criar uma *StatefulKnowledgeSession* que é baseada na base de conhecimento para que o contexto conforme discutido na seção 2.2.1.1 seja inserido e avaliado pela máquina de inferência. O *Object* p1 representa o contexto a ser inserido na sessão, porém quando este contexto é inserido, o próprio cliente precisa armazenar a sua referência (*FactHandle*) para que todas as atualizações referentes ao contexto p1 não sejam perdidas. E como descrito pela linha oito, o contexto é atualizado utilizando a referência fh1 que foi armazenada pelo cliente previamente na inserção do objeto na sessão.

3.2.1 Usabilidade

A plataforma SCENE também não oferece integração com outras aplicações interessadas em situações detectadas pela plataforma, ou seja, toda aplicação que

deseja utilizar das funcionalidades de SCENE deve acoplar a plataforma a cada aplicação desenvolvida, limitando assim o desenvolvimento das aplicações que desejam utilizar a plataforma. É também de responsabilidade do desenvolvedor da aplicação sensível ao contexto lidar com as ações que precisam ser tomadas quanto a uma situação de interesse. Se alguma situação desejada foi ativada, a aplicação não é notificada, ou seja, a aplicação deve acessar a informação sobre a situação diretamente na *working memory*. A biblioteca de SCENE não oferece nenhum mecanismo de callback ou implementação do padrão *observer/listener* para ser chamado no momento da detecção de uma situação.

3.3 REQUISITOS

Para conseguir atender a demanda exigida pelos clientes da plataforma, foi necessário um estudo para o levantamento de requisitos, para determinar os melhoramentos desejados.

A tabela a seguir lista os requisitos apontados pelos clientes devido as dificuldades encontradas para a utilização da plataforma.

Requisito	
RQ01	Permitir a criação/atualização e remoção de uma aplicação sensível a situação (Aplicação SCENE).
RQ02	Permitir a definição de modelo de contexto e situação baseado num dialeto (DRL) e associação do mesmo a uma aplicação SCENE.
RQ03	Fornecer maneiras de inserir e gerenciar contexto de uma aplicação.
RQ04	Gerenciar aplicações sensíveis a situações com contextos isolados, a alteração de uma aplicação não interfere no funcionamento de outra.
RQ05	Permitir que outras aplicações ou serviços declararem interesse em situações de uma aplicação SCENE.
RQ06	Oferecer mecanismo para publicação de situações e notificação para serviços interessados como descrito em RQ05.
RQ07	Permitir consulta dos tipos situação oferecidos por uma aplicação em tempo de execução.
RQ08	Permitir consulta de todos os dados contextuais envolvidos numa aplicação em tempo de execução da plataforma.

TABELA 1 - TABELA DE REQUISITOS

Utilizando um modelo *PaaS* (seção 2.4.2) podemos focar na construção de uma nova plataforma voltada a atender múltiplos clientes de uma só vez, de tal forma que o cliente não precise incorporar toda a lógica da plataforma SCENE em sua aplicação, removendo a limitação de desenvolvimento, possibilitando ao cliente desenvolver sua aplicação fora dos padrões que a plataforma SCENE exigia.

3.4 ATUALIZAÇÃO DA VERSÃO DO JBOSS DROOLS

Uma atualização da plataforma SCENE foi feita neste trabalho com a intenção de manter a *JBoss Drools* atualizado com as novas tecnologias, porém nenhuma alteração no comportamento da plataforma foi feita devido a esta atualização, somente uma adaptação de funções. Nas seções seguintes são discutidas as mudanças causadas pela atualização da versão do *Drools*.

3.4.1 Knowledge is Everything (KIE)

KIE é o novo nome usado para agrupar os projetos relacionados. Como a família continua a crescer. O KIE também é usado para as partes genéricas da API unificada. Tais como construção, implantação e carregamento. Isso substitui as palavras-chave *droolsjbpm* e conhecimento que teriam sido usadas antes.

Na Figura está descrita a nova arquitetura KIE e por quais módulos ela é utilizada.

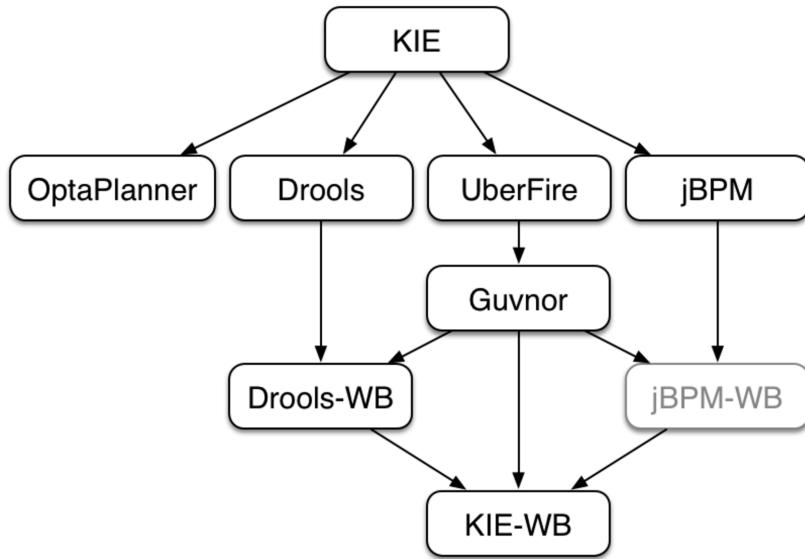


FIGURA 3.2 - ARQUITETURA KIE (DRL, 2016)

3.4.2 Convenção e configuração tomadas

Todas as funções começando com *Knowledge* foram substituídas por *Kie* e a criação do 'kmodule.xml' com as informações básicas da *KieBase* e *KieSession*. O 'kmodule.xml' fornece configuração declarativa para projetos KIE. Estes padrões são usados para reduzir a quantidade de configuração necessária. O código 3.2 apresenta a estrutura composta no padrão citado acima

```

01. <kmodule xmlns="http://www.drools.org/xsd/kmodule">
02.   <kbase name="kbase1" packages="org.mypackages">
03.     <ksession name="ksession1"/>
04.   </kbase>
05. </kmodule>
  
```

CÓDIGO 3.2 - USO DA KIESESSION

Uma aplicação em Drools 6.x é composta através de um módulo (*KieModule*), estão contidos todas as Bases de conhecimento (*KieBase*) e as sessões que as instanciam.

3.5 ARQUITETURA PROPOSTA

Após o estudo dos requisitos levantados, para que os melhoramentos pudessem ser feitos, algumas mudanças teriam que acontecer, pois a nova plataforma está sendo voltada para um uso coletivo e compartilhado. Portanto um novo modelo de situação foi desenvolvido.

3.5.1 Modelo

O modelo foi desenhado para ser independente de plataforma, diferentemente do modelo anterior (Figura 2.3), que utilizava das classes internas de JBoss Drools. A figura 3.2 apresenta o modelo conceitual proposto para a nova arquitetura proposta para a plataforma SCENE.

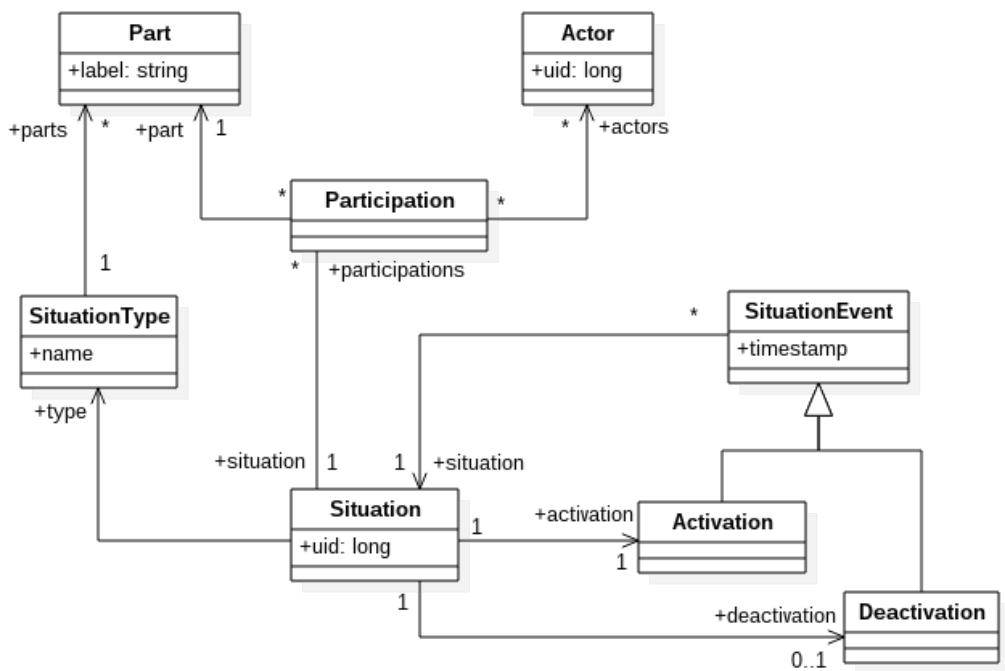


FIGURA 3.3 - MODELO DA NOVA ARQUITETURA DE SCENE

Com o modelo proposto é possível resolver uma dualidade existente no *SituationFactType* no modelo anterior que tinha o papel de tipo e instância. Não faz sentido que um Tipo de Situação possua eventos, por exemplo. Esta é uma característica exclusiva de situações (instâncias), por isso, resolveu-se pela separação de *SituationType* e *Situation*, o primeiro leva consigo a relação com as Parts, análogos a *SituationRoles* no modelo antigo, e o último mantém as relações

de ativação e desativação, além de ter relação explícita com *Participation*, que representa a manifestação de um indivíduo ao cumprir um papel (*Part*) numa determinada situação (*Situation*). Levando em consideração o modelo de detecção de febre, temos *febrile* como *Part*, ou seja, uma classificação do papel que a *Participation* exerce em cada *Situation*. Uma *Participation* surge assim que uma *Person* tem um episódio de febre com temperatura maior ou igual à 37°C. *Person* age como *Actors*, que são representações das instâncias das classes que compõem uma *Participation* e um *Actor* também pode ter vários tipos diferentes de *Participations*. Uma *Situation* pode ter dois tipos de *SituationEvent* sendo eles a *Activation* e *Deactivation*, estes eventos estão relacionados diretamente com a ativação (*Person* com episódio de febre) e desativação (após um episódio de febre, a temperatura de uma determinada *Person* retorna a um valor menor que 37°C) de uma situação. Cada *SituationEvent* também possui uma *Situation*, pois agora, todos os eventos estão diretamente relacionados a situações.

3.5.2 Regras de SCENE

Para que a plataforma SCENE ative e desative situações, algumas regras foram inseridas na base de regras da plataforma, regras que analisam as *OnGoingSituations* em ordem de gerar as instâncias de situações com seus respectivos eventos de ativação e desativação.

O código 3.3 apresenta regras que analisam as *OngoingSituations* e que também são uma das funcionalidades mais importantes da plataforma.

```

01. rule "SituationActivation"
02.   salience 1
03.   when
04.     $act: OnGoingSituation(situation == null, $type: type, $cast: cast,
05.     $timestamp: timestamp)
06.     then
07.       $act.setSituation(SituationHelper.activateSituation(drools, $cast,
08.         $type, $timestamp));
09.       update($act);
10.   end
11.
12. rule "SituationDeactivation"
13.   salience 999
14.   when
15.     $sit: Situation(active==true)
16.     not (exists OnGoingSituation(situation == $sit))
17.   then
18.     SituationHelper.deactivateSituation(drools, (Object) $sit);
19.   end

```

CÓDIGO 3.3 - REGRAS DE AVALIACAO DE SITUAÇÕES

Na regra *SituationActivation* todas as *OnGoingSituations* que tem um tipo (*SituationType*), participantes (*SituationCast*), e um *timestamp* atribuídos ativam esta regra que chama a função *activateSituation* (Código 3.8), fazendo com que uma instância da classe implementada pelo usuário que extenda *Situation* seja criada, um evento de ativação (*Activation*) é atribuído e inserida na *working memory*.

A regra *SituationDeactivation* é ativada quando uma *Situation* está ativa, mas não existe nenhuma *OnGoingSituation* atrelada a ela, então a função *deactivateSituation* (Código 3.9) é chamada e um evento de desativação é atrelado a *Situation* que ativou a regra.

3.5.3 Aplicação SCENE

A aplicação SCENE ou *SceneApplication* pode ser representada como uma instância de uma classe que utiliza as funcionalidades do scene-core (seção 3.4.4) para o gerenciamento de contexto e situações envolvendo um domínio em particular.

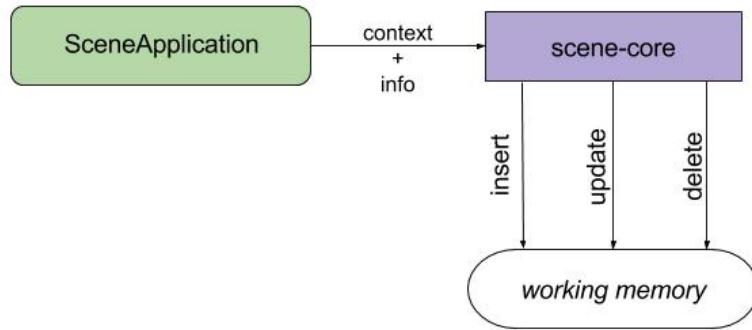


FIGURA 3.4 - ESTRUTURA DA APLICAÇÃO SCENE

No Código 3.4 podemos observar como a criação de uma `SceneApplication` é feita.

```

01. public SceneApplication(String name, KieSession ksession) {
02.     this.name = name;
03.     this.ksession = ksession;
04.     injectSceneMetamodel();
05. }
  
```

CÓDIGO 3.4 - CONSTRUTOR DA CLASSE SCENEAPPLICATION

Podemos observar que a instância da classe recebe um nome e uma `KieSession`, esta `KieSession` contém uma `KieBase`, estão contidas todas as informações sobre o modelo situacional usado para a descrição do domínio. Auxiliando no gerenciamento das situações (3.1.2) a função `injectSceneMetamodel` (Código 3.5) é chamada.

```

01. private void injectSceneMetamodel() {
02.     List<SituationType> types = findSituationTypes(ksession.getKieBase());
03.     for (SituationType type: types) {
04.         for(Part part: type.getParts()) {
05.             ksession.insert(part);
06.         }
07.         ksession.insert(type);
08.     }
09. }
  
```

CÓDIGO 3.5 - FUNÇÃO QUE INSERE O METAMODELO NA SESSÃO DE DROOLS

A função `injectSceneMetamodel` captura uma lista com todos os `SituationTypes` da `KieBase` contida na `KieSession` e a percorre inserindo todas as

Parts relacionadas a cada *SituationType* e por fim insere também cada *SituationType* na *KieSession*.

Do ponto de vista de SCENE, os tipos de metadados como *Part*, *Participation* e *SituationType* devem estar dentro da *working memory*, disponíveis para descrever as condições da regra, bem como as instâncias de situação e seus eventos já existentes.

3.5.4 Gerenciamento de Situações

O gerenciamento das situações é feito através de uma combinação de informações entre a *KieSession* e funções providas pelo *scene-core*. Uma das classes fundamentais do *scene-core* para a criação de uma situação é o *SituationCast* que é uma extensão de um *HashMap*, que também associa variáveis de bind de regras Drools que representam *Parts* de um *SituationType* e *Actors* (que são representados como *FactTypes* na *working memory*). A função de *put* que faz o cálculo da hash (Código 3.6) foi alterada fazendo com que cada *SituationCast* seja única.

```

01.  @Override
02.  public Object put(String key, Object value) {
03.      this.hash = 17;
04.      this.hash = this.hash + (key.hashCode() +
05.          value.getClass().hashCode() + value.hashCode());
06.      this.hash = 31*this.hash + key.hashCode();
07.      this.hash = 31*this.hash + value.getClass().hashCode();
08.      this.hash = 31*this.hash + value.hashCode();
09.      return super.put(key, value);
10. }
```

CÓDIGO 3.6 - SITUATIONCAST FUNÇÃO PUT

O cálculo da *hash* é o que auxilia o TMS (2.2.3) a detectar se uma *OnGoingSituation* está ainda atendendo as condições da regra que a criaram. O cálculo é iniciado com um número primo e é adicionado ao hash da chave os valores de hash da classe e do objeto a ser inserido na *HashMap* e depois novamente

adicionado a cada um dos itens citados acima, porém a cada adição o valor da hash é multiplicado por 31.

Toda regra escrita em *Drools* que é relacionada à detecção de uma situação deve ter em sua RHS a função *situationDetected* recebendo como parâmetro o *KnowledgeHelper* padrão chamado “drools”.

No Código 3.7 está descrita a função *situationDetected*, que sempre é chamada no RHS de uma regra de detecção (exemplos de regras em 2.2.2).

```

01. public static void situationDetected(KnowledgeHelper khelper) throws Exception {
02.     RuleImpl rule = khelper.getRule();
03.     String packageName = rule.getPackageName();
04.     String className = (String) rule.getMetaData().get("type");
05.     SituationType type = getSituationType(khelper.getKieRuntime(),
06.                                             packageName + '.' + className);
07.     OnGoingSituation ongoing = new OnGoingSituation(type,
08.                                                       khelper.getKieRuntime().getSessionClock().getCurrentTime(),
09.                                                       new SituationCast(khelper.getMatch(), type));
10.    khelper.insertLogical(ongoing);
11. }
```

CÓDIGO 3.7 - FUNÇÃO PARA DETECÇÃO DE UMA SITUAÇÃO

No inicio da função é chamada a função *getRule* para que a regra que chamou a função seja capturada. O nome do pacote da regra é obtido para ser combinado com o nome da classe que é obtido através dos metadados (seção 3.5). Uma *OnGoingSituation* é criada passando como parâmetros um *SituationType* que foi recuperado através da combinação do pacote da regra com seu respectivo nome da situação. Um *timestamp* de acordo com as funções de drools e um *SituationCast* criado através de um evento de ativação e um *SituationType*. Então, finalmente a *OnGoingSituation* criada é inserida logicamente na *Working Memory* e, de acordo com a unicidade da *SituationCast*, não serão inseridas duas *OnGoingSituations* iguais na *KieSession*. Como a *OnGoingSituation* está sendo inserida logicamente, ou seja, o sistema TMS (seção 2.2.3) está sendo utilizado, assim que as condições que ativaram a regra deixam de ser consistentes a *OnGoingSituation* adicionada é removida automaticamente.

Após a inserção de uma *OnGoingSituation* com seus parâmetros preenchidos, é feita uma verificação se não existe uma instância da classe *Situation*

relacionada a ela (ou seja, não há situações ativas relacionadas a este *OnGoingSituation*), uma regra interna ao *scene-core* (Código 3.3) é ativada e em seu RHS a função *activateSituation* (Código 3.8) é chamada.

```

01. public static Situation activateSituation(KnowledgeHelper khelper,
02.                                         SituationCast cast,
03.                                         SituationType type,
04.                                         long timestamp) {
05.     KieRuntime runtime = khelper.getKieRuntime();
06.     long evn_timestamp = runtime.getSessionClock().getCurrentTime();
07.     Activation activation = new Activation(evn_timestamp);
08.     Situation situation = ((SituationTypeImpl) type).newInstance(activation, cast);
09.     activation.setSituation(situation);
10.    runtime.insert(activation);
11.    for (Participation participation: situation.getParticipations()) {
12.        runtime.insert(participation);
13.    }
14.    runtime.insert(situation);
15.    return situation;
16. }
```

CÓDIGO 3.8 - FUNÇÃO QUE CRIA E ATIVA A SITUAÇÃO

Após todos os parâmetros auxiliares terem sido obtidos, são criadas instâncias do evento de ativação (*Activation*) e da classe *Situation* através de reflection, com o evento de ativação e o *SituationCast* como parâmetros de entrada. Então como a instância da classe *Situation* foi construída, pode agora ser atribuída também ao evento de ativação como demonstrado no modelo (seção 3.4.1). Então o evento de ativação, a instancia de situação e todas as *Participations* atribuidas à instância de *Situation* são inseridas na *Working Memory*.

Se as condições que tornaram a situação ativa não são mais consistentes, a instância da classe *OnGoingSituation* vai ser removida da *Working Memory* e, portanto, a instância da classe *Situation* vai estar com estado ativo sem uma instância da classe *OnGoingSituation* relacionada a ela. Neste momento, uma regra interna ao *scene-core* é ativada e em seu RHS a função *deactivateSituation* (Código 3.9) é chamada.

```

01. public static void deactivateSituation(KnowledgeHelper khelper, Object sit) {
02.     long evn_timestamp = khelper.getKieRuntime()
03.         .getSessionClock().getCurrentTime();
04.     Deactivation deactivation = new Deactivation(evn_timestamp);
05.     deactivation.setSituation((Situation) sit);
06.     ((Situation) sit).setDeactivation(deactivation);
07.     khelper.getKieRuntime().insert(deactivation);
08.     khelper.getKieRuntime().update(khelper.getKieRuntime().getFactHandle(sit), sit);
09. }
```

CÓDIGO 3.9 - FUNÇÃO QUE DESATIVA A SITUAÇÃO

No inicio da função, um *timestamp* é criado a partir das classes de *Drools* e uma instância do evento de desativação (*Deactivation*) é criada e a instância da situação (*Situation*) é atribuída a este evento de desativação. De maneira similar, é feita uma atribuição do evento de desativação na instância da situação. Feitas as devidas atribuições, o evento de desativação é então inserido na *Working Memory* e a situação também é atualizada, ou seja, agora ela está carregando um evento de desativação com ela.

3.6 SCENE DRL

Para atender ao requisito **RQ02** (seção 3.2), uma estrutura dentro de um DRL (seção 2.2.2) foi criada para que o cliente não precise se preocupar em criar implementações em *Java* que tenham que ser compatíveis com seu modelo, podendo agora descrever o seu domínio e a lógica por traz da detecção das situações desejadas em um formato padrão.

O Código 3.10 apresenta a estrutura formada para que o cliente possa descrever o seu domínio e usar a plataforma.

```

01. // Selecting the Package
02. package package.path
03.
04. // Importing all the Classes that will be needed
05. import package.path.Class
06.
07. declare Clazz
08.     attr: Attribute
09. ...
10. end
11.
12. declare SituationClazz extends Situation
13.     attr1: Attribute @part
14.     attr2: Attribute @part(label = "someLabel")
15. ...
16. end
17.
18. rule "RuleName"
19.     @role(situation)
20.     @type(SituationClazz)
21.     when
22.         someLabel: Clazz(someTest(attr))
23.     then
24.         SituationHelper.situationDetected(drools);
25. end

```

CÓDIGO 3.10 - ESTRUTURA DRL VOLTADA À SCENE

A estrutura descrita acima necessita de um pacote relacionado à aplicação juntamente com todas as importações de classes necessárias para o funcionamento da mesma. Feitas todas as importações necessárias, o usuário poderá declarar as classes, com seus respectivos atributos, pois *Drools* se encarregará de criar seu construtor e seus *getters* e *setters*, para cada atributo relacionado. As declarações das classes que serão situações não são muito diferentes na hora da criação, exceto que elas devem obrigatoriamente estender a classe *Situation*, e seus atributos devem estar com a tag *@part* (indicando as *Parts* relacionadas àquela situação em particular). Feitas as definições de classes do domínio e classes de situação, o usuário poderá criar regras de situação. Em particular, para uma regra que ativa uma situação, as tags *@role* e *@type* devem ser usadas como informados no código 3.10 *@role* sempre com *situation* como parâmetro e *@type* com o tipo da classe *Situation* atribuída. O LHS pode ser exatamente igual ao de *Drools* (seção 2.2.2),

porém o RHS deve conter a função interna do *scene-core* (*situationDetected*), como explicado na seção 3.4.4.

O código 3.11 está descrevendo um modelo de febre e também exemplificando a estrutura apresentada no código 3.10.

```

01. package scene
02.
03. import br.ufes.inf.lprm.scene.model.impl.Situation;
04. import br.ufes.inf.lprm.scene.util.SituationHelper;
05. import br.ufes.inf.lprm.situation.annotations.part;
06.
07. declare Person
08.     name: String
09.     temperature: int
10. end
11.
12. declare Fever extends Situation
13.     febrile: Person @part(label = "febrile")
14. end
15.
16. rule "FeverSituation"
17.     @role(situation)
18.     @type(Fever)
19.     when
20.         febrile: Person(temperature > 37)
21.     then
22.         SituationHelper.situationDetected(drools);
23. end

```

CÓDIGO 3.11 – EXEMPLIFICAÇÃO DA ESTRUTURA DO CÓDIGO 3.10

A estrutura descrita acima está usando o pacote *scene* relacionado à aplicação de interesse. Todas as importações de classes disponibilizadas pelo *scene-core* foram feitas, então o usuário declara a classe *Person*, com seus respectivos atributos. A declaração da classe da situação *Fever* é feita estendendo a classe *Situation*, e seu atributo com a tag *@part* (indicando a *Part* febrile relacionada àquela situação). Então a regra que detecta uma situação de febre é criada e as tags *@role*, *situation* define aquela regra como uma regra que detecta situações e o *@type* que descreve qual o *SituationType* pertencente à classe *Situation* que será detectado na regra.

3.7 EXECUÇÃO E ASPECTOS COMPORTAMENTAIS

Na Figura 3.5 está demonstrado o *working flow* que em SCENE é descrito como todo o gerenciamento de situações é feito, com base no exemplo de febre citado em capítulos anteriores.

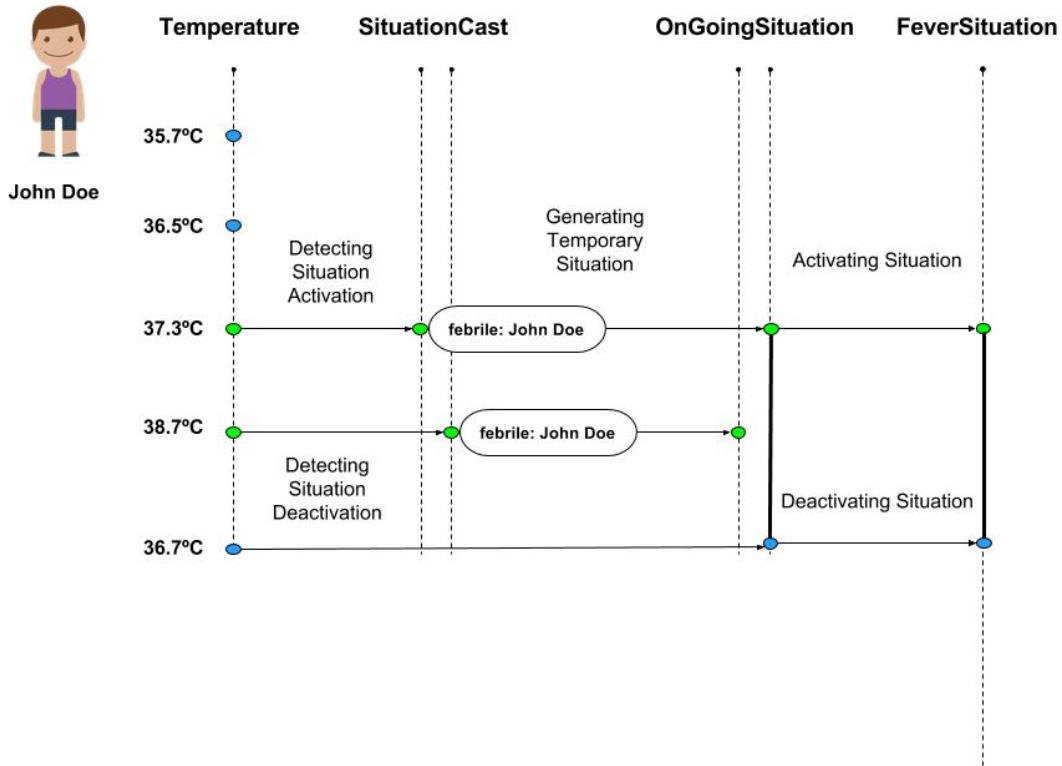


FIGURA 3.5 - SCENE WORKING FLOW

Na execução, o Drools irá construir a árvore de regras antes que qualquer objeto seja inserido na *Working Memory*. Depois que os objetos são inseridos, as regras serão avaliadas de acordo com os objetos inseridos e se uma delas for uma regra de situação e sua avaliação for válida, a execução da regra chamará o método *situationDetected* (Código 3.7).

Como a *OnGoingSituation* está agora na *Working Memory*, ela pode ser avaliada pelas regras *SituationActivation* e *SituationDeactivation*, a função chamada em seus RHS estão contidas respectivamente em Código 3.8 e 3.9.

3.8 CONSIDERAÇÕES

Todo o ciclo de gerenciamento da situação na nova plataforma SCENE se encontra descrito neste capítulo, juntamente com toda a nova arquitetura usada para evoluir a plataforma, também seu novo modelo de gestão de situações que agora passa a ser independente de plataforma.

4 SITUATION MANAGEMENT AS SERVICE

4 SITUATION MANAGEMENT AS A SERVICE (SMAAS)

Após a reformulação da plataforma SCENE, em conjunto com as novas tecnologias surgindo dando suporte a vários tipos de evolução, uma questão foi levantada: como SCENE seria capaz de interagir com outros sistemas sem que suas funcionalidades fossem comprometidas?

O desenvolvimento de SCENE, por ser baseado em JBoss Drools, ficou atrelado à ferramenta. Para contornar este problema, estamos propondo uma camada de aplicação responsável por gerenciar todas as comunicações dos clientes SCENE com a plataforma, como demonstrado na Figura 4.1. As funcionalidades do “scene-core” não estão disponíveis diretamente para o cliente, fazendo com que o desenvolvimento dos clientes seja independente de plataforma e em consequência tornando essa camada em um serviço. SCENE, então, atuaria somente como uma “caixa preta” para seus clientes, na qual a troca de informação entre cliente e plataforma é definida pela camada de aplicação citada acima. Eis que a plataforma se torna um serviço, podendo atender mais de uma requisição ao mesmo tempo, permitindo que aplicações que o utilizam possam ser escritas em diferentes linguagens ou baseadas em diferentes ferramentas, gerando, portanto, desacoplamento entre clientes e a plataforma.

A Figura 4.1 exemplifica como uma aplicação cliente pode vir a se comunicar com o serviço (SMaaS) proposto.

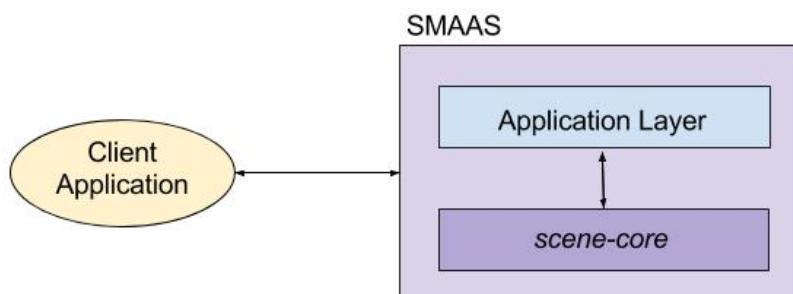


FIGURA 4.1 - COMUNICAÇÃO CLIENTE-SERVIÇO

4.1 VISÃO GERAL

O serviço SCENE, denominado SMaaS propõe eliminar a dependência de plataforma que os clientes da primeira versão de SCENE vivenciaram. Com a criação da camada de aplicação (seção 4.3.2), os clientes estão aptos a interagir com o serviço utilizando diferentes tipos de plataforma, porém seguindo um mesmo protocolo de comunicação. Detalhes do protocolo de comunicação com o serviço são discutidos na seção 4.2.

4.1.1 Funcionalidades

O SMaaS tem um conjunto de funcionalidades que dão suporte a seus clientes à inserção e gerência do funcionamento de suas respectivas aplicações. A seguir listamos o conjunto de funcionalidades oferecidas pelo serviço:

Gerenciamento de aplicações

Uma nova aplicação é inserida no serviço, o gerenciamento dos dados desta dependem inteiramente da aplicação cliente que a adicionou.

1. Inserção de contexto – Permite que os dados de uma aplicação cliente sejam inseridos remotamente em uma aplicação SCENE correspondente no serviço.
2. Atualização de contexto – Atualiza os dados já inseridos em uma dada aplicação SCENE, fazendo as modificações de acordo com a escolha da aplicação cliente.
3. Exclusão de contexto – Apaga os dados de uma dada aplicação SCENE no serviço.

Observação do comportamento de uma aplicação

Um cliente diferente do cliente que adicionou a aplicação SCENE pode observar todo o comportamento da mesma sem interferir em seus objetos, não tendo autonomia para controlar a aplicação.

4.1.2 Clientes do Serviço SCENE

Os clientes que podem gerenciar dados de uma aplicação têm acesso a todos os dados primitivos de suas respectivas aplicações. Por exemplo, se uma aplicação define uma classe Pessoa, o cliente seria capaz de manipular somente os dados dos objetos criados a partir da classe Pessoa. Portanto todos os objetos criados para o gerenciamento das situações estão protegidos de interferências externas. Abaixo estão descritos os tipos de clientes suportados pelo serviço.

Administrador - O que faz o gerenciamento das aplicações, ou seja, insere e manuseia os dados primitivos de uma ou mais aplicações. Atendendo ao requisito **RQ04** (seção 3.2), pois o Administrador só pode gerenciar ma aplicação de cada vez e somente as aplicações de sua própria autoria.

Assinante - Após cadastrar-se, recebe os dados da aplicação desejada, sem intervir nos mesmos. Atendendo ao requisito **RQ05** (seção 3.2) e levando em consideração o modelo de febre. O cliente assinante da aplicação pode saber sobre qualquer situação de febre envolvendo qualquer pessoa, saber sobre qualquer situação de febre de uma determinada pessoa, ou saber sobre qualquer situação em que uma pessoa esteja envolvida.

4.2 INTERAÇÃO DE CLIENTES COM O SERVIÇO

Como fora discutido nas seções anteriores, as funcionalidades do serviço e os papéis de cada tipo de cliente, podemos descer o nível de abstração e entrar nas particularidades do serviço.

Na Figura 4.2 abaixo, podemos observar os papéis do cliente e como a interação entre clientes e o serviço é feita.

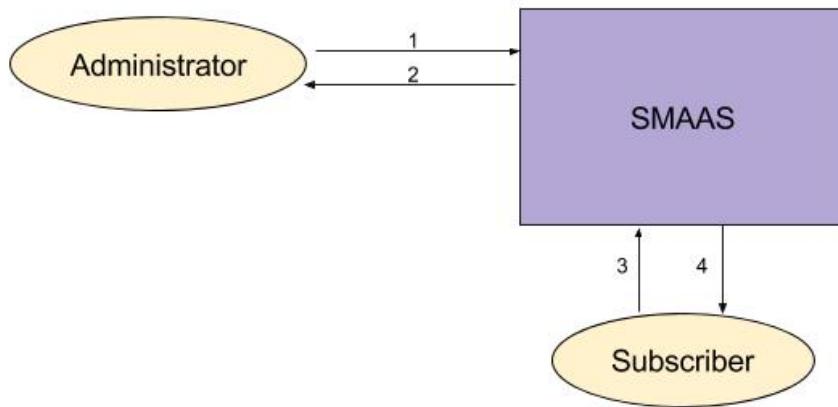


FIGURA 4.2 - INTERAÇÃO COM O SERVIÇO

SMaas adota uma *API RESTful* que usa integralmente mensagens HTTP para se comunicar e todas as mensagens tem um corpo definido por um *JSON Schema* (seção 2.4.5) relacionado. Todos os *JSON Schema* citados nesta seção estão disponíveis no Apêndice 1.

Dois métodos comumente usados para uma interação do tipo *request-response* entre um cliente e servidor são GET e POST.

GET - Solicita dados de um recurso especificado.

POST - Submete dados a serem processados por um recurso especificado.

4.2.1 Inserindo aplicação

Nesta seção será abordada a maneira que os clientes podem inserir uma nova aplicação no serviço e suas respectivas respostas.

Os clientes SMaaS podem enviar uma nova aplicação para o servidor, criando uma requisição POST indicado na Figura 4.2 na seta de número 1, também especificado abaixo e atendendo ao requisito **RQ01** (seção 3.2).

POST /app

No corpo da requisição estarão contidas as informações necessárias para que a aplicação seja executada com sucesso. No campo "application" o nome da aplicação é requerido, no campo "version" a versão da aplicação em questão é requerida, no campo "description" a descrição da aplicação não é necessária, pois é um campo extra, no vetor "files" são incluídos todos os arquivos necessários para

que a aplicação funcione. Para cada nó arquivo é necessário incluir o nome, pacote e o conteúdo do arquivo nos respectivos campos "*name*", "*package*", "*content*". Para mais detalhes sobre o JSON Schema do corpo da requisição, veja código disponível na requisição 1, no Apêndice 1.

A resposta para esta requisição é o número identificador da aplicação no serviço (na tag "*id*") e o nome da aplicação (na tag "*app*"). Para mais detalhes sobre o JSON Schema para a resposta disponível em requisição 1 em Apêndice 1.

4.2.2 Inserir, atualizar e excluir contexto de uma Aplicação Cliente

Nesta seção será abordada a maneira que os clientes podem gerenciar dados de uma aplicação, e suas respectivas respostas. Atendendo ao requisito **RQ03** (seção 3.2) que se refere ao gerenciamento de contexto dentro de uma aplicação.

Os clientes SMaaS podem enviar dados para uma aplicação, criando uma requisição POST indicado na Figura 4.2 na seta 1 e também descrito abaixo, sendo *{appId}* o número identificador da aplicação.

POST /app/{appId}

No corpo da requisição estarão contidas as informações necessárias para o gerenciamento dos dados de uma aplicação com identificador igual a *{appId}*. No campo "*type*", o cliente pode escrever "insert" para inserção de dados, "update" para atualização de dados ou "delete" para a exclusão de dados. Os outros campos são preenchidos com os objetos das classes de acordo com o modelo da aplicação cliente. Seus respectivos atributos também são preenchidos juntamente com um atributo obrigatório chamado "*id*", para a identificação e manuseio do objeto. Para mais detalhes sobre o JSON Schema do corpo da requisição, veja código disponível na requisição 2, no Apêndice 1.

A resposta para esta requisição é uma mensagem de sucesso ou de erro para o gerenciamento desejado, especificada no campo "*success*". Para mais detalhes sobre o JSON Schema para a resposta, veja código disponível na requisição 2, no Apêndice 1.

4.2.3 Coletando informações do Serviço

Nesta seção será abordada a maneira com que os clientes podem receber informações sobre uma aplicação. Atendendo aos requisitos **RQ07** e **RQ08** (seção 3.2), pois tratam de consultas ao serviço, sobre informações de uma aplicação desejada.

Os clientes SMaaS podem receber as identificações e nomes de todas as aplicações ativas no servidor, criando uma requisição GET indicado na seta 2 da Figura 4.2 e descrito abaixo.

GET /app

A resposta para esta requisição é uma lista de itens com campos "name" para nome da aplicação, "id" para número identificador da aplicação no servidor e "description" para a descrição. Para mais detalhes sobre o JSON Schema do corpo da requisição, veja código disponível na requisição 3, no Apêndice 1.

Os clientes SMaaS podem receber o modelo de uma aplicação no servidor com todas as informações sobre suas classes, criando uma requisição GET indicado na seta 2 da Figura 4.2 e como descrito abaixo, sendo {appId} o número identificador da aplicação.

GET /app/{appId}/model

A resposta para esta requisição é uma lista com a descrição de todas as situações possíveis para aquela aplicação, contendo também a descrição de cada objeto participante de cada situação. Cada item da lista contém um campo "name" (o nome da classe da situação) e um campo "parts" contendo a lista dos objetos de classe primária que compõem a situação em questão. Para cada um destes objetos, o campo "name" indica o nome da classe deste objeto, o "kind" indica o tipo da classe e a lista "fields", contendo "name" que indica o nome da variável e "type" indicando o tipo da variável para cada atributo da classe. Para mais detalhes sobre o JSON Schema para a resposta, veja código disponível na requisição 4, no Apêndice 1.

Os clientes SMaaS podem receber informações sobre todas as situações de uma aplicação, juntamente com suas participações (*Participations*) - Objetos que as ativaram e as desativaram. Para isto, o cliente especifica uma requisição GET indicado na Figura 4.2 na seta 2, como descrito abaixo, sendo {appId} o número identificador da aplicação.

GET /app/{appId}/situations

A resposta para esta requisição é uma lista das situações contidas no servidor, e cada item tem um "name" representando o nome da classe da situação, um campo "active" que é um booleano indicando se a situação está ativada (true) ou não (false), um campo "activation" que indica o exato momento em que a situação foi ativada (em milissegundos), "participations" uma lista das classes que compuseram a ativação daquela situação e cada item contém "kind" nome da classe do objeto, e todos seus atributos e seus respectivos valores listados. Para mais detalhes JSON Schema para a resposta disponível em requisição número 5 em Apêndice 1.

Os clientes SMaaS podem receber informações sobre todas as situações e objetos que estão presentes em uma aplicação, criando uma requisição GET indicado na seta 2 da Figura 4.2, e como descrito abaixo, sendo {appId} o número identificador da aplicação.

GET /app/{appId}/dump

A resposta para esta requisição é uma lista das situações contidas no servidor e também a lista de objetos que se encontram instanciados dentro do servidor. Cada objeto tem um campo "kind" que indica o nome da classe do objeto e todos os seus outros atributos e seus respectivos valores listados. Para mais detalhes sobre o JSON Schema para a resposta, veja código disponível na requisição 6, no Apêndice 1.

4.2.4 Assinatura para receber informações do Serviço

Nesta seção será abordada a maneira com que os clientes podem assinar uma aplicação afim de receber suas informações. Atendendo aos requisitos **RQ05** (seção 3.2), pois trata de aplicações clientes externas declararem interesse em uma aplicação SCENE dentro do serviço.

Os clientes SMaaS podem se cadastrar em uma determinada aplicação SCENE, as informações geradas são automaticamente redirecionadas para o cliente que se cadastrou. Criando uma requisição POST indicado na seta 3 da Figura 4.2 e como descrito abaixo, sendo {appId} o número identificador da aplicação.

POST /app/{appId}/subscriber

No corpo da requisição estarão contidas as informações necessárias, para que o serviço envie as informações de maneira correta a aplicação cliente que solicitou o cadastro. O campo “webhook” é sempre necessário, pois a aplicação necessita saber aonde retornar à informação para o cliente que solicitou o cadastro. Se apenas este campo é enviado, significa que o usuário está interessado em todas as situações disponíveis. Se o campo “situation” é preenchido juntamente com o campo “webhook”, significa que o cliente deseja receber informações de uma determinada situação contida naquela aplicação. Se o campo “actor” é preenchido juntamente com o campo “webhook”, significa que o cliente deseja todas as informações envolvendo aquele *Actor* (Seção 3.4.1) em qualquer situação dentro da aplicação. O campo “actor” necessita de um objeto contendo o tipo da classe “type” e o “id” relacionado aquela instância. Se todos os campos mencionados acima são preenchidos, significa que o cliente deseja receber informações sobre um determinado *Actor* em uma determinada *Situation* (seção 3.4.1). Para mais detalhes sobre o JSON Schema do corpo da requisição, veja código disponível na requisição 7, no Apêndice 1.

A resposta para esta requisição é o número identificador da subscrição no serviço (na tag "subscriberId") indicado na seta 4 da Figura 4.2. Para mais detalhes sobre o JSON Schema para a resposta disponível em requisição 7 em Apêndice 1.

Os clientes SMaaS podem cancelar uma assinatura previamente feita, criando uma requisição DELETE indicado na seta 2 da Figura 4.2 e como descrito abaixo, sendo `{appId}` o número identificador da aplicação e `{subscriberId}` é o número identificador da aplicação cliente que deseja informações sobre alguma aplicação rodando no serviço.

DELETE /app/{appId}/subscriber/{subscriberId}

A resposta para esta requisição é um objeto com somente um campo “`subscriberId`” com um número relacionado é retornado, quando a assinatura é cancelada. Para mais detalhes sobre o JSON Schema para a resposta disponível em requisição 8 em Apêndice 1.

4.3 ARQUITETURA INTERNA DO SERVICO

O SMaaS foi implementado usando o *Play Framework* (seção 4.3.1) possibilitando o seu acesso por qualquer outro tipo de plataforma que deseja gerenciar situações com apenas uma simples troca de dados. A comunicação entre cliente e serviço é feita por JSON.

4.3.1 Play Framework

O Play é um framework de aplicativos web Java e Scala de alta produtividade que integra componentes e APIs necessários para o desenvolvimento de aplicativos web modernos. Tem uma arquitetura web-friendly e possui consumo de recursos mínimos (CPU, memória, threads) para aplicações altamente escaláveis graças ao seu modelo reativo, baseado em *Akka Streams*. (*Play Framework*, 2016)

O framework foi utilizado para construir um caminho de comunicação entre o cliente e a plataforma, dando suporte à comunicação via web, como explicado nas próximas seções.

4.3.2 Estrutura e Implementação

A estrutura do serviço consiste em utilizar o SCENE core combinado com o Play framework para gerenciar as comunicações entre o cliente e a plataforma usando mensagens formato JSON.

Na Figura 4.3 está descrita a arquitetura do serviço como um todo, em uma visão de desenvolvedor.

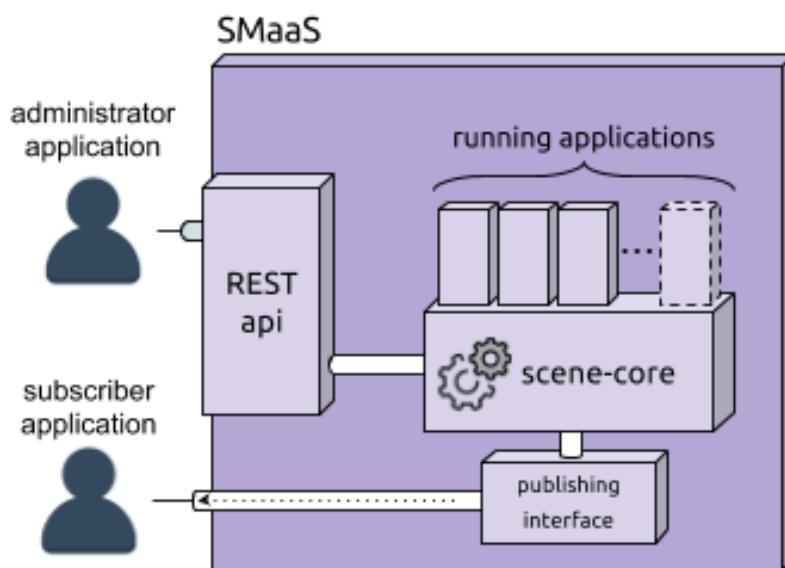


FIGURA 4.3 - ARQUITETURA DO SERVIÇO SCENE

As aplicações criadas dentro do *Play framework* (representadas pelas “*running applications*” na Figura 4.3) fazem o gerenciamento das situações por métodos de comunicação com a camada de aplicação providos pelo *scene-core*. Quando o serviço é levantado nenhuma aplicação ainda existe.

O Código 4.1 é a representação da implementação da Figura 4.3 no Play Framework, o *Singleton Scene* faz a comunicação entre o framework e o *scene-core* e a classe *Application* se comunica com o cliente externo a aplicação.

```
01. @Singleton
02. public class Scene {
03.     private final SceneManager manager;
04. }
05.
06. public class Application extends Controller {
07.     @Inject
08.     Scene scene;
09. }
```

CÓDIGO 4.1 - COMUNICAÇÃO ENTRE O PLAY FRAMEWORK E SCENE-CORE

4.3.2.1 Inserindo Aplicação

Assim que um cliente envia uma solicitação para a criação de uma aplicação (seção 4.2.3), algumas validações de dados são necessárias para que a aplicação seja adicionada à lista de aplicações conhecidas. Portanto, uma validação dos campos dos JSON recebidos deve ser consistente com o schema provido.

O Código 4.2 representa a validação do JSON recebido, verificando todos os campos requeridos pelo schema e, caso algo esteja fora do padrão, uma mensagem de "bad request" é enviada ao cliente com uma notificação sobre o campo em específico que o mesmo deve alterar para que a aplicação possa ser inserida com sucesso.

```

01.     if(!node.has("application"))
02.         return badRequest("There is no application field.");
03.     if(!node.has("files"))
04.         return badRequest("There is no files field.");
05.     else {
06.         JsonNode files = node.get("files");
07.         if(files.size() == 0)
08.             return badRequest("There is no described files inside files field.");
09.         else
10.             for(int i = 0; i < files.size(); i++) {
11.                 if(!files.get(i).has("name"))
12.                     return badRequest("File number " + i + " has no name described.");
13.                 if(!files.get(i).has("package"))
14.                     return badRequest("File number " + i + " has no package described.");
15.                 if(!files.get(i).has("content"))
16.                     return badRequest("File number " + i + " has no content described.");
17.             }
18.     }

```

CÓDIGO 4.2 - VALIDAÇÃO DO APLICAÇÃO RECEBIDA POR JSON

No processo de criação da aplicação, uma validação ao conteúdo de cada item pertencente à lista "files" recebidos pelo JSON da aplicação recebida deve ser feita. No entanto, somente o *JBoss Drools* consegue fazer esta validação, pois os arquivos enviados devem estar em formato DRL aceito pela plataforma. Para tanto, todos os conteúdos dos arquivos providos pelo JSON são inseridos em um *KieFileSystem* e então um *KieBuilder* é gerado. Durante a operação *buildAll*, descrita no Código 4.3, se algo estiver errado com qualquer um dos conteúdos providos, uma exceção é gerada e redirecionada ao *singleton SCENE* que provê uma mensagem adequada ao cliente.

O Código 4.3 representa a validação dos dados contidos na lista "files" providas pelo JSON da aplicação. O código também mostra a geração de uma exceção para ser tratada pelo *singleton SCENE*.

```

01.     KieBuilder kbuilder = kServices.newKieBuilder(kieFileSystem);
02.     kbuilder.buildAll();
03.     if (kbuilder.getResults().hasMessages()) {
04.         throw new IllegalArgumentException(kbuilder.getResults());
05.     }

```

CÓDIGO 4.3 - VALIDAÇÃO DE DADOS DRL

Após feitas todas as validações, a aplicação é construída e inserida na lista de aplicações conhecidas, e um identificador único é criado e retornado por meio de mensagem em formato JSON.

O Código 4.4 demonstra a criação e a inserção de uma aplicação (linhas 01 à 03). O código também mostra a criação de uma chave *hash* única atribuída à aplicação, juntamente com a montagem de uma mensagem de sucesso para o cliente sobre a operação desejada.

```

01. SceneApplication newApp = new SceneApplication();
02. ObjectNode answer = Json.newObject();
03. int hashCode = manager.putApp(newApp, newApp.hashCode());
04. answer.put("app", newApp.getName());
05. answer.put("id", hashCode);
06. return answer;

```

CÓDIGO 4.4 - INSERÇÃO DE APLICAÇÃO

4.3.2.2 Gerenciando Dados de uma Aplicação

Os dados de uma aplicação cliente só podem ser gerenciados através de três métodos que o serviço oferece, sendo eles *insert* para inserção de dados, *update* para atualização dos dados e *delete* para exclusão dos dados na *Working Memory* disponível no *scene-core*.

O JSON schema para a utilização dos três métodos é o mesmo, e, como na inserção de uma aplicação, os dados recebidos têm que ser validados. Portanto, antes de designar qual tipo de gerenciamento de dados será feito, uma verificação sobre a existência da aplicação é feita e, se a aplicação existe, logo em seguida uma verificação do campo *type* (tipo) é feita para se determinar qual operação deve ser realizada.

O Código 4.5 representa a validação dos campos recebidos pelo JSON. O código também mostra a montagem da mensagem de erro caso algum dos campos requeridos não esteja consistente.

```

01. SceneApplication app = manager.getApp(key);
02. ObjectNode answer = Json.newObject();
03.
04. if(app == null) {
05.     answer.put("error", "There is no application with key " + key);
06.     return answer;
07. }
08. if(!node.has("type")) {
09.     answer.put("error", "There is no type field.");
10.    answer.put("hint", "insert, update or delete.");
11.    return answer;
12. }

```

CÓDIGO 4.5 - VALIDAÇÃO DOS CAMPOS NECESSÁRIOS PARA GERENCIAMENTO DE DADOS

Uma validação dos atributos de cada objeto recebido pelo JSON não é possível, porque o Play Framework não tem acesso ao modelo da aplicação provida pelo cliente. Portanto, a validação dos dados enviados é feita pelo scene-core, e, se algum erro acontece na operação, o scene-core gera uma exceção que é tratada pelo Play framework para que o cliente receba uma informação mais detalhada do erro gerado.

4.3.2.3 Gerenciando Assinaturas

A requisição de assinatura, se correta, resulta na instanciação de um objeto Subscription (Código 4.6) e sua inserção na *working memory* de uma respectiva aplicação. No processo, é atribuido um id à subscrição, além relacioná-la a um SituationType e Actor quando a configuração presente na requisição assim o exigir.

Caso a requisição apresente o campo "situation", o serviço realizará uma busca do metadado de tipo de situação (*SituationType*) cujo nome corresponda ao valor fornecido, no contexto da aplicação. Se este campo não for informado, assume-se que a subscrição abrange qualquer tipo de situação da aplicação e o atributo *type* do objeto *subscription* é **nulo**. Caso a requisição apresente o campo "actor", o serviço buscará uma entidade referente à classe fornecida no subcampo "type" e cuja identificação seja igual ao valor do subcampo "id". Se o campo "actor"

não é fornecido, assume-se que a subscrição abrange qualquer indivíduo que participe de uma determinada situação e o atributo *actor* do objeto *subscription* torna-se nulo.

```

01. declare Subscription
02.   id: String
03.   webhook: String
04.   type: SituationType
05.   actor: Actor
06. end

```

CÓDIGO 4.6 - DECLARAÇÃO DA CLASSE SUBSCRIPTION

Uma vez bem sucedida, a requisição de subscrição é respondida informando o identificador do subscriber (*subscriberId*), o *id* do objeto de subscrição, atribuído pelo próprio serviço. Esse identificador servirá à aplicação *subscriber* caso a mesma deseje cancelar sua subscrição num outro momento.

No momento que o objeto de subscrição é inserido na *working memory* da aplicação, a regra de subscrição (Código 4.7) é passível de ser disparada, dada uma subscrição (*Subscription*) e uma dada situação (*Situation*) (de um tipo específico ou de qualquer tipo) onde exista uma participação (*Participation*) (de um participante específico ou qualquer participante) na mesma.

```

01. rule Subscription
02. when
03.   $sub: Subscription($type: type, $actor: actor)
04.   $sit: Situation($type == null || type == $type)
05.   exists (Participation((situation == $sit) && ($actor == null || actor == $actor)))
06. then
07.   ws.url($sub.webhook).post(Json.toJson($sit));
08. end

```

CÓDIGO 4.7 - REGRA DE ASSINATURA

Sua consequência é efetuar um requisição POST para o serviço origem da subscrição, fornecido pela sua url *webhook*. O conteúdo da requisição é será um snapshot da situação como o exemplo em (Código 4.8). Vale ressaltar que uma a

regra de subscrição é capaz disparar tanto para a ativação quanto a desativação de uma instância de situação.

```

01. {
02.     "name": "br.ufes.inf.lprm.scene.examples.feverjson.Fever",
03.     "active": true,
04.     "activation": 1481164482975,
05.     "participations": [
06.         {
07.             "kind": "br.ufes.inf.lprm.scene.examples.feverjson.Person",
08.             "label": "f1",
09.             "name": "Maria",
10.             "temperature": 38,
11.             "id": 2
12.         }
13.     ]
14. }
```

CÓDIGO 4.8 - REQUISIÇÃO PARA SERVIÇO DE ORIGEM

Para o cancelamento da assinatura, assim que a requisição DELETE (seção 4.2.4) é recebida com o “subscriberId” do assinante a query mostrada no Código X entrega as informações necessárias para que o serviço, possa excluir o *FactHandle* correspondente ao objeto da assinatura.

```

01. query SubscriptionQuery (String subscriberId)
02.     subscription: Subscription(id == subscriberId)
03. end
```

CÓDIGO 4.9 - QUERY PARA LOCALIZAÇÃO DO OBJETO SUBSCRIPTION

A query acima faz uma busca dentro da *Working Memory* por fatos que são do tipo *Subscription* com o id esperado, para que o serviço possa identificar qual assinatura cancelar.

5 ESTUDO DE CASO

5 ESTUDO DE CASO

Medicamentos ou produtos de tratamento estético que precisam ser permanentemente monitorados devem ser mantidos em geladeira ou freezer devido às especificações e recomendações estabelecidas pelos fabricantes. Dessa forma, o médico precisa ser informado/alertado periodicamente sobre a temperatura do produto, principalmente na ocorrência de situações indesejadas. O monitoramento da temperatura do produto deve, preferencialmente, ser feito desde o recebimento do produto até o término do uso. Em caso de ocorrência de aumento da temperatura do produto, por qualquer motivo, fora da faixa especificada pelo fornecedor, os interessados devem ser informados para que seja tomada alguma medida para evitar a perda do produto.

Aplicações dessa natureza que fazem uso de IoT para automatizar o monitoramento da temperatura de produtos envolvem e demandam o uso de tecnologias de hardware e software e vários profissionais interessados no desenvolvimento e uso da aplicação final.

O estudo de caso se baseia em um monitoramento e controle de temperatura de frascos de Botox (Toxina Botulínica, 2016) que são utilizados por médicos dermatologistas para tratamentos estéticos ou de saúde.

Normalmente, o médico que faz aplicação de Botox em pacientes, faz compras mensais ou programadas do produto, adquirindo uma quantidade maior que precisa ser mantida em freezer antes do uso e, em geladeira, após a diluição. Uma compra programada pode conter aproximadamente 20 frascos de 100 unidades cada. Dessa forma, o médico precisa armazenar e monitorar a temperatura dos frascos de Botox que normalmente são recebidos por serviços rápidos de entrega de encomendas.

Os produtos são recebidos em um isopor de tamanho pequeno com barras de gelo em gel. Com isolamento e lacres apropriados o produto pode ficar aproximadamente 24 horas em transporte sem afetar a qualidade. Esse período de transporte é garantido pelo fabricante do produto.

Após o recebimento do isopor, contendo aproximadamente 10 caixas com o Botox é preciso armazenar em freezer em uma temperatura menor ou igual a -5°C. Após a diluição com o soro para o uso conforme bula e protocolo adotado pelo

médico, o produto deve ser mantido somente em geladeira sob um intervalo de temperatura entre 2°C e 8°C por um período de no máximo de 3 dias.

Neste estudo de caso o foco será apenas para o controle de temperatura do frasco de Botox a vácuo que ainda não foi utilizado ou diluído. Dessa forma, a temperatura do aparelho de refrigeração deve ser igual ou inferior a -5°C.

Assim que o isopor com os frascos é recebido o próprio médico deve verificar as condições do produto e, se for possível, fazer a medição da temperatura do interior do isopor com algum medidor manual de temperatura para verificar se está igual ou inferior a -5°C. Após a verificação o produto deve ser armazenado no freezer do consultório do médico que deve ser cadastrado no sistema assim que o produto for registrado, pois o sistema deve iniciar o monitoramento imediatamente após o armazenamento do produto dentro do freezer do consultório do médico. Toda vez que os produtos atingirem uma temperatura maior que -5°C o Médico deve ser informado para que alguma providência seja tomada. Normalmente, quando se identifica algum problema no Freezer é preciso transportar o produto para outro Freezer.

Baseado na informação obtida pelo cenário proposto acima, podemos identificar três situações de relevância. Sendo suas classificações descritas abaixo condizentes com o modelo apresentado abaixo (seção 5.1).

1. *IncreasingTemperature* – Quando a temperatura registrada por um sensor aumenta em um grau celcius dentro de uma janela de três minutos de tempo.
2. *IncreasingDistance* – Quando a distância aumenta em 50 unidades de medida dentro de uma janela de dez minutos.
3. *IncreasingBoth* – Quando as duas situações acima estão acontecendo ao mesmo tempo e seus respectivos *Actors* (seção 3.4.1) são os mesmos.

5.1 MODELO

Na Figura 5.1 está descrito o modelo proposto para o cenário acima.

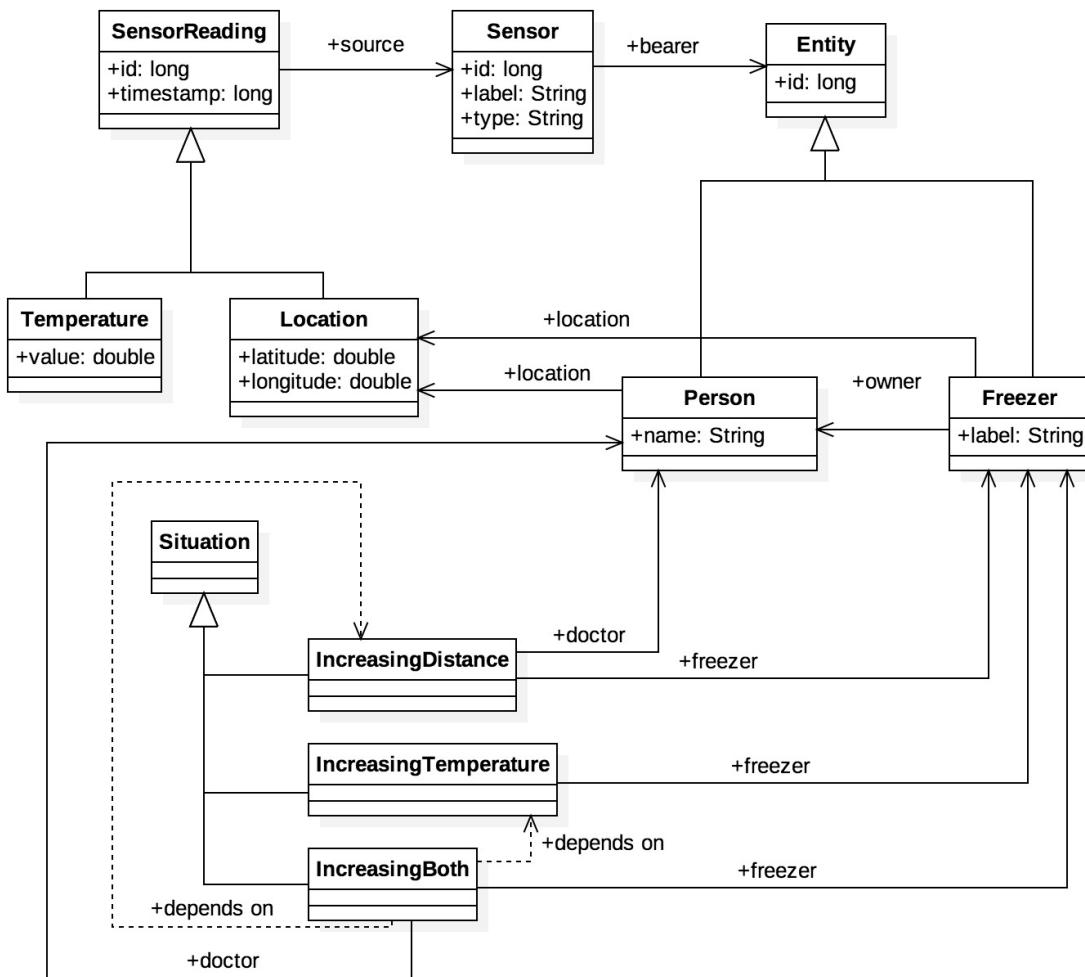


FIGURA 5.1 - MODELO DO ESTUDO DE CASO

A classe *Entity* tem um identificador “id” e através de sua hierarquia, podemos observar as classes *Person* e *Freezer*. Ambas as classes derivadas de *Entity* possuem uma relação de localidade com a classe *Location*, onde nada mais é que uma posição dada em latitude e longitude. Sensores do tipo *Sensor* estão relacionados com uma *Entity*, pois é necessário saber qual entidade que o sensor está monitorando. Cada *Sensor* tem um *SensorReading* atribuído como sua fonte de aquisição de dados. Portanto, por serem descendentes de *SensorReading*, *Temperature* e *Location*, além de seus dados, também possuem uma identificação e um timestamp relacionados.

Através da hierarquia de *Situation*, podemos observar o tipo de situação *IncreasingTemperature* que possui um *Freezer* (atuando como *actor*), cujo sensor de temperatura é utilizado para o monitoramento do comportamento do freezer ao longo do tempo. E também temos *IncreasingDistance* que possui um *Freezer* e uma *Person*, ambos possuem uma *Location* relacionada, para tornar possível o monitoramento da distância entre eles.

Podemos verificar que a situação *IncreasingBoth* tem uma relação de dependência com as situações *IncreasingTemperature* e *IncreasingDistance*, além de suas relações com *Person* e *Freezer*. Isto se dá, pelo motivo que as duas situações tem que estar acontecendo para uma mesma pessoa que possui um freezer relacionado.

5.2 REGRAS DAS SITUAÇÕES

As regras descritas abaixo fazem a detecção de situações de acordo com o cenário e seu modelo descrito na seção 5.1.

```

01. rule IncreasingTemperature
02.   @role(situation)
03.   @type(IncreasingTemperature)
04. when
05.   $freezer: Freezer()
06.   $sensor: Sensor(bearer == $freezer, type == "temperature")
07.   Number( doubleValue > 1 )
08.     from accumulate( Temperature( source == $sensor, $value: value )
09.   over window:time( 3m ), variance( $value ) )
10.   then
11.     SituationHelper.situationDetected(drools);

```

CÓDIGO 5.1 - REGRA DE DETECÇÃO DE AUMENTO DE TEMPERATURA

O código 5.1 mostra a regra de situação em DRL para o tipo de situação *IncreasingTemperature*. Se um *Sensor* analisa temperatura (linha 06) e pertence a um *Freezer* (linha 05) e essa temperatura analisada aumenta em um grau celcius numa janela de tempo de três minutos cresce em um grau celcius uma situação do tipo *IncreasingTemperature* é detectada.

Em ordem para calcular o total do aumento dessa temperatura em um período de três minutos, a função *accumulate* de Drools é usada. O seu condicional tem três partes, uma define o padrão, uma usa funções pré-definidas e uma terceira define uma restrição. Na linha 08 um padrão é definido na primeira parte da função *accumulate*, visando capturar todas as temperaturas relacionadas com o *Sensor* previamente descrito numa janela de tempo de três minutos para que todas as temperaturas relacionadas com o *Sensor* sejam capturadas. Na ultima parte da função uma função calcula a variância entre todos os valores acumulados e checa se essa variância é maior que o valor 1 na linha 07, onde 1 significa um grau celcius.

```

01. rule IncreasingDistance
02.   @role(situation)
03.   @type(IncreasingDistance)
04.   when
05.     $doctor: Person(location != null)
06.     $freezer: Freezer(owner == $doctor, location != null)
07.     $doctorGPS: Sensor(bearer == $doctor, type=="location")
08.     Number( doubleValue > 50 )
09.       from accumulate( $doctorLoc: Location( source == $doctorGPS ) over
10.         window:time( 10m ), variance( distance($doctorLoc, $freezer.getLocation() ) ) )
11.   then
12.     SituationHelper.situationDetected(drools);
13.   end

```

CÓDIGO 5.2 - REGRA DE DETECÇÃO DE AUMENTO DE DISTÂNCIA

Se uma *Person* (linha 05) tem uma localização diferente de nula e possui um *Freezer* (linha 06) com uma localização também válida e um GPS (linha 07) relacionado. Quando esta *Person* se distanciando 50 metros em uma janela de dez minutos a situação *IncreasingDistance* é ativada.

Em ordem para calcular o total do aumento dessa distância em um período de dez minutos, a função *accumulate* de Drools é usada. Na linha 09 um padrão é definido na primeira parte da função *accumulate*, visando capturar todas as *Locations* relacionadas com o *Sensor* previamente descrito uma janela de tempo de dez minutos para que todas as temperaturas relacionadas com o *Sensor* sejam capturadas. Na ultima parte da função uma função calcula a variância entre todos os

valores acumulados e checa se essa variância é maior que o valor 50 na linha 07, onde 50 significa cinquenta metros.

```
01. rule IncreasingBoth
02.   @role(situation)
03.   @type(IncreasingBoth)
04.   when
05.     $doctor: Person()
06.     $freezer: Freezer(owner == $doctor)
07.     $distance: IncreasingDistance(freezer == $freezer, doctor == $doctor)
08.     $temperature: IncreasingTemperature(freezer == $freezer)
09.   then
10.     SituationHelper.situationDetected(drools);
11.   end
```

CÓDIGO 5.3 - REGRA DE SITUAÇÃO DE SITUAÇÕES PARA DETECÇÃO DE AUMENTO DE TEMPERATURA E DISTÂNCIA AO MESMO TEMPO

Se a existe uma situação de aumento de temperatura *IncreasingTemperature* (linha 08) e uma situação de aumento de distância *IncreasingDistance* (linha 07) relacionados a uma *Person* (linha 05) e *Freezer* (linha 06) que pertence à pessoa (*Person*), uma situação de situações *IncreasingBoth* é detectada.

6 CONCLUSÕES

6 CONCLUSÕES

Desde o seu desenvolvimento a plataforma SCENE (Pereira, 2013) foi empregada em diversos trabalhos como em (Costa, 2016) e (Moreira, 2015), principalmente em cenários de saúde pública. A primeira versão de SCENE obteve bom reconhecimento da comunidade acadêmica. Contudo, como em qualquer tecnologia, SCENE precisou de avanços. Graças ao considerável número de usuários SCENE (e, naturalmente, de reflexões de seus próprios desenvolvedores), foi possível obter feedback reportando as maiores dificuldades no uso da plataforma.

Partindo da necessidade do avanço da plataforma SCENE, este trabalho propôs um modelo independente de plataforma para a gerência de situações. Este modelo, juntamente com uma estrutura de escrita (seção 3.5) baseada em *Drools Rule Language (DRL)* concebida com o propósito de facilitar o uso da plataforma. Uma plataforma que permite aos clientes desenvolver, executar e gerenciar aplicações sem a complexidade de criar e manter a infraestrutura normalmente associada ao desenvolvimento e lançamento de uma aplicação. Assim uma nova versão da plataforma SCENE foi gerada, trazendo as melhorias mencionadas acima.

O desenvolvimento de SCENE, por ter sido baseado em JBoss Drools, ficou atrelado à ferramenta. Para contornar este problema, foi proposto uma camada de aplicação responsável por gerenciar todas as comunicações dos clientes SCENE com a plataforma. Portanto as funcionalidades do “scene-core” não estão disponíveis diretamente para o cliente, fazendo com que o desenvolvimento dos clientes seja independente de plataforma. SCENE atua somente como uma “caixa preta” para seus clientes, na qual a troca de informação entre cliente e plataforma é definida pela camada de aplicação do serviço. Portanto o serviço (SMaaS) pode atender mais de uma requisição ao mesmo tempo, permitindo que aplicações que o utilizam possam ser escritas em diferentes linguagens ou baseadas em diferentes ferramentas, um desacoplamento entre clientes e plataforma foi gerado. O cliente se comunica com o serviço através de uma aplicação RESTfull utilizando requisições HTTP tais como GET e POST. O cliente pode ter dois tipos de comportamento. Administradores, são clientes capazes de inserir aplicações, seus contextos são gerenciados isoladamente. Assinantes, são capazes de visualizar todo o contexto de uma determinada aplicação, mas sem alterá-lo.

6.1 TRABALHOS FUTUROS

Com relação à SCENE alterações feitas na plataforma, tornaram o editor de DRL um tanto inadequado em alguns casos. Por exemplo, nas declarações de operadores temporais entre situações, o analisador sintático não reconhece estas operações como válidas. Versões futuras podem prover alterações no analisador a fim de suportar plenamente uma nova sintaxe.

Com relação ao modelo de uma aplicação cliente dentro do serviço SMaaS, as aplicações inseridas no serviço não podem ter o seu modelo alterado. Portanto, uma melhoria necessária no serviço seria permitir atualização desses modelos.

Com relação ao serviço, o gerenciamento de uma aplicação não está completo, pois por enquanto o serviço suporta somente a inserção de aplicações (faltando as funcionalidades de modificação e retirada de aplicações). As próximas versões do serviço, podem implementar estas funcionalidades.

Não existe persistência de dados no serviço proposto, portanto deixando a quantidade de clientes limitada pela capacidade de memória da máquina de processamento. Determinar como a indicação desta persistência de dados será feita, o cliente ou o serviço tomará a iniciativa dessa persistência.

Durante o processo de desenvolvimento do serviço SMaaS, não houve uma estimativa do impacto da arquitetura sobre sua performance. Uma linha de trabalho futura é a de analisar as decisões arquiteturais à luz do desempenho do serviço, utilizando cenários de situação robustos, que se aproximem dos reais, inúmeras requisições seriam feitas dentre várias aplicações rodando ao mesmo tempo no serviço.

REFERÊNCIAS

- Schilit, Bill, Norman Adams, and Roy Want. "Context-aware computing applications." *Mobile Computing Systems and Applications, 1994. WMCSA 1994. First Workshop on.* IEEE, 1994.
- H. Lieberman and T. Selker, *Out of context: Computer systems that adapt to, and learn from, context*, (2000).
- Abowd, Gregory D., et al. "Towards a better understanding of context and context-awareness." *International Symposium on Handheld and Ubiquitous Computing*. Springer Berlin Heidelberg, 1999.
- Costa, Patricia Dockhorn, et al. "Situations in conceptual modeling of context." *2006 10th IEEE International Enterprise Distributed Object Computing Conference Workshops (EDOCW'06)*. IEEE, 2006.
- Dockhorn Costa, Patrícia. *Architectural support for context-aware applications: from context models to services platforms*. University of Twente, 2007.
- Anagnostopoulos, Christos B., Yiorgos Ntarladimas, and Stathes Hadjiefthymiades. "Situational computing: An innovative architecture with imprecise reasoning." *Journal of Systems and Software* 80.12 (2007): 1993-2014.
- Costa, Patricia Dockhorn, et al. "A model-driven approach to situations: Situation modeling and rule-based situation detection." *Enterprise Distributed Object Computing Conference (EDOC), 2012 IEEE 16th International*. IEEE, 2012.
- Pereira, Isaac SA, Patrícia Dockhorn Costa, and João Paulo A. Almeida. "A rule-based platform for situation management." *Cognitive Methods in Situation Awareness and Decision Support (CogSIMA), 2013 IEEE International Multi-Disciplinary Conference on*. IEEE, 2013.
- J. Ye, S. Dobson & S. McKeever, *Situation identification techniques in pervasive computing: A review*, (2011)
- Deyo, Jeremy. "Software as a Service (SaaS)." (2008).
- Stankov, Ivo E., and Rastislau Datsenka. "Platform-as-a-Service as an Enabler for Global Software Development and Delivery." *MKWI*. 2010.
- Pichler, Mario, and Diethard Leber. "On the formalization of expert knowledge: a disaster management case study." *2014 25th International Workshop on Database and Expert Systems Applications*. IEEE, 2014.
- Rendon, Oscar Mauricio Caicedo, et al. "Rich dynamic mashments: An approach for network management based on mashups and situation management." *Computer Networks* 94 (2016): 285-306.

Jakobson, Gabriel. "On modeling context in situation management." *2014 IEEE International Inter-Disciplinary Conference on Cognitive Methods in Situation Awareness and Decision Support (CogSIMA)*. IEEE, 2014.

Costa, Patrícia Dockhorn, et al. "Rule-based support for situation management." *Fusion Methodologies in Crisis Management*. Springer International Publishing, 2016. 341-364.

Moreira, João LR, et al. "Towards ontology-driven situation-aware disaster management." *Applied Ontology* 10.3-4 (2015): 339-353.

DRL. Disponível em: <<http://docs.jboss.org/drools/release/6.5.0.Final/drools-docs/html/ch08.html>>. Acesso em: 02 de nov. 2016.

Drools. Disponível em: <<http://docs.jboss.org/drools/release/6.5.0.Final/drools-docs/html/>>. Acesso em: 02 de nov. 2016.

Representational State Transfer. Disponível em: <<https://www.w3.org/2001/sw/wiki/REST>>. Acesso em: 02 de nov. 2016.

Simple Object Access Protocol. Disponível em: <<https://www.w3.org/TR/soap/>>. Acesso em: 02 de nov. 2016.

Web Services Description Languages. Disponível em: <<https://www.w3.org/TR/wsdl>>. Acesso em: 02 de nov. 2016.

Javascript Object Notation. Disponível em: <<http://www.json.org>>. Acessado em 29 de set. 2016.

Javascript Object Notation Schema. Disponível em: <<http://json-schema.org>>. Acessado em 29 de set. 2016.

Play Framework. Disponível em: <<https://www.playframework.com>>. Acessado em 29 de set. 2016.

Toxina Botulinica. Disponível em: <http://www.allergan.com.br/Bulas/Documents/Botox_profissional.pdf>. Acesso em: 02 de nov. 2016.

APÊNDICE 1

APÊNDICE 1 - REQUISIÇÕES

Este apêndice contém todos os JSON Schemas das requisições utilizadas pelo serviço proposto, juntamente com suas respectivas respostas.

1 - POST /app

Request Body:

```

01. {
02.   "type": "object",
03.   "properties": {
04.     "application": {
05.       "type": "string"
06.     },
07.     "version": {
08.       "type": "string"
09.     },
10.     "files": {
11.       "type": "array",
12.       "items": {
13.         "type": "object",
14.         "properties": {
15.           "name": {
16.             "type": "string"
17.           },
18.           "package": {
19.             "type": "string"
20.           },
21.           "content": {
22.             "type": "string"
23.           }
24.         },
25.         "required": [
26.           "name",
27.           "package",
28.           "content"
29.         ]
30.       }
31.     },
32.     "required": [
33.       "application",
34.       "version",
35.       "files"
36.     ]
37.   }
38. }
```

Response:

```

01. {
02.   "type": "object",
03.   "properties": {
04.     "app": {
05.       "type": "string"
06.     },
07.     "id": {
08.       "type": "integer"
09.     }
10.   },
11.   "required": [
12.     "app",
13.     "id"
14.   ]
15. }
```

2 - POST /app/{appId}**Request Body:**

```

01. {
02.   "type": "object",
03.   "properties": {
04.     "type": {
05.       "type": "string"
06.     },
07.     "Object": {
08.       "type": "array",
09.       "items": {
10.         "type": "object",
11.         "properties": {}
12.       }
13.     }
14.   },
15.   "required": [
16.     "type",
17.     "Object"
18.   ]
19. }
```

Response:

```

01. {
02.   "type": "object",
03.   "properties": {
04.     "success": {
05.       "type": "string"
06.     }
07.   },
08.   "required": [
09.     "success"
10.   ]
11. }
```

3 - GET /app**Response:**

```

01. {
02.   "type": "object",
03.   "properties": {
04.     "apps": {
05.       "type": "array",
06.       "items": {
07.         "type": "object",
08.         "properties": {
09.           "id": {
10.             "type": "integer"
11.           },
12.           "name": {
13.             "type": "string"
14.           },
15.           "description": {
16.             "type": "string"
17.           }
18.         },
19.         "required": [
20.           "id",
21.           "name",
22.           "description"
23.         ]
24.       }
25.     }
26.   },
27.   "required": [
28.     "apps"
29.   ]
30. }
```

4 - GET /app/{appId}/model**Response:**

```
01. {
02.     "type": "object",
03.     "properties": {
04.         "model": {
05.             "type": "array",
06.             "items": {
07.                 "type": "object",
08.                 "properties": {
09.                     "name": {"type": "string"},
10.                     "parts": {
11.                         "type": "array",
12.                         "items": {
13.                             "type": "object",
14.                             "properties": {
15.                                 "label": {"type": "string"},
16.                                 "kind": {"type": "string"},
17.                                 "fields": {
18.                                     "type": "array",
19.                                     "items": {
20.                                         "type": "object",
21.                                         "properties": {}
22.                                     }
23.                                 }
24.                             },
25.                             "required": ["label", "kind", "fields"]
26.                         }
27.                     }
28.                 },
29.                 "required": ["name", "parts"]
30.             }
31.         }
32.     },
33.     "required": ["model"]
34. }
```

5 - GET /app/{appId}/situations

Response:

```
01. {
02.     "type": "object",
03.     "properties": {
04.         "situations": {
05.             "type": "array",
06.             "items": {
07.                 "type": "object",
08.                 "properties": {
09.                     "name": {"type": "string"},
10.                     "active": {"type": "boolean"},
11.                     "activation": {"type": "integer"},
12.                     "deactivation": {"type": "integer"},
13.                     "participations": {
14.                         "type": "array",
15.                         "items": {
16.                             "type": "object",
17.                             "properties": {}
18.                         }
19.                     }
20.                 },
21.                 "required": ["name", "active", "activation",
22.                             "deactivation", "participations"]
23.             }
24.         }
25.     },
26.     "required": ["situations"]
27. }
```

6 - GET /app/{appId}/dump

Response:

```
01. {
02.     "type": "object",
03.     "properties": {
04.         "situations": {
05.             "type": "array",
06.             "items": {
07.                 "type": "object",
08.                 "properties": {
09.                     "name": {"type": "string"},
10.                     "active": {"type": "boolean"},
11.                     "activation": {"type": "integer"},
12.                     "deactivation": {"type": "integer"},
13.                     "participations": {
14.                         "type": "array",
15.                         "items": {
16.                             "type": "object",
17.                             "properties": {}
18.                         }
19.                     }
20.                 },
21.                 "required": ["name", "active", "activation",
22.                             "deactivation", "participations"]
23.             }
24.         },
25.         "objects": {
26.             "type": "array",
27.             "items": {
28.                 "type": "object",
29.                 "properties": {}
30.             }
31.         },
32.         "required": ["situations", "objects"]
33.     }
34. }
```

7 - POST /app/{appId}/subscriber**Request body:**

```

01. {
02.     "type": "object",
03.     "properties": {
04.         "webhook": {
05.             "type": "string"
06.         },
07.         "situation": {
08.             "type": "string"
09.         },
10.         "actor": {
11.             "type": "object",
12.             "properties": {
13.                 "type": {
14.                     "type": "string"
15.                 },
16.                 "id": {
17.                     "type": "integer"
18.                 }
19.             },
20.             "required": [
21.                 "type",
22.                 "id"
23.             ]
24.         }
25.     },
26.     "required": ["webhook"]
27. }
```

Response

```

01. {
02.     "type": "object",
03.     "properties": {
04.         "subscriberId": {
05.             "type": "integer"
06.         }
07.     },
08.     "required": [
09.         "subscriberId"
10.     ]
11. }
```

8 – POST**/app/{appId}/subscriber/{subscriberId}****Response:**

```

01. {
02.     "type": "object",
03.     "properties": {
04.         "subscriberId": {
05.             "type": "integer"
06.         }
07.     },
08.     "required": [
09.         "subscriberId"
10.     ]
11. }
```

