UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

EDUARDO GISCHKOW POTTER

Simulador de Ambientes Inteligentes: foco em *home care* para pessoas com idade avançada

Trabalho de Graduação.

Prof. Dr. Leandro Krug Wives
Orientador

Júlia Kikuye Kambara da Silva
Coorientador

Porto Alegre, dezembro de 2013.

AGRADECIMENTOS

Agradeço a minha família, especialmente meus pais e irmão, por todo apoio e compreensão durante o trabalho.

Também agradeço aos meus amigos, por entenderem as minhas várias ausências neste semestre.

Por fim, deixo meus sinceros agradecimentos ao meu orientador e coorientadora, pela disponibilidade e dedicação, e por terem ajudado sempre que precisei no trabalho.

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS

AJAX	Asynchronous Javascript and XML
CSS	Cascade Style Sheets
ER	Entidade-Relacionamento
HTML	HyperText Markup Language
MVC	Model-view-controller
ORM	Object-Relational Mapping
SGBD	Sistema de Gerenciamento de Banco de Dados
SMTP	Simple Mail Transfer Protocol
SOAP	Simple Object Access Protocol
WSDL	Web Services Description Language
XML	Extensible Markup Language

LISTA DE FIGURAS

LISTA DE TABELAS E QUADROS

RESUMO

Atualmente há grande interesse pelos sistemas de *home care*. Tais sistemas visam a permitir que pessoas com necessidades de assistência possam continuar vivendo em suas próprias casas, mesmo que sua saúde e autonomia estejam diminuindo. Essa tecnologia inclui sensores, dispositivos, dados e infraestrutura computacional, além de sistemas inteligentes para o seu controle. O objetivo deste trabalho é propor um simulador de ambiente inteligente que torne possível simular sistemas de *home care,* focado em pessoas com idade avançada. No simulador deverá ser possível configurar dinamicamente a residência desejada, adicionando cômodos, sensores, dispositivos, objetos e pessoas que se encontram na casa. Para as pessoas, é possível descrever suas deficiências. Os dispositivos, objetos e sensores oferecem serviços através de *web services*, sendo possível interagir com eles. Com isso, o sistema permite simular diferentes configurações de ambientes, e tentar validar diversas situações propostas.

Palavras-Chave: *home care*, simulador, ambientes inteligentes.

Smart Environment Simulator: Focus on Home Care for People at an Advanced Age

ABSTRACT

Currently, there is a great interest for home care systems. These systems allow people with disabilities and that need personal assistance to live their own life in their own homes, even if their health and autonomy were diminishing. This technology includes sensors, appliances, data and computational infrastructure, besides intelligent systems to provide their control. The goal of this work is to propose a smart environment simulator that allows us to simulate home care systems focused on elderly people. The simulator allows setting up dynamically the desired house, adding rooms, people, devices, sensors and appliances. It is possible to define the disabilities of each people, define the functionalities of each device, appliance and sensor. The functionalities are represented by Web services, and through them, it is possible to interact with the devices that were added to the house in the set up phase, so that it is possible to simulate different environments, and validate several situations.

Keywords: home care, simulator, smart environment.

1 INTRODUÇÃO

Em 1950, segundo a UNFPA (2012), existiam 205 milhões de pessoas no mundo com 60 anos de idade ou mais. Até 2012, o número de pessoas com essa idade tinha subido para quase 810 milhões. Projeta-se, que, até 2050, esse número tenha ultrapassado os dois bilhões de pessoas.

Dentro desse contexto de aumento da população idosa, surgiram os sistemas de *home care*. De acordo com McGee-Lennon (2008), sistemas de *home care* podem ser definidos como tecnologia utilizada para dar suporte ao cumprimento de tarefas de rede, provendo os meios para coletar, distribuir, analisar e gerenciar informações relacionadas ao cuidado ou assistência. Essa tecnologia tipicamente inclui sensores, dispositivos, visores, dados, redes e infraestrutura computacional. Sistemas de *home care* visam a permitir que pessoas com necessidades de assistência possam continuar vivendo em suas próprias casas, mesmo que sua saúde e autonomia estejam diminuindo.

Muitos artigos foram propostos no assunto, como Machado et al. (2013), Kambara-Silva et al. (2014). Eles sugerem maneiras para modelar casos de interesse, nos quais os sistemas de home care tentam resolver problemas do cotidiano do paciente. Entretanto, esses modelos são teóricos, e ainda não foram verificados. Segundo Machado et al. (2013), ambientes inteligentes equipados com sensores e atuadores representados por *Web Services* poderiam ser controlados utilizando-se processos de negócio.

Diante disso, este trabalho tem como objetivo desenvolver um simulador de ambientes de *home care*. A ideia geral consiste em permitir a especificação e configuração dos elementos (sensores e atuadores) presentes em um ambiente inteligente. Através dele, será possível simular processos de negócio aplicados a diferentes configurações.

Os dispositivos serão representados como *web services*, conforme proposto por Machado et al. (2013). Assim, será possível simular diferentes situações em um ambiente *home care*, possibilitando, com isso, a verificação do comportamento de alguns processos de negócio nessas situações específicas.

O trabalho está dividido em cinco capítulos. O segundo capitulo abordará os requisitos do sistema, mostrando o que deve ser feito para o sistema atingir seu objetivo. No terceiro capítulo será apresentado como o sistema foi desenvolvido: as tecnologias utilizadas, sua arquitetura, o modelo do banco de dados, a estrutura de classes do sistema e imagens mostrando as diversas telas do sistema. No quarto capítulo serão apresentadas duas simulações de exemplo, feita utilizando processos de negócio. Por fim, no quinto e último capítulo, serão reafirmados os objetivos iniciais deste trabalho, bem como sugeridos alguns trabalhos futuros.

2 REQUISITOS DO SISTEMA

Neste capítulo será proposto um sistema capaz de simular um ambiente inteligente utilizando *Web services* para simular os dispositivos. Nele serão apresentados os requisitos funcionais que o sistema deve ter para atingir o seu objetivo.

2.1 Requisitos Funcionais

Nesta seção serão apresentados os requisitos funcionais necessários ao simulador, para que o objetivo inicial seja atingido, isto é, para que seja possível simular diferentes processos de negócio com diversas configurações de ambiente. Os requisitos serão apresentados na forma de *user stories*, que basicamente descrevem uma funcionalidade de valor para o cliente ou comprador de um sistema, segundo Cohn (2004).

2.1.1 User Stories

Aprofundando o conceito citado anteriormente, tem-se que segundo Cohn (2004) uma boa história deve ser independente, negociável, agregar valor na visão do usuário, estimável, pequena e testável.

A Tabela 2.1 apresenta as *user stories* definidas para este projeto. As suas respectivas colunas serão explicadas a seguir:

- Id: É apenas um identificador único para cada *user story*.
- Como: representa o papel do usuário que executa a ação. Existem dois papéis, que são o papel de administrador, que configura o ambiente, e o papel de cliente, que é aquele que utiliza os Web services disponibilizados pelo ambiente.
- Quero: representa o que o sistema deve fazer, ou seja, a funcionalidade desejada pelo usuário.
- Para: é o objetivo final da *user story*. Também pode ser visto como o benefício. Nem sempre é obrigatório.

Tabela 2.1: *User stories* previstas para este sistema.

Id	Como	Quero	Para
1	administrador	adicionar um quarto	montar a estrutura inicial do ambiente a ser simulado
2	administrador	remover um quarto	
3	administrador	editar um quarto	
	administrador	adicionar dispositivo a um	configurar o ambiente de

Id	Como	Quero	Para
		quarto	simulação
3	administrador	remover dispositivo de um quarto	
4	administrador	editar um dispositivo	mudar a localização do dispositivo ou os seus status
5	administrador	adicionar pessoa	
6	administrador	remover pessoa	
7	administrador	editar pessoa	mudar o tipo, deficiências e status de uma pessoa
8	administrador	adicionar organização	
9	administrador	editar organização	
10	administrador	remover organização	
11	administrador	ver o log do sistema	saber sobre as diversas simulações que já aconteceram
12	administrador	acessar a tela de simulação atual	obter feedback visual sobre os status atuais dos diversos dispositivos e pessoas
13	administrador	iniciar uma simulação	habilitar os web services
14	administrador	finalizar uma simulação	configurar e iniciar uma nova simulação
15	administrador	visualizar lista de dispositivos	saber dispositivos que podem ser adicionados a um quarto
16	administrador	visualizar wsdl de um dispositivo	
17	administrador	visualizar funcionalidades de um dispositivo	saber quais são as pré condições das funcionalidades
18	administrador	adicionar objeto	adicionar objeto a um cômodo
19	administrador	remover objeto	remover objeto de um cômodo
20	cliente	executar operações sobre os dispositivos	simular um ambiente inteligente
2	cliente	chamar organização	
21	cliente	saber em quanto tempo a unidade de uma organização chegará a casa	
22	cliente	saber em que quarto uma pessoa está	interagir com a pessoa através de dispositivos
23	cliente	saber em que quarto um dispositivo está	
24	cliente	listar todos dispositivos que podem executar determinada funcionalidade	
25	cliente	listar todos dispositivos em	saber quais dispositivos

Id	Como	Quero	Para
		um quarto	posso usar em quarto
27	cliente	listar todos os quartos	saber os quartos da casa
28	cliente	listar todas as pessoas	saber as pessoas da casa
29	cliente	saber os dados de uma pessoa	saber os detalhes de uma pessoa
30	cliente	listar objetos em um cômodo	
31	cliente	saber a localização de um objeto	
32	cliente	fazer as requisições utilizando SOAP	
33	cliente	consultar uma WSDL para cada dispositivo	poder simular diversas situações diferentes
34	cliente	consultar uma WSDL para cada funcionalidade diferente	

No próximo capítulo será apresentado o projeto do simulador em si. Serão mostradas as tecnologias utilizadas no sistema, a arquitetura escolhida para o sistema, a modelagem do banco de dados utilizada, através de um diagrama ER (Entidade-Relacionamento). Também será mostrado a estrutura de classes do sistema, e, por fim, as diversas telas do sistema serão apresentadas e explicadas através de figuras.

3 PROJETO E DESENVOLVIMENTO DO SIMULADOR

No capítulo anterior foram apresentados os requisitos funcionais necessários ao simulador para que ele atingisse os seus objetivos. Este capítulo apresenta em detalhes o sistema que foi construído.

Primeiramente, serão apresentadas as tecnologias utilizadas no sistema e também as diversas decisões de projeto tomadas. A segunda seção descreve a arquitetura que foi utilizada na aplicação. A terceira seção apresenta a modelagem de banco de dados que foi utilizada, através de um diagrama ER. A quarta seção explica a estrutura de classes do sistema. Por fim, na quinta seção, são mostradas imagens do sistema desenvolvido e também uma análise e explicação dessas diversas telas.

3.1 Ferramentas Utilizadas

Esta seção mostra as ferramentas utilizadas no desenvolvimento do sistema proposto, junto com decisões de projeto que foram tomadas com base nos requisitos listados no capítulo anterior.

3.1.1 Listagem das Ferramentas Utilizadas

O simulador foi desenvolvido para o ambiente Web, em uma arquitetura cliente-servidor. A interface do simulador foi feita utilizando HTML, Javascript e CSS. Para o banco de dados, foi utilizado o PostgreSQL. Como linguagem para o servidor, foi utilizado Ruby on Rails. O simulador está hospedado no servidor Heroku (http://heroku.com). Os Web services utilizam o protocolo SOAP (*Simple Object Access Protocol*) e estão descritos em WSDL (*Web Services Description Language*).

3.1.2 Web

Por ser mundialmente utilizada, e ter a vantagem de ser multiplataforma, a Web foi o ambiente escolhido para desenvolvimento do sistema.

Além disso, a Web oferece protocolos abertos e comumente aceitos, tornando o sistema interoperável através de Web services (serviços Web). Os Web services desenvolvidos para o simulador aqui desenvolvido são disponibilizados via servidor. Além disso, optou-se por fazer uma interface de configuração das simulações utilizando-se as facilidades que a Web proporciona, como, por exemplo, o fato de por estar hospedado na web o sistema pode ser acessado remotamente. Com isso, todas as partes que integram o sistema estão hospedadas na Web.

3.1.3 CSS

Como foi decidido que o sistema estaria hospedado na Web, o seu leiaute foi feito utilizando CSS (*Cascade Style Sheets*). CSS é uma linguagem utilizada para descrever a apresentação de uma página HTML. O principal benefício do CSS é prover a separação entre o formato e o conteúdo de um documento. Com a formatação fora do documento, quando o desenvolvedor deseja modificar alguma coisa no leiaute da aplicação, ele precisa apenas modificar o arquivo de estilos. Essa separação, além de deixar o código mais legível e facilitar a manutenção, permite que um mesmo estilo seja compartilhado por várias páginas.

Para o desenvolvimento do simulador foi utilizado o Bootstrap (http://getbootstrap.com/), que é um conjunto de diversas ferramentas para criar websites e aplicações Web, sendo disponibilizado gratuitamente. Ele contém diversos temas prontos que auxiliam na estilização de uma página.

3.1.4 Javascript

Javascript é uma linguagem de programação interpretada. É mais popularmente utilizada em navegadores, e, nesse contexto, permite que o código possa interagir com o usuário, controlar o navegador e alterar o conteúdo do documento que aparece dentro da janela do navegador. É normalmente chamado de *client-side* Javascript para enfatizar que o código executa apenas do computador do cliente, sem passar pelo servidor (FLANAGAN, 2006).

O fato de os *scripts* poderem ser executados apenas no lado do cliente é uma grande vantagem. Além disso, a invenção do AJAX (*Asynchronous Javascript and XML*) tornou o Javascript ainda mais poderoso. Segundo Ullman (2012), AJAX é o uso do Javascript para fazer requisições assíncronas ao servidor. Com isso, as páginas Web se tornam mais interativas, já que o conteúdo pode ser carregado modificando apenas as informações que realmente sofreram mudanças, sem a necessidade de recarregar toda a página.

Diversos frameworks e bibliotecas foram desenvolvidas para Javascript. Uma delas, e que foi utilizada no trabalho, é a JQuery (http://jquery.com/). Basicamente, JQuery é uma biblioteca feita para funcionar em todos os navegadores e que tenta simplificar o código Javascript.

As três tecnologias citadas acima foram utilizadas em conjunto neste trabalho visando criar uma aplicação que tornasse a interação entre o usuário e o sistema mais simples e intuitiva, além de simplificar o código.

3.1.5 PostgreSQL

O sistema de gerenciamento de banco de dados escolhido para garantir a persistência foi o PosgreSQL. De acordo com Black (2001), PostgreSQL é um SGBD desenvolvido com projeto de código aberto, sendo hoje, dentre os SGBDs com código aberto, um dos mais avançados.

O PostgreSQL foi escolhido por ser um ótimo SGBD de código aberto, e também porque pode ser facilmente integrado com o *Active Record*, a camada do Ruby On Rails responsável pela interoperabilidade entre a aplicação e o banco de dados.

3.1.6 Ruby On Rails

Segundo Flanagan (2008), Ruby é uma linguagem de programação dinâmica com uma complexa mas expressiva gramática e uma biblioteca núcleo com uma poderosa e rica API. Ruby é inspirado em Lisp, Smalltalk e Perl, mas utiliza uma gramática que é fácil para programadores Java e C aprenderem. Ruby é uma linguagem puramente orientada a objetos, mas também aplicável para estilos de programação funcional ou procedural.

Ruby On Rails, também conhecido apenas como Rails, é um framework que torna mais fácil o desenvolvimento, *deployment* e manutenção de aplicações Web. Durante os meses que seguiram seu lançamento inicial, Rails passou de uma ferramenta desconhecida a um fenômeno mundialmente conhecido e, mais importante, tornou-se o framework de escolha para implementação de uma grande quantidade das aplicações Web, segundo Sam Ruby (2011).

Rails utiliza o padrão de arquitetura MVC (*Model View Controller*), que basicamente tenta separar os dados do modelo da aplicação, a interface de usuário e o controle lógico em três componentes, de maneira que as modificações realizadas em um desses componentes interfira minimamente nos outros. Essa arquitetura será aprofundada na seção 3.2.

Sobre a parte de modelo, de acordo com Sam Ruby (2011), Rails utiliza uma camada ORM (*Object-Relational Mapping*) para mapear tabelas de banco de dados em classes. Linhas na tabela correspondem a objetos dessa classe, e colunas da tabela correspondem a atributos dessa classe. Active Record é a camada ORM utilizada com o Rails, e segue a convenção ORM padrão que foi citada acima, com tabelas mapeando para classes, linhas para objetos e colunas para atributos. Quando um objeto é modificado, seu registro será atualizado. O fato de Rails já possuir o padrão Active Record incorporado faz com que seja muito fácil trabalhar com banco de dados em aplicações do framework.

Ruby possui atualmente uma comunidade muito ativa, trazendo com isso diversos benefícios. Essa comunidade criou o conceito de *Gems*. *Gems* são pequenas aplicações empacotadas, que resolvem algum tipo de problema. Elas possuem código aberto, e a grande maioria delas possui seu código hospedado no Github (https://github.com/). Diversas *Gems* foram utilizadas no trabalho, e elas são listadas no Quadro 3.1.

Quadro 3.1: Lista de Gems utilizadas no trabalho

Gem	Descrição
wash_out	Essa Gem foi muito importante para o trabalho. Ela é responsável por toda a parte de Web services que foi implementada, e precisou ser modificada para poder preencher alguns requisitos, como por exemplo o fato de um WSDL ser necessário por dispositivo.
best_in_place	É uma gem que utiliza jQuery para que edição e visualização de algum objeto possam ser feitos na mesma tela.
anjlab-bootstrap-rails	Serve para criar templates HTML já baseadas no bootstrap.
pg	É a interface Ruby para o PostgreSQL.

O sistema utilizou o Rails 4.0.0 e o Ruby 2.0.0, que são suas versões mais atuais.

3.1.7 Heroku

Heroku (https://www.heroku.com) é um sistema na nuvem de plataforma como serviço que suporta diversas linguagens de programação. Começou a ser desenvolvido em 2007 inicialmente suportando apenas a linguagem Ruby.

Foi o servidor escolhido para hospedar a aplicação, devido a facilidade de *deployment* que ele oferece e também pelo fato de ser gratuito para aplicações pequenas e com poucos acessos.

3.1.8 Web Services

De acordo com Alonso (2004), um *Web service* é um tipo específico de serviço que é identificado por um URI (*Uniform Resource Identifier*) e possui as seguintes características: o serviço é exposto através da Internet utilizando linguagens padrões e protocolos da própria Internet. Além disso, eles podem ser implementados através de uma interface autodescritiva, baseada em padrões abertos da Internet, como interfaces XML (*Extensible Markup Language*), por exemplo.

Neste trabalho, foram utilizadas as tecnologias SOAP (*Simple Object Access Protocol*) e WSDL *(Web Services Description Language*) para implementar os Web services. Tais tecnologias são descritas nas subseções seguintes.

3.1.9 SOAP

De acordo com Curbera (2002), SOAP foi inicialmente criado pela Microsoft e depois desenvolvido em colaboração com as empresas Developmentor, IBM, Lotus e UserLand. SOAP é um protocolo baseado em XML para troca de mensagens e chamada de procedimentos remotos. Em vez de definir um novo protocolo de transporte, SOAP trabalha com os protocolos já existentes, como HTML e SMTP (*Simple Mail Transfer Protocol*).

Segundo Curbera (2002), uma mensagem SOAP possui uma estrutura bem simples, um elemento XML com dois elementos filhos, um dos quais contém o cabeçalho e o outro o corpo, os quais são compostos por um XML arbitrário.

Em adição à estrutura básica da mensagem, a especificação SOAP define um modelo que dita como os recipientes devem processar a mensagem SOAP. O modelo da mensagem também inclui atores que indicam quem deve processar a mensagem, conforme Curbera (2002).

3.1.10 WSDL

Web Services Description Language(WSDL) é um formato XML, desenvolvido pela IBM e Microsoft, utilizado para descrever Web Services como conjuntos de pontos de comunicação que podem trocar certas mensagens. Em outras palavras, um documento WSDL descreve a interface de um Web Service e provê um ponto de contato ao usuário (CURBERA, 2002).

Neste trabalho, é possível consultar um WSDL com todas as funcionalidades do simulador, assim como consultar o WSDL de cada dispositivo ou ainda um WSDL de uma única operação.

3.2 Arquitetura do Sistema

Esta seção apresenta a arquitetura do sistema proposto. Basicamente, o sistema interage com o usuário, via interface Web, e com uma aplicação cliente através de requisições utilizando os Web services. Isso é mostrado na figura seguinte.

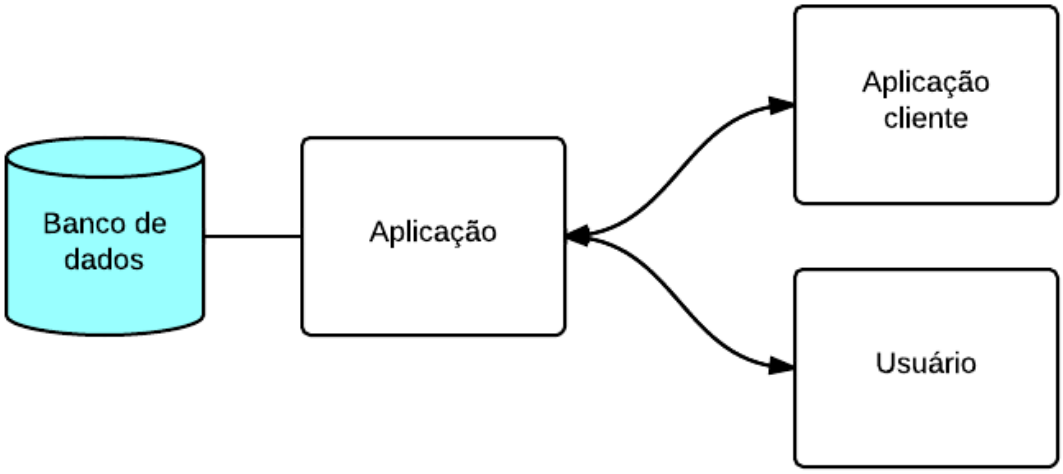


Figura 3.1: Arquitetura do sistema

O sistema, por estar na Web, é baseado na arquitetura Cliente-Servidor e segue, por ter sido desenvolvido em Ruby on Rails, o padrão MVC descrito anteriormente.

3.2.1 Cliente-Servidor

É o modelo clássico de arquitetura utilizado na Web. Basicamente, existem dois componentes, o cliente e o servidor, interagindo através de uma rede. O cliente envia uma ou mais requisições ao servidor, que é responsável por processá-las e devolver uma resposta baseada na requisição recebida.

No caso do sistema, o navegador Web utilizado pelo usuário para acessar a aplicação faz o papel de uma instância do componente cliente. Quando o usuário utiliza algum dos serviços via SOAP, a aplicação que faz a requisição também é uma instância de um cliente. O servidor que recebe essas requisições deve processá-las da maneira correta.

O fato de existir uma centralização no servidor traz algumas vantagens. Com o armazenamento de dados em apenas um serviço, fica mais fácil controlar o acesso aos recursos, garantindo assim maior segurança nas transações. Por outro lado, essa abordagem também pode sobrecarregar o servidor, e qualquer falha crítica nessa máquina pode implicar na inutilização da aplicação.

3.2.2 Padrão Model View Controller

Quando se constrói aplicações interativas, a modularidade dos componentes traz enormes benefícios. Segundo Krasner (1988), isolar unidades funcionais de outras o máximo possível faz com que seja mais fácil modificar uma das unidades, sem precisar entender todas as outras. Uma das maneiras de prover modularidade consiste em separar os componentes da aplicação em três partes: uma que representa o modelo da aplicação, outra que representa a visualização da aplicação e outra que representa o modo como o usuário interage com a aplicação, que é o controlador. Essa separação é denominada MVC (*Model View Controller*).

Quando utilizado em uma aplicação Web, a camada de Controle recebe a requisição feita pelo usuário, e é responsável então por se comunicar com as outras camadas, e a resposta final é feita pela camada de Visualização.

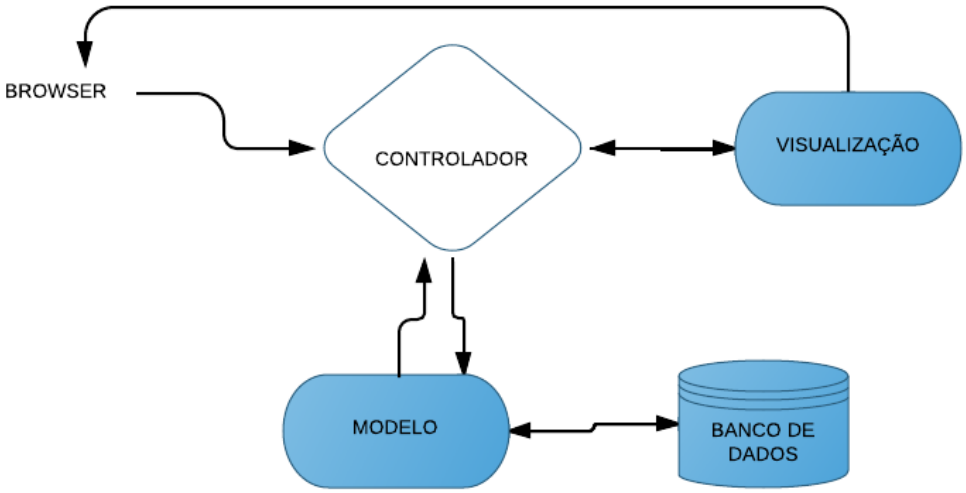


Figura 3.2: Esquema para padrão MVC voltado para Web

A camada de Modelo é a responsável pelos dados da aplicação. Ela acessa os dados, armazena-os, define os métodos que vão manipular ou alterar esses dados, além de ser responsável pelas validações e associações.

A camada de Visualização pode ser definida como aquilo que o usuário enxerga. Ela apresenta os dados do Modelo ao usuário.

A camada de controle é responsável pela comunicação entre a camada de modelo e de visualização. Ela é responsável por buscar na camada de modelo os dados corretos, e organizá-los de uma forma que se encaixe nas necessidades da camada de visualização.

3.3 Modelagem do Banco de Dados

A Figura 3.2 mostra o diagrama ER do banco de dados utilizado neste trabalho. As entidades e seus respectivos atributos são descritos nas subseções seguintes. Como todas as tabelas no banco de dados foram criadas utilizando o framework Rails, por padrão elas já possuem os atributos created_at, updated_at e id, que descrevem quando as entidades foram criadas e atualizadas, além de lhes prover um identificador único.

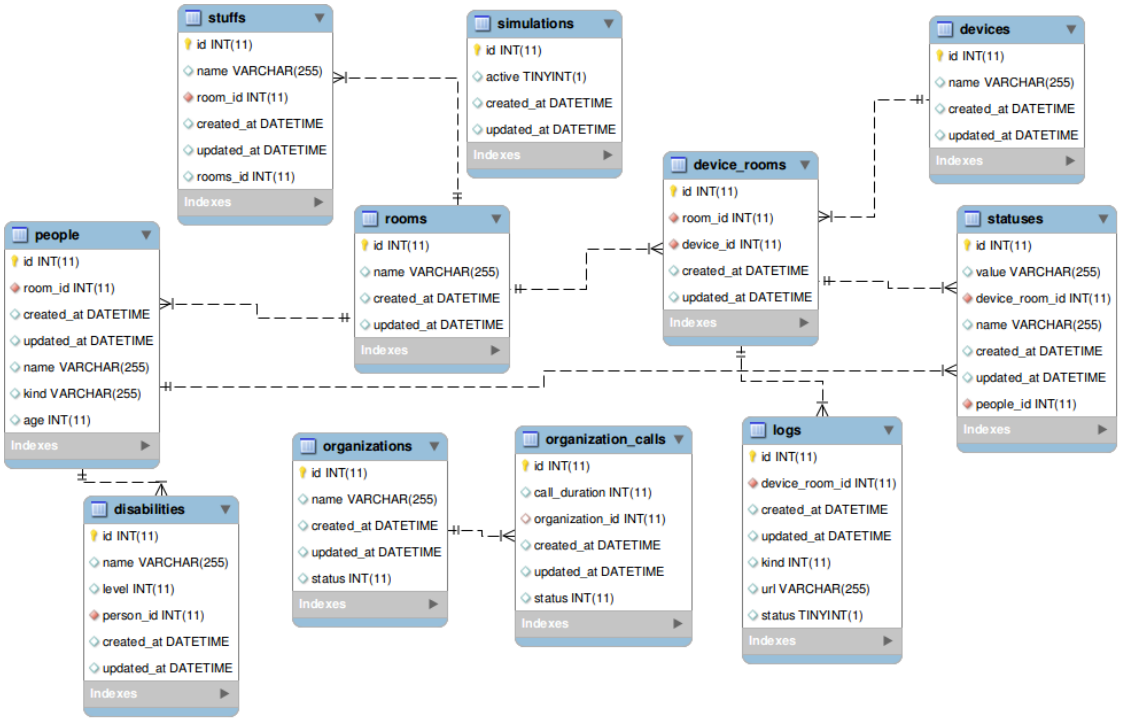


Figura 3.3: Modelagem ER do Banco de Dados.

As entidades são descritas a seguir.

3.3.1 Entidade Simulations

Esta entidade basicamente serve para marcar o início e fim das simulações. Quando uma simulação está em andamento, o atributo *active* daquela tupla é colocado como *true*. Nunca pode existir mais de uma simulação com o valor *active* igual a *true*, pois isso seria um erro de lógica, já que só pode existir uma simulação ativa a cada momento.

Quando uma simulação é iniciada, uma nova linha na tabela Simulations é criada, com o seu atributo *active* estabelecido com o valor *true*. Por padrão, todas as tabelas criadas através do framework Rails vêm com os atributos created_at e updated_at. Nesse caso, eles são úteis para saber a ordem das simulações.

3.3.2 Entidade Rooms

Apesar de possuir apenas um atributo específico, que é o atributo name, esta entidade é muito importante para o sistema. Ela serve para representar os cômodos de uma casa, e possui associação com as entidades People e Device_rooms.

3.3.3 Entidade Devices

Esta entidade representa um tipo de dispositivo que poderá ser instanciado em algum cômodo da casa, através da associação device_rooms, que será explicada na próxima subseção.

O atributo name é responsável por identificar os tipos de dispositivo, e deve ser único. Atualmente, existem 31 dispositivos cadastrados no sistema, como, por exemplo, micro-ondas, televisão, computador, etc. Além disso, atualmente essa lista é estática e

não pode ser editada. Em trabalhos futuros, espera-se torna-la dinâmica através de um editor.

3.3.4 Entidade Device_rooms

Esta entidade serve para implementar uma associação do tipo muitos para muitos entre a entidade Devices e a entidade Rooms. Essa associação é feita através dos identificadores únicos dessas duas tabelas.

Ela é uma entidade muito importante para o sistema, pois representa um dispositivo que é instanciado em algum quarto da casa. Possui ainda uma relação com a entidade *statuses*, que será explicada na subseção 3.3.7.

3.3.5 Entidade Stuffs

Esta entidade serve para representar objetos que existam na casa, mas diferentemente dos dispositivos, não possuem nenhuma operação que possa ser feita sobre eles.

Possui dois atributos: name e room_id. Name serve para representar o nome de um objeto, e room_id para indicar em que cômodo da casa um objeto se encontra.

3.3.6 Entidade Logs

Entidade que serve para armazenar informações sobre operações que foram executadas no sistema.

Existem 4 atributos que são importantes: device_room_id, description, kind e status. Device_room_id faz associação desta tabela com device_rooms, e é utilizado quando alguma operação do Web service foi chamada, como ligar televisão, por exemplo. Description é a descrição da operação que foi chamada. Kind serve para indicar qual é o tipo de informação que está sendo armazenada, ou seja, se é uma operação sobre dispositivo, se uma nova simulação foi iniciada, etc. Status indica se uma operação teve sucesso ou não, quando essa operação atuar sobre dispositivos. No exemplo anterior de ligar uma televisão, se ela já estiver ligada, esse atributo indicará que houve erro.

3.3.7 Entidade Statuses

Esta tabela serve para representar os status tanto de uma entidade People quanto de um objeto da entidade device_rooms. Ela é crucial ao sistema e armazena diversas informações importantes.

É utilizando essa tabela que será determinado se uma determinada operação, chamada através de um Web service, poderá ser executada. Por exemplo, para se chamar uma operação de ligar uma televisão, é necessário que ela esteja desligada. É consultando essa tabela que poderá ser possível obter essa informação.

Para saber os status de uma pessoa, por exemplo, deve-se consultar os status da pessoa, e, através do nome pode-se escolher o status desejado. Por exemplo, se for necessário saber o estado de agitação da pessoa, deve-se procurar a linha da tabela statuses que possui person_id igual ao valor desejado, e name igual a agitationLevel. Com isso, encontra-se a linha correta na tabela e, através da coluna value, é possível saber qual é o estado de agitação da pessoa. Como alguns status podem assumir diversos valores, a coluna value é do tipo string, possibilitando assim que todos os status sejam representados por uma dupla name e value.

Além disso, por ter uma associação com a entidade Persons, também armazena as informações do status do paciente. Por exemplo, se ele está agitado, se está tendo um ataque súbito, etc.

3.3.8 Entidade People

Esta entidade representa as pessoas que estão em uma simulação. Possui os atributos name e kind. Name é o nome da pessoa e deve ser único; kind representa o tipo da pessoa e pode assumir os valores de Patient, Caregiver, Visitor ou Doctor, indicando se a pessoa é um paciente, cuidador, visitante ou ainda médico. People possui ainda uma relação um para muitos com a entidade disabilites, que será explicada a seguir.

3.3.9 Entidade Disabilities

Esta entidade serve para associar uma deficiência a uma pessoa. O atributo name serve para indicar o tipo de deficiência, que pode ser visual, cognitiva, motora ou auditiva. O atributo level serve para indicar quão severa é essa deficiência, podendo variar de 1 a 3, sendo 3 o grau mais severo. Por fim, o atributo person_id associa uma deficiência a uma pessoa, já que uma pessoa pode possuir várias deficiências.

3.3.10 Entidade Organizations

Esta entidade serve para representar organizações. Existem dois atributos importantes: name e status. Name é o nome da organização, e deve ser único. Status serve para indicar se uma organização foi chamada.

3.3.11 Entidade Organization_calls

Esta entidade serve para representar quando uma organização é chamada, e funciona como se fosse uma unidade daquela organização. O atributo call_duration serve para indicar quanto tempo a unidade vai demorar até chegar ao destino. O atributo status indica se ela já chegou ou ainda está a caminho, e o atributo organization_id associa a unidade à uma determinada organização.

3.4 Estrutura de Classes

O sistema foi estruturado de acordo com o padrão MVC proposto pelo Ruby on Rails, conforme especificado anteriormente. As seções seguintes apresentam as classes e páginas criadas para esta aplicação e como elas estão relacionadas com os componentes de Modelo, Visualização e Controle descritos anteriormente.

3.4.1 Modelo da Aplicação

A Figura 3.3 mostra a estrutura de classes do sistema que representa o componente Modelo da aplicação.

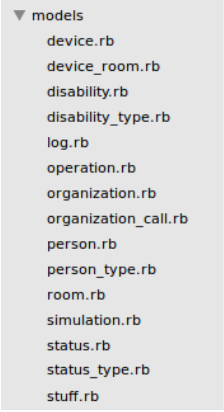


Figura 3.4: Componente Modelo da aplicação

Cada modelo corresponde a uma classe que representa uma tabela no banco de dados. Essas classes possuem o mesmo nome que o da tabela original, mas no singular. Todas as classes que representam uma tabela no banco de dados herdam do componente ActiveRecord do Rails, fazendo todo o trabalho de sincronização com o banco.

Pode-se perceber que, além das onze tabelas utilizadas no banco de dados, existem três classes especiais, que estão nos arquivos person_type.rb, disability_type.rb e status_type.rb. Essas três classes estão no modelo, mas não herdam do componente ActiveRecord, e portanto não representam uma tabela do Banco de Dados. Elas servem para definir tipos complexos de pessoa, deficiência e status que são utilizados na definição dos arquivos WSDL.

Tipos complexos no WSDL são utilizados quando precisamos de mais informações que apenas um tipo simples pode representar. Quando precisamos responder uma requisição e informar todos os dados de uma pessoa, precisamos informar o nome da pessoa, o cômodo da casa em que ela está, suas deficiências e os seus status. Isso não é possível utilizando apenas os tipos simples inerentes ao WSDL. Para isso, precisamos dos tipos complexos de pessoa, status e deficiências, que estão definidos nos arquivos que foram citados no parágrafo anterior.

3.4.2 Visualização da Aplicação

A Figura 3.4 mostra a estrutura de classes que representa o componente de visualização do padrão MVC.

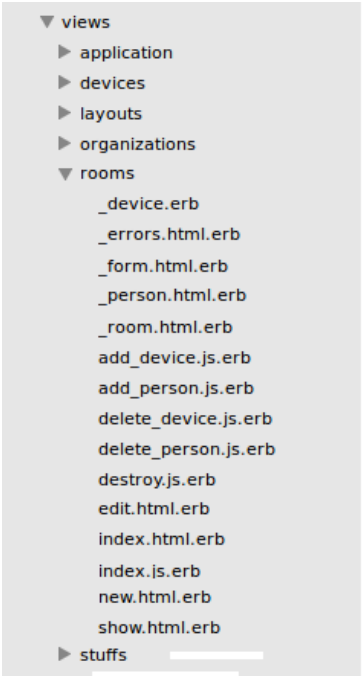


Figura 3.5: Componente Visualização da aplicação

Existem diversas pastas, nas quais os arquivos de visualização estão armazenados. Por padrão, o Rails cria uma pasta para cara arquivo controlador que é gerado, para armazenar seus arquivos de visualização nelas.

Como pode ser notado, existem basicamente dois tipos de arquivo utilizados no sistema, que são os html.erb e os js.erb. Arquivos html.erb facilitam a inserção de código Ruby dentro do HTML. Cada uma dessas páginas representa uma tela diferente, que pode ser renderizada por um controlador. Pode-se notar que alguns arquivos começam o seu nome com o caractere underscore. Esses são arquivos que utilizam o conceito de Partials do rails, isto é, são arquivos de visualização que, para simplificação do código, podem ser utilizados dentro de outro arquivo de visualização, como é mostrado na Figura 3.6.

```
<% @rooms.each do |room| %>
  <%= render :partial => 'room', :locals => {room: room} %>
<% end %>
```

Figura 3.6: Exemplo de utilização de um Partial

Os arquivos js.erb são arquivos que permitem que código ruby seja utilizado dentro de um arquivo Javascript. No trabalho, eles são utilizados principalmente quando requisições Ajax são feitas ao controlador, que responde então com algum código Javascript para ser executado através dos arquivos js.erb.

3.4.3 Controlador da Aplicação

A Figura 3.7 mostra a estrutura de classes que representa o componente controlador do padrão MVC.

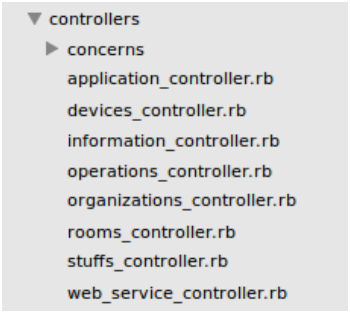


Figura 3.7: Componente Controlador da aplicação

As classes contidas na pasta chamada controllers representam o componente controlador do padrão MVC. Cada uma dessas classes possui diversas ações, que são acessadas, por exemplo, quando o usuário pressiona algum botão ou quando ele acessa um determinado endereço.

Existem dois controladores, information_controller e web_service controller, que são responsáveis pela parte de *Web services* do sistema. É neles que estão definidas as operações que podem ser acessadas via SOAP.

Aqui, foi utilizada a gem Wash_out, que é responsável tanto por gerar os arquivos WSDL como por tratar as requisições feitas via *web service*. Ela possibilita que as requisições *web service* sejam tratadas da mesma maneira que as requisições feitas por um navegador são tratadas. Por exemplo, segue o código do *web service* responsável por devolver a localização de uma pessoa.

```
soap_action "get_localization_from_person", :args => {"person_name" => :string}, :return => :string
def get_localization_from_person
  render_action and return if @error
  if params["person_name"]
    @result = Person.find_by_name(params["person_name"]).room.name
  else
    @error = "person_id was not passed"
  end
  render_action
end
```

Figura 3.8: Código que implementa um método disponibilizado via web service

Podemos ver na Figura 3.8 a implementação do método e também a definição da soap action, que, através da gem Wash_out, faz com que as definições desse método sejam escritas no arquivo WSDL. Temos acesso normal ao modelo da aplicação, e através dele podemos consultar informações no banco de dados.

3.5 Interface de navegação

Esta seção apresenta uma análise das telas do sistema, explicando o funcionamento e navegação das mesmas.

3.5.1 Tela de configuração da simulação

Esta é a primeira tela vista pelo usuário ao acessar a URL do sistema. É a principal tela do sistema, onde o usuário pode especificar diversos dados em relação à simulação. Através do menu superior, ele pode escolher entre adicionar um cômodo, adicionar um objeto, adicionar uma organização ou adicionar uma pessoa.

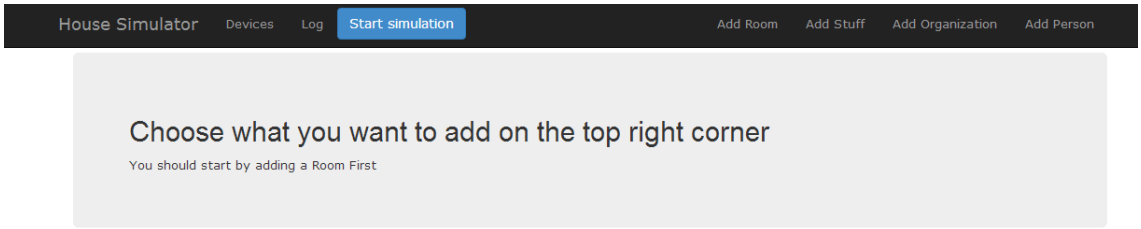


Figura 3.9: Tela inicial do sistema

Conforme é explicado na tela, o usuário deve escolher algo para adicionar a simulação. Inicialmente deve-se adicionar um cômodo, já que só será possível adicionar dispositivos e objetos depois de existirem cômodos na casa. Após termos adicionados pelo menos um cômodo, é possível executar todas as outras operações de adição.

3.5.1.1 Adicionar cômodo

Ao clicar em Add Room, o usuário irá enxergar o formulário para adicionar um cômodo. O usuário deve escolher um nome e, então, ao clicar em Add Room, o cômodo aparecerá imediatamente na tela, como é mostrado na Figura 3.10.

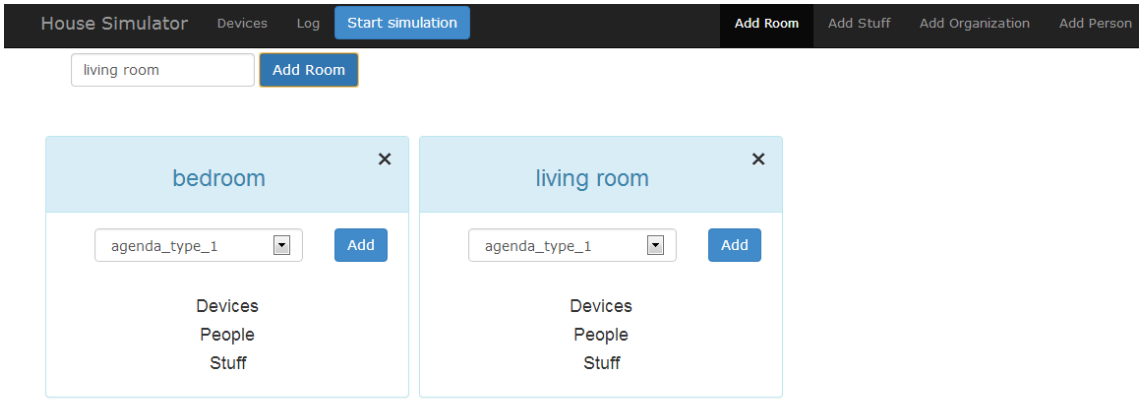


Figura 3.10: Adicionando um cômodo à simulação

O dispositivo pode ser removido pelo usuário clicando no X no canto superior direito de cada cômodo. Uma confirmação da ação será necessária, como é mostrado na Figura 3.11.

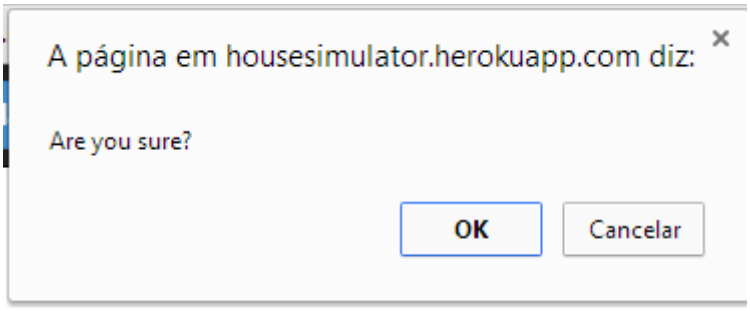


Figura 3.11: Confirmação é necessária para remover cômodo

3.5.1.2 Adicionar dispositivo a um cômodo

Como pode ser visto na Figura 3.12, o usuário pode escolher um dos dispositivos e então clicar em Add, e esse será imediatamente adicionado àquele cômodo.

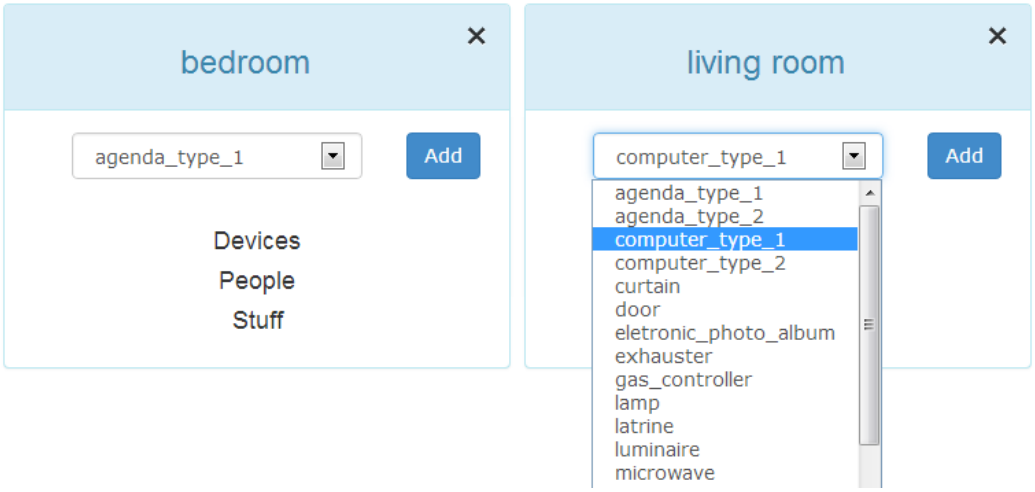


Figura 3.12: Selecionar tipo de dispositivo para adicionar

Uma vez adicionado, o dispositivo vai para a lista de dispositivos do respectivo cômodo, conforme pode ser visualizado na figura seguinte.

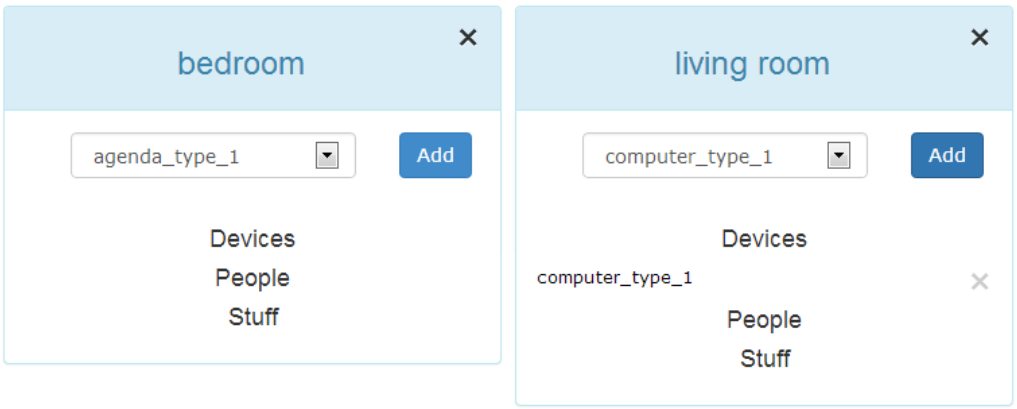


Figura 3.13: Dispositivo computer_type_1 foi adicionado a living room

Após ter adicionado um dispositivo, o usuário pode facilmente removê-lo clicando no X ao lado do seu nome. Uma confirmação é solicitada, como no caso anterior.

3.5.1.3 Adicionar objeto a um cômodo

Usando novamente o menu localizando no canto superior direito da tela, é possível clicar em Add Stuff. Um novo formulário vai aparecer, no qual o usuário pode dar um nome ao objeto e selecionar em qual cômodo esse objeto estará localizado. A seguir, basta clicar em Add Stuff e o objeto será imediatamente adicionado ao cômodo selecionado, conforme pode ser visualizado na figura seguinte. Da mesma maneira que ocorreu para dispositivos, também é possível remover objetos clicando-se no X ao lado do seu nome.

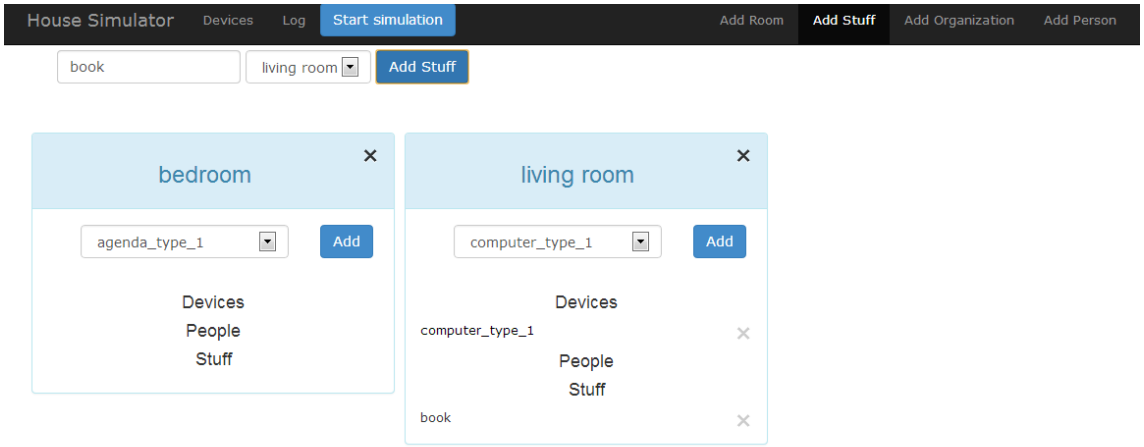


Figura 3.14: Tela após adicionar-se um objeto book ao cômodo living room

3.5.1.4 Adicionar organização

Clicando em Add Organization no menu superior direito, o usuário poderá adicionar uma organização à simulação. Para isso, ele deverá preencher o formulário com um nome para a organização, e clicar em Add Organization ao lado do campo Name. Com isso, a nova organização irá aparecer na lista de organizações.

Figura 3.15: Adicionar Organização

Novamente, para remover uma organização, basta clicar no X e confirmar a intenção de removê-la.

3.5.1.5 Adicionar pessoa

Selecionando Add Person no menu superior, o formulário de adição de pessoa será mostrado. Pará adicionar uma pessoa, basta preencher o campo name, selecionar o quarto, selecionar o tipo e então clicar em Add Person. Se for do desejo do usuário, uma ou mais deficiências podem ser adicionadas para a pessoa antes de adicioná-la. Também é possível remover deficiências adicionadas previamente.

Cabe salientar que atualmente a lista de possibilidade de deficiências é fixa. Em trabalhos futuros pretende-se adicionar um cadastro de deficiências, possibilitando uma gama maior de simulações e opções.

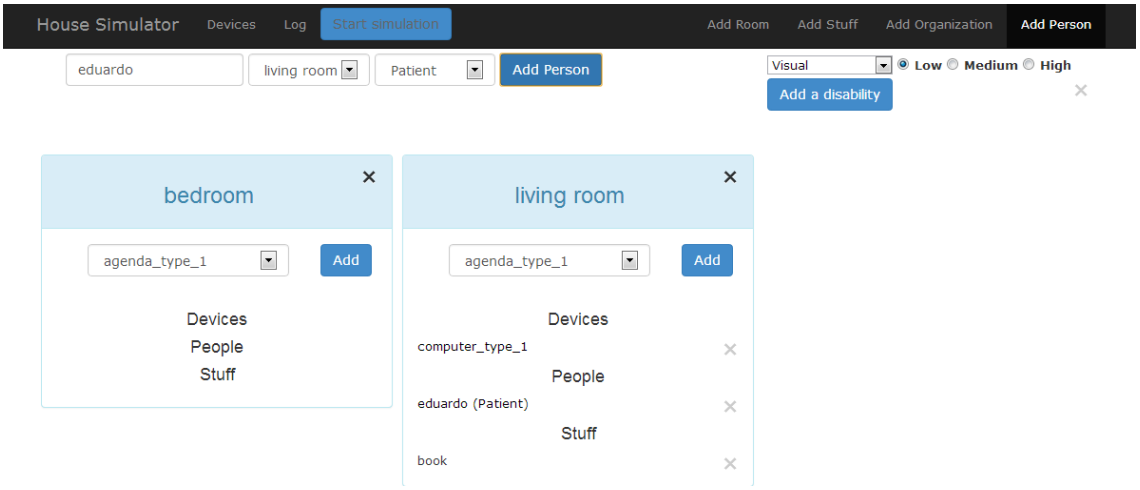


Figura 3.16: Adicionar pessoa a um cômodo

3.5.2 Lista de Dispositivos

Ao clicar em Devices no menu superior o usuário chegará à tela da lista de dispositivos. Basicamente, essa tela lista todos os tipos de dispositivo que estão cadastrados no sistema.

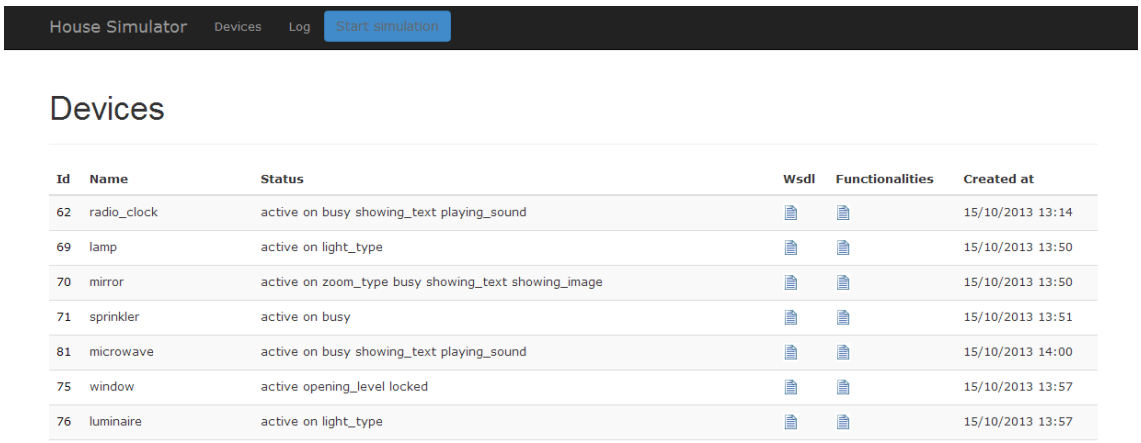


Figura 3.17: Lista de dispositivos

Através dessa lista o usuário pode acessar tanto o arquivo WSDL de cada dispositivo como uma lista de funcionalidades que aquele dispositivo possui, que será descrita na subseção seguinte.

Os WSDL são gerados automaticamente através da gem wash_out. Algumas modificações precisaram ser feitas para que fosse possível gerar um WSDL para cada dispositivo e também para cada funcionalidade, já que por padrão, a gem inclui no arquivo WSDL todos os métodos existentes no controlador. Através de um arquivo de configuração existente no sistema, são definidas quais as funcionalidades de casa dispositivo. Com isso, podemos filtrar apenas as funcionalidades que desejamos para um determinado dispositivo, e assim conseguimos gerar a sua WSDL. Para o caso de desejarmos o WSDL de uma funcionalidade específica, é mais simples, pois basta mostrar apenas o método que já está definido no controlador.

3.5.3 Lista de funcionalidades

A lista de funcionalidades mostra todas as funcionalidades de um determinado tipo de dispositivo, junto com as precondições e efeitos de cada uma dessas funcionalidades. Para exibição dessa lista foram utilizados dois arquivos, um que contem a informação das precondições de cada funcionalidade, e outro que mostra o resultado da execução dessa funcionalidade. Entretanto, estes arquivos não são utilizados atualmente na execução da funcionalidade em si, e sim apenas para mostrar essas informações ao usuário. As funcionalidades estão definidas no modelo da aplicação, para que possam ser acessadas pelo controlador quando uma requisição *web service* for feita.

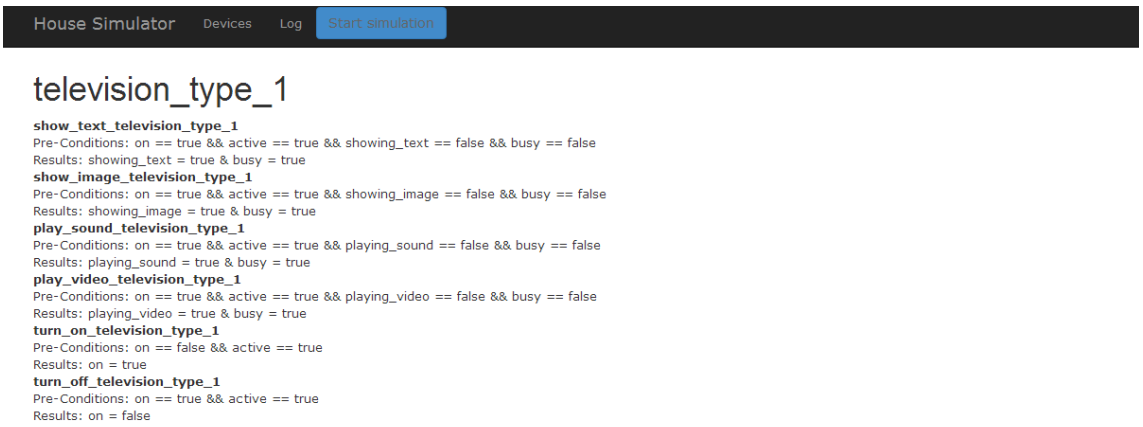


Figura 3.18: Lista de funcionalidades

É possível notar, na Figura 3.18, que temos a lista de funcionalidades do dispositivo television_type_1. Para uma dessas funcionalidades, por exemplo, a funcionalidade turn_on_television_type_1, podemos notar que ela só poderá ser executada caso o status on do dispositivo tenha valor false e o status active tenha valor true. Se essas duas precondições forem atendidas, então o status on da televisão passará a ter valor true.

3.5.4 Log da aplicação

Através do menu superior é possível acessar a tela de logs da aplicação. Ela mostra operações que ocorreram sobre o sistema organizadas por data.

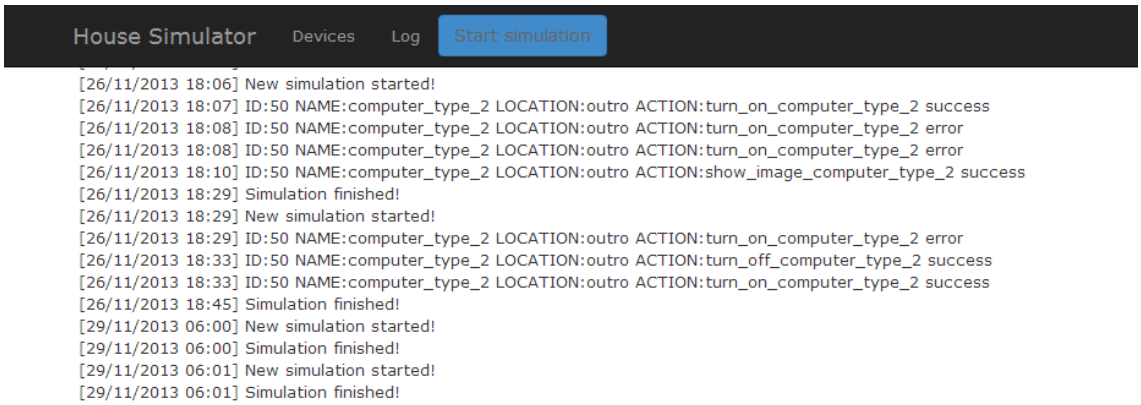


Figura 3.19: Log da aplicação

Como é visto na Figura 3.19, o log da aplicação mostra o início e fim das simulações. Ele também mostra chamadas web service sobre os dispositivos. Por

exemplo, é possível ver na segunda linha uma chamada do web service turn_on_computer_type_2 e a indicação de sucesso.

3.5.5 Tela de edição de dispositivo

A partir da tela principal é possível clicar em um dispositivo que está associado a um cômodo e editá-lo. Os atributos que podem ser modificados são o cômodo em que ele se localiza e os seus status. Através do identificador único, podemos saber utilizando o componente modelo da aplicação qual é o quarto selecionado, e também os status do dispositivo.

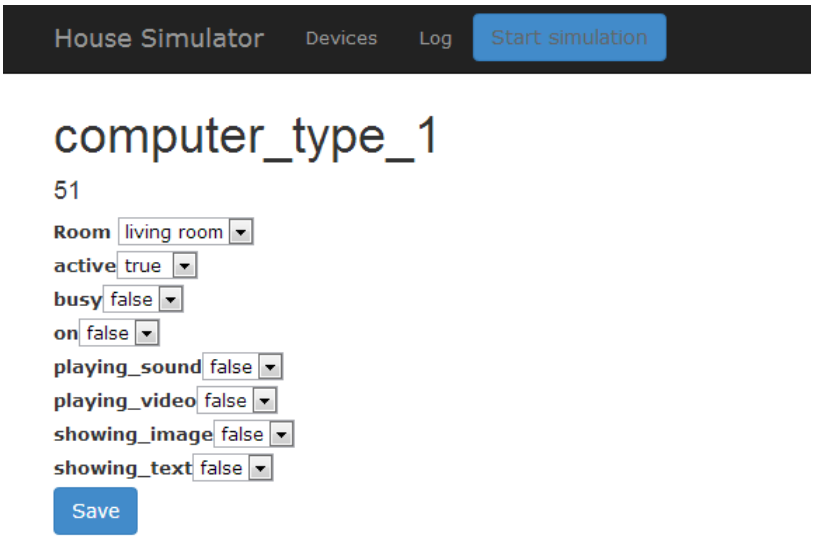


Figura 3.20: Edição de dispositivo

3.5.6 Tela de edição de pessoas

A tela de adição de pessoas pode ser acessada através da tela inicial, bastando para isso clicar em alguma pessoa que está em um dos cômodos da casa. Através dela é possível mudar o nome da pessoa, adicionar, remover ou editar alguma deficiência que já exista e mudar algum dos status daquela pessoa.

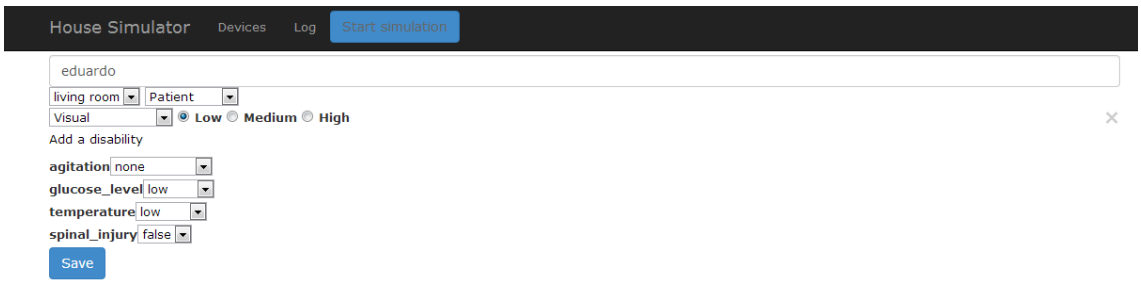


Figura 3.21: Edição de pessoa

3.5.7 Tela de Feedback

A tela de feedback fica acessível após o usuário clicar em Start Simulation, e iniciar a simulação.

É importante ressaltar que os Web services e arquivos WSDL estão sempre disponíveis. Entretanto, se tentarmos acessar qualquer Web service antes de uma

simulação ser iniciada, um erro será gerado dizendo que a simulação deve ser iniciada primeiro.

Esta tela mostra os status dos dispositivos, das organizações e do paciente, além de mostrar o log da simulação atual.

Figura 3.22: Tela de Feedback

Acima está a tela inicial, quando o usuário a acessa. Abaixo apresenta-se a mesma tela após uma operação sobre o computer_type_1 localizado na living room ter sido chamada.

Figura 3.23: Mudança de status na tela de feedback

Como pode ser notado, quando o status de um dispositivo muda, a sua cor muda por 30 segundos para indicar que uma operação foi chamada via Web service sobre aquele dispositivo. Abaixo segue a tela após uma organização ter sido chamada através de um Web service.

Figura 3.24: Organização polícia foi chamada

No próximo capítulo será apresentado o conceito de processo de negócio. Além disso, duas simulações diferentes serão mostradas, correspondentes a dois processos de negócio propostos por Kambara-Silva et al. (2014).

4 PROCESSOS DE NEGÓCIO SIMULADOS

No capítulo anterior descreveu-se como o simulador foi desenvolvido. Foram apresentadas *user stories* iniciais que descrevem o sistema. Também foram descritas as tecnologias utilizadas no sistema, bem como a sua arquitetura. Por fim, foi apresentado um guia mostrando as telas do sistema.

Neste capítulo serão demonstrados dois exemplos de utilização do sistema. Primeiramente será definido o conceito de processo de negócio, visto que são eles que determinam como o ambiente inteligente é controlado (em especial quando uma situação não desejada é detectada). Depois, serão mostrados dois processos de negócios e as suas simulações utilizando o sistema.

4.1 Processo de Negócio

Segundo Weske (2012), processos de negócio consistem em um conjunto de atividades que são executadas e coordenadas em um ambiente técnico e organizacional, que juntos tentam realizar um objetivo de negócio.

Um modelo de processo de negócio consiste de uma série de modelos de atividade e restrições de execução entre eles, segundo Weske (2012). Nesse contexto, BPMN (*Business Process Model and Notation*) é a maneira mais comum de se representar os processos de negócio.

A figura seguinte mostra os elementos BPMN usados neste trabalho.

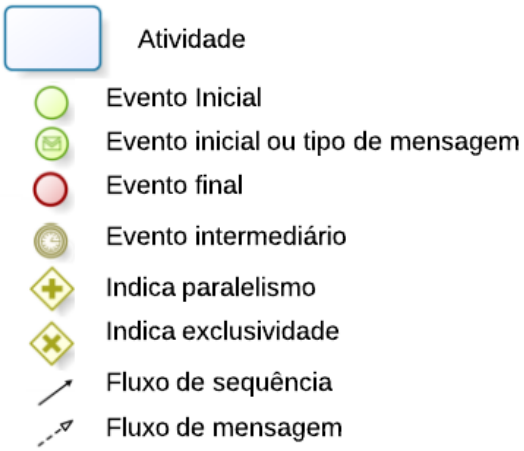


Figura 4.1: Elementos BPM utilizados no trabalho

4.2 Simulação do Processo de Negócio de Agitação

Conforme descrito por Kambara-Silva et al. (2014), os processos de negócio são utilizados para descrever como um sistema de gerência de ambientes inteligentes deve se comportar quando detecta uma situação indesejada no ambiente. Para tanto, o ambiente é provido de sensores e atuadores que disponibilizam funcionalidades (operações) através de Web services. Além disso, existe um controlador que gerencia o ambiente, analisando os dados coletados dos sensores e detectando quando uma situação indesejada ocorre. Finalmente, quando o controlador detecta uma situação indesejada ele chama uma aplicação que executa o plano de negócio em questão.

O modelo de processo de negócio apresentado a seguir basicamente descreve o comportamento esperado quando o ambiente detecta um paciente em estado de agitação. A figura seguinte descreve processo de negócio de agitação, o qual foi proposto por Kambara-Silva et al. (2014), sendo uma adaptação do modelo proposto por Gassen et al. (2012).

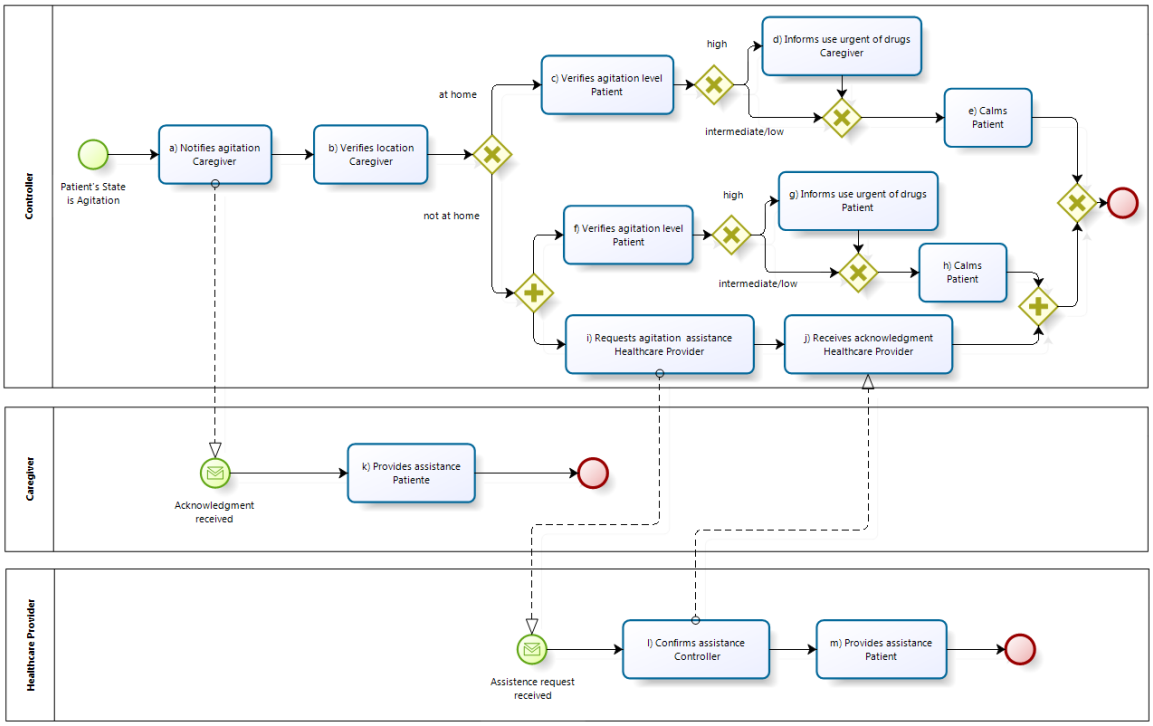


Figura 4.2: Processo de negócio para paciente agitado.

Fonte: Kambara-Silva et al. (2014).

Como pode ser percebido na figura, esse processo tenta acalmar o paciente se ele está agitado. É importante salientar que o tipo de medida tomada muda caso o cuidador do paciente esteja na casa ou não, pois, se o cuidador não estiver em casa, as indicações são dadas diretamente ao paciente. O processo também varia conforme o nível de agitação do paciente, pois, se o nível for alto, é necessário indicar que o paciente deve tomar algum medicamento.

Para a parte do simulador, só o controlador é importante, pois é o controlador que gerencia os dispositivos dentro da casa.

Primeiramente deve-se configurar a simulação. A simulação foi configurada com dois cômodos: cozinha e quarto. O paciente está na cozinha, e, além dele, há um

cuidador que não se encontra na residência. Cada um dos cômodos possui dois dispositivos, como pode ser observado na Figura 4.3.

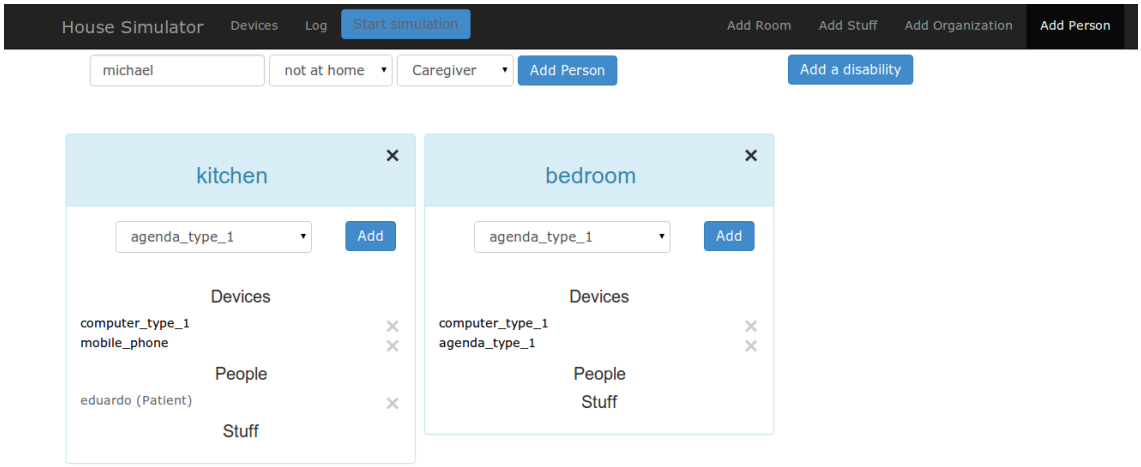


Figura 4.3: Configuração inicial da aplicação

Após essa configuração inicial, clica-se em "*Start simulation*", e a simulação é iniciada. Com isso, a tela de feedback da aplicação é mostrada na tela, como é mostrado na figura seguinte.

Figura 4.4: Tela de feedback inicial

Agora é possível utilizar todos os Web services providos pelo simulador, os quais correspondem às operações disponibilizadas pelos dispositivos e sensores adicionados à simulação.

Após configurar o ambiente e inicializar a simulação, é necessário desenvolver uma aplicação que simule o processo de negócio em execução. Essa aplicação vai atuar como cliente dos Web services disponibilizados, chamando suas operações. Dependendo da chamada realizada, os status dos dispositivos e sensores podem mudar. Isso é refletido na interface do simulador. Além das operações dos dispositivos, o simulador também oferece uma série de Web services que retornam informações relevantes sobre a simulação em si.

Os Web services que são oferecidos para coletar informações sobre a simulação são:

- Descobrir localização de uma pessoa, que recebe como argumento o nome da pessoa e retorna a sua localização;
- Descobrir localização de um dispositivo, que recebe como argumento o identificador do dispositivo e retorna a sua localização;
- Descobrir os dispositivos ativos em um cômodo, que recebe a localização como argumento e retorna uma lista de identificadores de dispositivo;
- Descobrir os objetos em um cômodo, que recebe a localização como parâmetro e retorna uma lista de objetos;
- Descobrir os dispositivos ativos por uma funcionalidade, que recebe a funcionalidade como argumento e retorna uma lista de identificadores de dispositivo;
- Descobrir as pessoas, que retorna todas as pessoas ta simulação;
- Descobrir os cômodos, que retorna todos os cômodos da simulação.

Do ponto de vista da aplicação cliente, inicialmente identificam-se as pessoas da casa para depois checar o estado de agitação do paciente. Isso pode ser feito com uma requisição ao simulador para que ele retorne quais são as pessoas disponíveis na simulação, seus tipos e deficiências. De posse dessa lista é possível verificar o tipo de cada pessoa, como é evidenciado na figura seguinte.

```
41
42  client = Savon.client(wsdl: 'http://localhost:3000/information/wsdl')
43
44  patient = nil
45  caregiver = nil
46  r = client.call(:get_people)
47  people = r.body[:get_people_response][:value]
48  people.each do |person|
49      if person[:kind] == "Patient"
50          patient = person
51      elsif person[:kind] == "Caregiver"
52          caregiver = person
53      end
54  end
```

Figura 4.5: Requisição para listar pessoas da simulação

Pode-se perceber uma chamada à operação get_people, que retorna uma lista de todas as pessoas na simulação. De posse do paciente, seu estado de agitação pode ser consultado. Como os dados da pessoa já foram recuperados, não é preciso fazer uma nova requisição ao serviço.

```
def get_status_level(status_name, patient)
    answer = nil
    patient[:status].each do |status|
        answer = status[:value] if status[:name] == status_name
    end
    answer
end
agitation_level = get_status_level("agitation", patient)
```

Figura 4.6: Descobrir estado de agitação do paciente

Para a parte principal da aplicação, como pode ser visto na Figura 4.4, o estado de agitação do paciente é alto. Com isso, o primeiro passo a ser tomado, conforme descrito no modelo do processo de negócio, é notificar o cuidador. As seguintes telas mostram a requisição e a tela de feedback da aplicação após a requisição ter sido realizada.

```
if agitation_level != "none"
    puts "Patient State is agitation"
    r = client.call(:notify_caregiver, message: {"caregiver_name" => caregiver[:name], "text" => "Patient state is agitation!"})
    if r.body[:notify_caregiver_response][:value] == "ok"
        puts "Caregiver was notified"
    end
```

Figura 4.7: Requisição para notificar o cuidador

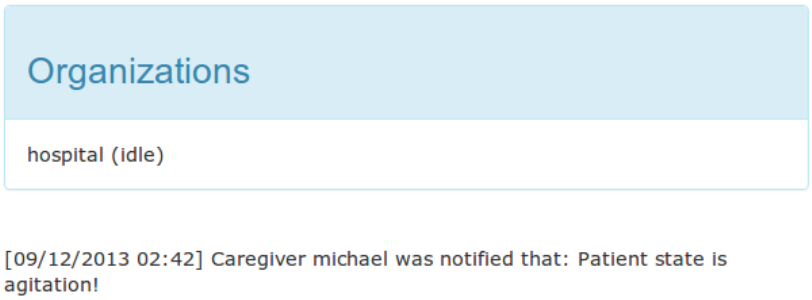


Figura 4.8: Tela de feedback após a notificação do cuidador

Agora, conforme descrito, tem-se duas situações diferentes, caso o cuidador esteja ou não na residência. Como ele não está, deve-se então chamar a organização responsável, e, além disso, notificar o paciente sobre uso de drogas, caso ele esteja com estado de agitação alto, e esse é o caso simulado. O processo de negócio indica que isso deve ser feito paralelamente, mas a aplicação foi feita sequencialmente, por motivos de simplificação. Abaixo seguem imagens contendo o código das requisições feitas.

```
def inform_use_of_drugs(client,person)
    r = client.call(:get_active_devices_from_localization, message: {"room_name" => person[:location]})
    r2 = client.call(:get_active_devices_by_functionality, message: {"functionality" => "show_text"})
    useful_devices = r.body[:get_active_devices_from_localization_response][:value] & r2.body[:get_active_devices_by_functionality_response][:value]
    device_name = client.call(:get_device_name, message: {"device_id" => useful_devices.first})
    device_name = device_name.body[:get_device_name_response][:value]
    operation = "show_text_" + device_name
    operation = operation.to_sym
    new_client = Savon.client(wsdl: 'http://localhost:3000/web_service/wsdl?'+device_name)
    st_response = new_client.call(operation, :message => {"device_id" => useful_devices.first, "text" => "Urgent use of drugs!"})
    st_response.body["#{operation}_response".to_sym][:value] == "ok"
end

if caregiver[:location] == "not at home"
    puts "Caregiver is not at home"
    if agitation_level == "high"
        puts "Use of drugs informed to Patient" if inform_use_of_drugs(client, patient)
    end
    client.call(:call_for_organization, :message => {"organization_name" => "hospital"})
    puts "Hospital was called"
```

Figura 4.9: Requisição para informar paciente sobre uso de drogas e chamar organizacão

Como pode ser notado, o método utilizado para notificar o paciente foi o método de mostrar texto. Para isso foi feita uma pesquisa por todos os dispositivos que estão ativos e então foi feita a requisição. Também é possível notar que o hospital foi chamado. Abaixo segue a figura que mostra a tela de feedback após essa requisição.

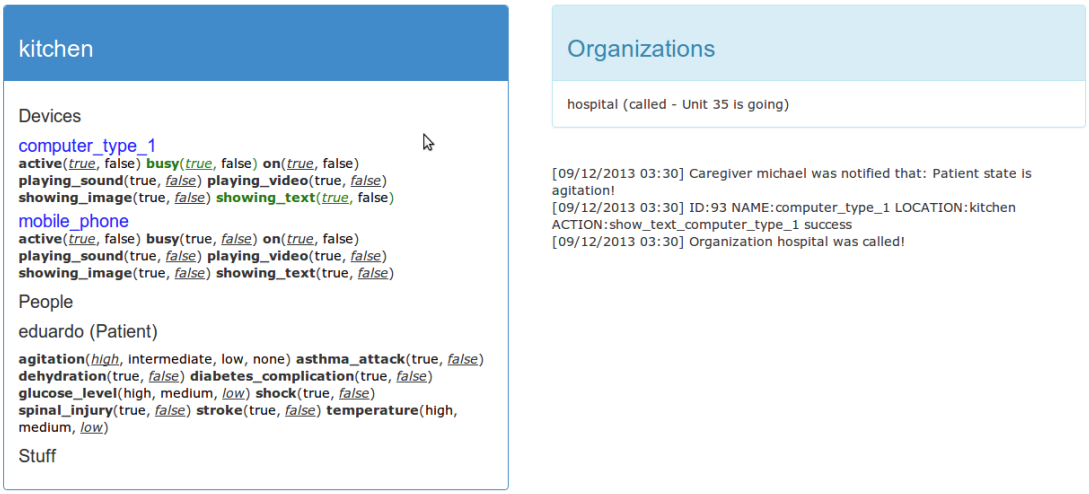


Figura 4.10: Tela de feedback após informar paciente e chamar organizacão

Como pode ser visto, o dispositivo está mostrando o texto e uma unidade da organizacão está a caminho da residência. Conforme o processo de negócio, o passo seguinte consiste em acalmar o paciente. A maneira escolhida para isso foi tocar um som. A requisição e a tela de feedback podem ser vistas nas figuras seguintes.

```
def calm_patient(client,person)
    r = client.call(:get_active_devices_from_localization, message: {"room_name" => person[:location]})
    r2 = client.call(:get_active_devices_by_functionality, message: {"functionality" => "play_sound"})
    useful_devices = r.body[:get_active_devices_from_localization_response][:value] & r2.body[:get_active_devices_by_functionality_response][:value]
    device_name = client.call(:get_device_name, message: {"device_id" => useful_devices.last})
    device_name = device_name.body[:get_device_name_response][:value]
    operation = "play_sound_" + device_name
    new_client = Savon.client(wsdl: 'http://localhost:3000/web_service/wsdl?'+device_name)
    st_response = new_client.call(operation.to_sym, :message => {"device_id" => useful_devices.last, "sound" => "Base64.encode64(File.binread('/home/e
    st_response.body["#{operation}_response".to_sym][:value] == "ok"
end
puts "Patient is gettting calm" if calm_patient(client, patient)
```

Figura 4.11: Requisição para acalmar o paciente

Figura 4.12: Tela de feedback após acalmar o paciente

Com isso, a simulação do processo de negócio está pronta. É possível ver que o computador foi utilizado para mostrar o texto, notificando paciente sobre a necessidade de uso de drogas (medicamentos), enquanto o telefone celular foi utilizado para tocar um som com o fim de acalmar o paciente.

A mesma simulação com uma configuração inicial diferente poderia ser facilmente implementada. A seguir, a título de exemplo, a mesma simulação é realizada, mas com

a diferença de que o cuidador se localiza agora na residência. As figuras seguintes mostram a tela de feedback para a simulação feita dessa maneira.

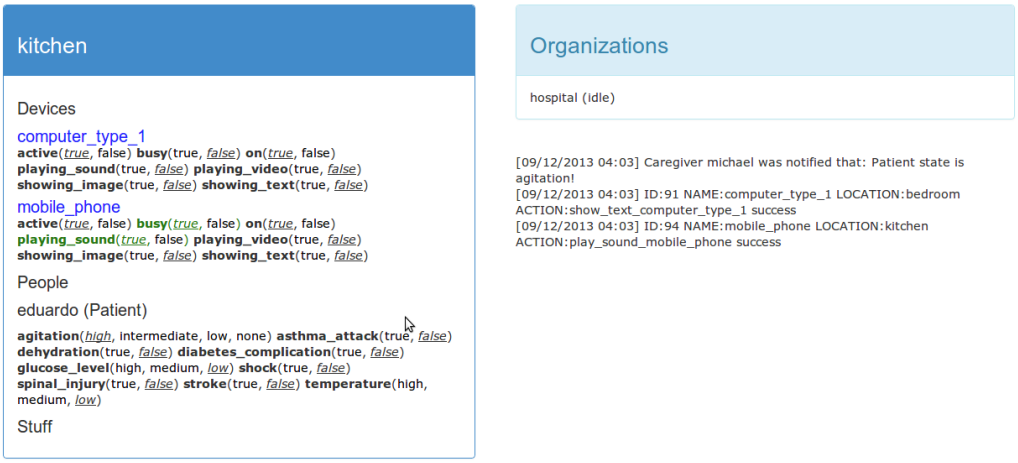


Figura 4.13: Tela de feedback caso o cuidador não esteja em casa

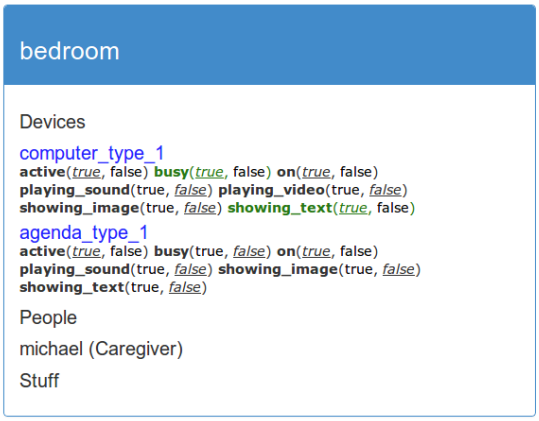


Figura 4.14: Continuação da tela de feedback

Diferentemente do caso anterior, quem deve ser notificado agora é o cuidador, e por isso o dispositivo utilizado é o computador que se encontra no outro cômodo. Além disso, nenhuma organizacão precisou ser chamada, já que o cuidador estava na residência.

4.3 Simulação do Processo de Negócio de Esquecer o Fogão Ligado

Este é um processo um pouco mais simples e foi desenvolvido para o caso em que um fogão é esquecido ligado (com um bocal aceso). O modelo do processo de negócio, que foi proposto por Kambara et al. (2014), é ilustrado na figura seguinte.

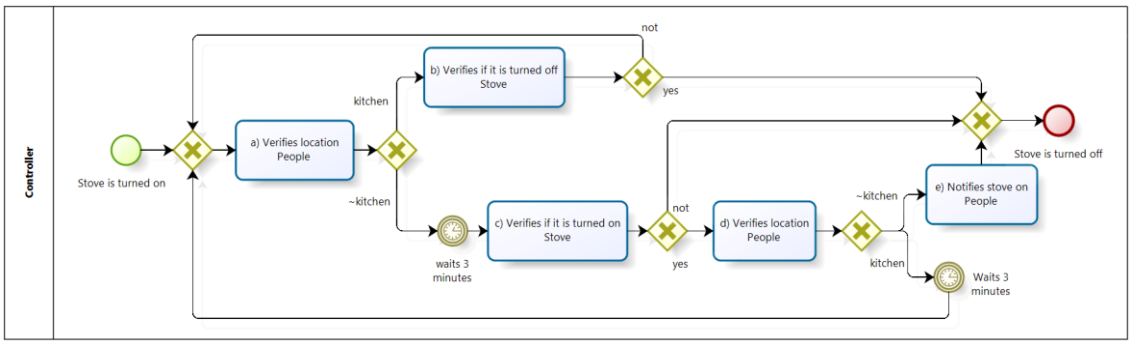


Figura 4.15: Processo de negócio de esquecer o fogão ligado.

Fonte: Kambara-Silva et al. (2014)

Basicamente, o processo de negócio verifica se o fogão está ligado. Caso ele esteja ligado há mais de três minutos, e ninguém mais esteja na cozinha, então alguma pessoa da casa é notificada sobre o fogão ligado.

Para esta simulação foi utilizada uma configuração bem simples, com apenas um paciente na casa e dois cômodos. Alguns dispositivos também são necessários, sendo um deles o fogão, que está localizado na cozinha. Isso é demostrado na figura seguinte.

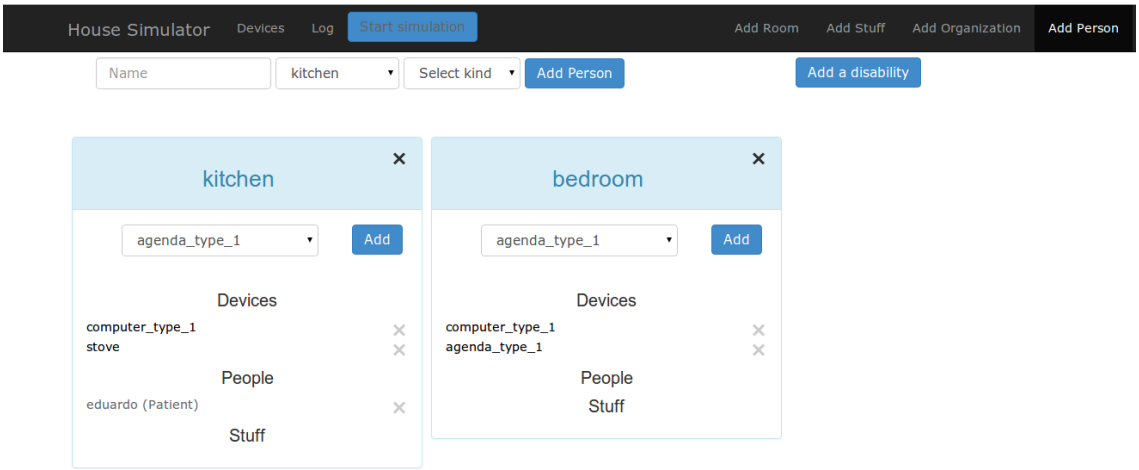


Figura 4.16: Configuração inicial da simulação

Com o simulador configurado e inicializado, pode-se utilizar uma aplicação cliente para monitorar o estado do fogão a fim de saber se ele está ou não ligado. Para tanto, uma aplicação em Ruby foi desenvolvida e é apresentada na Figura seguinte. Nessa aplicação, enquanto existir uma pessoa na cozinha, o monitoramento continua sendo feito e só é interrompido quando a pessoa for notificada ou se o fogão for desligado. Os tempos foram diminuídos em relação ao modelo descrito na Figura 4.15 para facilitar a simulação. A seguir segue a figura que mostra o código da aplicação cliente que implementa o BPMN descrito na Figura 4.15.

```ruby
client = Savon.client(wsdl: 'http://localhost:3000/information/wsdl')


def notify_person(client)
    r = client.call(:get_active_devices_from_localization, message: {"room_name" => "bedroom"})
    r2 = client.call(:get_active_devices_by_functionality, message: {"functionality" => "show_text"})
    useful_devices = r.body[:get_active_devices_from_localization_response][:value] & r2.body[:get_active_devices_by_functionality_response][:value
    device_name = client.call(:get_device_name, message: {"device_id" => useful_devices.first})
    device_name = device_name.body[:get_device_name_response][:value]
    operation = "show_text_" + device_name
    operation = operation.to_sym
    new_client = Savon.client(wsdl: 'http://localhost:3000/web_service/wsdl?'+device_name)
    st_response = new_client.call(operation, :message => {"device_id" => useful_devices.first, "text" => "Stove is turned on!"})
end

def is_stove_turned_on?(client)
    stove_status = client.call(:get_status_device, message: {"device_id" => 95, "status" => "on"})
    stove_status = stove_status.body[:get_status_device_response][:value]
    stove_status == "true" || stove_status == true
end

def is_person_in_kitchen?(client)
    r = client.call(:get_localization_from_person , message: {"person_name" => "eduardo"})
    localization = r.body[:get_localization_from_person_response][:value]
    localization == "kitchen"
end

while is_stove_turned_on?(client)
    if is_person_in_kitchen?(client)
        puts "Person is in kitchen"
    else
        puts "Person is not in kitchen"
        sleep 5
        if is_stove_turned_on?(client)
            if !is_person_in_kitchen?(client)
                notify_person(client)
                puts "Person was notified"
                break
            else
                sleep 5
            end
        end
    end
end
```

Figura 4.17: Aplicação cliente para o processo de negócio esquecer o fogão ligado

Como pôde ser visto na aplicação cliente, o *loop* só termina se o fogão for desligado ou quando o paciente for notificado, que foi o caso simulado. A figura seguinte mostra a saída da execução da aplicação cliente.

```
1 Person is in kitchen
2 Person is in kitchen
3 Person is in kitchen
4 Person is in kitchen
5 Person is in kitchen
6 Person is in kitchen
7 Person is not in kitchen
8 Person was notified
```

Figura 4.18: Tela de saída da aplicação cliente

Como o paciente saiu da cozinha para o quarto, é preciso notificá-lo de que o fogão está ligado através de algum dos dispositivos. As figuras seguintes mostram a tela de feedback para a simulação feita, após a notificação ter sido feita realizada.

Figura 4.19: Visão da tela de feedback após a aplicação cliente executara

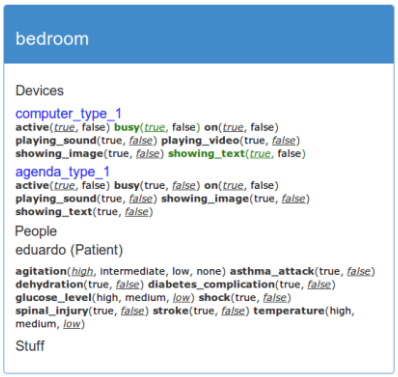


Figura 4.20: Continuação da tela de feedback

Como pôde ser notado com essas duas figuras, a aplicação estava rodando e monitorando o estado do fogão. Ele estava ligado, e, com isso, o *loop* iniciou. Enquanto o paciente estivesse na cozinha, o estado do fogão continuava sendo checado. Quando o paciente mudou de quarto, esperou-se um tempo e o fogão continuava ligado. Com isso, verificou-se novamente que o paciente ainda estava no quarto. Então ele foi notificado através de um texto no computador de que o fogão estava ligado.

Para essa simulação, era interessante que o paciente mudasse de cômodo. Para que isso acontecesse, foi feito dentro do simulador com que o paciente mudasse de quarto a cada cinco segundos, já que era interessante testar o caso em que o paciente era notificado.

5 CONCLUSÃO

Este trabalho apresentou o projeto e desenvolvimento de um simulador para uma residência inteligente, voltado para *home care*. A sua princial motivação estava em possibilitar a simulação de diferentes situações de interesse, sendo possível modificar o ambiente simulado facilmente.

O objetivo deste trabalho foi propor um simulador que fosse facilmente configurado e pudesse simular, através de Web services, dispositivos e objetos que estariam na casa, pessoas e seus diversos estados, e também organizações, que poderiam eventualmente ser chamadas. A ideia é tornar possível validar soluções para situações indesejadas em home care, como, por exemplo, a situação de agitação que foi simulada. Além disso, muitas soluções foram propostas, entretanto não é possível verificá-las atualmente, e esse é o objetivo principal do simulador.

O resultado do trabalho foi satisfatório já que, utilizando o simulador proposto, foi possível simular dois processos de negócio retirados do artigo de Kambara-Silva et al. (2014) e verificar que o simulador possibilitou testar esses processos, podendo facilmente modificar a configuração inicial do ambiente e fazer com que diferentes medidas fossem tomadas para tentar resolver a situação.

Existem algumas limitações no trabalho. Por tratar-se de um simulador, é muito difícil simular o comportamento real de uma pessoa. Para o caso de agitação, não é possível determinar se realmente o paciente foi acalmado ou não, por exemplo. Também é difícil de ser simulado o comportamento de uma organização, que pode recusar o pedido, pode demorar muito ou pouco tempo, e nem sempre que a organização chegar à residência o problema estaria resolvido. No simulador, dispositivos e organizações não interferiram no comportamento das pessoas.

Apesar de ter sido possível simular os processos de negócio desejados, ainda existem formas do simulador ser melhorado. São deixadas como sugestões para trabalhos futuros uma interface que mostre a casa e seus dispositivos graficamente, poder vincular uma funcionalidade com alguma alteração específica de status, seja ela de dispositivo ou pessoa, poder adicionar dinamicamente um dispositivo, poder adicionar dinamicamente uma funcionalidade, poder adicionar dinamicamente um status ao dispositivo e poder adicionar dinamicamente status a uma pessoa. Também seria interessante tentar simular de forma mais realista a interação entre pessoas e dispositivos na casa, de uma forma que valores aleatórios simulassem essa interação.

REFERÊNCIAS

BLACK, STEPHANIE. **Review: PostgreSQL: introduction and concepts.** Journal, Linux Journal archive, 2001.

COHN, MIKE. **User Stories Applied: For Agile Software Development**. Redwood City, CA, USA: Addison Wesley Longman Publishing Co, 2004.

CURBERA, FRANCISCO; DUFTLER, MATTHEW; KHALAF, RANIA; NAGY, WILLIAM; MUKHI, NIRMAL; WEERAWARANA, SANJIVA. **Unraveling the Web Services Web: An Introduction to SOAP, WSDL, and UDDI**. IEEE Internet Computing, 6(2), 86–93, 2002.

FLANAGAN, DAVID; FERGUSON, PAULA. **JavaScript: The Definitive Guide**. 3th ed. Sebastopol, CA, USA: O'Reilly & Associates, 1998.

FLANAGAN, DAVID; MATSUMOTO, YUKIHIRO. **The Ruby Programming Language**. 1st ed. O'Reilly, 2008.

GASSEN, JONAS. B.; MACHADO, ALENCAR; THOM, LUCINÉIA H; OLIVEIRA, JOSÉ P.M. **Ontology Support for Home Care Process Design.** In: Proceedings of the 14th International Conference on Enterprise Information Systems. SciTePress - Science and and Technology Publications, pp. 84–89, 2012.

KAMBARA-SILVA, JULIA K.; MACHADO, GUILHERME M.; THOM, LUCINÉIA H.; WIVES, LEANDRO K. **Business Process Modeling and Instantiation in Home Care Environments.** In: Proceedings of the 16th International Conference on Enterprise Information Systems, 2014. (Artigo submetido para avaliação)

KOTONYA, GERALD; SOMMERVILLE, IAN. **Requirements Engineering: Processes and Techniques**. New York, NY, USA: John Wiley & Sons, Inc, 2008.

KRASNER, GLENN E.; POPE, STEPHEN T. **A Description of the Model-View-Controller User Interface Paradigm in the Smalltalk-80 System.** Plymouth Street Mountain View, CA, 1988.

MACHADO, ALENCAR; PERNAS, ANA M.; AUGUSTIN, IARA; THOM, LUCINÉIA H.; WIVES, LEANDRO K,; OLIVEIRA, JOSÉ P.M. **Situation-awareness as a Key for Proactive Actions in Ambient Assisted Living.** Proceedings of the 15th International Conference on Enterprise Information Systems: SciTePress - Science and Technology Publications, 2013, p. 418–426.

MCGEE-LENNON, M.R. **Requirements engineering for home care technology**. In: Proceeding of the Twenty-Sixth Annual CHI Conference on Human Factors in Computing Systems - CHI ’08. New York: ACM Press, 2008, p. 1439–1442.

PAPAZOGLOU, MIKE P. **Service-Oriented Computing: Concepts, Characteristics and Directions.** Proceeding WISE '03 Proceedings of the Fourth International Conference on Web Information Systems Engineering, IEEE Computer Society Washington, DC, USA 2003.

RUBY, SAM; THOMAS, DAVE; HANSSON, DAVID H. **Agile Web Development with Rails 4**. 3rd ed., Pragmatic Bookshelf, 2011.

ULLMAN, CHRIS; DYKES, LUCINDA. **Beginning Ajax.** Wrox Press Ltd, Birmingham, UK, 2007.

UNFPA; INTERNATIONAL, HELPAGE. **Ageing in the Twenty-First Century: A Celebration and a Challenge.** UNFPA and HelpAge International, 2012.

WESKE, M**. Business Process Management**. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

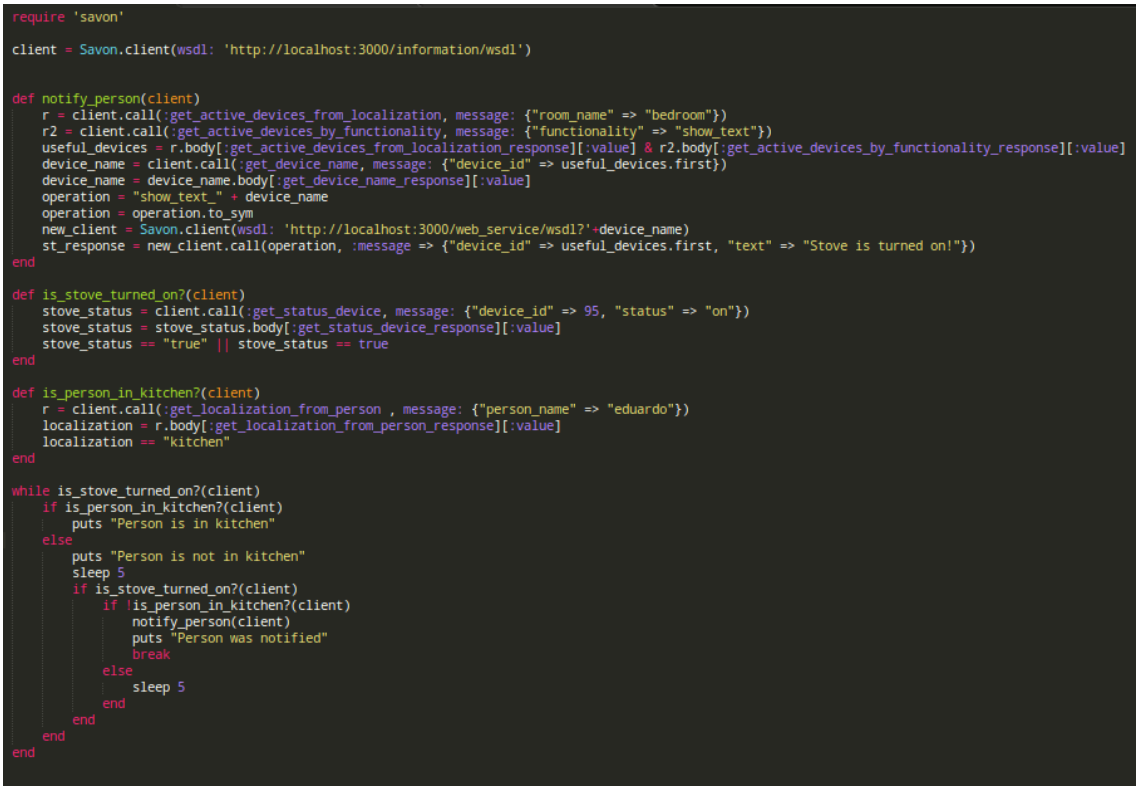
APÊNDICE CÓDIGO DAS APLICAÇÕES CLIENTE

Este apêndice apresenta o código Rails que foi utilizado no processo de negócio de agitação. Seguem as figuras que mostram o código do processo de negócio cuja simulação foi descrita na seção 4.2.

```ruby
require 'savon'

def get_status_level(status_name, patient)
    answer = nil
    patient[:status].each do |status|
        answer = status[:value] if status[:name] == status_name
    end
    answer
end

def inform_use_of_drugs(client,person)
    r = client.call(:get_active_devices_from_localization, message: {"room_name" => person[:location]})
    r2 = client.call(:get_active_devices_by_functionality, message: {"functionality" => "show_text"})
    useful_devices = r.body[:get_active_devices_from_localization_response][:value] & r2.body[:get_active_devices_by_functionality_response][:value]
    device_name = client.call(:get_device_name, message: {"device_id" => useful_devices.first})
    device_name = device_name.body[:get_device_name_response][:value]
    operation = "show_text_" + device_name
    operation = operation.to_sym
    new_client = Savon.client(wsdl: 'http://localhost:3000/web_service/wsdl?'+device_name)
    st_response = new_client.call(operation, :message => {"device_id" => useful_devices.first, "text" => "Urgent use of drugs!"})
    st_response.body["#{operation}_response".to_sym][:value] == "ok"
end

def calm_patient(client,person)
    r = client.call(:get_active_devices_from_localization, message: {"room_name" => person[:location]})
    r2 = client.call(:get_active_devices_by_functionality, message: {"functionality" => "play_sound"})
    useful_devices = r.body[:get_active_devices_from_localization_response][:value] & r2.body[:get_active_devices_by_functionality_response][:value]
    device_name = client.call(:get_device_name, message: {"device_id" => useful_devices.last})
    device_name = device_name.body[:get_device_name_response][:value]
    operation = "play_sound_" + device_name
    new_client = Savon.client(wsdl: 'http://localhost:3000/web_service/wsdl?'+device_name)
    st_response = new_client.call(operation.to_sym, :message => {"device_id" => useful_devices.last, "sound" => "Base64.encode64(File.binread('/home/edu
    st_response.body["#{operation}_response".to_sym][:value] == "ok"
end
```

```ruby
client = Savon.client(wsdl: 'http://localhost:3000/information/wsdl')
r = client.call(:get_people)
people = r.body[:get_people_response][:value]
people.each do |person|
    if person[:kind] == "Patient"
        patient = person
    elsif person[:kind] == "Caregiver"
        caregiver = person
    end
end

agitation_level = get_status_level("agitation", patient)
if agitation_level != "none"
    puts "Patient State is agitation"
    r = client.call(:notify_caregiver, message: {"caregiver_name" => caregiver[:name], "text" => "Patient state is agitation!"})
    if r.body[:notify_caregiver_response][:value] == "ok"
        puts "Caregiver was notified"
    end
    if caregiver[:location] == "not at home"
        puts "Caregiver is not at home"
        if agitation_level == "high"
            puts "Use of drugs informed to Patient" if inform_use_of_drugs(client, patient)
        end
        client.call(:call_for_organization, :message => {"organization_name" => "hospital"})
        puts "Hospital was called"
    else
        puts "Caregiver is at home"
        if agitation_level == "high"
            inform_use_of_drugs(client, caregiver)
            puts "Use of drugs informed Caregiver"
        end
    end
    puts "Patient is gettting calm" if calm_patient(client, patient)
end
```

Abaixo segue a figura da aplicação cliente responsável por implementar o processo de negócio de esquecer o fogão ligado, descrito na seção 4.3.



```ruby
require 'savon'

client = Savon.client(wsdl: 'http://localhost:3000/information/wsdl')


def notify_person(client)
    r = client.call(:get_active_devices_from_localization, message: {"room_name" => "bedroom"})
    r2 = client.call(:get_active_devices_by_functionality, message: {"functionality" => "show_text"})
    useful_devices = r.body[:get_active_devices_from_localization_response][:value] & r2.body[:get_active_devices_by_functionality_response][:value]
    device_name = client.call(:get_device_name, message: {"device_id" => useful_devices.first})
    device_name = device_name.body[:get_device_name_response][:value]
    operation = "show_text_" + device_name
    operation = operation.to_sym
    new_client = Savon.client(wsdl: 'http://localhost:3000/web_service/wsdl?'+device_name)
    st_response = new_client.call(operation, :message => {"device_id" => useful_devices.first, "text" => "Stove is turned on!"})
end

def is_stove_turned_on?(client)
    stove_status = client.call(:get_status_device, message: {"device_id" => 95, "status" => "on"})
    stove_status = stove_status.body[:get_status_device_response][:value]
    stove_status == "true" || stove_status == true
end

def is_person_in_kitchen?(client)
    r = client.call(:get_localization_from_person , message: {"person_name" => "eduardo"})
    localization = r.body[:get_localization_from_person_response][:value]
    localization == "kitchen"
end

while is_stove_turned_on?(client)
    if is_person_in_kitchen?(client)
        puts "Person is in kitchen"
    else
        puts "Person is not in kitchen"
        sleep 5
        if is_stove_turned_on?(client)
            if !is_person_in_kitchen?(client)
                notify_person(client)
                puts "Person was notified"
                break
            else
                sleep 5
            end
        end
    end
end
```