**UNIVERSIDADE ESTADUAL DA PARAÍBA**
**CAMPUS I**
**CENTRO DE CIÊNCIAS E TECNOLOGIA - CCT**
**CURSO DE COMPUTAÇÃO**

**JEFFERSON SAMPAIO DE MEDEIROS**


# DESENVOLVIMENTO DE UMA ONTOLOGIA PARA RASTREAMENTO DE ATIVIDADES EM SISTEMAS ASSISTIDOS NA UNIVERSAAL


**CAMPINA GRANDE**
**2018**

JEFFERSON SAMPAIO DE MEDEIROS

# DESENVOLVIMENTO DE UMA ONTOLOGIA PARA RASTREAMENTO DE ATIVIDADES EM SISTEMAS ASSISTIDOS NA UNIVERSAAL

Trabalho de Conclusão de Curso apresentado ao Departamento de Computação da Universidade Estadual da Paraíba, como requisito parcial à obtenção do título de Bacharel em Computação.

**Área de concentração:** Internet das Coisas.

**Orientador:** Prof. Dr. Paulo Eduardo e Silva Barbosa.

CAMPINA GRANDE
2018

É expressamente proibido a comercialização deste documento, tanto na forma impressa como eletrônica. Sua reprodução total ou parcial é permitida exclusivamente para fins acadêmicos e científicos, desde que na reprodução figure a identificação do autor, título, instituição e ano do trabalho.

M488d Medeiros, Jefferson Sampaio de.

Desenvolvimento de uma ontologia para rastreamento de atividades em sistemas assistidos da Universaal [manuscrito] / Jefferson Sampaio de Medeiros. - 2018.

77 p. : il. colorido.

Digitado.

Trabalho de Conclusão de Curso (Graduação em Computação) - Universidade Estadual da Paraíba, Centro de Ciências e Tecnologia , 2018.

"Orientação : Prof. Dr. Paulo Eduardo e Silva Barbosa , Coordenação do Curso de Computação - CCT."

1. Internet das coisas. 2. Atividade física . 3. Interoperabilidade . 4. Gerenciamento de informação. I. Título

21. ed. CDD 003.5

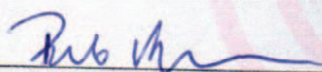
Elaborada por Gabriella D. de Oliveira - CRB - 15/840 BC/UEPB

JEFFERSON SAMPAIO DE MEDEIROS

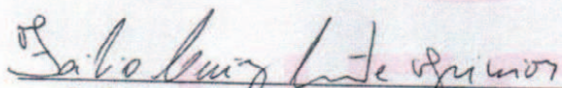
# DESENVOLVIMENTO DE UMA ONTOLOGIA PARA RASTREAMENTO DE ATIVIDADES EM SISTEMAS ASSISTIDOS NA UNIVERSAAL

Trabalho de Conclusão de Curso de Graduação em Ciência da Computação da Universidade Estadual da Paraíba, como requisito à obtenção do título de Bacharel em Ciência da Computação.

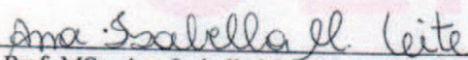
Aprovada em 22 de Novembro de 2018.



Prof. Dr. Paulo Eduardo e Silva Barbosao (DC - UEPB)
Orientador(a)



Prof. MSc. Fábio Luiz Leite Júnio (DC - UEPB)
Examinador(a)



Prof. MSc. Ana Isabella Muniz Leite (DC - UEPB)
Examinador(a)

Aos meu pais, pelo apoio incondicional, amizade e incentivo sempre, DEDICO.

## AGRADECIMENTOS

À minha família, principalmente meus pais, que sempre me apoiaram e incentivaram a continuar em momentos difíceis dessa jornada.

Aos meus amigos da época do ensino fundamental / médio, e aos novos que fiz ao longo do curso que tornaram esses anos um pouco menos difíceis.

À equipe do NUTES, principalmente Aislan e Alex que me guiaram quando estava um pouco perdido com relação a este trabalho e me ajudaram nos estágios realizados por lá.

Ao meu orientador, professor Dr. Paulo Eduardo e Silva Barbosa, pelas oportunidades dadas no laboratório e pelo auxílio dado no meu trabalho.

*"A imaginação é mais importante que a ciência, porque a ciência é limitada, ao passo que a imaginação abrange o mundo inteiro."*

*Albert Einstein*

**RESUMO**

Com o grande crescimento da Internet das Coisas cresceu também a variedade de plataformas que podem ser utilizadas nesta área do conhecimento, sendo uma delas a *universAAL* que propõe a criação de Ambientes de Vida Assistida e utiliza de ontologias como forma de representar o conhecimento. Esta foi a plataforma IoT escolhida dentro do projeto OCARIoT no qual o NUTES está participando, e que tem basicamente o objetivo de lutar contra a obesidade infantil, realizando o monitoramento de atividades físicas realizadas por crianças através de *smartwatches*. O objetivo principal deste trabalho é o desenvolvimento de uma ontologia que modele um conjunto de atividades que possa ser utilizado para representar o conhecimento em futuros módulos deste projeto. A metodologia utilizada no desenvolvimento em questão incluiu o uso do processo SCRUM, com alguns elementos não inclusos, mas contendo os principais aspectos como a ocorrência de reuniões com alguma frequência, seja com a equipe completa ou apenas com as pessoas envolvidas no desenvolvimento deste módulo, e foi utilizada a ferramenta de desenvolvimento *universAAL Studio*, um *plug-in* integrado ao *Eclipse*. Foram desenvolvidas algumas versões desta ontologia de atividades, chamada de *Activity*, onde uma destas versões foi utilizada em um sistema de aquisição de dados que tinha como objetivo principal validar uma arquitetura de microserviços, e alguns elementos desta arquitetura, como o uso do *smartwatch* Fitbit, da plataforma *universAAL*, entre outros. Os resultados obtidos com isto foram satisfatórios, visto que o modelo de dados criado pôde ser utilizado com sucesso no sistema que, por sua vez, também conseguiu ter seu objetivo alcançado. Durante este processo foram descobertos alguns pontos de melhoria na modelagem da ontologia, que poderiam simplificar o projeto e atender de forma ainda melhor as necessidades do sistema já estabelecido. Algumas modificações já foram realizadas e algumas são indicadas como trabalhos futuros.



**Palavras-Chave**: Ontologia. Atividade Física. Internet das Coisas.

## ABSTRACT

With the great growth of the Internet of Things has also grown the variety of platforms that can be used in this area of knowledge, one of them is the universAAL that proposes the creation of Assisted Living Environments and uses ontologies as a way of representing knowledge. This was the IoT platform chosen within the OCARIoT project in which NUTES is participating, and that aims to fight against childhood obesity by carrying out children's awareness activities through smartwatches. The main objective of this work is the development of an ontology that models a set of activities that can be used to represent knowledge in future modules of this project. The methodology used in the development in question included the use of the SCRUM process, with some elements not included, but containing the main aspects such as the occurrence of meetings with some frequency, either with the complete team or only with the people involved in the development of this module and using the development tool UniversAAL Studio, a plug-in built into Eclipse. Some versions of this ontology of activities have been developed, called Activity, where one of these versions was used in a data acquisition system whose main objective was to validate a microservice architecture, and some elements of this architecture, such as the use of the Fitbit smartwatch, the universAAL platform, among others. The results obtained with this were satisfactory, since the created data model could be successfully used in the system, which, in turn, also succeeded in achieving its goal. During this process some points of improvement in ontology modeling were discovered, which could simplify the project and better serve the needs of the already established system. Some modifications have already been made and others are indicated as future work.

**Keywords:** Ontology. Physical Activity. Internet of Things.

## LISTA DE ILUSTRAÇÕES


Figura 1 – Camadas da universAAL ........................................ 15
Figura 2 – Modelo de um Exporter de um dispositivo qualquer ........................................ 16
Figura 3 – Exemplo de um ContextEvent ........................................ 17
Figura 4 – Funcionamento do ContextBus ........................................ 18
Figura 5 – Modelo compartilhado entre solicitante e provedor ........................................ 18
Figura 6 – Fluxo de solicitação/resposta no ServiceBus ........................................ 19
Figura 7 – Camadas da arquitetura da Web Semântica ........................................ 22
Figura 8 – Exemplo de uma relação em um grafo conceitual ........................................ 26
Figura 9 – Modelo Ontológico ........................................ 29
Figura 10 – Classe Activity ........................................ 30
Figura 11 – Método setProperty(String propURI, Object value) ........................................ 31
Figura 12 – Classe ActivityFactory ........................................ 31
Figura 13 – Classe ActivityActivator ........................................ 32
Figura 14 – Definindo tipo de uma propriedade utilizando o URI de uma classe ........................................ 32
Figura 15 – Definindo tipo de uma propriedade utilizando a classe *TypeMapper* ........................................ 33
Figura 16 – Instalando o AAL Studio ........................................ 36
Figura 17 – Visualização do projeto no Package Explorer ........................................ 38
Figura 18 – Arquivo ".uml" da ontologia ........................................ 39
Figura 19 – Arquivo ".di" da ontologia ........................................ 39
Figura 20 – Paleta de elementos do Papyrus ........................................ 40
Figura 21 – Editando um class ........................................ 41
Figura 22 – Editando uma property ........................................ 42
Figura 23 – Definindo tipo de uma propriedade ........................................ 43
Figura 24 – Editando uma associação ........................................ 44
Figura 25 – Model Explorer ........................................ 45
Figura 26 – Importando ontologia (Tela 1) ........................................ 46
Figura 27 – Importando ontologia (Tela 2) ........................................ 46
Figura 28 – Estrutura depois de uma nova ontologia ter sido importada ........................................ 47
Figura 29 – Estrutura depois de uma nova ontologia ter sido importada ........................................ 48
Figura 30 – Operação de deletar associação feita com PersonWeight no UML ........................................ 49
Figura 31 – Transformando o modelo em código ........................................ 51

Figura 32 – Opção que possibilita a transformação Java > OWL .................................. 51
Figura 33 – Erro causado pela ausência da classe HeartRate ........................................ 52
Figura 34 – Adicionando HealthMeasurement como dependência do projeto ................ 52
Figura 35 – Código após a transformação .................................................................... 53
Figura 36 – Código após a restrição ter sido ajustada .................................................. 53
Figura 37 – Código de inserção da ontologia no repositório ......................................... 55
Figura 38 – Pasta da ontologia no repositório criado .................................................... 55
Figura 39 – Modelo inicial de Activity ........................................................................... 57
Figura 40 – Elemento PhysicalExerciseActivity ............................................................ 58
Figura 41 – Elemento WalkActivity ............................................................................... 59
Figura 42 – Elementos do tipo Device ........................................................................... 60
Figura 43 – Activity no Tools Log Monitor ................................................................... 61
Figura 44 – Arquitetura da PoC ..................................................................................... 62
Figura 45 – ContextEventPattern da aplicação Subscriber da PoC ................................ 64
Figura 46 – Criação do arquivo JSON ........................................................................... 64
Figura 47 – Classe SendActivity .................................................................................... 65
Figura 48 – Pacotes necessários para o módulo universAAL ........................................ 65
Figura 49 – Execução da PoC ........................................................................................ 66
Figura 50 – Chegada dos eventos vista no Log Monitor ................................................ 66
Figura 51 – Adição de novos repositórios ao Karaf ....................................................... 67
Figura 52 – Adição de novas dependências ................................................................... 68
Figura 53 – Execuções do plug-in do Karaf .................................................................. 68
Figura 54 – Arquivo “features.xml” .............................................................................. 68
Figura 55 – Adição de recursos pré-instalados .............................................................. 69
Figura 56 – Assistente para projetos de ontologia do AAL Studio (Tela 1) ................... 75
Figura 57 – Assistente para projetos de ontologia do AAL Studio (Tela 2) ................... 75

## LISTA DE ABREVIATURAS E SIGLAS

| | |
|---|---|
| AAL | Ambient Assisted Living |
| API | Application Programming Interface |
| CGIF | CG Interchange Format |
| DF | Display Form |
| IDE | Integrated Development Environment |
| IOT | Internet of Things |
| JAR | Java Archive |
| JDK | Java Development Kit |
| KAR | Karaf ARchive |
| KIF | Knowledge Interchange Format |
| MQTT | Message Queuing Telemetry Transport |
| NUTES | Núcleo de Tecnologias Estratégicas em Saúde |
| OCARIoT | Smart Childhood Obesity Caring Solution using IoT potential |
| OKBC | Open Knowledge Base Connectivity |
| OSGI | Open Services Gateway Initiative |
| OWL | Ontology Web Language |
| POC | Proof of Concept |
| RDF | Resource Description Framework |
| RDFS | RDF Schema |
| SPARQL | SPARQL Protocol And RDF Query Language |
| TURTLE | Terse RDF Triple Language |
| URI | Uniform Resource Identifier |
| XML | Extensible Markup Language |

**SUMÁRIO**


1 **INTRODUÇÃO** ........ 12
1.1 OBJETIVO GERAL ........ 13
1.2 OBJETIVOS ESPECÍFICOS ........ 13
1.3 ESTRUTURAÇÃO DO TRABALHO ........ 13
**2 FUNDAMENTAÇÃO TEÓRICA** ........ 14
2.1 UNIVERSAAL ........ 14
2.2 ONTOLOGIAS ........ 19
**2.2.1 Benefícios do uso de ontologias** ........ 23
**2.2.2 Linguagens para ontologias** ........ 24
**2.2.3 Ontologias na universAAL** ........ 29
**3 METODOLOGIA** ........ 34
3.1 ESTUDO DA PLATAFORMA UNIVERSAAL ........ 34
3.2 INSTALAÇÃO E EXPLORAÇÃO DO AAL STUDIO ........ 34
3.3 DESENVOLVIMENTO DA ONTOLOGIA ........ 36
3.4 USO DE ACTIVITY COMO MODELO DE DADOS DE UM SISTEMA ...... 37
**4 DESENVOLVIMENTO DA ONTOLOGIA ACTIVITY** ........ 38
4.1 MODELAGEM DA ONTOLOGIA ........ 38
4.2 PROBLEMAS E LIMITAÇÕES DA FERRAMENTA ........ 48
4.3 GERAÇÃO DO CÓDIGO ATRAVÉS DO MODELO ........ 50
4.4 INSERÇÃO DA ONTOLOGIA EM UM REPOSITÓRIO MAVEN ........ 54
**5 RESULTADOS OBTIDOS** ........ 56
5.1 DESCRIÇÃO DA ONTOLOGIA ACTIVITY ........ 56
5.2 VISÃO GERAL DA POC ........ 61
5.3 MÓDULO UNIVERSAAL DA POC ........ 63
5.4 DISTRIBUIÇÃO PERSONALIZADA DA UNIVERSAAL ........ 67
**6 CONCLUSÃO** ........ 70
**REFERÊNCIAS** ........ 71
**APÊNDICE A – UNIVERSAAL STUDIO** ........ 74

# 1 INTRODUÇÃO

A Internet das Coisas tem crescido bastante nos últimos anos, assim como o número de aplicações que já foram ou estão sendo desenvolvidas nas mais diversas áreas, como comércio, saúde, e entre outras, ambientes domésticos. Estes últimos estão associados a um conceito chamado *Ambient Assisted Living* (AAL), o conceito de um ambiente que está rodeado por sensores que têm a responsabilidade de perceber as características do local e também por atuadores que usam essa percepção para tentar influenciar o ambiente da maneira configurada pelo usuário (STOCKLÖW, 2018a).

Assim como essas aplicações estão aumentando em quantidade e proporção, devem existir ferramentas que suportem as necessidades de tais aplicativos. Falando sobre os ambientes de vida assistida, uma das ferramentas para sua construção é a plataforma *universAAL* que possibilita a conexão de vários dispositivos diferentes em uma rede única (STOCKLÖW, 2018a). Plataforma esta que foi a escolhida como parte da tecnologia IoT do OCARIoT[1], projeto que conta com a participação de várias instituições, sendo uma delas o NUTES (Núcleo de Tecnologias Estratégicas em Saúde) e que tem a iniciativa de desenvolver aplicativos associados a sensores, que irão monitorar em tempo real, informações sobre o dia a dia de crianças, como gasto de energia, refeições feitas, atividades físicas praticadas, entre outras. E, como o NUTES irá participar, dentre outras tarefas brevemente descritas no site do projeto, na parte de aquisição de dados via IoT, foi necessário o estabelecimento de uma competência na plataforma *universAAL* dentro da empresa (RNP, 2017).

Esta última é uma plataforma bem recente que vem crescendo junto a esses novos conceitos trazidos pela Internet das Coisas. Ela conta com o formalismo chamado de ontologia como sua forma de representar o conhecimento e com isto ela consegue especificar fortemente um determinado domínio. É uma plataforma que promete muita interoperabilidade, muita facilidade na manipulação (e criação) de um ambiente de vida assistida, e também um desenvolvimento rápido tendo como base conceitos como o reuso de código (há uma coleção com várias ontologias prontas disponibilizadas pela *universAAL*), sendo assim uma tecnologia que pode ser bastante utilizada até mesmo no contexto de outros projetos no futuro, visto que essa área IoT parece estar apenas começando sua expansão sobre o cotidiano das pessoas.

[1] Site do projeto OCARIoT - http://ocariot.com/

Com a definição da *universAAL* como plataforma do OCARIoT, surgiu a necessidade do desenvolvimento de uma ontologia que modelasse um conjunto de atividades para representação do conhecimento nos futuros módulos do *software*. Em um *workshop* do mesmo realizado em março de 2018 em Fortaleza, um modelo inicial dessa ontologia foi mostrado em uma apresentação da própria equipe da *universAAL*.

E é justamente onde entra a justificativa principal deste trabalho, o desenvolvimento de uma versão inicial dessa ontologia de atividades que deve servir como pontapé para versões mais maduras que sejam integradas no projeto de acordo com as necessidades encontradas.

## 1.1 OBJETIVO GERAL

Desenvolver uma ontologia de atividades que sirva como modelo de dados para os futuros aplicativos do OCARIoT.

## 1.2 OBJETIVOS ESPECÍFICOS

- Aprofundar os conhecimentos sobre o uso das ontologias na *universAAL*.
- Estudar a ferramenta de criação de ontologias da *universAAL*, *universAAL Studio*.
- Utilizar a ontologia criada como modelo de dados de um sistema que será desenvolvido pelo Laboratório de Computação Biomédica do NUTES/UEPB.

## 1.3 ESTRUTURAÇÃO DO TRABALHO

O texto foi estruturado da forma que segue. A seção 2 provê uma fundamentação teórica acerca da plataforma *universAAL*; das ontologias, seus benefícios, suas linguagens e como são utilizadas na plataforma em questão; ao final do trabalho há um apêndice sobre o *universAAL Studio*, ferramenta que promove um auxílio no desenvolvimento dessas ontologias na *universAAL*. A seção 3 mostra a metodologia utilizada para o processo de desenvolvimento do trabalho. A seção 4 detalha o processo de desenvolvimento da ontologia. A seção 5 apresenta os resultados obtidos com uma seção descrevendo a ontologia e também com a demonstração da utilização desta em um sistema de aquisição de dados. E, por fim, na seção 6 temos as considerações finais acerca dos principais pontos do trabalho.

# 2 FUNDAMENTAÇÃO TEÓRICA

O texto a seguir descreve a base teórica acerca das tecnologias utilizadas no trabalho, que envolvem a *universAAL*, e sobre as ontologias em si, forma de representação do conhecimento utilizada por essa plataforma IoT. A subseção 1 trata da *universAAL* em si e a 2 sobre as ontologias de forma geral e no contexto *universaal*.

## 2.1 UNIVERSAAL

Como já é sabido, tudo muda muito rápido no mundo da Computação. Como o cientista Mark Weiser já havia previsto, os antigos e enormes computadores de antigamente se tornaram *chips* muito pequenos e menos caros, *chips* esses que estão começando a nos rodear e auxiliar sem que suas presenças sejam notadas, e sem necessariamente estarem presentes em computadores *desktop,* mas até mesmo em uma TV ou geladeira. Essa situação leva ao conceito de ambientes que, entre outras denominações, são chamados de ambientes de vida assistida (*Ambient Assisted Living* – AAL) e que têm em seu funcionamento mais básico a ideia de perceber o ambiente com sensores e tentar influenciá-lo com atuadores de forma que o mesmo fique o mais desejável possível ao gosto dos usuários. (Cf. STOCKLÖW, 2018a).

Ainda sobre os AAL, Girolami et al. (2014) afirmam que um Serviço AAL se refere a artefatos de *software*, itens de *hardware* e recursos humanos que, em conjunto, compõem um serviço útil para idosos.

Para Lim et al. (2018) a sociedade, principalmente os idosos, tendem a se atrair cada vez mais por essas soluções AAL. Dizem ainda que tem havido muito esforço para fornecer infraestrutura de *software* e *middleware* para esses ambientes assistidos.

A necessidade de suporte a esse conceito de Ambientes de Vida Assistida está cada vez maior, e com isso chega-se a plataforma *universAAL*, que em uma definição bem simplista é um *software* que ajuda a construir esses ambientes conectando vários dispositivos diferentes a uma única rede (STOCKLÖW, 2018a).

Lim et al. (2018) dizem que ela é a solução mais promissora quando se fala em serviços AAL, apresentando benefícios como ser personalizada, fácil de configurar e acessível, o que leva a um desenvolvimento mais fácil e barato.

A *universAAL* deve ajudar a criar esses ambientes nos quais, por exemplo, um sensor de movimento detecte a presença de uma pessoa em um cômodo e outro sensor de brilho

detecte que este cômodo está escuro, e através dessa percepção do ambiente o sistema de assistência fique sabendo que esse não é um estado desejável pelo usuário e ligue as luzes do cômodo, se assim estiver configurado, claro. Mas, para que isto aconteça, é natural que todos esses sensores e atuadores estejam conectados em uma mesma rede. Na plataforma, cada dispositivo deste é chamado de nó, e essa integração pode ocorrer de duas maneiras, sendo a primeira, a instalação do *middleware*, uma de suas peças que contém a infraestrutura necessária para comunicação, no dispositivo. E, sendo outra maneira, o uso de um "nó escravo" que executa esse *middleware*, agindo como um nó intermediário para a comunicação entre um dado dispositivo e a rede *universaal* desejada. Sendo que esta última situação acontece nos casos em que um dispositivo desejado em um sistema assistido não possa instalar o *middleware* da *universAAL*. (Cf. STOCKLÖW, 2018a).

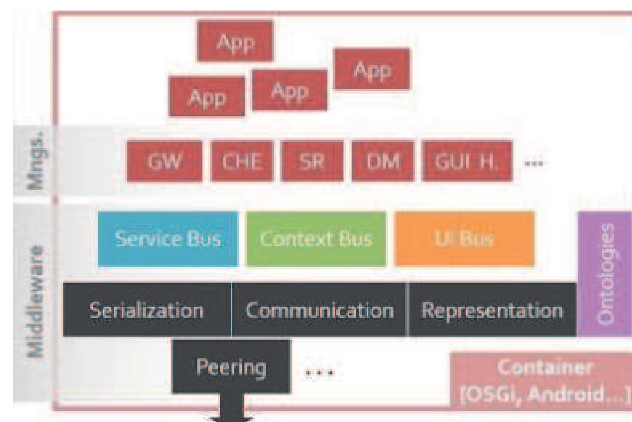
Muito foi falado no último parágrafo sobre o *middleware* da plataforma, então faz-se necessária uma breve explicação do que ele é basicamente:

> O Middleware é a parte central da plataforma universAAL e cuida para que todos os nós universaais em um Space possam cooperar uns com os outros. Estabelece comunicações peer-to-peer entre eles para que possam compartilhar os diferentes tipos de comunicação semântica na universAAL: Contexto, Serviço e Interação com o Usuário, seguindo o Modelo Ontológico compartilhado. (STOCKLÖW, 2018b).

A partir daqui quando a palavra *middleware* for mencionada será em referência ao componente específico da *universAAL*, a menos que seja dito o contrário. O mesmo serve para o termo "plataforma", que sempre estará sendo usado para falar da *universAAL*.

Abaixo tem-se uma figura onde são mostrados os componentes da plataforma distribuídos em camadas.

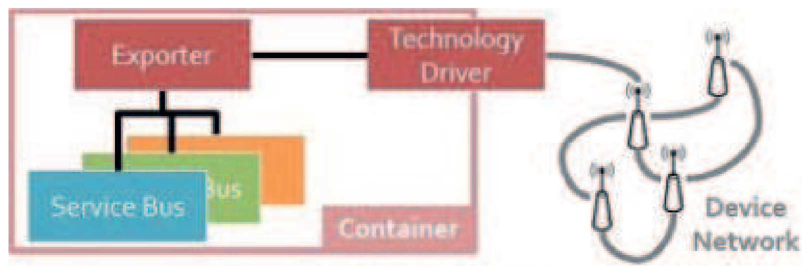
**Figura 1 – Camadas da *universAAL***




Fonte: Stocklöw (2018b)

Na figura acima, tem-se o *Container* que permite a execução da plataforma em diferentes ambientes, sendo que atualmente, são suportados apenas dispositivos que executem OSGi ou *Android*. O *Peering* interconecta e comunica as instâncias do *middleware* usando tecnologias como *jSLP* e *jGroups*. *Communication* contém a lógica que permite o fluxo semântico entre nós *universaais* e o componente *Serialization* criptografa e analisa mensagens nos nós. Os *Managers* são aplicativos de baixo nível que são necessários para o funcionamento perfeito de toda a plataforma e, na camada mais alta, tem-se os aplicativos *universaais* que são quaisquer *softwares* que possam ser executados no *Container* e que faça uso dos barramentos ou *Managers*. Para concluir o modelo da figura basta falar sobre os sensores e atuadores já mencionados como parte do sistema assistido construído com a ajuda da *universAAL*. Eles são conectados através de *Exporters* que, naturalmente, exportam as informações sobre esses dispositivos para a plataforma, como visto na figura abaixo. (Cf. STOCKLÖW, 2018b).

**Figura 2 – Modelo de um *Exporter* de um dispositivo qualquer**




Fonte: Stocklöw (2018b)

O conceito de *Space* (ou *uSpace*) também é mencionado na definição do *middleware*, mas qual papel esse elemento teria na plataforma?

A definição oficial de um *uSpace* é

> Um ambiente inteligente centrado em seus usuários humanos, no qual um conjunto de artefatos de rede embarcados, tanto hardware quanto software, realizam coletivamente o paradigma da Inteligência Ambiental, principalmente por fornecer reconhecimento de contexto e personalização, reatividade e pró-atividade. (STOCKLÖW, 2018b).

Agora que foram explicadas as camadas da plataforma, e o conceito de um *Space*, de forma mais alto nível, falta abranger os tipos de comunicação (Contexto, Serviço...) e o Modelo Ontológico, mencionados na definição oficial do *middleware*.

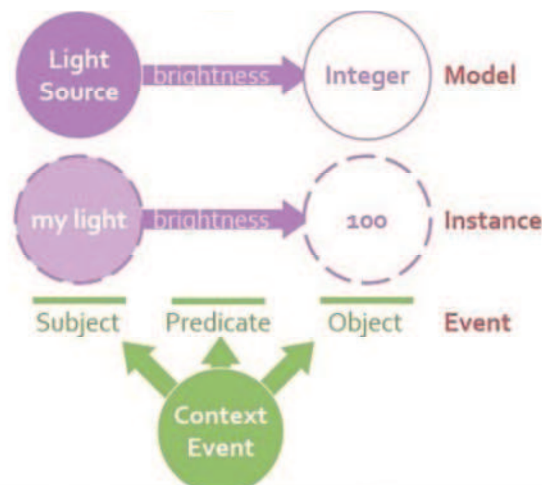
Sobre os tipos de comunicação, Contexto, Serviço e Interação com o Usuário, há de se saber que existe um barramento na plataforma para cada um, sendo eles, respectivamente, *Context Bus*, *Service Bus* e *UI (User Interaction) Bus*. E ainda um *Control Bus*, não muito citado nas documentações da plataforma, que é responsável pelo gerenciamento dos nós em um *uSpace*, pela descoberta de nós novos e pela implementação de artefatos de *software* (STOCKLÖW, 2018a).

O *Context Bus* é um barramento baseado em eventos, que recebe informações de um aplicativo e também as publica, sem se importar com a existência ou não de receptores reais. O *Service Bus* é um barramento baseado em chamadas que recebe solicitações de serviço (*ServiceRequests*) e busca um ou mais perfis de serviço (*ServiceProfiles)* que forneçam esse serviço solicitado. E o *UI Bus* é o barramento que manipula toda e qualquer mensagem que tenha a ver com a interação explícita com o usuário. (Cf. STOCKLÖW, 2018a).

Sobre a comunicação no *Context Bus*, Stocklöw (2018b) diz que ela é feita basicamente através de Eventos de Contexto (*ContextEvents*) (figura 3) que são enviados por Editores de Contexto (*ContextPublishers*) e recebidos por Assinantes de Contexto (*ContextSubscribers*), da forma que segue na figura 4.

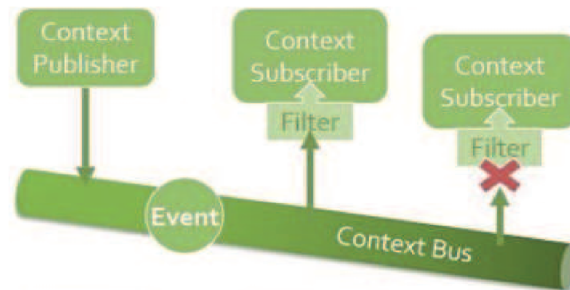
Um evento nesse barramento é composto por três propriedades, assunto (*Subject*), predicado (*Predicate*) e objeto (*Object*). O assunto é o conceito sobre o qual o evento está dizendo algo, o predicado é uma propriedade e identifica a informação exata de interesse dentro desse assunto, e o objeto é o valor dessa propriedade. (Cf. STOCKLÖW, 2018g).

**Figura 3 – Exemplo de um *ContextEvent***




Fonte: Stocklöw (2018b)

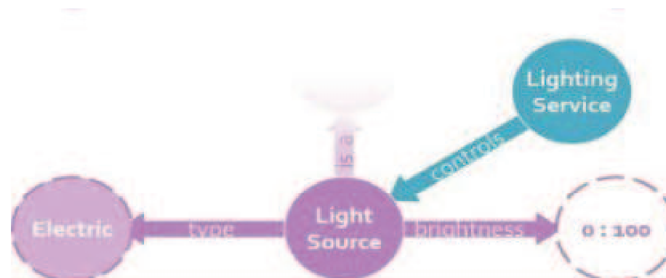
**Figura 4 – Funcionamento do *ContextBus***




Fonte: Stocklöw (2018b)

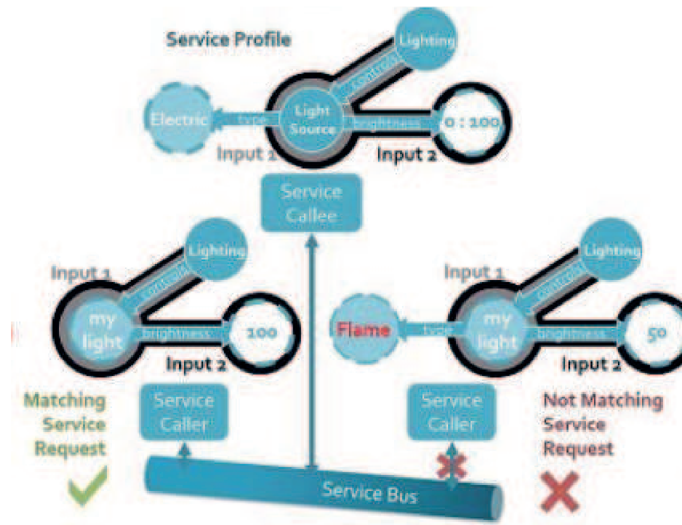
E sobre o *Service Bus*, Stocklöw (2018b) fala que os aplicativos que fornecem certos serviços através da criação de perfis de serviço são chamados de *ServiceCallees* e existem também os *ServiceCallers* que solicitam serviços, solicitações essas que são combinadas com os perfis registrados no barramento e se forem equivalentes ontologicamente, o *ServiceCallee* que os registrou será chamado e dará uma resposta (figura 6). Como é notado na última frase, há de existir um modelo compartilhado entre o solicitante e provedor para que exista uma equivalência quando houver uma solicitação, sendo um exemplo deste modelo, o elemento "*LightingService*" da figura 5.

**Figura 5 – Modelo compartilhado entre solicitante e provedor**




Fonte: Stocklöw (2018b)

**Figura 6 – Fluxo de solicitação/resposta no *ServiceBus***




Fonte: Stocklöw (2018b)

O tipo de comunicação referente à interação com o usuário não será melhor abordado porque ainda não é tão usado, fugindo assim do escopo do trabalho. Na verdade, o *Context Bus* será o mais mencionado daqui para frente, o *Service Bus* só recebeu algum detalhamento pois também é importante para o entendimento da importância da *universAAL*.

Após o aprofundamento desta última, basta mencionar que ela é uma das melhores opções quando o assunto trata de soluções AAL. E com a Internet das Coisas ganhando cada vez mais peso, além desse aumento da demanda de ambientes domésticos autômatos, o uso dessa plataforma só tende a crescer.

Na próxima subseção as ontologias serão melhor descritas, assim como suas origens, seus benefícios, as linguagens que podem ser usadas para representá-las e como elas são usadas como modelo de dados na *universAAL*.

## 2.2 ONTOLOGIAS

Com a necessidade cada vez maior de representação de domínios na Computação, principalmente na área da Inteligência Artificial, surgiu uma subárea que passou a ser chamada de Representação do Conhecimento que tenta fazer com que conceitos de um determinado domínio sejam entendidos por máquinas e programas inteligentes.

Assim, essa questão do conhecimento se tornou crítica, visto que a área da Inteligência Artificial, por exemplo, dependia totalmente disso para avançar seus conceitos, sendo assim foi surgindo uma demanda cada vez maior relacionada ao conhecimento e à sua

representação, sendo exemplos de linguagens utilizadas para isto, as Redes Semânticas, Frames, Regras de Produção, Redes Neurais, entre outras.

Dentre estas linguagens há as ontologias que suportam algumas operações que podem ser realizadas sobre o conhecimento, como por exemplo, a sua captura, seu processamento, reutilização e comunicação. Sendo assim, o uso delas se tornou uma das formas mais poderosas de suportar essas necessidades dentro do campo da Representação do Conhecimento. (Cf. OBITKO, 2007).

Para se entender o porquê delas serem tão poderosas nesse sentido é preciso saber que elas foram usadas na filosofia clássica para estudar basicamente a estrutura da existência das coisas, ou seja, sempre foram utilizadas nesse sentido de representação, de como que uma determinada existência pode ser especificada. Segundo Almeida (2014, p.242-258), o termo já começava a ser utilizado no campo da Representação do Conhecimento desde a década de 60.

Nas palavras de Gruber (1993, p.199-220) uma ontologia seria uma "uma especificação explícita de uma conceituação", onde essa conceituação seria, nesse contexto, uma visão simples e abstrata de um mundo que se deseja representar para algum fim. O autor ainda estabelece uma ponte entre o conceito de ontologia nos dois grandes campos, Ciência da Computação e Filosofia, quando afirma que para sistemas baseados em conhecimento, tudo aquilo que existe é justamente aquilo que pode ser representado. Para o autor, quando um formalismo declarativo representa um domínio, o conjunto de objetos que podem ter uma representação é chamado de universo do discurso, e esse conjunto junto as relações formalizadas entre eles são refletidos em um vocabulário representacional com o qual um programa baseado em conhecimento representa esse conhecimento. Neste sentido, no contexto da IA uma ontologia pode ser descrita com a definição de um conjunto de termos representacionais de um determinado domínio.

Vale destacar um conceito importante relacionado a elas que é o chamado compromisso ontológico que são acordos impostos sobre os objetos e relações que estão sendo discutidos entre agentes, em interfaces de módulos de software ou em bases de conhecimento. Sendo assim, falando em um sentido mais próximo das linguagens de programação, as ontologias poderiam ser vistas como declarações de tipos que são globais em uma API, e os compromissos ontológicos seriam as restrições de tipo impostas sobre as entradas e saídas de um programa. (Cf. GRUBER, 1993, p.199-220).

Tendo-se em mente que uma ontologia na Computação pode ser vista basicamente como a definição de um conjunto de termos representacionais e das relações existentes entre eles, há uma definição que parece se encaixar nesse conceito e ainda trazer um dos benefícios

do uso desse formalismo (o de compartilhar conhecimento), essa definição é vista nas palavras de Noy e Mcguinness (2001): “Uma ontologia define um vocabulário comum para pesquisadores que precisam compartilhar informações em um domínio. Inclui definições interpretáveis por máquina de conceitos básicos no domínio e relações entre eles.”

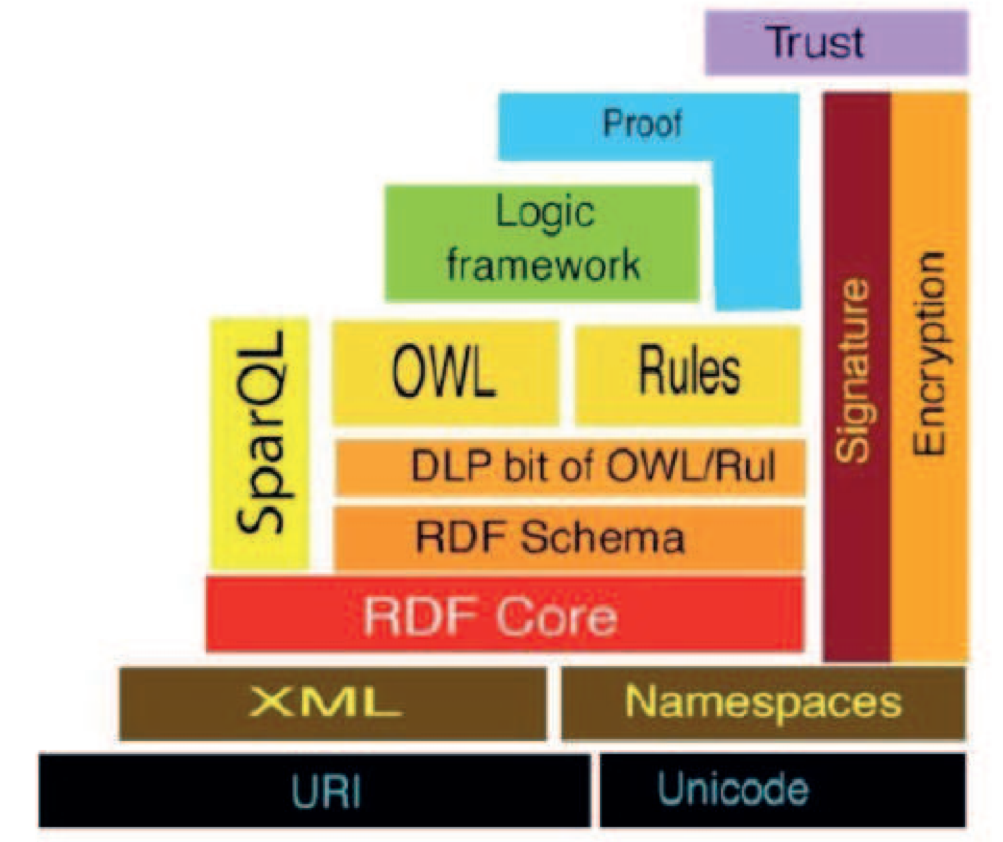
Temos nas palavras de Obitko (2007) mais uma definição que concretiza um pouco mais o escopo deste formalismo, onde ele diz que uma ontologia fornece uma base para o conhecimento modelado em sistemas baseados em conhecimento e que pode ser definida em uma outra visão como um corpo de conhecimento (e não como uma forma de descrevê-lo) que descreve algum domínio, tipicamente o domínio do conhecimento do senso comum (OBITKO, 2007).

Logo, percebe-se que apesar de existirem algumas visões um pouco divergentes quanto à relação entre as ontologias e a Computação, de qualquer forma esta acaba girando em torno da representação de informações de um domínio específico, e que não está tão presente atualmente apenas por causa de sua relação com a IA, mas também pelo aumento da força e presença de tecnologias como a Web 2.0 e a Web Semântica. Falando sobre esta última, ela promove que a Web deve ser vista como um repositório único de informação, ao invés de um conjunto de sites, levando muitos a pensar que isto implica em uma “única e grande ontologia ou esquema organizacional, um anel para dominar a todos” (BERGMAN, 2007).

Mas, em uma pequena reflexão este último pensamento anterior é invalidado, bastando perceber a alta diversidade de domínios que existem, onde cada um deles pode ser representado ontologicamente, sendo que Bergman (2007) define uma possível equação para uma provável quantidade de ontologias existentes, que seria: domínio x perspectiva x esquema.

Ainda sobre a Web Semântica, há uma pirâmide que representa sua arquitetura e que mostra mais sobre as ontologias nesse contexto, ela pode ser vista na figura abaixo e foi a versão proposta pela W3C em 2005. Como tudo na Computação está em constante evolução, há outras versões desta arquitetura que podem ser vistas com mais detalhes em Ramalho (2006).

**Figura 7 – Camadas da arquitetura da Web Semântica**




Fonte: Ramalho (2006)

Na figura pode-se ver os elementos da Web Semântica que são definidos por Ramalho (2006) da seguinte forma:

- O **Unicode** e o **URI**, que são respectivamente um esquema padrão para codificar caracteres e um identificador exclusivo de recursos na Web;
- ***Signature*** que tem a função de simular a assinatura de uma pessoa de forma digital e ***Encryption***, um processo que cifra as informações para fornecer mais segurança;
- **XML**, uma linguagem computacional que permite que dados sejam estruturados através da definição de atributos;
- ***Namespaces***, uma coleção de nomes identificados por um **URI** utilizados em XML para validação de atributos e elementos;
- **RDF *Core***, núcleo que envolve as especificações do modelo e a sintaxe da linguagem RDF e **RDF *Schema***, utilizada para a representação de um vocabulário RDF na Web;
- ***SparQL***, uma linguagem de consultas em estruturas RDF;

- **DLP**, uma tecnologia que funciona como ponte entre a **OWL** e a **F-Logic**, onde a OWL é a linguagem padrão para desenvolvimento de ontologias definida pela W3C, e a F-Logic também é usada para representar conhecimento e ontologias, sendo que a DLP junto da SparQL formam um conjunto de tecnologias que possibilitam a integração da camada de ontologias às outras, uma dificuldade encontrada nas arquiteturas anteriores;
- ***Rules***, uma camada que fornece a definição de regras lógicas para os recursos;
- ***Logic Framework***, que permite a definição de regras mas dessa vez para as informações descritas nos níveis inferiores;
- ***Proof***, que funciona como uma espécie de validador da coerência lógica dos recursos; e
- ***Trust***, camada que tenta prover um grau de confiabilidade no modo como as informações são representadas.

Onde cada uma dessas camadas deve ser compatível com as camadas inferiores e não depender das camadas superiores, proporcionando uma estrutura escalonável (RAMALHO, 2006).

Assim, vê-se que as ontologias também aparecem como forma de representar o conhecimento em áreas mais novas como a Web Semântica, por conta da forte relação entre esta última com informações.

É notável que o conceito mais recorrente relacionado a elas na Computação é o dado por Gruber (1993), o de serem uma "especificação explícita de uma conceituação" mas também existem outras razões pelas quais um desenvolvedor resolve criar uma ontologia, e na próxima subseção serão descritos brevemente alguns benefícios obtidos com o uso delas na Computação.

### 2.2.1 Benefícios do uso de ontologias

Noy e Mcguinness (2001) listam alguns pontos positivos do uso de ontologias, são eles:

- Compartilhar o entendimento comum da estrutura de informações entre agentes de software
- Permitir a reutilização do conhecimento de um domínio
- Tornar as suposições de um domínio explícitas

- Separar o conhecimento de domínio do conhecimento operacional
- Analisar o conhecimento de um domínio

Aqui se faz necessária uma explicação acerca dos pontos três e quatro listados acima, onde "Tornar as suposições de um domínio explícitas" facilitam muito em um cenário no qual o conhecimento do domínio possa mudar no futuro (NOY; MCGUINNESS, 2001), e "Separar o conhecimento de domínio do conhecimento operacional" torna mais prático o desenvolvimento, visto que o uso de uma ontologia desenvolvida para um único domínio pode abranger muitos tipos de cenários operacionais.

Bergman (2007) também lista alguns aspectos positivos relacionados à utilização das ontologias, e, entre outros (bem semelhantes ao já listados anteriormente), estão:

- Dependendo de seu grau de formalismo, as ontologias ajudam a deixar claro o escopo, sua definição, a linguagem e a semântica de um determinado domínio;
- Se hierarquicamente estruturadas, podem fornecer o poder da herança;
- Fornecem orientação sobre como colocar informações de forma correta em relação a outras informações nesse domínio;
- Também podem ser uma fonte de vocabulários estruturados e controlados, úteis para um contexto ambíguo, dependendo claro de seu grau de formalismo;
- E, podem atuar como estruturas orientadoras para navegação dentro de um domínio.

Listados esses aspectos, percebe-se que a maioria deles gira em torno de aperfeiçoar a relação humano-informação que está cada vez mais forte com os novos campos que estão surgindo e se consolidando rapidamente como as tecnologias predominantes, como a Web Semântica bastante mencionada nesse texto, e, não menos importante, a Internet das Coisas, que tem no seu âmago a necessidade de lidar com informações, a uma alta velocidade e com grande nível de segurança. Sendo assim, as ontologias aparecem na Ciência da Computação, como foco nesse apoio às melhores formas de lidar com o conhecimento, apoio fundamental para a consolidação das novas tecnologias da Indústria 4.0.

A seguir serão descritas algumas linguagens que podem ser usadas para a representação de uma ontologia.

### 2.2.2 Linguagens para ontologias

Existem diversas linguagens descritas na literatura que podem ser usadas para expressar ontologias e algumas delas serão apresentadas nesta seção. É importante dizer que

serão apresentadas de forma simples, não será feita nenhuma descrição extremamente aprofundada de nenhuma delas, sendo o escopo deste texto apenas indicar que existem e que podem ser usadas a fim de se especificar uma ontologia.

Há vários níveis de linguagens formais para se modelar uma ontologia, ou se expressar o conhecimento com base em uma ontologia, e, logicamente, quanto menos formal for essa linguagem, mais fácil será implementar uma. No entanto, à medida que essa formalidade aumenta, não só a dificuldade de desenvolvimento é elevada, mas também o poder de expressividade da ontologia, assim como sua capacidade de compartilhamento/reusabilidade. (Cf. OBITKO, 2007).

Um primeiro formalismo que pode ser citado é o de modelos *Frame-Based*, ou modelos baseados em quadros que usam *frames* como entidades e suas propriedades como uma primitiva de modelagem, onde esta última seria um conjunto formado por um *frame* e *slots* aplicados apenas a este *frame* para o qual foram definidos. Cada atributo pode ter restrições definidas sobre si, que são conhecidas aqui como *facets* (facetas) e os *frames* são o meio de se modelar os aspectos do domínio desejado. Uma característica importante desse formalismo é a possibilidade de herança entre os *frames*. (Cf. OBITKO, 2007).

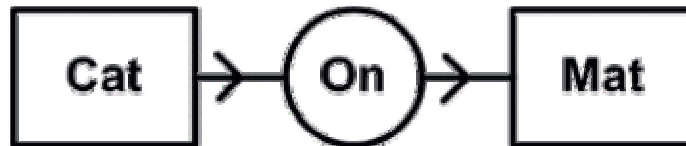
Um exemplo de modelo *Frame-Based* é o OKBC (*Open Knowledge Base Connectivity*), uma API para acessar sistemas de representação de conhecimento, com todos elementos descritos no parágrafo anterior presentes e ainda alguns novos. É independente de linguagem e fornece implementações em *Common* LISP, Java e C (SRI International, 1995).

Outra linguagem formal é chamada de rede semântica que pode ser definida como um grafo, onde os vértices representam conceitos e as ligações entre eles (arestas) representam as relações definidas entre esses conceitos. Sendo que os tipos de relações existentes nas redes semânticas são em quatro: *synonym*, onde um conceito A expressa a mesma ideia que um conceito B; *antonym*, onde os conceitos A e B expressam ideias opostas; *meronym* e *holonym*, que são relações de tipo, "parte de" e "tem parte", respectivamente; e *hyponym* e *hypernym*, que representam o intervalo semântico em ambos sentidos entre os conceitos. Foram criadas com a intenção de expressar a interlíngua, um idioma comum que pudesse ser usado para tradução entre vários outros idiomas. (Cf. OBITKO, 2007).

Um exemplo de rede semântica é o *WordNet*, um grande banco de dados léxico do inglês, onde substantivos, adjetivos, verbos e advérbios são agrupados em conjuntos de sinônimos cognitivos onde cada um expressa um conceito distinto, formando uma rede de palavras e conceitos altamente relacionados (PRINCETON University, 2005).

Ontologias ainda podem ser representadas através de grafos conceituais, um formalismo lógico que inclui classes, indivíduos, relações e quantificadores e têm tradução direta para a lógica de predicados de primeira ordem. Esses grafos têm uma representação gráfica (figura 8), e são baseados nas redes semânticas. (Cf. OBITKO, 2007).

**Figura 8 – Exemplo de uma relação em um grafo conceitual**




Fonte: Obitko (2007)

Na figura acima pode-se ver uma representação da frase "um gato está na esteira" na forma chamada de *Display Form* (DF), onde os retângulos representam conceitos, as elipses representam as relações e as arestas orientadas representam o sentido da relação fazendo a ligação entre os conceitos (OBITKO, 2007). Como foi dito, esse formalismo pode ser traduzido para a lógica de predicados de primeira ordem e isso se dá da forma que segue.

A frase expressa em DF pode ser também expressa na linguagem formal CGIF (CG *Interchange Form*) na forma [Cat: *x] [Mat: *y] (On ?x ?y) ou (On [Cat] [Mat]), onde *y define uma variável e ?y uma referência para esta. Sendo que estas expressões têm o mesmo sentido da seguinte sentença $\exists$ x,y: Cat(x) ^ Mat(x) ^ on(x, y) na lógica de predicado. (Cf. OBITKO, 2007).

Outro formalismo muito importante na construção de ontologias é formado pelo conjunto da Ontolingua e o KIF (*Knowledge Interchange Format*). Esta última é uma linguagem criada para troca de conhecimento entre sistemas diferentes, que tem a sintaxe semelhante à da linguagem LISP e foi definida dentro do projeto Ontolingua, um sistema para descrição de ontologias que permite compatibilidade entre múltiplos formalismos (OBITKO, 2007).

Um exemplo de definição de um conceito de uma ontologia na Ontolingua/KIF é mostrado no exemplo abaixo retirado, literalmente, da obra de Gruber (1993):

```
(define-class AUTHOR (?author)
      "An author is a person who writes things. An author must have created
      at least one document. In this ontology, an author is known by his or
      her real name."
:def (and person ?author)
      (= (value-cardinality ?author AUTHOR.NAME) 1)
```

```
(value-type ?author AUTHOR.NAME biblio-name)
(>= (value-cardinality ?author AUTHOR.DOCUMENTS) 1)
(<=> (author.name ?author ?name)
        (person.name ?author ?name))))
```

*AUTHOR* é um dos conceitos (denota uma classe) de uma ontologia para informação bibliográfica, ontologia essa que tem a finalidade de apoiar tarefas como compartilhamento de conhecimento, integração de dados bibliográficos com outros bancos de dados, entre outras. A sentença *def* define as condições necessárias para um objeto individual ser uma instância dessa classe, sendo que na própria linha da palavra-chave *def*, a sentença que segue diz que um autor deve ser uma pessoa. Na linha que segue é definido que um autor deve ter somente um nome associado, isto através da relação *AUTHOR.NAME*. Após isso é definido uma restrição de tipo (*value-type*) para que o valor da relação *AUTHOR.NAME* aplicada a cada autor seja do tipo *biblio-name*. A próxima linha define que deve existir pelo menos um documento associado a cada autor (*AUTHOR.DOCUMENTS*). E a última definição especifica que os nomes definidos em *AUTHOR.NAME* e *PERSON.NAME* devem ser iguais (Cf. GRUBER, 1993, p.199-220).

Neste exemplo pode-se ver como são definidos elementos como classe, restrições, cardinalidade, entre outros, na linguagem KIF. E é importante ressaltar que assim como com a linguagem CGIF, também pode-se traduzir uma sentença de KIF para a lógica de predicados (OBITKO, 2007), sendo este talvez o elo mais forte entre a maioria desses formalismos e justamente o que permite a tradução entre os mesmos.

Ainda é necessário dizer que a Ontolingua não suporta consultas sendo este um passo realizado por sistemas especializados, e reforçar que seu principal propósito é tornar as ontologias portáteis (GRUBER, 1993, p.199-220), uma característica que como foi dito na seção anterior é um dos benefícios que o uso de ontologias traz, essa portabilidade do conhecimento.

Por fim, é importante falar da OWL (*Ontology Web Language*, já citada neste documento) que é uma das mais conhecidas quando se fala em linguagens para ontologias. Essa linguagem foi projetada pelo W3C (*World Wide Web Consortium*) para a Web Semântica, com o objetivo de representar conhecimento sobre coisas, grupos de coisas e relações entre coisas, seja esse conhecimento simples ou complexo, e está na versão 2 conhecida como OWL 2 (W3C, 2013a).

OWL é uma linguagem baseada em lógica computacional que faz parte do conjunto de tecnologias da Web Semântica, que inclui outras tecnologias como RDF, RDFS e SPARQL

(W3C, 2013a), que são, respectivamente, um modelo padrão para troca de dados na WEB (W3C, 2014) que pode ser representado por alguns formalismos como o Turtle; uma linguagem geral para representação de vocabulários RDF simples na WEB (W3C, 2010); e uma linguagem de consulta para várias fontes de dados, sejam eles armazenados como RDF de forma nativa ou visualizados como RDF por meio de um *middleware* (W3C, 2013b).

OWL é a linguagem padrão para desenvolvimento de ontologias, é muito poderosa e possui vários documentos descrevendo sua sintaxe, portanto, para mais informações aprofundadas, as referências dadas podem ser consultadas. Apenas a cargo de uma pequena exemplificação, um pequeno trecho de código com a sintaxe OWL retirado da obra de Obitko (2007) é mostrado abaixo:

```
Namespace(p = <http://example.com/pizzas.owl#>)
Ontology( <http://example.com/pizzas.owl#>
Class(p: Pizza partial
      restriction(p:hasBase someValuesFrom(p:PizzaBase)))
DisjointClasses(p:Pizza p:PizzaBase)
Class(p:NonVegetarianPizza complete
      intersectionOf(p:Pizza complementOf(p:VegetarianPizza)))
ObjectProperty(p:isIngredientOf Transitive
      inverseOf(p:hasIngredient))
)
```

No exemplo acima retirado de uma ontologia de pizzas, pode ser visto o uso de elementos como: conceitos (*PizzaBase*), propriedades (*isIngredientOf*) que são características dos conceitos, e até a definição de algumas restrições como na quarta linha com o uso da palavra reservada *restriction* restringindo os valores para “p” como sendo do tipo *PizzaBase*.

Assim, pode-se concluir que existe uma grande variedade de linguagens/formalismos para se expressar ontologias, sendo apenas alguns deles citados neste texto. Mesmo assim, dá para se ter uma pequena ideia de como funciona esse mundo computacionalmente falando. Essas linguagens geralmente têm um denominador comum (a lógica de predicados) que proporcionam a possibilidade de tradução entre elas, sendo assim, pode-se expressar o conhecimento computacionalmente (um dos principais objetivos do uso de ontologias) sem se preocupar com qual idioma está sendo utilizado.

E, falar sobre todas essas formas de se expressar ontologias, é basicamente o mesmo que dizer que surgiram ao longo dessas últimas décadas formalismos que passaram a apoiar cada vez mais o campo da Representação do Conhecimento, o que por sua vez dá um maior apoio à IA e seu avanço. No ponto que a área está, o conhecimento já tem muito suporte no

que diz respeito à sua representação, talvez os próximos problemas a serem resolvidos digam respeito a como utilizá-lo de maneira inteligente a fim de avançar nesse sentido, ou seja, como integrá-lo com outras tecnologias para alcançar objetivos maiores e mais concretos.

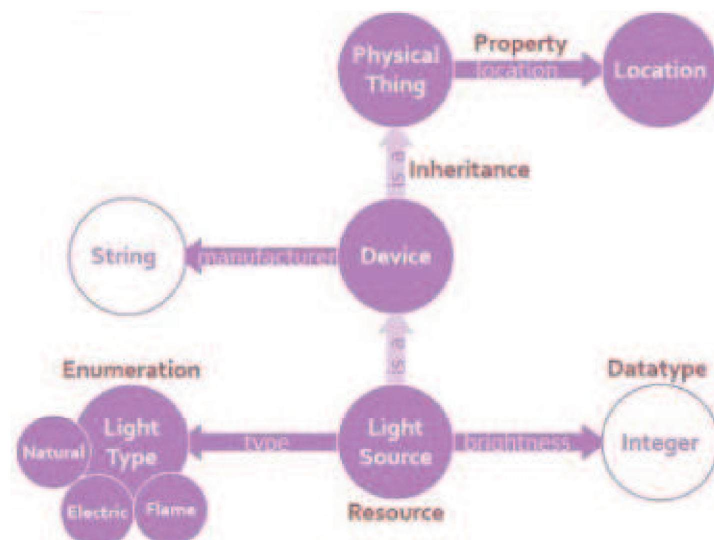
A próxima subseção irá abranger a forma pela qual as ontologias são representadas na *universAAL*.

### 2.2.3 Ontologias na universAAL

Como já foi dito na seção sobre a *universAAL*, uma ontologia no âmbito desta plataforma é a forma de se modelar o conhecimento, mas como elas são representadas nesse contexto é o questionamento que esta seção procura abranger.

Para Stocklöw (2018c) uma ontologia pode ser vista como uma rede de conceitos que são ligados através de propriedades, e que pode ser representada em alguns formatos, como por exemplo o *RDF*, mas, para que possa ser manipulada pelo *middleware*, é representada em código *Java* no contexto *universaal*. Geralmente são agrupadas em um campo mais geral do conhecimento como *Device* (domínio de dispositivos), *Health* (domínio relacionado à saúde), entre outros. O Modelo Ontológico da plataforma é mostrado na figura que segue.

**Figura 9 – Modelo Ontológico**




Fonte: Stocklöw (2018b)

*Resources* são como os conceitos são representados, sendo identificados por um *URI* (*Uniform Resource Identifier*) e podendo herdar de outros *resources*, além de terem propriedades (*properties*) associadas. Essas *properties* são ligações entre os *resources*, que

também são identificadas por *URIs* e têm a possibilidade de herança, sendo que pode haver restrições sobre elas. Os *Datatypes* são os formatos de dados nativos, como *String*, *Float*, entre outros e as *Enumerations* são conjuntos de instâncias de *resources*, representando alguns valores que uma *Property* pode ter. (Cf. STOCKLÖW, 2018b).

No geral uma ontologia depois de criada deve ter uma coleção de classes de ontologia (usadas para representar *resources*), uma classe de ontologia (mais geral) que define todas as *properties* de toda essa coleção e um elemento *Factory* para serialização (serve basicamente para instanciar as classes que representam os conceitos da ontologia). (STOCKLÖW, 2018c).

Segue um trecho de código que exemplifica algumas das características mencionadas acima:

**Figura 10 - Classe *Activity***

Activity.java

```
1  package org.universAAL.ontology.activity;
2
3  import javax.xml.datatype.XMLGregorianCalendar;
5
6
7  /**
8   * Ontological representation of Activity in the activity ontology.
9   * Methods included in this class are the mandatory ones for representing an
10  * ontological concept in Java classes for the universAAL platform. In addition
11  * getters and setters for properties are included.
12  *
13  * @author
14  * @author Generated by the OntologyUML2Java transformation of AAL Studio
15  */
16 public class Activity extends ManagedIndividual {
17     public static final String MY_URI = ActivityOntology.NAMESPACE
18             + "Activity";
19     public static final String PROP_NAME = ActivityOntology.NAMESPACE
20             + "name";
21     public static final String PROP_STOP = ActivityOntology.NAMESPACE
22             + "stop";
23     public static final String PROP_DURATION = ActivityOntology.NAMESPACE
24             + "duration";
25     public static final String PROP_START = ActivityOntology.NAMESPACE
26             + "start";
27
28
29     public Activity () {
30         super();
31     }
32
33     public Activity (String uri) {
34         super(uri);
35     }
36
37     public String getStart() {
38         return (String)getProperty(PROP_START);
39     }
```

Fonte: Elaborada pelo autor (2018).

No código acima pode-se ver um *resource* (a classe *Activity*), o seu URI (*MY_URI*), e algumas propriedades *PROP_NAME*, *PROP_STOP*, *PROP_DURATION* e *PROP_START*, além do método auxiliar *getStart()* para adquirir o valor da propriedade *PROP_START*.

E na imagem abaixo, um exemplo de como usar o método genérico *setProperty(String propURI, Object value)* para alterar a propriedade de um objeto, sendo que os objetos que estão sendo passados como segundo parâmetro foram criados anteriormente e são do tipo *String* (*start* e *stop*) e *Long* (*duration*), foram omitidos por questões de simplificação da imagem:

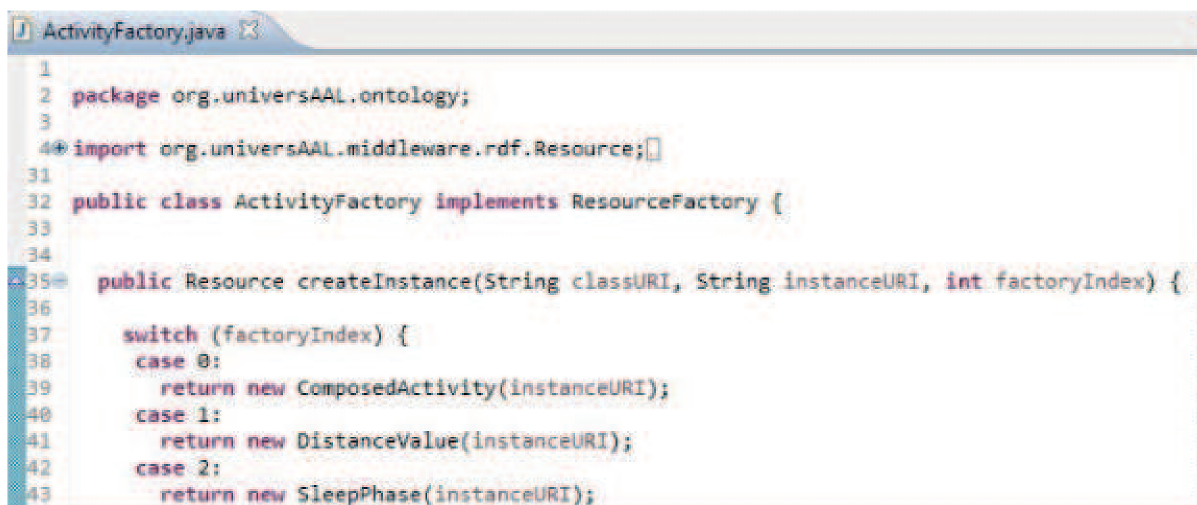
**Figura 11 - Método *setProperty(String propURI, Object value)***

```
67        // Setting up properties of my EatingActivity
68        eatingAct.setProperty(Activity.PROP_START, start);
69        eatingAct.setProperty(Activity.PROP_STOP, stop);
70        eatingAct.setProperty(Activity.PROP_DURATION, duration);
```

Fonte: Elaborada pelo autor (2018).

Um exemplo de *Factory* para a ontologia que está sendo utilizada de exemplo seria da forma que segue, onde cada um dos *returns* está instanciando uma classe de ontologia diferente:

**Figura 12 - Classe *ActivityFactory***




Fonte: Elaborada pelo autor (2018).

E um exemplo de *enumeration* é dado no seguinte trecho de código: *new Enumeration(new Integer[] {new Integer(0), new Integer(100)})*, onde os valores adotados por uma *property* qualquer seriam 0 ou 100.

Abaixo segue um exemplo de como é o *Activator* de uma ontologia, ou seja, a classe que contém os métodos de *start* e *stop* que, respectivamente, registra ou cancela o registro de uma ontologia na instância da *universAAL* que estiver sendo executada.

**Figura 13 - Classe *ActivityActivator***

```
ActivityActivator.java
1
2  package org.universAAL.ontology;
3
4  import org.universAAL.middleware.container.ModuleContext;
8
9
10 public class ActivityActivator implements ModuleActivator {
11
12   ActivityOntology _activityOntology = new ActivityOntology();
13
14
15   public void start(ModuleContext mc) throws Exception {
16     OntologyManagement.getInstance().register(mc, _activityOntology);
17   }
18
19   public void stop(ModuleContext mc) throws Exception {
20     OntologyManagement.getInstance().unregister(mc, _activityOntology);
21   }
22 }
```

Fonte: Elaborada pelo autor (2018).

E por fim, tem-se que exemplificar de que forma aquela classe mais geral, mencionada um pouco acima, define o tipo de uma *property* de alguma classe de ontologia.

**Figura 14 - Definindo tipo de uma propriedade utilizando o URI de uma classe**

```
184    oci_DistanceCalculator.setResourceComment("");
185    oci_DistanceCalculator.setResourceLabel("DistanceCalculator");
186    oci_DistanceCalculator.addSuperClass(ActivityDevice.MY_URI);
187    oci_DistanceCalculator.addObjectProperty(DistanceCalculator.PROP_HAS_MEASURED_DISTANCE_VALUE).setFunctional();
188    oci_DistanceCalculator.addRestriction(MergedRestriction
189      .getAllValuesRestrictionWithCardinality(DistanceCalculator.PROP_HAS_MEASURED_DISTANCE_VALUE,
190          DistanceValue.MY_URI, 1, 1));
```

Fonte: Elaborada pelo autor (2018).

Neste trecho de código, há a definição da *property* *PROP_HAS_MEASURED_DISTANCE_VALUE* de uma classe chamada *DistanceCalculator* (na linha 187). E a adição de uma restrição a ela que acontece no método *addRestriction* (linha 188) chamado por um objeto do tipo *OntClassInfoSetup* (interface para criação de novas classes OWL), que está basicamente dizendo que os valores dessa propriedade serão do tipo *DistanceValue* (outra classe da coleção de classes em questão). Bem, neste caso vê-se

que o tipo da propriedade é um definido como uma das classes da ontologia, mas se o tipo de dado for nativo do *Java*, a classe *TypeMapper* é usada para a obtenção do URI de um desses tipos como nas linhas 134 e 138 da figura abaixo (estão exemplificados os tipos *String* e *XMLGregorianCalendar*):

**Figura 15 - Definindo tipo de uma propriedade utilizando a classe *TypeMapper***

```
128    oci_SleepPhase.setResourceComment("");
129    oci_SleepPhase.setResourceLabel("SleepPhase");
130    oci_SleepPhase.addSuperClass(ManagedIndividual.MY_URI);
131    oci_SleepPhase.addDatatypeProperty(SleepPhase.PROP_REMARKS).setFunctional();
132    oci_SleepPhase.addRestriction(MergedRestriction
133      .getAllValuesRestrictionWithCardinality(SleepPhase.PROP_REMARKS,
134          TypeMapper.getDatatypeURI(String.class), 1, 1));
135    oci_SleepPhase.addDatatypeProperty(SleepPhase.PROP_DURATION).setFunctional();
136    oci_SleepPhase.addRestriction(MergedRestriction
137      .getAllValuesRestrictionWithCardinality(SleepPhase.PROP_DURATION,
138          TypeMapper.getDatatypeURI(XMLGregorianCalendar.class), 1, 1));
```

Fonte: Elaborada pelo autor (2018).

A próxima seção descreve a metodologia usada para o desenvolvimento da ontologia resultante do esforço empreendido no uso dessa ferramenta e das necessidades já citadas na introdução do trabalho. E a seguinte à da metodologia, mostra o processo de desenvolvimento em si exemplificando melhor alguns benefícios e funcionalidades que foram ditas nesta seção e até aprofundando melhor algumas que não foram citadas aqui por serem mais “técnicas”.

# 3 METODOLOGIA

Pode-se dizer que houve de início uma pesquisa de caráter exploratório para o desenvolvimento deste trabalho, pois como será descrito nas próximas subseções houveram estudos acerca da plataforma e suas nuances antes do desenvolvimento em si. Depois disso o trabalho passou a ter um caráter muito forte de desenvolvimento, com o avanço na ferramenta e com a criação de versões do modelo da ontologia, assim como do uso desta como modelo de dados de um sistema e também da criação de um módulo de *Subscriber* na plataforma *universAAL* para compor este sistema (que será descrito na seção 5).

As subseções a seguir descrevem como se deu o desenvolvimento deste trabalho.

## 3.1 ESTUDO DA PLATAFORMA UNIVERSAAL

O desenvolvimento que é descrito neste trabalho se deu, de certa forma, como resultado de um estudo que permeou os Estágios Curriculares Obrigatórios I e II da grade do curso, estes que foram realizados no NUTES. Então, é plausível pôr o estudo da plataforma como a primeira atividade que foi realizada, mesmo que no início não houvesse nenhuma pretensão quanto ao desenvolvimento de uma ontologia.

Pelo menos nos dois primeiros meses (março e abril de 2018), o foco daquele que era o primeiro estágio foi bem direcionado a tentar entender como se dava o funcionamento da *universAAL*, isto através de leituras em documentações, criação de exemplos iniciais nos barramentos de contexto e de serviço, entre outras atividades, sempre com muita discussão acerca dos problemas encontrados com quem já havia utilizado um pouco a plataforma. E, como as ontologias são neste caso a forma de se representar o conhecimento seja ele relacionado a dispositivos de saúde, a perfis de usuário, ou a medidas, de certa forma já estava havendo um "treinamento" sobre como utilizar este formalismo em sistemas *universaais*.

Nos meses seguintes houve uma prática maior relacionada a ontologias, com o uso da ferramenta de desenvolvimento *AAL Studio* como descrito na próxima subseção.

## 3.2 INSTALAÇÃO E EXPLORAÇÃO DO AAL STUDIO

De maio em diante houve um foco maior no estudo das ontologias em si, como se dava o seu funcionamento interno, a representação de conceitos (um usuário (*User*), por exemplo, na ontologia *Profile*), o seu registro na instância em execução da plataforma, entre

outras características. Além disso, houve uma exploração no *AAL Studio*, a ferramenta que foi utilizada para modelagem e geração do código de *Activity*, sendo a instalação do mesmo o primeiro desafio enfrentado.

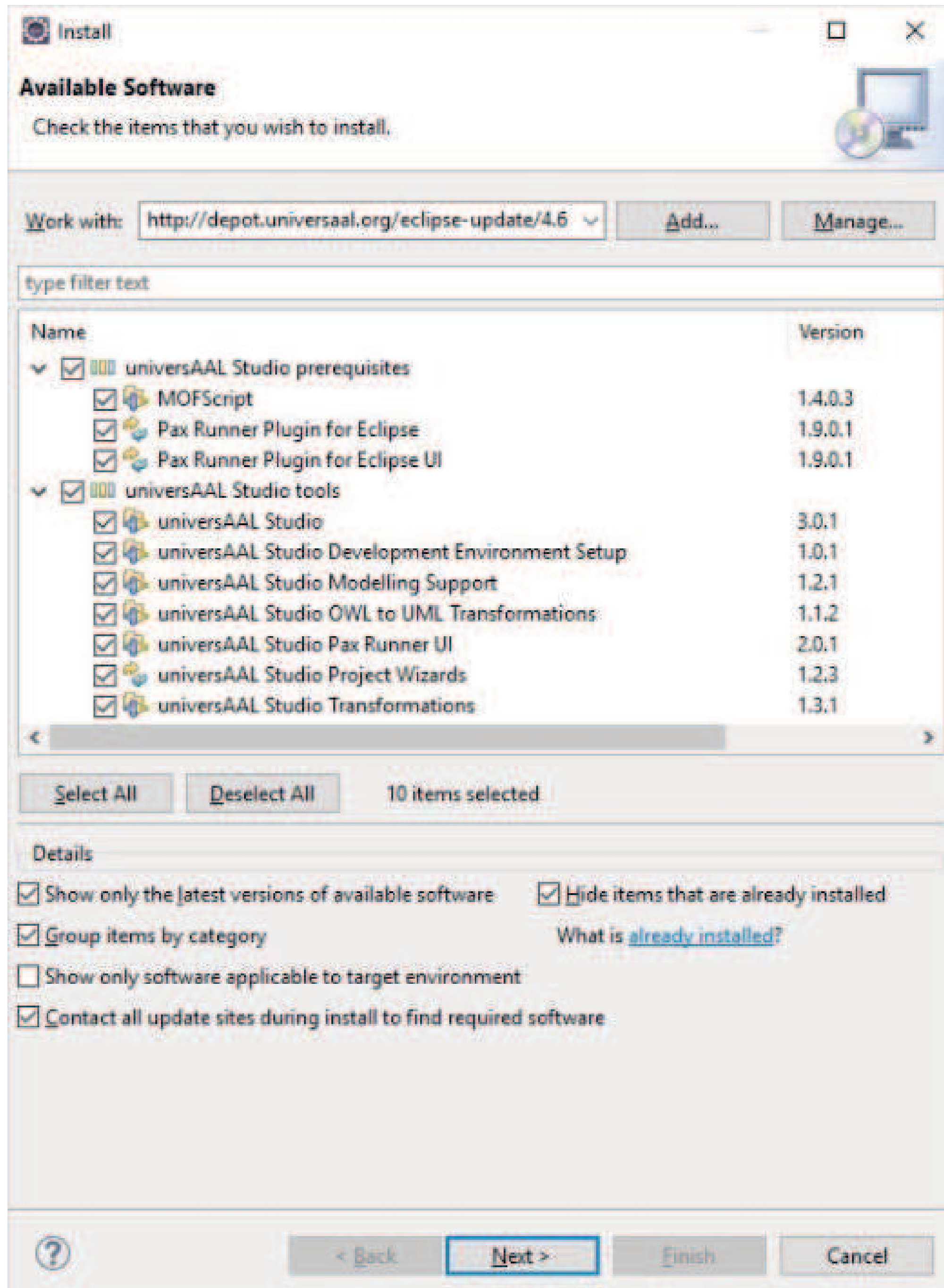
Os principais pré-requisitos para essa instalação são o JDK do *Java*, de preferência uma versão mais nova (7 acima, para abranger tipos como o *XMLGregorianCalendar*), e uma versão do *Eclipse* com o *Maven* instalado, de preferência a versão *Modeling Tools* que já fornece alguns *plug-ins* necessários pré-configurados. Ele pode ser instalado no *Eclipse* no menu *Help* > *Install New Software*… preenchendo o campo de nome "*Work with:*" com o link http://depot.universaal.org/eclipse-update/4.6 para a versão *Neon* do *Eclipse* ou http://depot.universaal.org/eclipse-update/3.7 para a versão *Indigo*, e depois selecionando todas as opções como na figura 16, e clicando em *Next* > *Finish*.

Esta etapa foi desafiadora, pois a instalação do pacote na versão *neon* do *Eclipse* não deu certo e quando iria haver a tentativa de outras formas de instalação (pensou-se até em desenvolvimento linha por linha de código), uma versão da IDE foi disponibilizada neste link[2] pela própria equipe da *universAAL*, versão essa que já continha o *plug-in* instalado. Isto foi de grande ajuda, pois como já dito neste texto, essa ferramenta consegue propor um grande aumento na agilidade do desenvolvimento, além de minimizar a ocorrência de erros.

Após essa instalação, foi feita uma investigação na documentação da *universAAL* relacionada à ferramenta com a finalidade de entender melhor como funcionava esse processo e se ele realmente funcionava, isto com a criação de alguns projetos de teste apenas para explorar melhor as funcionalidades. Projetos esses que acabaram por revelar alguns problemas e limitações do *plug-in* que serão melhor descritos na seção sobre o desenvolvimento da ontologia de atividades.

[2] Link para *Eclipse* com *AAL Studio* instalado: https://owncloud.lst.tfo.upm.es/index.php/s/6Lcb9yTIutlIFJq

**Figura 16 – Instalando o *AAL Studio***




Fonte: Elaborada pelo autor (2018).

## 3.3 DESENVOLVIMENTO DA ONTOLOGIA

A ontologia em si foi modelada e transformada em código utilizando a versão do *Eclipse* já citada na subseção anterior, após isso foi compilada e integrada à instância de execução da *universAAL* da máquina utilizada no estágio. Mas, como o processo é um tanto quanto longo e entra como o cerne principal deste trabalho, a seção que segue a esta foi

reservada apenas para esta descrição, onde serão detalhados todos os passos seguidos, os problemas encontrados e as soluções/*workarounds* utilizados no processo de utilização do *AAL Studio* para essa construção.

## 3.4 USO DE ACTIVITY COMO MODELO DE DADOS DE UM SISTEMA

Como validação do módulo criado, leia-se a ontologia *Activity*, esta última foi utilizada como o modelo de dados em um protótipo que fez parte de uma PoC (*Proof of Concept* – prova de conceito) para o projeto OCARIoT[3]. Esta PoC foi realizada com o objetivo de demonstrar os benefícios do uso de uma arquitetura de microserviços, um conceito recente que traz basicamente a ideia da divisão de um sistema em vários módulos que têm responsabilidades individuais, que podem ser desenvolvidos com tecnologias diferentes, e que ao final do desenvolvimento devem ser todos integrados, tornando o desenvolvimento independente de linguagens e/ou tecnologias. E também a importância de alguns elementos como a *universAAL*, plataforma IoT que foi adotada no projeto.

Segue abaixo uma lista com os requisitos detalhados de forma bem básica que a ontologia deve atender como módulo de representação do conhecimento do sistema:

- Suportar atividades físicas em geral;
- As atividades que serão recebidas no sistema podem ser do tipo *WalkActivity* que é uma atividade física que contém um atributo relacionado à quantidade de passos, outro relacionado à distância percorrida e informações sobre a localização de início e término dela;
- Se a atividade não for do tipo *WalkActivity*, ela deve tratada como uma atividade física geral;
- Os dados referentes a quaisquer atividades físicas sempre têm atributos relacionados à sua intensidade e à energia consumida.
- Dados referentes a frequência cardíaca e pressão sanguínea devem ser modelados, mas de início não serão utilizados no sistema.

O desenvolvimento da ontologia será mostrado na seção que segue, já seu modelo, a sua visualização na plataforma, e sua utilização neste sistema serão mostrados na seção “Resultados Obtidos”.

---

[3] ocariot.com

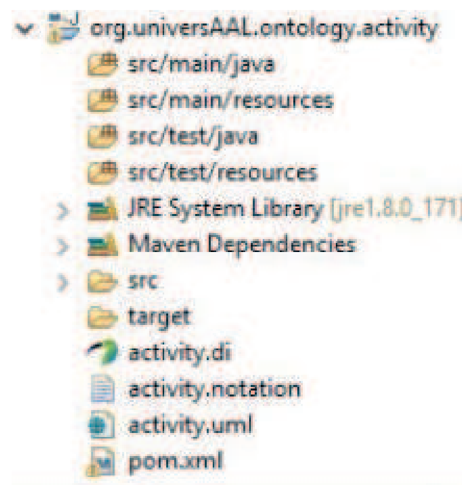
# 4 DESENVOLVIMENTO DA ONTOLOGIA *ACTIVITY*

Nesta seção serão descritas as peculiaridades do processo de modelagem e geração do código de uma ontologia utilizando o *universAAL Studio*, ferramenta de modelagem de ontologias da *universAAL* que é melhor detalhada no Apêndice A, assim como as dificuldades enfrentadas. O texto segue de forma que as ilustrações mostram os passos iniciais de cada etapa do processo de construção da ontologia, e estas etapas são descritas textualmente sempre com a ontologia em questão sendo referenciada como exemplificação.

## 4.1 MODELAGEM DA ONTOLOGIA

O processo de modelagem da ontologia começou após ser finalizado o assistente de criação de projetos do *AAL Studio*, onde o seguinte projeto passou a ser mostrado no *Package Explorer* do *Eclipse*:

**Figura 17 – Visualização do projeto no *Package Explorer***




Fonte: Elaborada pelo autor (2018)

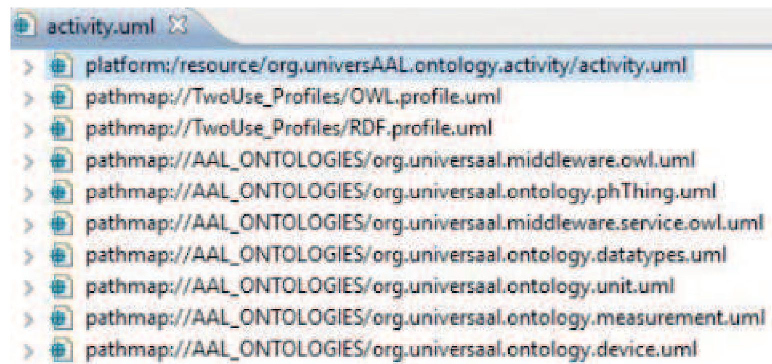
Pode-se ver que já existiam alguns arquivos iniciais, sendo eles "*activity.di*", "*activity.notation*", "*activity.uml*" e "*pom.xml*", arquivos esses que teriam de ser criados manualmente e inseridos no projeto caso o assistente do *AAL Studio* não fosse utilizado. Os próximos parágrafos tratarão de mostrar as características dos arquivos "*.uml*" e "*.di*", necessários para o entendimento do processo de modelagem.

Começando pelo arquivo de terminação "*.uml*" que permite a visualização dos elementos da ontologia que está sendo modelada e das ontologias superiores que já são

importadas na criação do projeto, esta visualização acontece da forma vista na figura 18. E o "*.di*" é o arquivo no qual o modelo é, de fato, editado. Quando inicializado, a perspectiva do *plug-in Papyrus* é aberta mostrando os elementos vistos na figura 19 (se o arquivo estiver sendo aberto pela primeira vez).

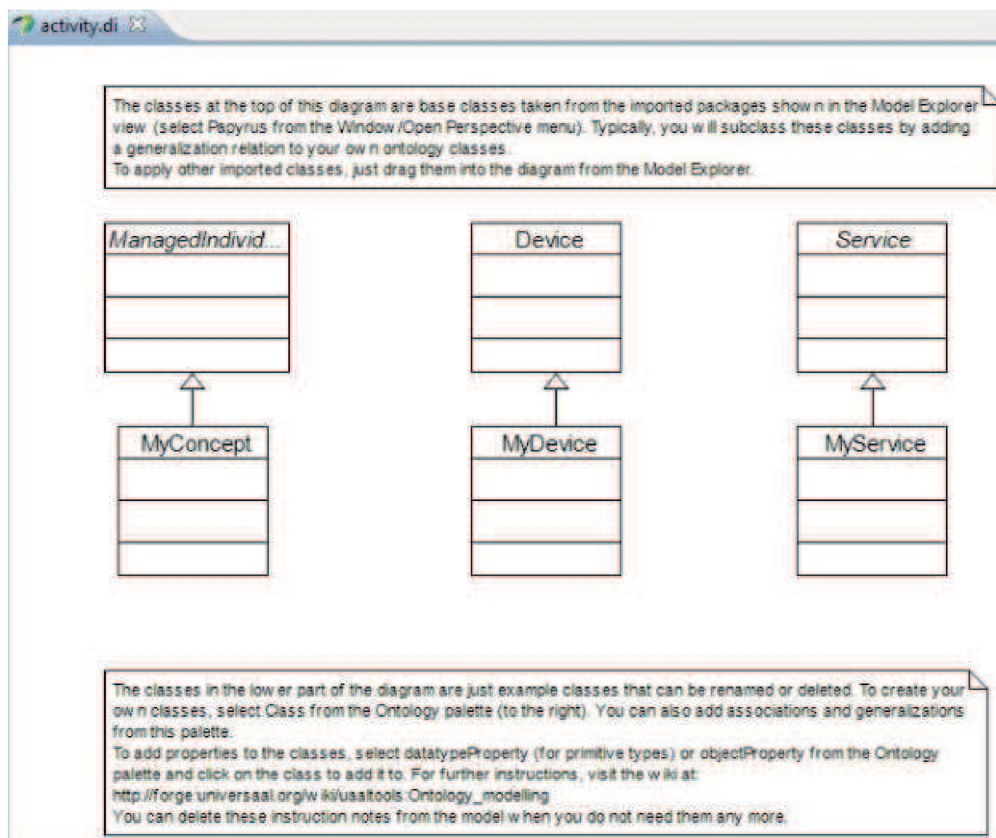
É importante que o arquivo de terminação "*.di*" esteja fechado para que os outros dois possam ser abertos.

**Figura 18 – Arquivo "*.uml*" da ontologia**




Fonte: Elaborada pelo autor (2018)

**Figura 19 – Arquivo "*.di*" da ontologia**

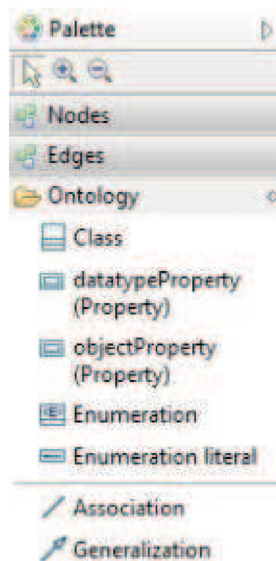



Fonte: Elaborada pelo autor (2018)

Agora falando sobre os elementos iniciais do modelo, tem-se *ManagedIndividual* que funciona no *middleware* como a classe mãe de toda e qualquer classe de ontologia, além de *Device* e *Service*, que modelam exatamente a ideia abstraída de seus nomes. E também três classes iniciais, *MyConcept*, *MyDevice* e *MyService* que já vêm inseridas, herdando dos tipos superiores que são importados de início no modelo. Aqui vale mencionar que com exceção do tipo *ManagedIndividual*, os outros dois não precisam necessariamente estar lá para que uma ontologia seja criada e funcione normalmente, tudo vai depender da base de conhecimentos que se deseja modelar.

Foram usados apenas os elementos da aba *Ontology* da paleta de elementos do *Papyrus*, com pena de não ser possível a geração do código ao fim da modelagem caso o contrário ocorresse, afinal esse *plug-in* é usado para modelagens UML em geral e tudo que está em *Nodes* e *Edges* serve exatamente para isso. A paleta pode ser visualizada abaixo.

**Figura 20 – Paleta de elementos do *Papyrus***




Fonte: Elaborada pelo autor (2018)

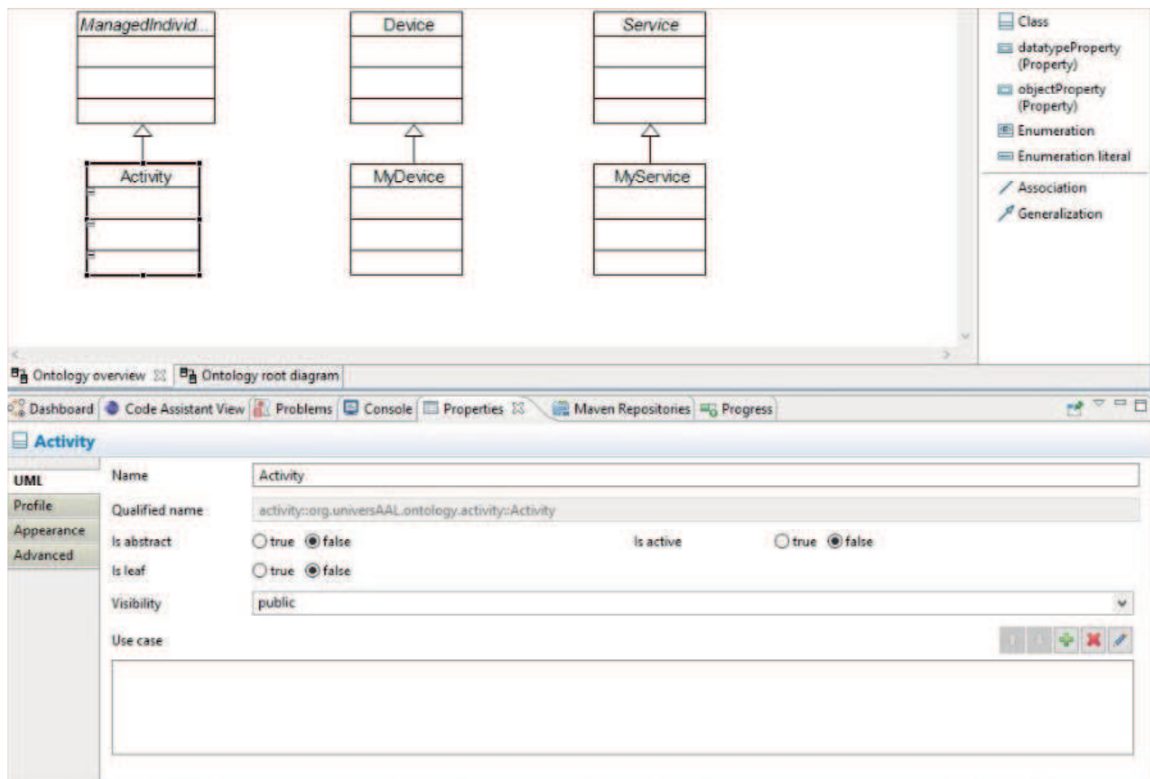
Sobre esses elementos, tem-se o de nome *Class* utilizado para inserir novas classes de ontologia no seu modelo; *datatypeProperty* para inserir uma propriedade de tipos primitivos UML (*String*, *Integer*, *Float*...) em um *Class*, assim como *objectProperty* para inserir uma propriedade que tenha como tipo outras classes de ontologia; *Enumeration* para representar enumerações e *Enumeration literal* para adicionar um literal em uma enumeração,

*Association* para associações entre as classes de ontologia e *Generalization* para o mecanismo de herança.

Para colocar um *datatypeProperty* ou um *objectProperty* em um elemento *Class*, ou um *Enumeration literal* em uma E*numeration*, basta clicar nele na paleta e depois sobre o devido elemento.

Para editar as propriedades de qualquer elemento, basta clicar nele e abrir a aba *Properties* do *Eclipse*. Na figura abaixo podem ser vistas as possibilidades de edição de um elemento do tipo *Class*, sendo que o elemento que inicialmente era chamado de *MyConcept* e herdava de *ManagedIndividual* teve seu nome alterado para *Activity*, a classe mais geral da ontologia que foi modelada.

**Figura 21 – Editando um *class***




Fonte: Elaborada pelo autor (2018)

Para as outras classes da ontologia como, por exemplo, a que modela a ideia de atividades fisicas chamada *PhysicalExerciseActivity*, o procedimento sempre foi o de apenas clicar em *Class* na paleta *Ontology* do *Papyrus* e depois clicar em algum local do modelo.

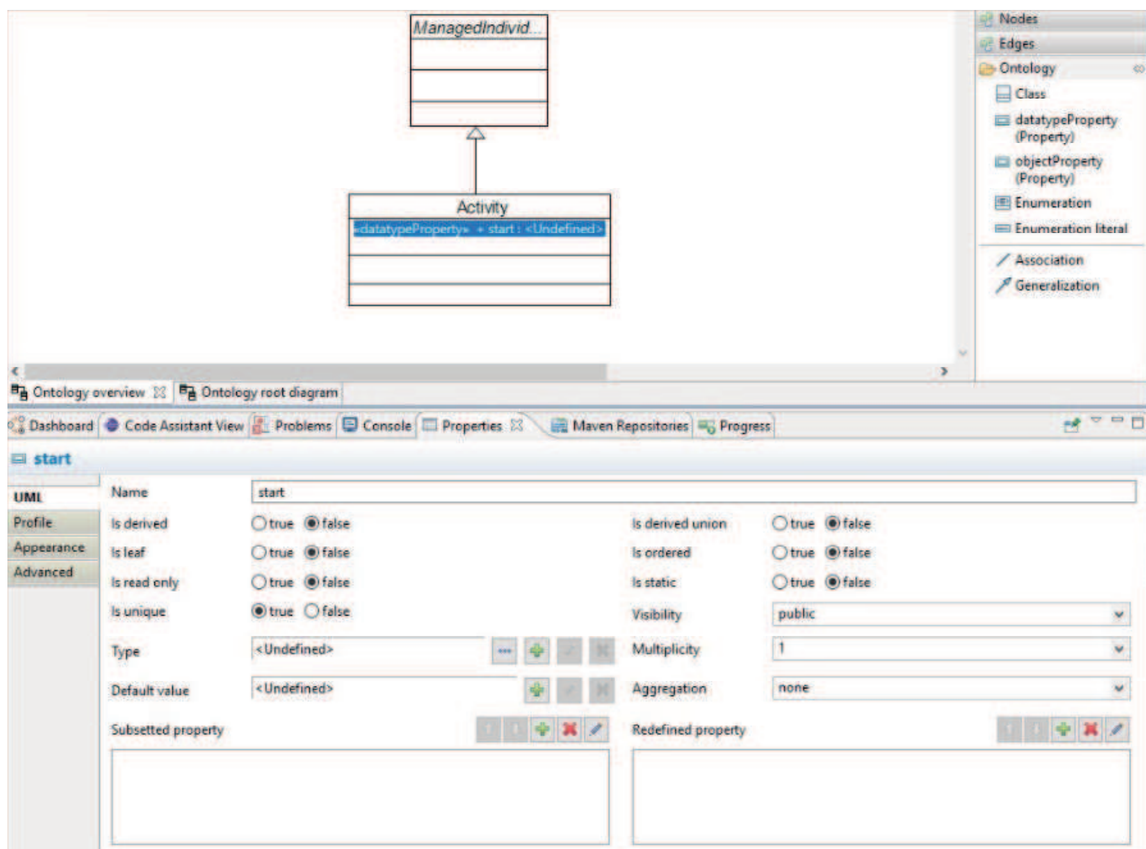
Muito foi falado sobre cada classe de ontologia ter propriedades associadas, sendo que neste modelo ficou decidido que uma *Activity* teria quatro propriedades, uma referente ao

nome, e as outras três relacionadas ao seu tempo, sendo uma para o registro da data de início (*start*), outra para a data de término (*stop*) e a última para a duração da atividade (*duration*).

Sobre essas propriedades ficou decidido que a duração seria do tipo *Long* e as datas seriam do tipo *String*, estas últimas foram do tipo *XMLGregorianCalendar* em algumas versões anteriores da modelagem, mas acabou por ficarem desta forma para facilitar a aquisição e manipulação desses dados por qualquer serviço que precisasse fazer isto. Como será visto na seção de resultados, existem vários microserviços na arquitetura que trabalham com esses dados.

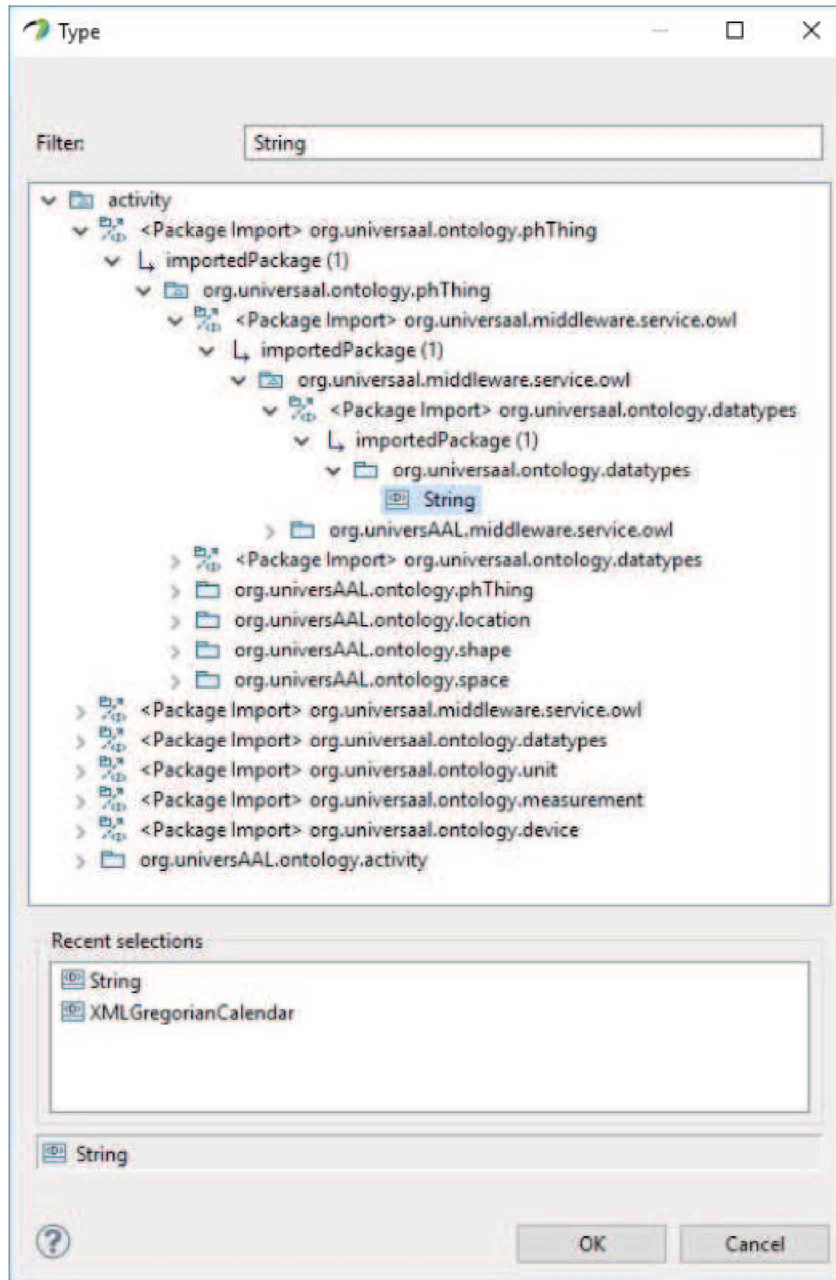
Como eram propriedades de tipos nativos (*Long* e *String*) o elemento *datatypeProperty* da paleta foi utilizado e inserido na classe *Activity*, mas se fosse necessário algum tipo que não fosse nativo, bastaria inserir o elemento de nome *objectProperty*, sendo que nestes dois casos é necessário adicionar o tipo da propriedade (figura 23), como ocorreu em toda modelagem, clicando no botão "..." ao lado do campo "*Type*" (visto na figura abaixo que mostra a tela de edição de uma propriedade, neste caso "*start*").

**Figura 22 – Editando uma *property***




Fonte: Elaborada pelo autor (2018)

**Figura 23 – Definindo tipo de uma propriedade**




Fonte: Elaborada pelo autor (2018)

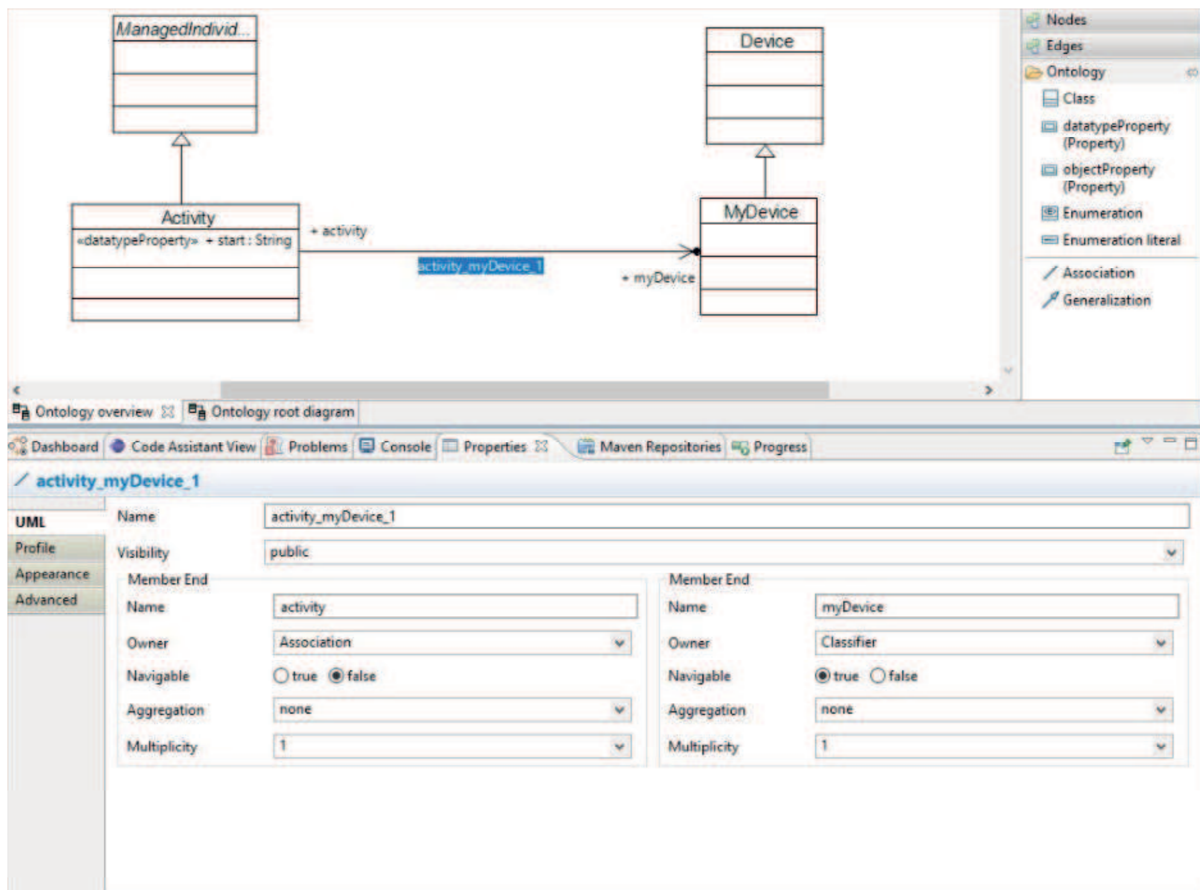
Aqui vale mencionar que não só classes de ontologias já existentes na *universAAL* poderiam ser usadas como tipo, mas também aquelas que foram recém-criadas no próprio modelo ou alguma outra que pudesse ser importada do mesmo *workspace* do projeto da ontologia modelada (passo que será descrito mais à frente).

Além destas possibilidades já mencionadas, ainda há os elementos de generalização e de associação, que foram usados no modelo que será visto na próxima seção, como por exemplo, quando todas as classes que modelam uma atividade herdam direta ou indiretamente

de *Activity*, ou no caso da classe chamada *ActivityTrackingProfile* que modela a ideia de um rastreador de atividades, e tem uma associação para com o elemento *Activity* do tipo um para muitos.

Na figura abaixo é vista a aba de edição de uma associação, onde podem ser editadas características como navegabilidade, multiplicidade e agregação.

**Figura 24 – Editando uma associação**



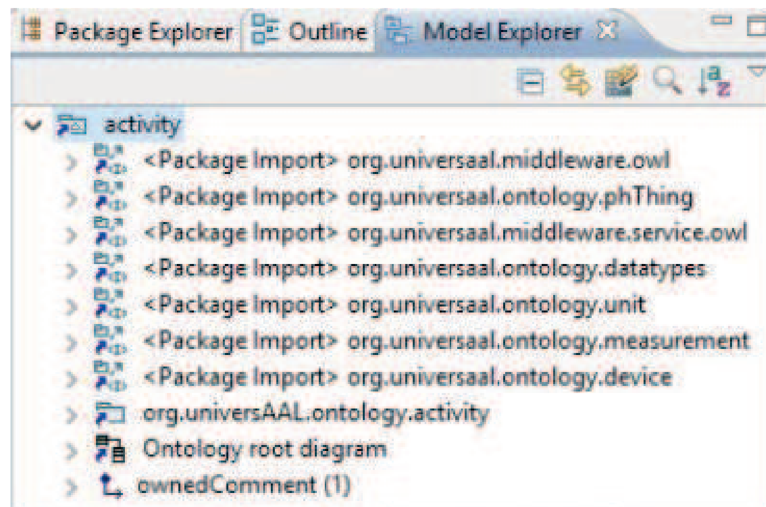

Fonte: Elaborada pelo autor (2018)

A ideia básica é que ao clicar em qualquer elemento no modelo e estando com a aba *Properties* aberta, as características desse elemento podem ser editadas e foi basicamente esse o processo de modelagem da ontologia, um processo de montagem das classes e das ligações entre elas, assim como a definição de propriedades, heranças e associações, faltando falar sobre um processo bastante importante já mencionado um pouco acima que é o de importar uma ontologia para ser usada no processo.

Durante a modelagem ficou claro que algumas atividades como a de exercício fisico precisavam de propriedades relacionadas a medidas de saúde, e como o repositório[4] de ontologias oficiais da *universAAL* já havia sido bastante explorado, sabia-se que existia uma ontologia pronta para isso chamada de *HealthMeasurement*. Foi aí que entrou a necessidade de usá-la, seja com alguns de seus elementos como tipo para propriedades de algumas destas classes de atividade ou até como classe pai de algumas novas medidas que foram criadas como *EnergyConsumption* (que modela a ideia de calorias consumidas em um exercício).

Nem todas as ontologias desse repositório podem ser importadas no modelo pois é necessário que exista seu arquivo "*.uml*" para isto. As ontologias que já estão importadas no modelo podem ser vistas abrindo a aba *Model Explorer* (figura 25) do *Eclipse*.

**Figura 25 – *Model Explorer***

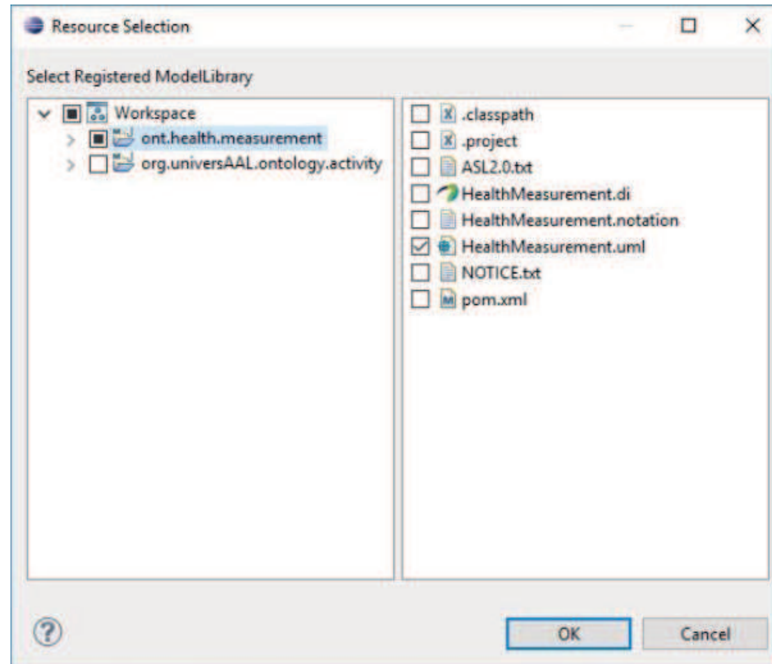



Fonte: Elaborada pelo autor (2018)

A ontologia de *HealthMeasurement* foi baixada do repositório, colocada no mesmo *workspace* do projeto de *Activity* e importada no modelo através da opção *Import* > *Import Package From Workspace* que aparece ao clicar com o botão direito sobre "*activity*". As figuras que seguem mostram o processo.

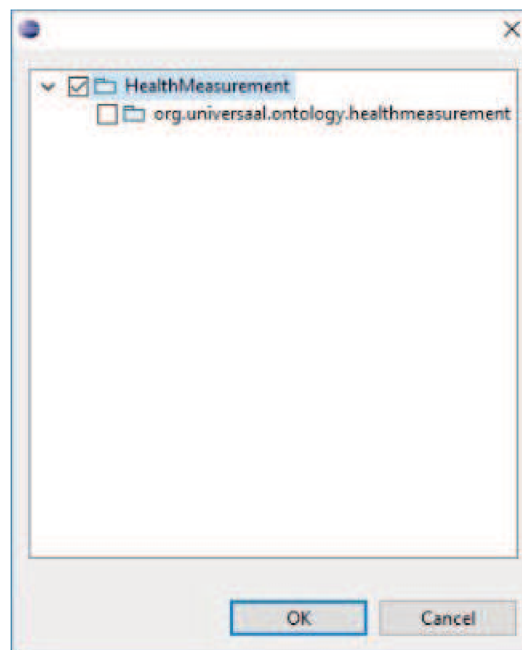
[4] Link para repositório de ontologias oficiais da *universAAL* - https://github.com/universAAL/ontology/

**Figura 26 – Importando ontologia (Tela 1)**




Fonte: Elaborada pelo autor (2018)

**Figura 27 – Importando ontologia (Tela 2)**

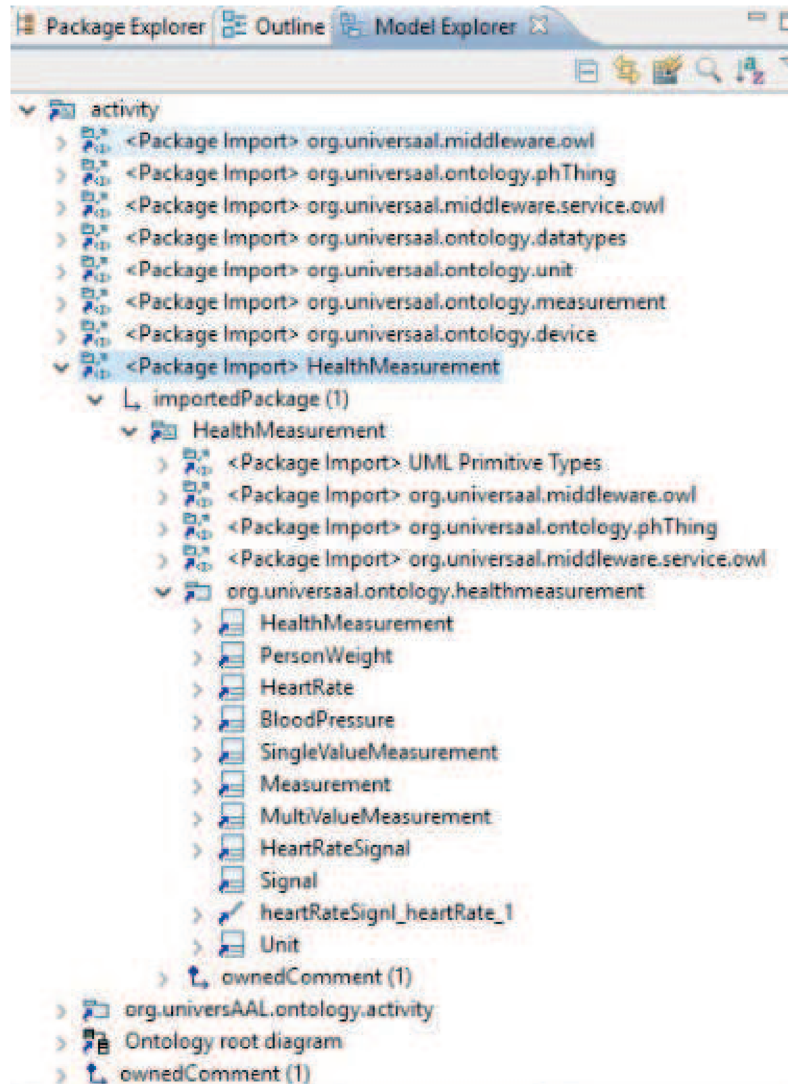



Fonte: Elaborada pelo autor (2018)

Com isso a visualização do *Model Explorer* ficou um pouco diferente como visto na figura 28. Com este passo realizado, qualquer elemento presente em *HealthMeasurement* passou a estar disponível para uso no modelo da ontologia *Activity*, onde bastava clicar sobre um elemento como *PersonWeight* no *Model Explorer* e depois clicar sobre o modelo da

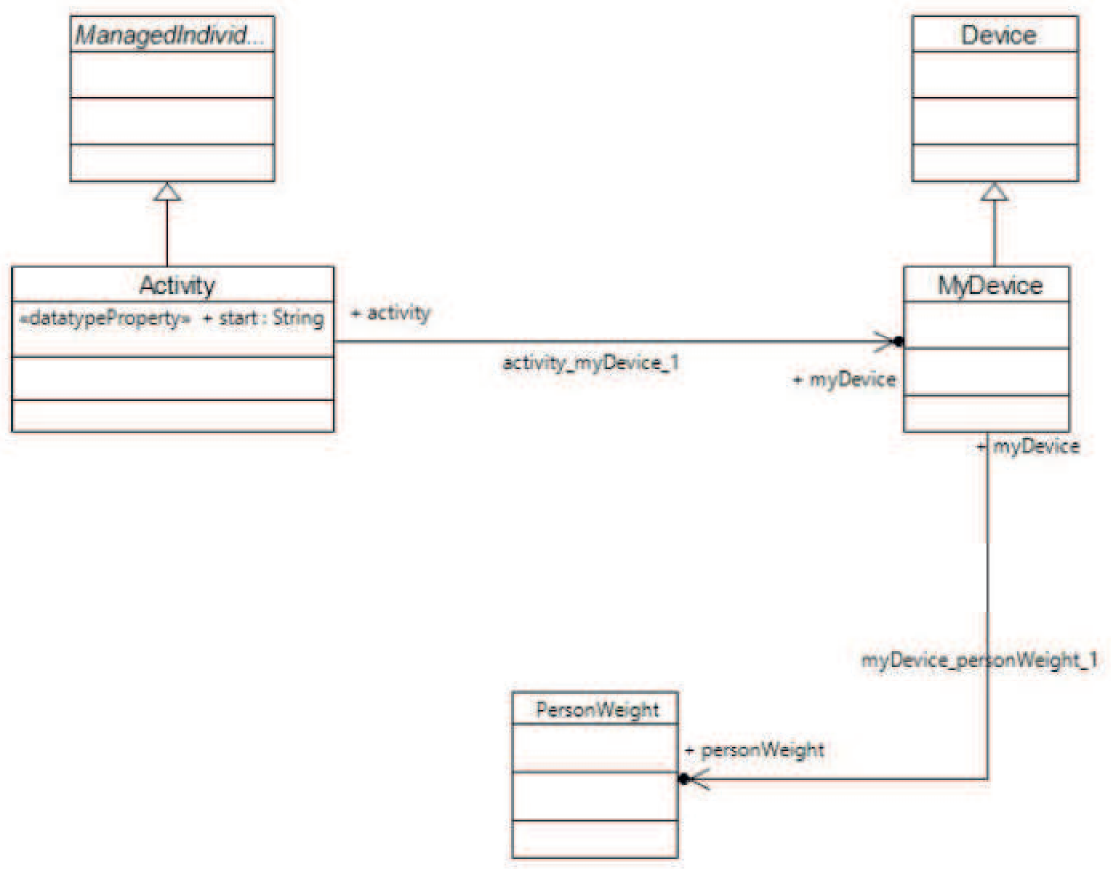
mesma forma que quando era inserida uma nova classe, gerando um resultado como o visto na figura 29 (onde também é inserida uma associação entre *PersonWeight* e *MyDevice*).

**Figura 28 – Estrutura depois de uma nova ontologia ter sido importada**




Fonte: Elaborada pelo autor (2018)

**Figura 29 – Utilizando elemento *PersonWeight* da ontologia importada**




Fonte: Elaborada pelo autor (2018)

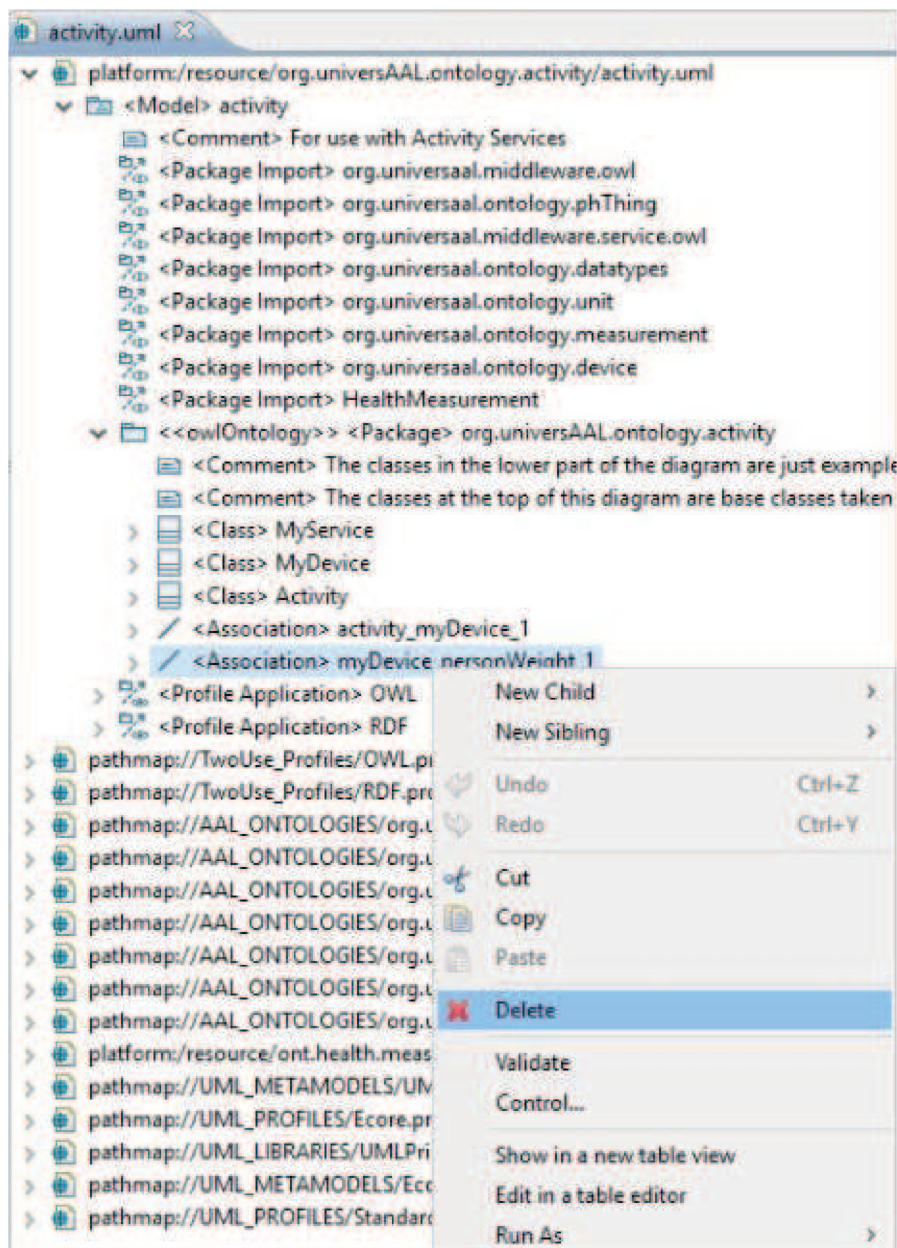
É bastante importante mencionar que o *AAL Studio* possui algumas limitações tanto na modelagem como na geração do código que precisam ser contornados. Isto é dito por experiência de alguém que chegou até mesmo a ter de remontar todo o diagrama por conta de alguns destes empecilhos. Mas sempre que algo do tipo ocorria, isto era documentado, e este detalhe do processo foi muito importante no sentido de agilizar e deixar mais seguro o ato de atualizar a ontologia, "seguro" porque após esses erros serem descobertos e documentados, não ocorreu mais retrabalho. Todos esses detalhes são descritos na próxima subseção pois são parte deste processo de desenvolvimento de ontologias com o *AAL Studio*.

## 4.2 PROBLEMAS E LIMITAÇÕES DA FERRAMENTA

Os problemas e limitações que serão descritos abaixo são uma ameaça à integridade de um dos principais objetivos do *AAL Studio*, o de agilizar o processo.

O primeiro problema encontrado foi o de excluir elementos do modelo como o *PersonWeight* do exemplo da figura 29, que mesmo após excluído aparecia no código gerado, ou até mesmo impedia a transformação UML > *Java* (por algum motivo que quebrasse as regras de uma ontologia *universaal*). Assim ele e/ou qualquer associação relacionada a ele teve de ser excluído também no arquivo "*.uml*". Neste caso, bastou deletar a caixa que representava *PersonWeight* e a associação criada selecionando-os e apertando a tecla *delete* do teclado no arquivo "*.di*" e só depois apagá-los do arquivo UML (figura 30).

**Figura 30 – Operação de deletar associação feita com *PersonWeight* no UML**




Fonte: Elaborada pelo autor (2018)

Com uma ressalva de que a visualização da figura 30 só foi possível porque o arquivo "*.di*" não estava aberto. E fazendo também a observação que qualquer exclusão desse tipo deve ser feita primeiramente no arquivo "*.di*", com pena de não ser mais possível editar este último, graças a uma exceção lançada pelo *Eclipse*, sendo que em ocorrências como essa, o modelo teve de ser gerado do zero.

Outra limitação encontrada foi na geração do código onde se o código já tivesse sido gerado e por qualquer motivo o modelo fosse alterado, sendo necessária uma nova geração, classes de mesmo nome eram substituídas completamente, apagando toda e qualquer alteração manual que tivesse sido feita. A recomendação é que se as alterações forem pequenas, que sejam feitas manualmente, mesmo se tendo em mente que essa prática pode facilmente incluir erros na ontologia. Até porque há a necessidade de algumas alterações manuais de qualquer forma, então se for para contornar essa limitação, basta que seja feita com cuidado que tudo ocorrerá normalmente.

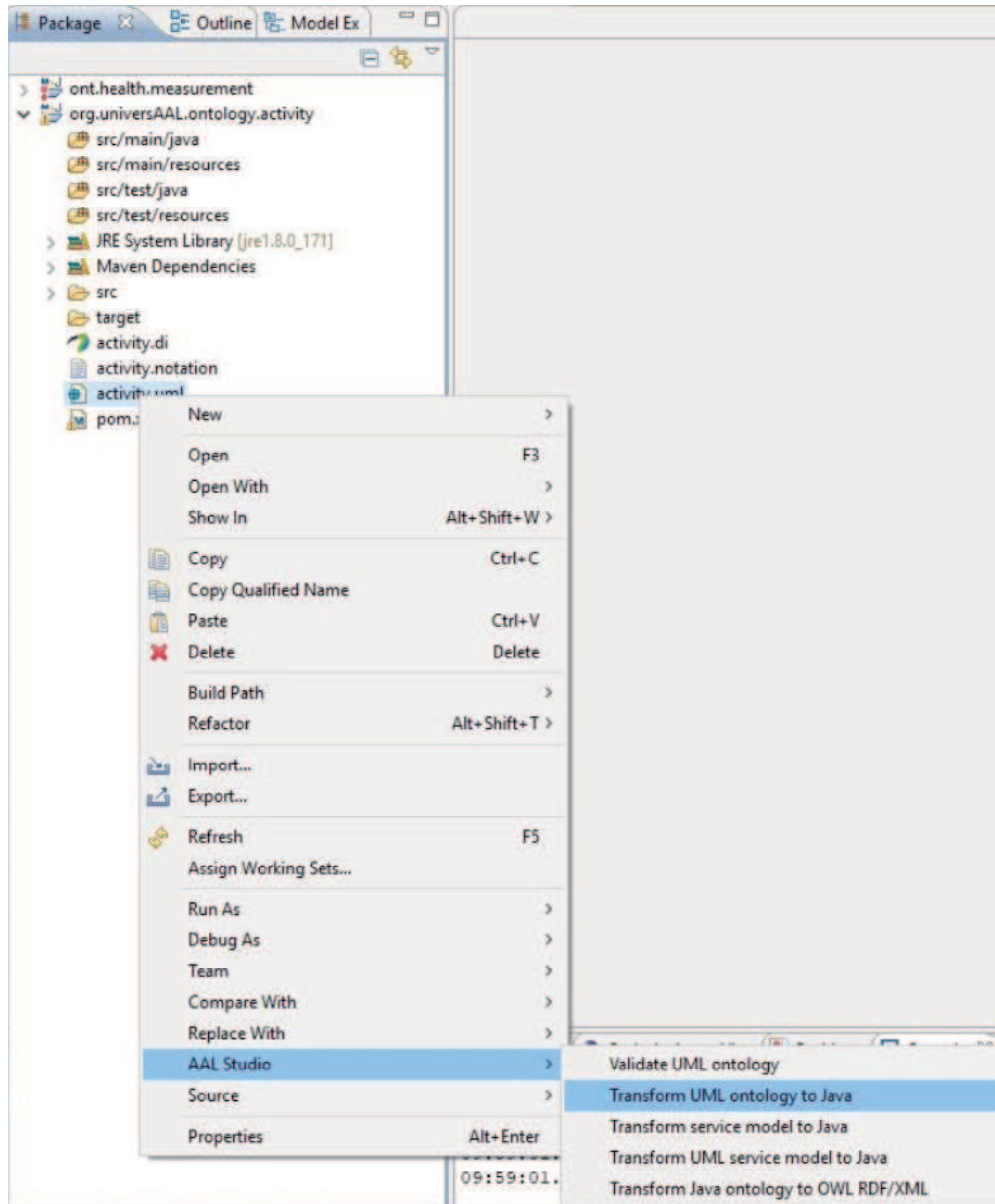
Houve ainda um erro relacionado a usar uma ontologia importada como tipo para uma propriedade, mas que teve de ser lidado após a transformação UML > *Java*, diretamente no código, visto que onde havia qualquer uso dessas classes, aparecia apenas um espaço vazio que devia ser preenchido manualmente. Mais detalhes serão dados a seguir.

## 4.3 GERAÇÃO DO CÓDIGO ATRAVÉS DO MODELO

Após a modelagem, o código foi gerado através do arquivo "*.uml*", processo visto na figura 31. A versão mais nova disponível (3_0_0) do *middleware* foi selecionada, e após isso o código gerado apareceu no caminho de saída *src/main/java*. Isto das vezes que o processo funcionou, ou seja, depois de os problemas terem sido contornados.

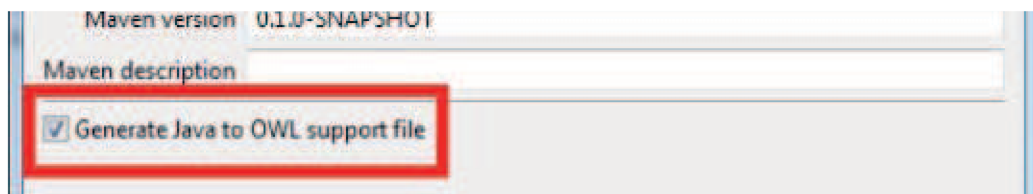
As classes geradas sempre são divididas em três pacotes, onde um deles contém o *Activator* e o *Factory* da ontologia, bem semelhantes aos já vistos na seção 2.2.3. Outro irá conter uma classe *Creator*, que se executada transformará o código da ontologia em um arquivo RDF/OWL, isso se a opção "*Generate Java to OWL support file*" estiver marcada como na figura 32 durante a criação do projeto. E, um último pacote com todas as classes do modelo, incluindo a "principal", que define todas as restrições de todas as propriedades da ontologia, sendo crucial para o funcionamento ideal do modelo recém transformado.

Figura 31 – Transformando o modelo em código




Fonte: Elaborada pelo autor (2018)

Figura 32 – Opção que possibilita a transformação *Java* > *OWL*




Fonte: Stocklöw (2018i)

Após o código ter sido gerado, foram necessárias algumas alterações manuais como alterar restrições, mudar valores das propriedades para ajustá-las de acordo com o que era desejado, e também importar/usar os elementos daquelas ontologias que foram importadas.

Na figura que segue pode ser visto o erro que era causado pelo uso de elementos da ontologia *HealthMeasurement*.

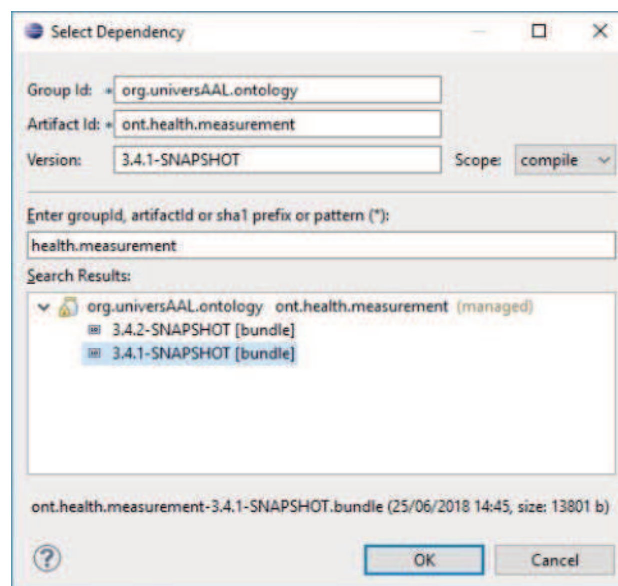
**Figura 33 – Erro causado pela ausência da classe *HeartRate***

```
57    public  getHeartRate() {
58      return ()getProperty(PROP_HEART_RATE);
59    }
60
61    public void setHeartRate( newPropValue) {
62      if (newPropValue != null)
63        changeProperty(PROP_HEART_RATE, newPropValue);
64    }
```

Fonte: Elaborada pelo autor (2018)

Neste caso foi necessário apenas substituir os espaços vazios de retorno de método, *casting*, e tipo de parâmetro por *HeartRate* e importá-la na classe para resolver esse problema. Sendo que como essa ontologia foi importada manualmente, ela não estava configurada como uma dependência no arquivo "*pom.xml*" gerando assim outro erro causado pela impossibilidade de se importar a classe em questão, sendo este outro passo a se realizar (figura 34).

**Figura 34 – Adicionando *HealthMeasurement* como dependência do projeto**




Fonte: Elaborada pelo autor (2018)

Após isso ter sido feito e o arquivo ter sido salvo, o uso de *HeartRate* passou a funcionar. Mas outro passo foi necessário para que o código representasse o que realmente estava modelado. Lá na classe *ActivityOntology*, que é aquela que define todas as restrições sobre todas as propriedades de todas as classes da ontologia criada, foi preciso adicionar o tipo que foi importado, nas propriedades que estivessem usando-o. As próximas imagens ilustram, respectivamente, a situação após a geração do código e após a restrição da propriedade *PROP_HEART_RATE* ter sido ajustada, onde basicamente ocorreu uma pequena alteração no método que põe a restrição sobre ela (*getCardinalityRestriction*).

**Figura 35 – Código após a transformação**

```
227    oci_PhysicalExerciseActivity.setResourceComment("");
228    oci_PhysicalExerciseActivity.setResourceLabel("PhysicalExerciseActivity");
229    oci_PhysicalExerciseActivity.addSuperClass(LocationBasedActivity.MY_URI);
230    oci_PhysicalExerciseActivity.addObjectProperty(PhysicalExerciseActivity.PROP_ENERGY_CONSUMPTION).setFunctional();
231    oci_PhysicalExerciseActivity.addRestriction(MergedRestriction
232      .getAllValuesRestrictionWithCardinality(PhysicalExerciseActivity.PROP_ENERGY_CONSUMPTION,
233         EnergyConsumption.MY_URI, 1, 1));
234    oci_PhysicalExerciseActivity.addObjectProperty(PhysicalExerciseActivity.PROP_HEART_RATE).setFunctional();
235    oci_PhysicalExerciseActivity.addRestriction(MergedRestriction.getCardinalityRestriction(PhysicalExerciseActivity.PROP_HEART_RATE, 1, 1));
236    oci_PhysicalExerciseActivity.addObjectProperty(PhysicalExerciseActivity.PROP_BLOOD_PRESSURE).setFunctional();
237    oci_PhysicalExerciseActivity.addRestriction(MergedRestriction.getCardinalityRestriction(PhysicalExerciseActivity.PROP_BLOOD_PRESSURE, 1, 1));
```

Fonte: Elaborada pelo autor (2018)

**Figura 36 – Código após a restrição ter sido ajustada**

```
227    oci_PhysicalExerciseActivity.setResourceComment("");
228    oci_PhysicalExerciseActivity.setResourceLabel("PhysicalExerciseActivity");
229    oci_PhysicalExerciseActivity.addSuperClass(LocationBasedActivity.MY_URI);
230    oci_PhysicalExerciseActivity.addObjectProperty(PhysicalExerciseActivity.PROP_ENERGY_CONSUMPTION).setFunctional();
231    oci_PhysicalExerciseActivity.addRestriction(MergedRestriction
232      .getAllValuesRestrictionWithCardinality(PhysicalExerciseActivity.PROP_ENERGY_CONSUMPTION,
233         EnergyConsumption.MY_URI, 1, 1));
234    oci_PhysicalExerciseActivity.addObjectProperty(PhysicalExerciseActivity.PROP_HEART_RATE).setFunctional();
235    oci_PhysicalExerciseActivity.addRestriction(MergedRestriction
236        .getAllValuesRestrictionWithCardinality(PhysicalExerciseActivity.PROP_HEART_RATE,
237           HeartRate.MY_URI, 1, 1));
238    oci_PhysicalExerciseActivity.addObjectProperty(PhysicalExerciseActivity.PROP_BLOOD_PRESSURE).setFunctional();
239    oci_PhysicalExerciseActivity.addRestriction(MergedRestriction
240        .getAllValuesRestrictionWithCardinality(PhysicalExerciseActivity.PROP_BLOOD_PRESSURE,
241           BloodPressure.MY_URI, 1, 1));
```

Fonte: Elaborada pelo autor (2018)

Percebe-se que o método em destaque da figura 35 não define nenhum tipo para a propriedade *PROP_ HEART_RATE*, o que de acordo com o modelo criado está incorreto, ela deveria ser do tipo *HeartRate*. A única coisa que está sendo definida, na verdade, é a cardinalidade dessa propriedade, então outro método (*getAllValuesRestrictionWithCardinality*) é usado e o tipo é devidamente definido (linhas 236 e 237 do código da figura 36).

Claro que estas exemplificações desta seção e das anteriores foram feitas com apenas alguns elementos do modelo. Durante o processo de desenvolvimento várias foram as

alterações manuais que foram necessárias no código até porque a ontologia começou a tomar proporções maiores com as atualizações que ocorriam. Mas, por fim, após estes processos de modelar, gerar o código, e corrigi-lo, bastou compilar o projeto seguindo o menu *Run As* > *Maven install* e gerando assim o arquivo *jar* da ontologia, onde este arquivo representava a sua portabilidade, bastando, no caso da plataforma *universAAL* sendo executada com o ambiente de execução *Apache Karaf*, copiá-lo e colá-lo no caminho "*distro.karaf/target/assembly/deploy*" para implantá-lo.

Um último detalhe é que a classe de teste *ArtifactIT* criada automaticamente pelo *AAL Studio* causava um erro de compilação, e já que neste escopo ela não era necessária, suas funcionalidades foram comentadas a fim de encerrar o processo normalmente.

Com isto, todo o processo de modelagem e geração de código da ontologia *Activity* foi detalhado. Após todos estes passos, a ontologia passou a estar disponível para ser utilizada em aplicações *universaais* como a que foi criada para o sistema que será descrito nos resultados.

*Activity* foi armazenada em um repositório *maven*, passo que será mostrado na próxima subseção, e assim passou a estar disponível para ser adicionada como dependência de qualquer aplicação, fazendo a ressalva que até o momento da escrita deste trabalho a aplicação *Android* da arquitetura está utilizando diretamente o arquivo jar para mapear atividades, pois alguns problemas de compilação estavam sendo encontrados ao tentar baixar a dependência diretamente do repositório.

## 4.4 INSERÇÃO DA ONTOLOGIA EM UM REPOSITÓRIO MAVEN

Um repositório *maven* foi criado utilizando o site de repositórios privados *myMavenRepo*[5] e o código da figura 37 foi adicionado ao arquivo "*pom.xml*" do projeto da ontologia para que ela fosse enviada ao repositório que tem seu link de escrita inserido na *tag* <*url*>. Aqui se faz necessário mencionar que isto só ocorre se o comando *maven deploy* for utilizado.

É possível ver na figura 38 o diretório onde está a ontologia no repositório criado, contendo todos os arquivos necessários para que quando uma aplicação a use através de uma dependência *maven*, tudo possa ocorrer normalmente.

---

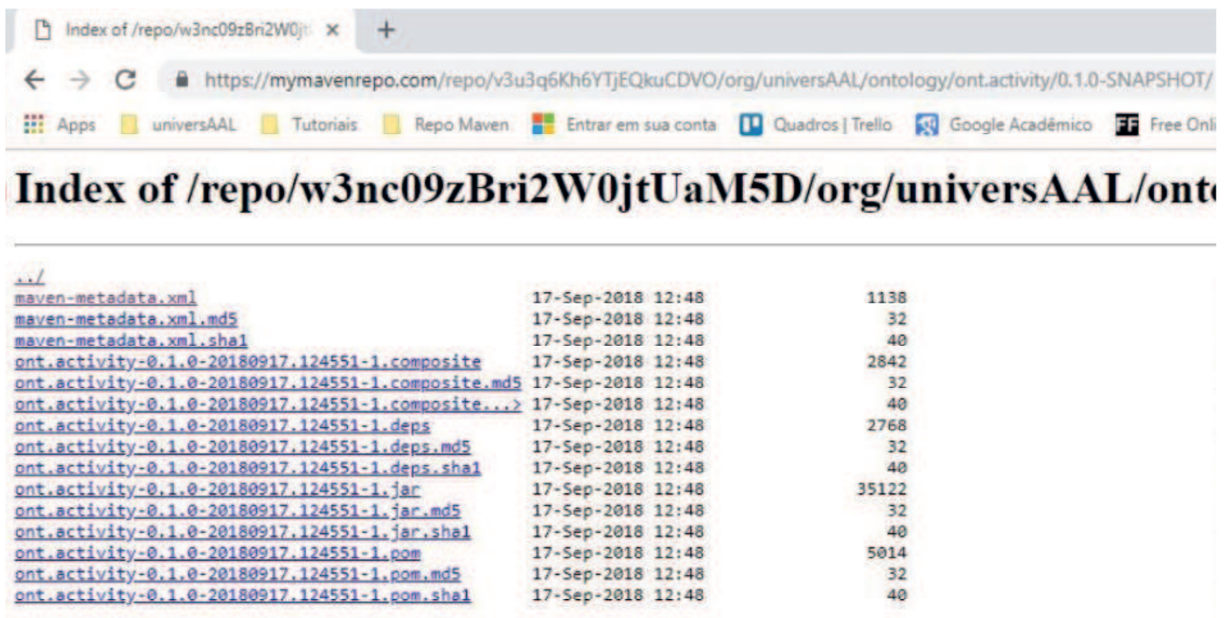
[5] Link myMavenRepo - https://mymavenrepo.com

**Figura 37 – Código de inserção da ontologia no repositório**

```
116⊖    <distributionManagement>
117⊖        <repository>
118             <id>nutes-ocariot</id>
119             <url>https://mymavenrepo.com/repo/ZSW49ih0rmexufEu2n3D/</url>
120         </repository>
121⊖        <snapshotRepository>
122             <id>nutes-ocariot</id>
123             <url>https://mymavenrepo.com/repo/ZSW49ih0rmexufEu2n3D/</url>
124         </snapshotRepository>
125     </distributionManagement>
```

Fonte: Elaborada pelo autor (2018)

**Figura 38 – Pasta da ontologia no repositório criado**




Fonte: Elaborada pelo autor (2018)

A próxima seção descreve melhor os elementos da ontologia obtida como resultado deste trabalho, assim como sua utilização na PoC já aqui mencionada.

# 5 RESULTADOS OBTIDOS

Como resultado desse trabalho tem-se uma primeira versão de uma ontologia de atividades que pode ser utilizada como modelo de dados de qualquer aplicação que seja capaz de utilizar dependências *maven*. E que como já dito na seção de metodologias, foi utilizada em uma primeira aplicação, que nada mais é do que um sistema de aquisição de dados que envolve várias linguagens e tecnologias em uma arquitetura de microserviços, com a finalidade de validar este tipo de estrutura e também alguns elementos como a plataforma *universAAL*. Tudo isto realizado como tarefa do já citado projeto OCARIoT.

É importante dizer que muito esforço foi empreendido neste desenvolvimento para que pudesse ser entregue uma modelagem que abarcasse os requisitos definidos na seção 3.4, na medida do possível, visto que havia um prazo bem definido para o término da prova de conceito.

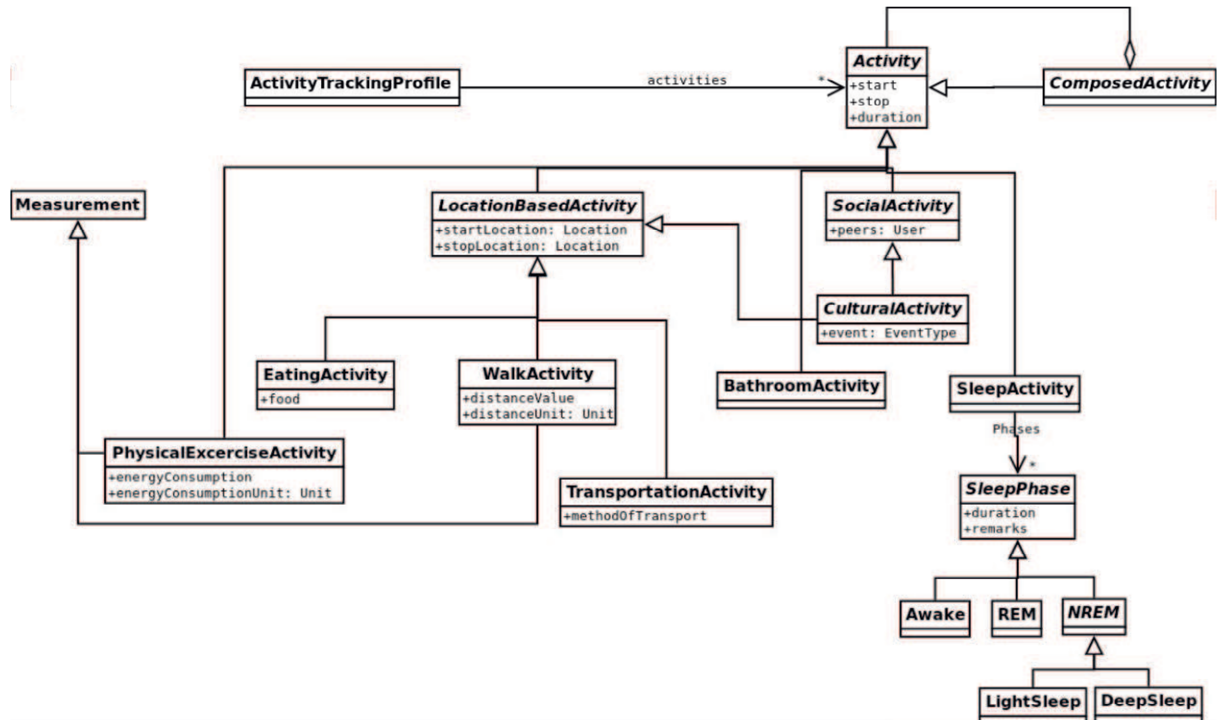
Na subseção abaixo são dados mais detalhes concretos acerca da ontologia.

## 5.1 DESCRIÇÃO DA ONTOLOGIA ACTIVITY

Esta ontologia modela um conjunto de atividades que variam de bem específicas como a atividade de andar até algumas mais gerais como uma "atividade baseada em local". Também traz a ideia de que há um rastreador de atividades, contendo nenhuma ou várias delas (1...*) e permite ainda a criação de atividades compostas, através do padrão de projeto *Composite*, sendo que uma atividade pode ser composta por 1 ou várias outras atividades (1...*).

Pode-se ver na figura abaixo o modelo inicial proposto pela própria equipe da *universAAL* em um *workshop* do OCARIoT realizado em março de 2018 em Fortaleza.

**Figura 39 – Modelo inicial de *Activity***




Fonte: Equipe da *universAAL* (2018)

Após algumas alterações durante o processo de desenvolvimento chegou-se a um modelo com diferenças significativas na parte esquerda do modelo, parte mais relacionada às atividades físicas, que se tratava do escopo necessário nos requisitos levantados para o modelo de dados na PoC. A ramificação do meio que parte para *SocialActivity*, e a da direita relacionada a atividades de dormir não receberam tantas alterações, sendo algo a ser realizado como trabalho futuro. Apesar disto, o modelo tomou proporções grandes o suficiente para não ser legível se inserido neste documento, sendo necessário acessá-lo neste link[6] (que também contém as outras versões da ontologia).

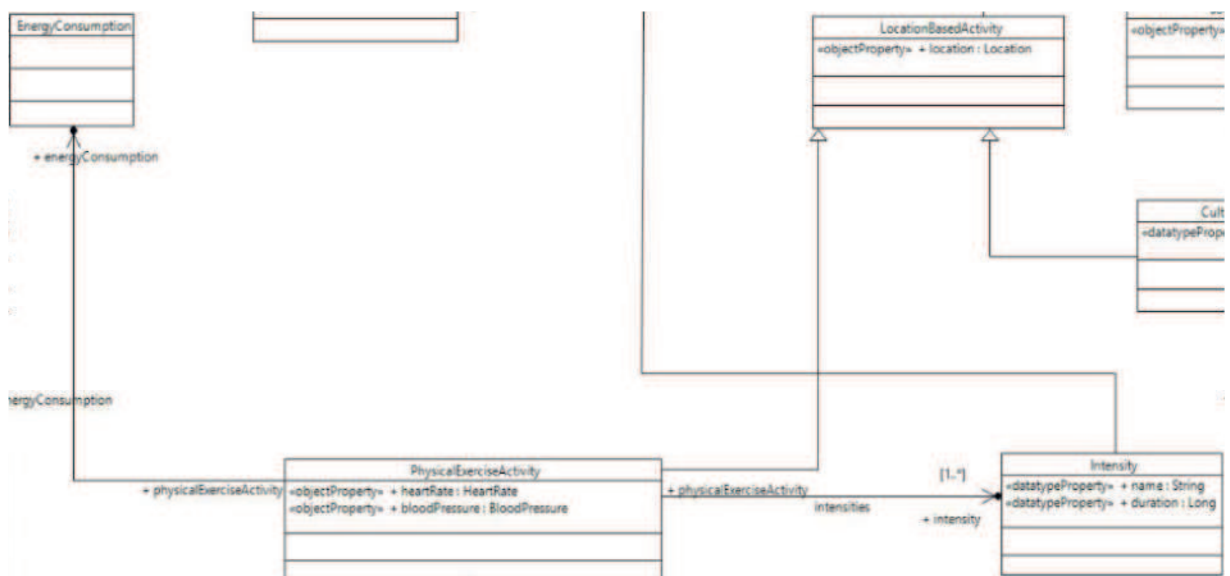
Como já especificado na seção 3.4 que o escopo mais bem trabalhado foi o de atividades físicas, a descrição focará esses pontos. É importante mencionar também que o modelo que será descrito a seguir foi a terceira versão, sendo que atualmente existe uma versão mais nova do modelo com algumas alterações. Foi escolhida esta versão porque foi a utilizada como modelo de dados da arquitetura da PoC.

Esta versão do modelo foi estruturada da seguinte forma: há a classe mais geral, *Activity*, onde qualquer outra classe de atividade que existir herdará dela, direta ou indiretamente, sendo seus atributos relacionados ao tempo (*start*, *stop* e *duration*) e ao seu

[6] Pasta da ontologia - https://drive.google.com/open?id=15XU_1HxYUCVTPiJ2df2evK_Op9EuHxuo

nome. Algumas atividades herdam diretamente de *Activity* como por exemplo, *LocationBasedActivity*, que modela atividades que têm uma localização associada e que tem um atributo do tipo *Location*. *PhysicalExerciseActivity*, que herda desta última e está associada a medidas de saúde (*HeartRate* e *BloodPressure*), a uma nova medida criada para este modelo chamada de *EnergyConsumption* que modela a ideia de calorias consumidas e a uma ou mais intensidades (*Intensity*) que têm um nome (*sedentary*, por exemplo) e uma duração (*Long*). Estas relações podem ser vistas na figura abaixo.

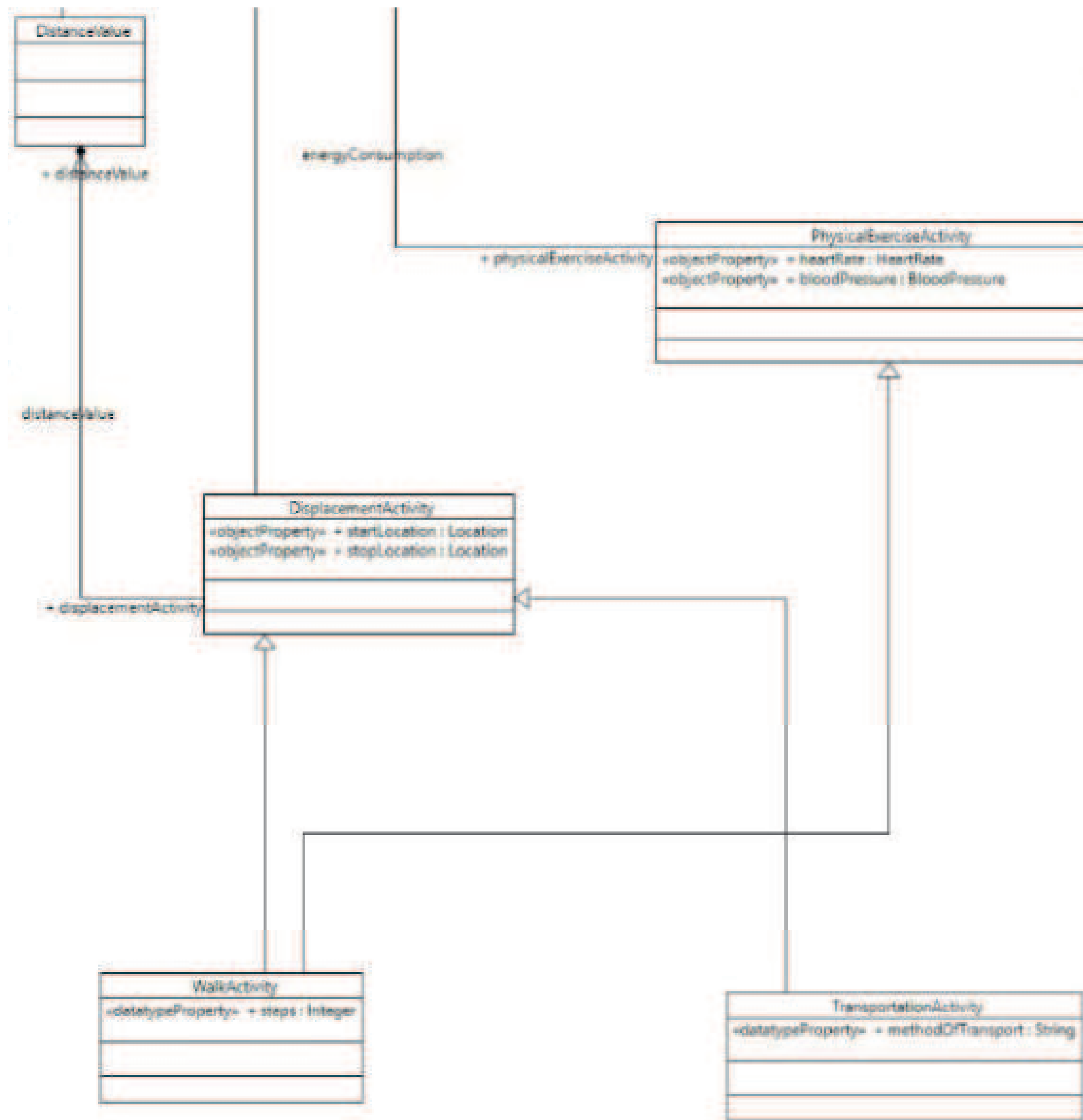
**Figura 40 – Elemento *PhysicalExerciseActivity***




Fonte: Elaborada pelo autor (2018)

Há também uma classe chamada *WalkActivity* (na parte inferior esquerda da figura 41) para a atividade de andar que herda de *PhysicalExerciseActivity* e *DisplacementActivity* (atividades que têm localização de início e término e uma medida de distância), e tem um atributo *steps* (passos dados). Neste ponto pode surgir um questionamento "se a *universAAL* utiliza Java e em Java não há herança múltipla, como que uma classe (neste caso, *WalkActivity*) pode herdar de outras duas?" Bem, a resposta é que o próprio *middleware* fornece essa possibilidade internamente, através de um método de assinatura '*addSuperClass(String superClassURI)*' que é utilizado na classe que define todas as características de todos os conceitos da ontologia, e que já é inserido durante a transformação do modelo em código.

**Figura 41 – Elemento *WalkActivity***

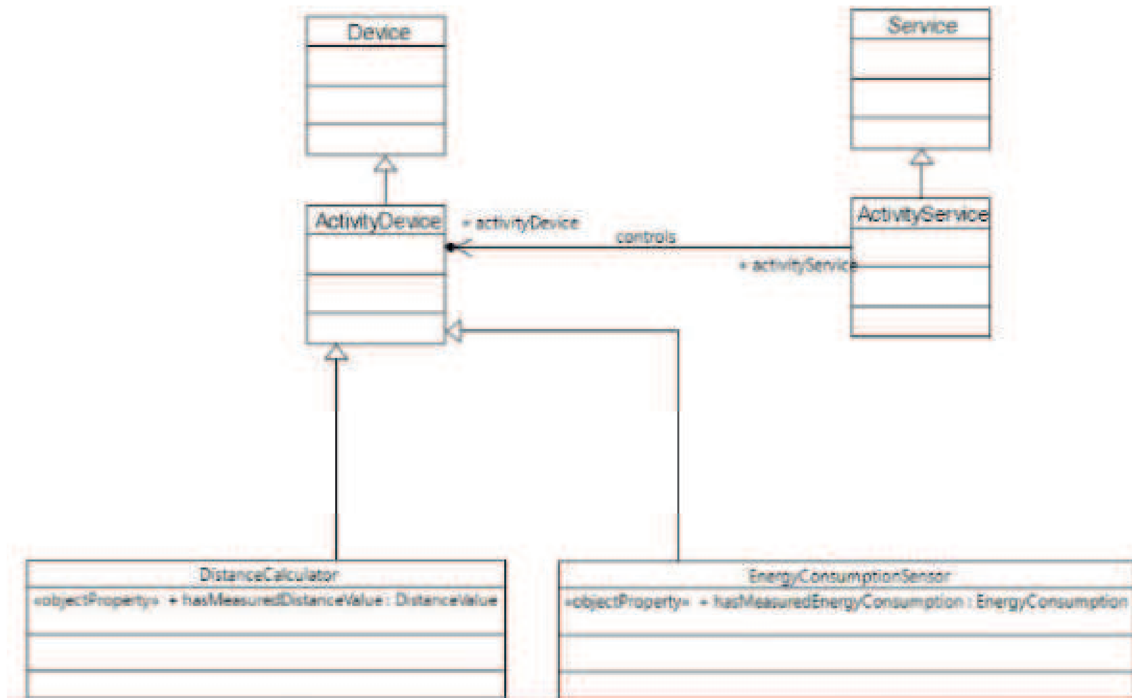



Fonte: Elaborada pelo autor (2018)

Foram criados dois *Devices* novos, *DistanceCalculator* e *EnergyConsumtpionSensor*, associados às novas medidas *DistanceValue* e *EnergyConsumption*, sendo que essas duas últimas herdam de *Measurement*, única classe da coleção oficial de ontologias da *universAAL* que foi utilizada como elemento do modelo (classes como *Location*, *HeartRate* e *BloodPressure* também fazer parte das ontologias já disponibilizadas pela *universAAL*, mas foram utilizadas aqui apenas como tipo de algumas propriedades). Estes novos *Devices* são controlados por uma classe de serviço como visto na figura 42, mas ainda não foram utilizados ativamente em nenhuma aplicação, são apenas aspectos da modelagem que podem vir a ser úteis se houver a necessidade de associar *Devices* que medem distância percorrida e energia consumida a futuras aplicações.

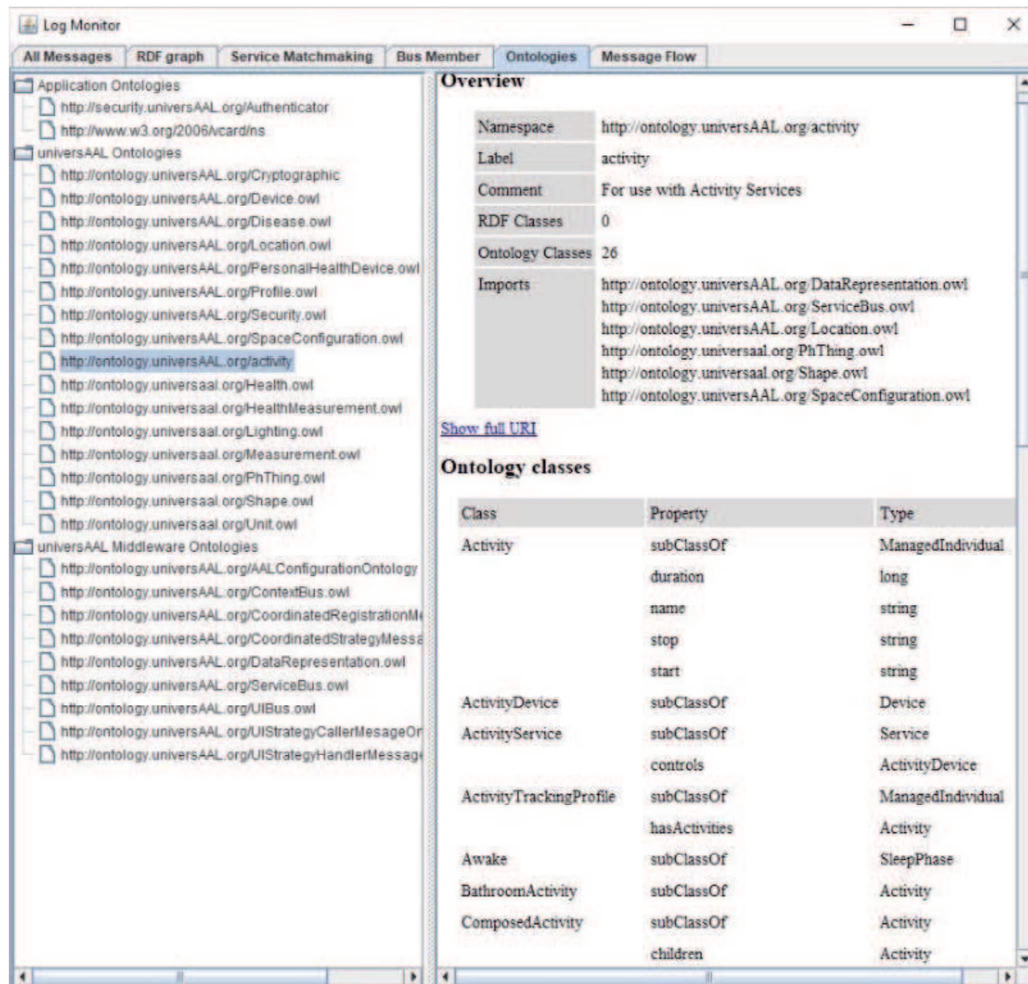
Por fim, basta mostrar alguns detalhes da ontologia no *Tools Log Monitor* (figura 43), ferramenta da plataforma que permite uma visualização de todas ontologias registradas, assim como elementos dos barramentos de contexto e serviço, *matches* de eventos, entre outras características do *middleware*.

**Figura 42 – Elementos do tipo *Device***




Fonte: Elaborada pelo autor (2018)

**Figura 43 – *Activity* no *Tools Log Monitor***



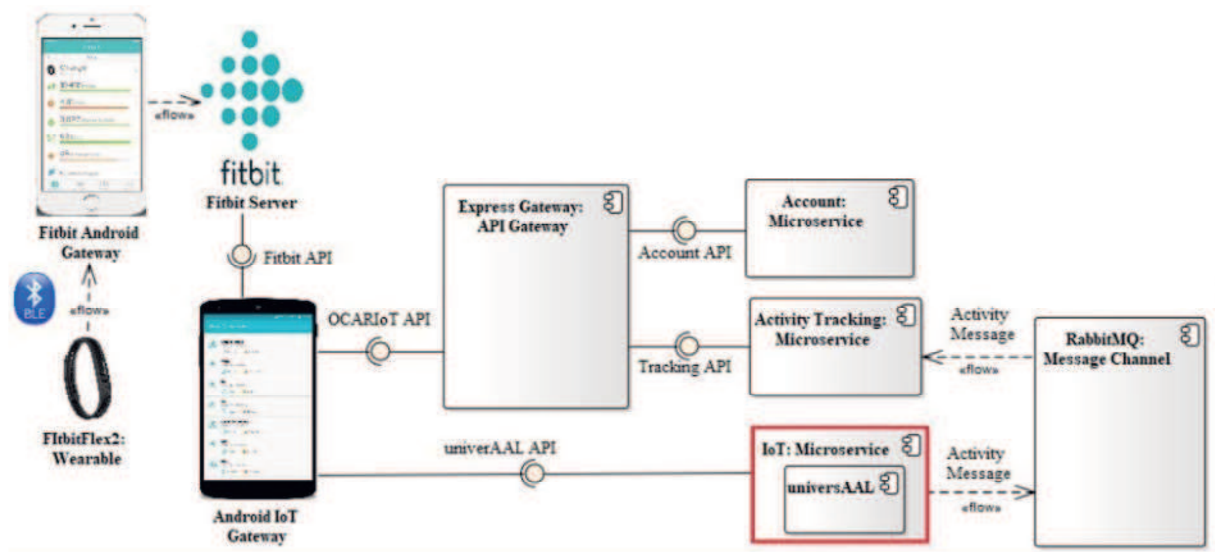

Fonte: Elaborada pelo autor (2018)

A próxima subseção trará mais detalhes sobre o uso da ontologia como modelo de dados na PoC realizada.

## 5.2 VISÃO GERAL DA POC

A ontologia desenvolvida foi utilizada para representar o conhecimento em um sistema que teve o objetivo de validar uma arquitetura de microserviços, como já citado neste trabalho. A arquitetura desta PoC pode ser vista na figura abaixo.

**Figura 44 – Arquitetura da PoC**




Fonte: NUTES/INSTITUTO ATLÂNTICO (2018)

O sistema acima recebe dados referentes a atividades registradas por um exemplar do dispositivo Fitbit Flex 2, através do servidor do Fitbit e uma aplicação *Android* envia esses dados modelados de acordo com a ontologia via REST. O elemento *universAAL* da arquitetura (marcado de vermelho) é uma aplicação *Subscriber* que será mais bem detalhada na próxima subseção, ela recebe este evento da REST, mapeia os dados para um formato JSON, e depois envia para uma fila de um *broker open source* de mensagens, o *RabbitMQ*, que suporta entre outros protocolos, o AMQP e o MQTT. Um microserviço chamado *tracking* faz o *Subscriber* desta fila de mensagens, e estes dados acabam voltando para o primeiro microserviço envolvido (aplicação *Android*), por intermédio de uma API *Gateway*. Isto não faz muito sentido em uma visão comercial, mas o sistema faz parte apenas de uma prova de conceito usada para a validação dessa arquitetura de microserviços.

Além do módulo *Subscriber* que será descrito mais adiante, a ontologia entrou como papel fundamental nessa arquitetura, pois ela definiu quais variáveis uma atividade deve ter, seja esta atividade de um nível mais geral com atributos apenas de nome, data de início e término, e duração, ou em níveis mais específicos, com níveis de energia consumida, de intensidade ou até mesmo quantidade de passos. E mesmo que apenas o módulo *Subscriber* (circulado de vermelho na figura 44) trabalhe com o formato ontológico, os outros microserviços precisaram modelar as informações recebidas/enviadas de acordo com essa especificação que foi definida na ontologia. Ou seja, a linguagem/tecnologia era diferente em

cada módulo, mas a semântica do sistema foi totalmente definida pela ontologia, permitindo uma forte especificação do domínio que era desejado.

## 5.3 MÓDULO UNIVERSAAL DA POC

A aplicação *Subscriber* que foi desenvolvida como elemento do microserviço IoT (*universAAL*) define um padrão de eventos (figura 45) para receber eventos que tenham apenas objetos do tipo *WalkActivity*, mesmo levando em consideração que outras atividades como andar de bicicleta e correr também foram recebidas pelo sistema. Foi escolhido esse tipo de objeto por ser o mais específico quando se trata de atividades físicas no modelo utilizado durante o desenvolvimento da PoC (no modelo mais novo *WalkActivity* não existe mais, assim como *DisplacementActivity*, ficando apenas o elemento *PhysicalExerciseActivity* responsável pelos atributos antes relacionados a estes dois). Se a atividade for, por exemplo, a de andar, o atributo que diz respeito ao número de passos pode ser utilizado, assim como a distância percorrida, mas se não basta que esses valores não sejam definidos, assim, de qualquer forma os atributos referentes a uma atividade física (energia consumida, frequência cardíaca, pressão sanguínea e intensidade) são abarcados.

Outra razão para isso é que um grande desafio foi encontrado na hora de criar o evento serializado no formato *Turtle* que seria enviado da aplicação *universaal* no *Android*. Este foi serializado com o objeto sendo do tipo *WalkActivity*, ou seja, para qualquer atividade enviada o objeto era do mesmo tipo. Caso essa estratégia não fosse utilizada seria necessário serializar um evento para cada tipo específico, além de modelar cada um dos que fossem necessários e ficaria difícil de automatizar, ou seja, enviar todos de uma única vez como é feito na execução atual do sistema.

O resto do *Subscriber* é composto apenas de um processo de inserção dos dados da atividade em um arquivo JSON (figura 46), da criação de um arquivo de saída usado apenas como elemento de depuração e depois da passagem do JSON como parâmetro na inicialização de uma classe chamada '*SendActivity*' (figura 47) responsável por definir toda configuração utilizada no *RabbitMQ*, e por enviar os dados para a fila compartilhada na arquitetura. Tudo isto feito dentro do método *handleContextEvent*(*ContextEvent event*) que só é ativado quando algum evento corresponde ao padrão definido.

**Figura 45 – *ContextEventPattern* da aplicação *Subscriber* da PoC**

```
ActivitySubscriber.java
1  package mainpackage;
2
3  // java.io
4⊕ import java.io.FileWriter;
25
26 public class ActivitySubscriber extends ContextSubscriber {
27
28⊖     private static ContextEventPattern[] getContextSubscriptionParams() {
29
30         // ContextEventPattern 1
31         ContextEventPattern cep = new ContextEventPattern();
32
33         cep.addRestriction(MergedRestriction.getAllValuesRestriction(ContextEvent.PROP_RDF_OBJECT, WalkActivity.MY_URI));
34
35         return new ContextEventPattern[] { cep };
36     }
37
38⊖    public ActivitySubscriber(ModuleContext context) {
39         super(context, getContextSubscriptionParams());
40     }
```

Fonte: Elaborada pelo autor (2018)

**Figura 46 – Criação do arquivo JSON**

```
ActivitySubscriber.java
113
114                saveFile.close();
115
116                /**
117                 * Populating JSON file with all information about WalkActivity
118                 */
119                // Output file
120                FileWriter writeFile = null;
121
122                // Populating JSON file
123                jsonObject.put("user_id", extractURI(event.getInvolvedUser().getURI()));
124                jsonObject.put("name", walkActivity.getProperty(Activity.PROP_NAME));
125                jsonObject.put("start_time", walkActivity.getProperty(Activity.PROP_START));
126                jsonObject.put("end_time", walkActivity.getProperty(Activity.PROP_STOP));
127                jsonObject.put("duration", walkActivity.getProperty(Activity.PROP_DURATION));
128                jsonObject.put("intensity_level", ((Intensity)
129                        walkActivity.getProperty(PhysicalExerciseActivity.PROP_INTENSITIES))
130                        .getProperty(Intensity.PROP_NAME));
131                jsonObject.put("distance", distanceValue.getProperty(Measurement.PROP_VALUE));
132                jsonObject.put("calories", energyConsumption.getProperty(Measurement.PROP_VALUE));
133                jsonObject.put("steps", walkActivity.getProperty(WalkActivity.PROP_STEPS));
134
135                try {
136                    writeFile = new FileWriter("saida.json");
137
138                    // Write the contents of the JSONObject in the output file and closes it
139                    writeFile.write(jsonObject.toJSONString());
140                    writeFile.close();
141
142                } catch(IOException e){
143                    e.printStackTrace();
144                }
145
146                try {
147                    new SendActivity(jsonObject);
148                } catch (Exception e) {
149                    e.printStackTrace();
150                }
```

Fonte: Elaborada pelo autor (2018)

Esta aplicação utilizou algumas bibliotecas, entre elas o cliente do *RabbitMQ* para o Java, foram elas:

- AMQP *Client*: http://central.maven.org/maven2/com/rabbitmq/amqp-client/4.0.2/amqp-client-4.0.2.jar

- SLF4J API: http://central.maven.org/maven2/org/slf4j/slf4j-api/1.7.21/slf4j-api-1.7.21.jar
- SLF4J *Simple*: http://central.maven.org/maven2/org/slf4j/slf4j-simple/1.7.22/slf4j-simple-1.7.22.jar
- JSON *Simple*: https://code.google.com/archive/p/json-simple/downloads

**Figura 47 – Classe *SendActivity***

```java
SendActivity.java
1  package RabbitMQ;
2
3⊕ import com.rabbitmq.client.ConnectionFactory;
8
9  public class SendActivity {
10     private final static String QUEUE_NAME = "activity_queue";
11
12     public SendActivity(JSONObject jsonObject) throws Exception {
13         sendActivity(jsonObject);
14     }
15
16     public void sendActivity(JSONObject jsonObject) throws Exception {
17         ConnectionFactory factory = new ConnectionFactory();
18         factory.setHost("rabbit1");
19         factory.setPort(5672);
20         factory.setVirtualHost("activity");
21         factory.setUsername("iot.app");
22         factory.setPassword("ocariot");
23         Connection connection = factory.newConnection();
24         Channel channel = connection.createChannel();
25
26         channel.queueDeclare(QUEUE_NAME, true, false, false, null);
27         channel.basicPublish("", QUEUE_NAME, null, jsonObject.toString().getBytes());
28         System.out.println(" [x] Sent '" + jsonObject + "'");
29
30         channel.close();
31         connection.close();
32     }
33 }
```

Fonte: Elaborada pelo autor (2018)

Foi necessário fazer o deploy do cliente do *RabbitMQ* na *universAAL*, ou seja, colocar seu arquivo jar na pasta *deploy* do *Karaf*, assim como com a ontologia e o próprio *Subscriber*. A figura abaixo mostra estes pacotes além da API REST necessária para o recebimento das atividades (e o *Apache CXF* necessário para essa API) na instância da plataforma.

**Figura 48 – Pacotes necessários para o módulo *universAAL***

```
488 | Active |  59 | 3.0.2             | Apache CXF Runtime HTTP Jetty Transport
489 | Active |  70 | 3.4.1.SNAPSHOT    | universAAL Remote Interoperability REST-API Manager
492 | Active |  80 | 4.0.2             | RabbitMQ Java Client
494 | Active |  80 | 0.1.0.SNAPSHOT    | universAAL Ontology Activity
495 | Active |  80 | 3.4.1.SNAPSHOT    | activitySubscriber
karaf@uAAL>_
```

Fonte: Elaborada pelo autor (2018)

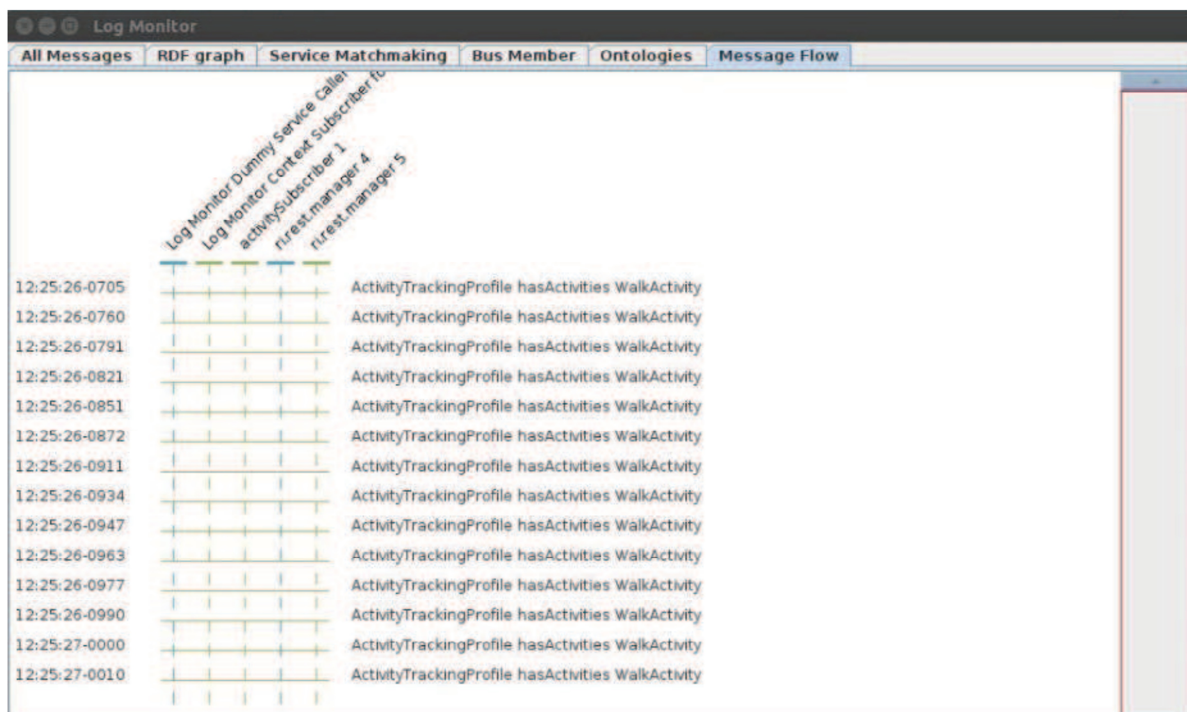
O resultado da chegada de atividades pode ser visto no *prompt* (figura 49) e no *Tools Log Monitor* (figura 50). O sistema operacional visto nas imagens é o Linux pois a execução foi registrada em outra máquina que estava rodando toda a arquitetura integrada com a tecnologia Docker. Tanto é que se pode ver na figura abaixo a autenticação no *RabbitMQ*, o envio da atividade pela *universAAL* para a fila de mensagens, o fechamento da conexão com o *RabbitMQ*, e, por fim, o armazenamento desses dados no serviço de *tracking*.

**Figura 49 – Execução da PoC**

```
rabbitmq             | 2018-09-21 15:08:35.710 [info] <0.1727.0> connection <0.1727.0> (172.19.0.7:51106 -> 172.19.0.3:5672): user 'iot.app
' authenticated and granted access to vhost 'activity'
uaal-karaf           |  [x] Sent '{"duration":21000,"start_time":"2018-07-18T10:29:43.000-03:00","intensity_level":"sedentary","distance":0
.0,"user_id":"5ba4f39c9d160b2df8b2a364","name":"Run","end_time":"2018-10-06T02:00:02.634","calories":0,"steps":4}'
rabbitmq             | 2018-09-21 15:08:35.725 [info] <0.1727.0> closing AMQP connection <0.1727.0> (172.19.0.7:51106 -> 172.19.0.3:5672, v
host: 'activity', user: 'iot.app')
rabbitmq             | 2018-09-21 15:08:35.728 [info] <0.1712.0> closing AMQP connection <0.1712.0> (172.19.0.7:51104 -> 172.19.0.3:5672, v
host: 'activity', user: 'iot.app')
tracking-service     | [DATA SAVED SUCESSFULLY]
```

Fonte: Elaborada pelo autor (2018)

**Figura 50 – Chegada dos eventos vista no *Log Monitor***




Fonte: Elaborada pelo autor (2018)

No *Tools Log Monitor* visto acima, os elementos do barramento de contexto são verdes e os do barramento de serviço são azuis, sendo que cada linha horizontal aparente na imagem representa um evento que foi registrado no barramento, neste caso com um rastreador de atividades (assunto) que tem uma atividade (predicado) do tipo *WalkActivity* (objeto).

Pode-se ver também os elementos registrados nos barramentos, como o "*activitySubscriber 1*" que representa o *Subscriber* mostrado nesta seção.

A seguir tem-se a descrição do uma distribuição customizada do *Apache Karaf* com alguns recursos pré-instalados, como o da ontologia, o da aplicação *Subscriber* e o do cliente do *RabbitMQ*. Processo esse que foi necessário para que a equipe pudesse ter um ambiente de execução *universAAL* com a possibilidade de ser baixado e compilado em qualquer máquina.

## 5.4 DISTRIBUIÇÃO PERSONALIZADA DA UNIVERSAAL

Uma distribuição do *Apache Karaf* foi customizada de acordo com as necessidades da equipe, sendo necessárias algumas edições basicamente no arquivo "*pom.xml*". Foi preciso adicionar a ontologia e o *Subscriber* (e o *RabbitMQ*, por consequência) como recursos padrões, sendo o primeiro passo adicionar os repositórios onde estavam, sendo que o *Subscriber* também foi adicionado no repositório *maven* criado seguindo os mesmos passos descritos na seção 4.5 e o *RabbitMQ* já se encontrava presente no repositório central do *maven*. As novas *tags* <*repository*> que foram colocadas dentro da tag <*repositories*> são vistas na figura abaixo.

**Figura 51 – Adição de novos repositórios ao *Karaf***

```
121    <repository>
122      <snapshots>
123        <enabled>false</enabled>
124      </snapshots>
125      <id>central</id>
126      <name>Central Maven Repository</name>
127      <url>http://repo1.maven.org/maven2</url>
128    </repository>
129    <repository>
130      <id>nutes-ocariot.Read</id>
131      <url>https://mymavenrepo.com/repo/v3u3q6Kh6YTjEQkuCDVO/</url>
132    </repository>
```

Fonte: Elaborada pelo autor (2018)

Com os repositórios disponíveis, foram adicionadas novas *tags* <*dependency*> para as três novas dependências necessárias (figura 52). E o *plug-in* do *Karaf* foi utilizado para primeiramente criar um arquivo chamado "*features.xml*" (linhas comentadas de número 245 a 253 do código da figura 53), que descrevia todos os recursos da *universAAL* e foi utilizado de exemplo para a descrição dos novos recursos.

**Figura 52 – Adição de novas dependências**

```
158        <dependency>
159            <groupId>org.universAAL.ontology</groupId>
160            <artifactId>ont.activity</artifactId>
161            <version>0.1.0-SNAPSHOT</version>
162        </dependency>
163        <dependency>
164            <groupId>org.universAAL.contextBus</groupId>
165            <artifactId>activitySubscriber</artifactId>
166            <version>3.4.1-SNAPSHOT</version>
167        </dependency>
168        <dependency>
169            <groupId>com.rabbitmq</groupId>
170            <artifactId>amqp-client</artifactId>
171            <version>4.0.2</version>
172        </dependency>
```

Fonte: Elaborada pelo autor (2018)

**Figura 53 – Execuções do *plug-in* do *Karaf***

```
239            <plugin>
240                <groupId>org.apache.karaf.tooling</groupId>
241                <artifactId>karaf-maven-plugin</artifactId>
242                <version>3.0.8</version>
243                <extensions>true</extensions>
244                <executions>
245                  <!--
246                  <execution>
247                    <id>generate-features-file</id>
248                    <phase>install</phase>
249                    <goals>
250                      <goal>features-generate-descriptor</goal>
251                    </goals>
252                  </execution>
253                   -->
254
255                  <execution>
256                    <id>create-kar-ActivityFeatures</id>
257                    <goals>
258                      <goal>features-create-kar</goal>
259                    </goals>
260                    <configuration>
261                      <featuresFile>${project.basedir}/src/main/resources/features.xml</featuresFile>
262                      <outputDirectory>${basedir}/target/assembly/deploy</outputDirectory>
263                    </configuration>
264                  </execution>
265
266                </executions>
267            </plugin>
```

Fonte: Elaborada pelo autor (2018)

A descrição dos novos recursos pode ser vista na figura abaixo.

**Figura 54 – Arquivo "*features.xml*"**

```
features.xml
1  <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
2  <features xmlns="http://karaf.apache.org/xmlns/features/v1.2.1" name="universAAL-Feature">
3      <feature name="uAAL-Ont.Activity" version="0.1.0-SNAPSHOT" description="For use with Activity services">
4          <details>For use with Activity services</details>
5          <bundle>mvn:org.universAAL.ontology/ont.activity/0.1.0-SNAPSHOT</bundle>
6      </feature>
7      <feature name="rabbitmq-amqp-client" version="4.0.2" description="RabbitMQ Client">
8          <details>RabbitMQ Client</details>
9          <bundle>mvn:com.rabbitmq/amqp-client/4.0.2</bundle>
10     </feature>
11     <feature name="uAAL-ActivitySubscriber" version="3.4.1-SNAPSHOT" description="Application that uses the Activity Ontology">
12         <details>Application that uses the Activity Ontology</details>
13         <bundle>mvn:org.universAAL.contextBus/activitySubscriber/3.4.1-SNAPSHOT</bundle>
14     </feature>
15 </features>
```

Fonte: Elaborada pelo autor (2018)

Alguns recursos nativos da *universAAL* foram configurados para também serem instalados, para isto bastou a adição deles na *tag* <*BootFeatures*>, como visto abaixo.

**Figura 55 – Adição de recursos pré-instalados**

```
271 <plugin>
272     <groupId>org.apache.karaf.tooling</groupId>
273     <artifactId>karaf-maven-plugin</artifactId>
274     <configuration>
275         <!-- <startupFeatures/> -->
276         <bootFeatures>
277             <feature>standard</feature>
278             <feature>management</feature>
279             <feature>uAAL-MW</feature>
280             <feature>uAAL-Ont.PhWorld</feature>
281             <feature>uAAL-Ont.Device</feature>
282             <feature>uAAL-Ont.CHe</feature>
283             <feature>uAAL-Ont.Profile</feature>
284             <feature>uAAL-Ont.Unit</feature>
285             <feature>uAAL-Ont.Cryptographic</feature>
286             <feature>uAAL-Ont.Profile</feature>
287             <feature>uAAL-Ont.Security</feature>
288             <feature>uAAL-Ont.Device</feature>
289             <feature>uAAL-Ont.Measurement</feature>
290             <feature>uAAL-Ont.Health.Measurement</feature>
291             <feature>uAAL-Ont.personalhealthdevice</feature>
292             <feature>uAAL-Ont.Health.Disease</feature>
293             <feature>uAAL-Ont.Profile.Health</feature>
294             <feature>uAAL-RI.RESTAPI</feature>
295         </bootFeatures>
296
297         <!-- <installedFeatures/>
298         <installedFeatures></installedFeatures>
299         -->
300
301     </configuration>
302 </plugin>
```

Fonte: Elaborada pelo autor (2018)

Assim estava pronta a distribuição customizada da *universAAL* que atendia as necessidades da equipe. Após isso o projeto foi armazenado no GIT da equipe, onde qualquer membro poderia cloná-lo e compilá-lo em sua máquina.

# 6 CONCLUSÃO

Neste trabalho foi desenvolvida uma ontologia que pudesse atuar como modelo de dados, disponibilizando um conjunto de atividades com atributos associados, afim de utilizá-lo principalmente em sistemas direcionados à assistência de usuários.

O objetivo geral foi conseguido, obtendo como resultado, a própria ontologia *Activity*, além de uma atuação eficaz e consistente, desta última, como modelo de dados no sistema desenvolvido para a PoC tantas vezes mencionada neste texto. E, falando sobre este sistema, pode-se dizer que seu objetivo também foi alcançado, visto que conseguiu validar, com sucesso, a arquitetura de microserviços e os módulos envolvidos nela, como a *universAAL*, o uso do Fitbit, o canal de mensagens, a API *Gateway*, entre outros.

A experiência de se trabalhar inicialmente com a plataforma já era bastante desafiadora, e quando surgiu a necessidade da criação de uma ontologia, o desafio só aumentou, mas graças às discussões realizadas com a equipe assim como o auxílio prestado, o objetivo principal foi conseguido e muita experiência foi adquirida, com ferramentas novas como a *universAAL*, o *universAAL Studio*, o ambiente de execução OSGi, Apache Karaf, para execução da plataforma, projetos *Maven* e suas configurações (dependências, arquivo "*pom.xml*", comandos de construção e instalação); com ferramentas já utilizadas durante o curso como a linguagem Java e o sistema de versionamento de código GIT; assim como uma adição bastante enriquecedora no senso de trabalho em equipe.

O desenvolvimento desta ontologia trouxe um conceito novo de representação do conhecimento, não historicamente falando, mas para o autor do trabalho, e abriu assim um leque de possibilidades de utilização desse conceito em possíveis projetos futuros.

Durante a etapa de utilizar *Activity* nesta PoC, ficou evidenciada a necessidade de estender seu modelo para outras atividades assim como a necessidade de se realizar alguns ajustes na modelagem, como por exemplo, simplificar algumas ramificações excluindo elementos não necessários, definir melhor as responsabilidades de cada atividade no modelo, entre outros aspectos, sendo estas possibilidades de trabalhos futuros. Como já dito no texto, alguns desses ajustes no modelo foram realizados e já existe um modelo diferente do que foi descrito na seção 5.1, mas sabe-se que ainda há muito o que aprimorar e adicionar nesta que pode ser uma forma muito útil de representar o conhecimento em futuros sistemas.

# REFERÊNCIAS


ALMEIDA, Maurício B. Uma abordagem integrada sobre ontologias: Ciência da Informação, Ciência da Computação e Filosofia. Perspectivas em Ciência da Informação, vol. 19, n.3, p.242-258, jul./set. 2014.

BERGMAN, Michael, K. An Intrepid Guide to Ontologies. 2007. Disponível em: <http://www.mkbergman.com/374/an-intrepid-guide-to-ontologies/>. Acesso em: 20 jun. 2018.

GIROLAMI, Michele. et al. universAAL: Provisioning Platform for AAL Services. Part of the Advances in Intelligent Systems and Computing book series, vol. 219, p. 105-112, 02 jun. 2014.

GRUBER, Thomas, R. A Translation Approach to Portable Ontology Specification. In: Knowledge Acquisition, vol. 5, p.199-220, 1993.

NOY, Natalya, F.; MCGUINNESS Deborah, L. Ontology Development 101: A Guide to Creating Your First Ontology. 2001. Disponível em: <https://protege.stanford.edu/publications/ontology_development/ontology101-noy-mcguinness.html>. Acesso em: 20 jun. 2018.

NUTES/INSTITUTO ATLÂNTICO. OCARIoT PLATFORM INTEGRATION: Proof of Concept (PoC) for Microservices Architecture Validation. 2018. 20 transparências.

OBITKO, Marek. *Translations between Ontologies in Multi-Agent Systems*. 2007. 149f. Dissertação de Ph.D. – Faculty of Electrical Engineering, Czech Technical University, Prague, 2007.

LIM, Yuto et al. Support for ECHONET-based smart home environments in the universAAL ecosystem. IEEE. USA. 12-14 jan. 2018.

PRINCETON University. WordNet: A Lexical Database for English. 2005. Disponível em: <https://wordnet.princeton.edu/>. Acesso em: 28 jun. 2018.

RAMALHO, Rogério A. S. *Web Semântica: aspectos interdisciplinares da gestão de recursos informacionais no âmbito da Ciência da Informação*. 2006. 121f. Dissertação de Pós-Graduação – Universidade Estadual Paulista, Marília, 2006.

RNP. Tecnologia baseada em Internet das Coisas promete redução da obesidade infantil. 2017. Disponível em: <https://www.rnp.br/destaques/tecnologia-baseada-internet-coisas-promete-reducao-obesidade-infantil>. Acesso em: 27 set. 2018.

SRI International. Open Knowledge Base Connectivity Home Page. 1995. Disponível em: <http://www.ai.sri.com/~okbc/>. Acesso em: 28 jun. 2018.

STOCKLÖW, Carsten. About UniversAAL. 2018a. Disponível em: <https://github.com/universAAL/platform/wiki/About-universAAL>. Acesso em: 11 ago. 2018.

STOCKLÖW, Carsten. Context Bus (Quick). 2018g. Disponível em: <https://github.com/universAAL/platform/wiki/Context-Bus-%28Quick%29>. Acesso em: 21 set. 2018.

STOCKLÖW, Carsten. Ontologies (Quick). 2018c. Disponível em: <https://github.com/universAAL/platform/wiki/Ontologies-%28Quick%29>. Acesso em: 11 ago. 2018.

STOCKLÖW, Carsten. Ontology Modelling. 2018e. Disponível em: <https://github.com/universAAL/tools.eclipse-plugins/wiki/Ontology-modelling>. Acesso em: 28 ago. 2018.

STOCKLÖW, Carsten. Ontology Project Wizard. 2018f. Disponível em: <https://github.com/universAAL/tools.eclipse-plugins/wiki/Ontology-Project-Wizard>. Acesso em: 28 ago. 2018.

STOCKLÖW, Carsten. Understanding UniversAAL. 2018b. Disponível em: <https://github.com/universAAL/platform/wiki/Understanding-universAAL>. Acesso em: 11 ago. 2018.

STOCKLÖW, Carsten. universAAL Studio overview and installation. 2018d. Disponível em: <https://github.com/universAAL/tools.eclipse-plugins/wiki/universAAL-Studio-overview-and-installation>. Acesso em: 18 ago. 2018.

STOCKLÖW, Carsten. Transformation OWL UML Java. 2018i. Disponível em: <https://github.com/universAAL/tools.eclipse-plugins/wiki/Transformation-OWL-UML-Java> Acesso em: 24 set. 2018.

W3C. OWL. 2013a. Disponível em: <https://www.w3.org/2001/sw/wiki/OWL>. Acesso em: 29 jun. 2018.

W3C. RDF. 2014. Disponível em: <https://www.w3.org/2001/sw/wiki/RDF>. Acesso em: 29 jun. 2018.

W3C. RDFS. 2010. Disponível em: <https://www.w3.org/2001/sw/wiki/RDFS>. Acesso em: 29 jun. 2018.

W3C. SPARQL. 2013b. Disponível em: <https://www.w3.org/2001/sw/wiki/SPARQL>. Acesso em: 29 jun. 2018.

# APÊNDICE A – UNIVERSAAL STUDIO

O *universAAL Studio*, também chamado de *AAL Studio*, é um conjunto de ferramentas de grande auxílio ao desenvolvedor, ferramentas essas que são integradas no *Eclipse* como *plug-ins*. Nesse conjunto estão disponíveis, entre outras, funcionalidades como assistentes para criação de projetos, e ferramentas de modelagem e transformação, que são as que devem ser usadas para criação de uma ontologia.

Segundo Stocklöw (2018d), é um ambiente de desenvolvimento baseado no *Eclipse* para construção de componentes de execução no *middleware*, que torna algumas tarefas de desenvolvimento mais eficientes e simples, deve ser usado para a modelagem e geração de código de ontologias a fim de reduzir não só o trabalho, mas também a ocorrência de erros. Sua ferramenta de modelagem de ontologias fornece uma interface simples e que direciona o desenvolvedor a se concentrar na ontologia e não no seu código, aumentando a agilidade no desenvolvimento e manutenção, e diminuindo a ocorrência de erros (STOCKLÖW, 2018e).

E, ainda conta com um assistente para a criação de um projeto, que segundo Stocklöw (2018f) facilita o início do desenvolvimento de ontologias para o desenvolvedor. Sendo que com a criação de um novo projeto utilizando este assistente, toda configuração inicial já vem inclusa, sendo esta formada por um projeto *Eclipse*, um arquivo *POM* para o *Maven* que é onde são configuradas todas as dependências do projeto, e um arquivo no *Papyrus*, *plug-in* de modelagem utilizado no processo deste trabalho. Além de possibilitar a configuração de informações sobre a ontologia como nome, descrição, versão, *groupId* e *ArtifactId*, entre outras, onde estas duas últimas juntas ao nome e a versão formam a dependência que pode ser utilizada por outros projetos que desejem utilizar a ontologia.

As figuras abaixo mostram as duas telas desse assistente. Na primeira tela (figura 56) é possível configurar todas as informações iniciais associadas ao projeto da ontologia. A partir destes, os campos da seção seguinte são derivados (se a opção "*Use derived values*" estiver marcada). Por fim, na última seção, basta colocar a versão do *Maven*, uma descrição (opcional) que estará no seu arquivo "*pom.xml*", e depois clicar em *Next*. A tela que segue mostra apenas as ontologias que já são importadas no modelo criado e instruções sobre como importar ontologias do *workspace* do *Eclipse*.

**Figura 56 – Assistente para projetos de ontologia do *AAL Studio* (Tela 1)**

Fonte: Stocklöw (2018f)

**Figura 57 – Assistente para projetos de ontologia do *AAL Studio* (Tela 2)**

Fonte: Stocklöw (2018f)