INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA INFORMÁTICA




Geração de UI para Controlo e Monitorização de Dispositivos Conectados

GUILHERME PINTO LEITE MAXIMIANO FERREIRA
Outubro de 2022

POLITÉCNICO
DO PORTO

Instituto Superior de Engenharia do Porto

Geração de UI para Controlo e Monitorização de Dispositivos Conectados

Guilherme Pinto Leite Maximiano Ferreira

Dissertação para obtenção do Grau de Mestre em Engenharia Informática, Área de Especialização em Engenharia de Software

Orientador: Prof. Paulo Maio

Porto, 15 de outubro de 2022

Dedicatória

Dedico este esforço a quem me é especial.

Resumo

A tecnologia é criada para melhorar a vida das pessoas e cada vez mais se tentam automatizar várias tarefas do quotidiano através da instalação de sistemas inteligentes. Estes suportam tarefas de monitorização de parâmetros (e.g. temperatura, pluviosidade, luminosidade, ligado/desligado, proximidade, movimento, leitura de uma tag NFC, leitura de dados biométricos, regulação de um potenciómetro) e reagem de alguma forma à alteração dos mesmos (e.g. alterar o estado do equipamento, enviar uma mensagem, comunicar com outro dispositivo).

Para facilitar a gestão dos dispositivos que constituem estes sistemas, existem algumas ferramentas disponíveis que ajudam a organizar informação recebida e apresentar estes valores em formato de gráfico, tabela ou outros, para que seja possível monitorizar estes. No entanto, estas ferramentas atuais não são fáceis de integrar em soluções mais pequenas, são complexas de configurar e, por norma, as interfaces gráficas para monitorização dos dispositivos são pouco ou nada personalizáveis.

Neste sentido, comunidades Faça você Mesmo (DIY) e autodidatas tendem a criar os seus próprios sistemas de gestão de dispositivos de forma a terem total controlo e poder de personalização. No entanto, isto implica que existam conhecimentos multidisciplinares - desenvolvimento dos dispositivos, criação e manutenção de um servidor de gestão de eventos, criação de interfaces gráficas e implementação de protocolos de comunicação.

Quer este projeto facilitar a criação de interfaces gráficas completamente personalizáveis e dinâmicas de forma a que sejam atualizadas sempre que um evento é espoletado pelos dispositivos conectados. Para este efeito é criada uma Linguagem Específica de Domínio (DSL) com o propósito de facilitar a personalização pretendida e a partir da qual é possível definir os elementos dos ecrãs, os dispositivos que se irão conectar ao mesmo e as ações a realizar mediante a chegada de eventos. Isto proporcionará maior facilidade na criação de soluções de integração de dispositivos numa interface gráfica completamente personalizável, reduzindo tempos de desenvolvimento para autodidatas e comunidades DIY dada a necessidade de menor conhecimento para o desenho e desenvolvimento deste tipo de soluções.

Com a solução desenvolvida é possível gerir e agregar a informação de vários dispositivos terminais num único dispositivo central. É ainda possível especificar uma interface gráfica na qual podem ser apresentados os dados recolhidos.

Palavras-chave: Geração de GUI, Linguagens Específicas de Domínio, Metamodelos, Gestão de Comunicações, IoT, DiY

Abstract

Technology is created to improve people's lives and more and more they try to automate several daily tasks through the installation of intelligent devices. These support tasks that rely on parameter monitorization (e.g. temperature, rainfall, brightness, on/off, proximity, movement, reading an NFC tag, reading biometric data, setting a potentiometer) and react in some way to changes (e.g. change the status of the equipment, send a message, communicate with a different device).

To facilitate the management of the devices that make up these systems, there are some tools available that help to organize received information and present these values in a graph, table or other format, so that it is possible to monitor them. However, these current tools are not easy to integrate into smaller solutions, they are complex to configure and, usually, the graphical user interfaces for monitoring the devices are little or not customizable at all.

In this sense, communities Faça você Mesmo (DIY) and self-educators tend to create their own device management systems in order to have total control and personalization power. However, this implies that there is multidisciplinary knowledge - device development, creation and maintenance of an event management server, creation of graphical interfaces and implementation of communication protocols.

This project wants to facilitate the creation of fully customizable and dynamic graphical interfaces so that they are updated whenever an event is triggered by the connected devices. For this purpose, a Linguagem Específica de Domínio (DSL) is created with the purpose of facilitating the writing of the desired personalization and from which it is possible to define the elements of the screens, the devices that will connect to it and the actions to be carried out upon arrival of events. This will make it easier to create device integration solutions in a completely customizable graphical interface, reducing development time and needed knowledge for self-educators and DIY communities, given the need for less knowledge for the design and development of this type of solutions.

With the solution developed it is possible to manage and aggregate the information from several terminal devices into a single central device. It is also possible to specify a graphical interface on which the collected data can be displayed.

Conteúdo

Lista de Figuras

Lista de Tabelas

Lista de Código

Lista de Acrónimo

AMQP	Advanced Message Queuing Protocol.
ANTLR	ANother Tool for Language Recognition.
API	Application Programming Interface.
AST	Abstract Syntax Tree.
CoAP	Constrained Application Protocol.
CSAT	Customer Satisfaction Score.
DI	Dependency Injection.
DIY	Faça você Mesmo.
DOM	Document Object Model.
DSL	Linguagem Específica de Domínio.
DTLS	Datagram Transport Layer Security.
EMF	Eclipse Modeling Framework.
GIL	Global Interpreter Lock.
GPL	Linguagem de Propósito Genérico.
GUI	Interface Gráfica de Utilizador.
GWT	Google Web Toolkit.
HOQ	House of Quality.
HTML	Hypertext Markup Language.
HTTP	Hypertext Transfer Protocol.
IDE	Integrated Development Environment.
IETF	Internet Engineering Task Force.
IoT	Internet das Coisas.
IP	Internet Protocol.
JVM	Java Virtual Machine.
LSP	Language Server Protocol.
MDE	Model Driven Engineering.
MOF	Meta Object Facility.
MPS	Meta Programming System.

MQTT	Message Queuing Telemetry Transport.
NDCM	New Concept Development Model.
OASIS	Organization for the Advancement of Structured Information Standards.
QFD	Quality Function Deployment.
REST	Representational State Transfer.
SaaS	System as a Service.
TCP	Transmission Control Protocol.
TEF	Textual Editing Framework.
TLS	Transport Layer Security.
TSL	Textual Syntax Language.
UCP	Unidade Central de Processamento.
UDP	User Datagram Protocol.
XAML	Extensible Application Markup Language.
XML	Extensible Markup Language.
XMPP	eXtensible Messaging and Presence Protocol.

Capítulo 1

Introdução

Este capítulo é dedicado à apresentação do contexto e do problema em que este trabalho assenta. São também apresentados os objetivos do mesmo, bem como a metodologia utilizada e, por fim, a estrutura dos restantes capítulos deste documento.

1.1 Contexto

Hoje em dia surgem no mercado cada vez mais e diversos equipamentos/dispositivos (e.g. sensores, *smart tv*'s, micro-controladores) com capacidade de transmitirem (quase) em tempo real dados/informação sobre o estado de algo (e.g. temperatura, pluviosidade, luminosidade, ligado/desligado, proximidade, movimento, leitura de dados biométricos, regulação de um potenciómetro) e, nalguns casos, também com capacidade de receberem dados/informação externa para atuarem sobre si próprios ou sobre outros equipamentos (e.g. alterar o estado de ligado para desligado ou vice-versa) (Chen et al. 2014).

Internet das Coisas (IoT)[1] é um conceito que representa os dispositivos conectados em rede (ou simplesmente online), que interagem com o nosso quotidiano e nos ajudam no dia-a-dia (Babun et al. 2021). A IoT encontra-se em constante crescimento por ser potenciado e por potenciar o surgimento de uma nova revolução industrial - Indústria 4.0 -, onde a comunicação e interação entre dispositivos mostrou ser fundamental (Okano 2017).

Com o maior acesso a este tipo de dispositivos eletrónicos (com especial relevância o fácil acesso a kits IoT que visam a experimentação com dispositivos como Arduínos e Raspberry Pi [Bajracharya e Hua 2020]), nota-se um aumento na quantidade de comunidades Faça você Mesmo (DIY)[2], isto é, comunidades que, independentemente do seu grau de conhecimento no tema, têm vontade de explorá-lo quer por ocupação, hobby, vontade de obter novas experiências, adquirir conhecimento ou, inclusive, criar algo novo (Kuznetsov e Paulos 2010).

[1] Traduzido do termo em inglês *Internet Of Things*
[2] Traduzido do termo em inglês *Do it Yourself*

1.2 Caracterização do Problema

A complexidade dos projetos IoT criados por comunidades DIY é tão maior quanto maior for a ambição e criatividade destes últimos. No entanto, a criação de um ecossistema baseado em dispositivos IoT passa, dependendo da complexidade do mesmo, por muitas e diferentes áreas da engenharia informática, por exemplo:

1. Programação do dispositivo IoT;
2. Tratamento de dados de sensores;
3. Comunicação na rede com outros dispositivos;
4. Recolha centralizada dos dados;
5. Criação de uma interface gráfica para um utilizador;

Embora o conceito DIY seja o de criação de algo novo ou reinvenção do que existe, nem todas as ferramentas são, normalmente, alvo de desenvolvimento e muitas soluções já existentes são reaproveitadas. Na lista que se segue encontram-se alguns projetos DIY na área de IoT com a respetiva enumeração de ferramentas existentes utilizadas.

1. **Sistema inteligente de irrigação**[3]: Ativa uma bomba de água para rega dependendo do nível da humidade do solo (com auxílio de um sensor de humidade). Recolhe ainda os valores obtidos de humidade no solo enviando-os para o ThingSpeak[4], uma plataforma de gestão, analise e visualização de dados. Esta plataforma oferece ainda uma Application Programming Interface (API) que poderá ser utilizada por outras aplicações para obter os dados recolhidos.
2. **Ventoinha controlada por um smartphone**[5]: Permite controlar a velocidade de rotação de uma ventoinha através da limitação da corrente. Neste projeto, foi desenvolvida uma aplicação Android que comunica a velocidade pretendida da ventoinha para a Firebase Realtime Database[6], a qual é subscrita pelo dispositivo IoT que controla a ventoinha e, mediante o valor existente nessa base de dados, faz alterar a velocidade de rotação da mesma.
3. **Estação Meteorológica**[7]: Através de sonsores de pressão do ar, de humidade e temparatura, alimentam uma página web suportada pelo próprio dispositivo para apresentação destes mesmos dados.

Vemos nestes projetos apresentados, bem como em vários outros existentes, que a utilização de soluções na nuvem[8] para centralização da informação (e acesso à

[3] https://circuitdigest.com/microcontroller-projects/iot-based-smart-irrigation-system-using-esp8266-and-soil-moisture-sensor

[4] https://thingspeak.com

[5] https://circuitdigest.com/microcontroller-projects/iot-based-ac-fan-speed-control-using-smartphone-with-nodemcu-and-google-firebase

[6] https://firebase.google.com/products/realtime-database

[7] https://circuitdigest.com/microcontroller-projects/esp12-nodemcu-based-iot-weather-station

[8] Traduzido do termo inglês *cloud* que define uma rede global de servidores

mesma na rede) são muito utilizadas. Para este ponto de centralização dos dados, nos exemplos apresentados, são utilizados protocolos comuns e não muito eficientes para dispositivos IoT, como o Hypertext Transfer Protocol (HTTP) (dado que é um protocolo pesado com muito *overhead*), e são escolhida plataformas como a ThingSpeak, Firebase, ou até mesmo, noutros projetos aparecem outras como: ThingWorx[9] ThingsBoard[10], AWS IoT Events[11], Hub IoT do Azure[12], Cloud IoT Core da Google[13], IoT Intellegent Applications da Oracle[14]. É de notar que, várias destas ferramentas suportam comunicação de dados através de vários protocolos diferentes, como por exemplo: HTTP, Message Queuing Telemetry Transport (MQTT), Advanced Message Queuing Protocol (AMQP), WebSockets, eXtensible Messaging and Presence Protocol (XMPP).

Os projetos expostos acima são, ainda assim, projetos simples onde apenas um dispositivo IoT (e.g. Rasberry Pi[15], Arduino[16], modelos Espressif[17]) comunica com os sensores existentes e, sozinho, comunica essa informação recolhida para um servidor que armazena estes dados. Existem, no entanto, casos onde o projeto levado a cabo por comunidades DIY vai além da comunicação de um conjunto pouco numeroso de dados (como é feito nos projetos listados) e passa sim por fazer ambientes onde, por vezes, vários destes projetos podem existir ao mesmo tempo numa única rede e comunicar e interagir entre si e, eventualmente, com um dispositivo central (existente numa rede local ou não). Deste modo, e para interligar dispositivos neste tipo de ambientes IoT, existem plataformas no mercado que possibilitam precisamente o estabelecimento de um canal de comunicação com os vários dispositivos na rede, fazer troca de informação (recolha de dados e/ou envio de comandos), e, algumas, possibilitam ainda a apresentação dos dados através da geração de uma API para consulta ou em interfaces gráficas com personalização muito reduzida (baseada em componentes estatísticos como gráficos e ou textos simples). Uma última característica limitadora de grande parte destas plataformas é o número reduzido de protocolos de comunicação suportados, embora esta seja, de facto, uma limitação que advém da falta de standards a este nível na área de IoT.

Podemos assim sistematizar as principais componentes que são de especial importância na criação de sistemas baseados em IoT:

1. Desenvolvimento dos próprios sensores/atuadores;
2. Agregação de informação proveniente dos vários dispositivos conectados;

[9] https://www.ptc.com/en/products/thingworx
[10] https://thingsboard.io/
[11] https://docs.aws.amazon.com/iotevents
[12] https://azure.microsoft.com/pt-pt/services/iot-hub
[13] https://cloud.google.com/iot-core
[14] https://www.oracle.com/pt/internet-of-things
[15] https://www.raspberrypi.org
[16] https://www.arduino.cc
[17] https://www.espressif.com

3. Controlo da comunicação entre os vários intervenientes e a segurança da informação;
4. Tratamento da informação recebida;
5. Apresentação de uma interface gráfica com acesso a estes dados.

Embora existam plataformas como a ThingWorx que possibilitam a agregação de dados dos dispositivos (cf. ponto 2), personalização da interface gráfica apresentada (embora muito orientada à análise de dados para casos de uso maioritariamente industriais) (cf. ponto 5) e que possui um vasto conjunto de protocolos de comunicação que inclui os mais utilizados da área (cf. ponto 2 e 3), não é possível utilizar esta ferramenta sem uma conexão à *internet* dado que esta serve apenas como solução na nuvem. Para projetos em redes locais, onde não exista o objetivo de comunicação com o - ou a partir do - exterior, não é uma ferramenta que seja possível de utilizar e, como esta, muitas outra existentes no mercado possuem esta característica.

Existem, desta forma, algumas limitações no âmbito de IoT a vários níveis que podemos sintetizar da seguinte forma:

- Ao nível do desenvolvimento dos dispositivos:
  - Capacidade de memória e processamento;
  - Interfaces de comunicação e conectividade;
  - Suporte de Sistema Operativo;
  - Segurança
  - Especificações de energia / Longevidade da bateria;
  - Tamanho e custo;
  - Qualidade dos dados recolhidos de sensores.
- Ao nível da implementação da camada de comunicação:
  - Falta de standards - cada plataforma tem os seus protocolos compatíveis;
- Ao nível da configuração das interfaces gráficas:
  - Plataformas existentes são normalmente direcionadas para casos de uso industriais, tendo as suas interfaces gráficas uma personalização limitada aos componentes existes para monitorização dos dispositivos;
- Ao nível da centralização dos dados:
  - Plataformas apenas focada em casos de uso na nuvem;

Os dispositivos IoT que se pretende controlar neste tipo de soluções são, em si, um dos principais problemas da área. Embora estes sejam acessíveis e haja cada

vez mais opções ou variantes no mercado, existem sempre preocupações a nível de hardware como indicado acima (Ojo et al. 2018).

No contexto deste trabalho distinguem-se dois tipos de dispositivos que serão deste ponto em diante denominados da seguinte forma:

- Dispositivos terminais - São os sensores ou dispositivos que comunicam diretamente com outras fontes de informação (e.g. outros dispositivos terminais, a Internet) e que enviam para a rede de dispositivos informação sobre os valores lidos.
- Dispositivo central - É o dispositivo que recebe e gere a informação proveniente dos dispositivos terminais.

Nos projetos realizados na área de IoT existe, como já referido, limitações/preocupações no que toca à qualidade dos dados recebidos. Esta última está diretamente relacionada com a qualidade dos próprios sensores, com a adequação do tipo de sensor à informação que se pretende obter (e.g. utilização de um sensor de humidade no ar junto de uma torneira para verificar se está aberta) e com o arredondamento feito muitas vezes nomeadamente em valores numéricos (e.g. leitura de potência recebida por pequenos electrodomésticos em quilowatt (kW) em vez de watt (W)).

No que toca à transmissão de dados existem ainda, vários protocolos que podem ser utilizados. Dado que o número de limitações destes dispositivos e a variedade dos mesmos, não tem sido possível estabelecer protocolos standard que satisfaçam todos os requisitos para os vários casos de uso possíveis. É ainda assim necessário que sejam seleccionado protocolos que sirvam, da melhor forma possível, os casos de uso que se pretende cumprir. A título de exemplo, num cenário onde seja necessária a criação de uma rede de dispositivos IoT num ambiente urbano, será necessário ter em atenção, entre outros (Chilipirea et al. 2016):

- O tipo de comunicações (e.g. Bluetooth, WIFI);
- A frequência utilizada para as comunicações (e.g. 2.4GHz, 5.0GHz);
- A distancia entre os dispositivos;
- O Número de outras redes que existam ao redor (evitar congestionamento de bandas).

Cada protocolo resolve problemas como os apontados acima de diferentes formas, pelo que atualmente a versatilidade das ferramentas existentes a este nível é um dos principais fatores de escolha.

1.3 Âmbito do Problema

Como foi mencionado anteriormente, no contexto de IoT existem várias problemáticas, limitações e diferentes áreas de desenvolvimento em jogo. Este projeto tem como público alvo comunidades DIY que podem ou não possuir conhecimento em

todas (ou algumas) áreas do desenvolvimento de sistemas IoT, mas que têm vontade de investir nas suas próprias soluções em termos de tempo de desenvolvimento, tempo para aprendizagem e, inclusive, investir nos próprios equipamentos a utilizar. Considera-se ainda assim que este público alvo deverá ser capaz de desenvolver os seus próprios dispositivos terminais, realizando a comunicação com sensores para obtenção de dados e implementação de uma camada de comunicação com um dispositivo central. Toda a gestão da informação, gestão dos dispositivos conectados e geração de interfaces gráficas serão processos que se assume que estas comunidades DIY não queiram reinventar dada a complexidade da área.

Deste modo, não será considerado o esforço de configuração e programação dos dispositivos terminais, sendo o âmbito apenas a:

- Criação de um programa que seja executado num dispositivo central para controlar e gerir informação proveniente de dispositivos terminas;
- Apresentação de uma interface gráfica para controlar e apresentar a informação obtida dos dispositivos centrais;

Assume-se, desta forma, que os dispositivos terminais serão programados (pelo o público alvo) de forma a que sejam compatíveis com o software executado pelo dispositivo central. Para isto, será necessário que este dispositivo utilize protocolo de comunicação popular ou standard para aumentar a sua versatilidade. Este projeto será desenvolvido tendo como dispositivo (central) alvo um equipamento que possua, à partida, capacidade de se conectar a uma rede e possibilidade de se conectar a um dispositivo periférico de saída de vídeo (e.g. HDMI, VGA).

1.4 Objetivos

O desenvolvimento deste projeto tem por objetivo geral contribuir com uma ferramenta genérica que suporte os processos inerentes a redes de dispositivos IoT:

- Gestão das comunicações com os vários dispositivos terminais;
- Centralização da informação dos vários dispositivos terminais num dispositivo central;
- Envio de mensagens/eventos do dispositivo central para os dispositivos terminais;
- Apresentação de uma interface gráfica dinâmica que apresente em (quase) tempo real os dados recebidos, sob especificação do utilizador.

De forma a cobrir o desenvolvimento dos processo acima descritos, são propostos os seguintes objetivos (O1, O2 e O3) e sub-objetivos (S1.1, S1.2, S3.1 e S3.2) que visam convergir numa solução que abstraia o seu público alvo das complicações inerentes a estes mesmos processos:

O1 Gestão das comunicações com os dispositivos conectados, incluindo gestão de eventos/mensagens recebidas/enviadas na comunicação com os dispositivos terminais.

S1.1 Criação de uma Linguagem Específica de Domínio (DSL)[18] que permita especificar a configuração de todos os dispositivos terminais da rede e respetivo meio de comunicação (e.g. protocolo de comunicação).

S1.2 A criação de um módulo que, através da especificação realizada no S1.1, realize a criação dos vários canais de comunicação com os dispositivos na rede, possibilitando o envio e receção de informação.

O2 A criação de uma camada de tratamento dos dados recebidos para que, sob configuração do utilizador, sejam considerados apenas os dados pretendidos;

O3 O desenvolvimento dos processos dependentes da interação com o utilizador, incluindo a possibilidade de especificar e customizar uma interface gráfica para apresentação da informação disponível obtida dos dispositivos terminais conectados;

S3.1 Criação de uma DSL para especificação de uma Interface Gráfica de Utilizador (GUI)[19] e os dados a apresentar na mesma;

S3.2 Uma aplicação que, com base numa instanciação da DSL especificada no ponto 1, construa a GUI e gira as comunicações com os diversos dispositivos conectados, incluindo a receção e envio de mensagens para os mesmos.

Esta ferramenta será dirigida essencialmente a entusiastas, praticantes de DIY, que queiram construir projetos na área de IoT e que pretendam uma mais fácil gestão dos seus dispositivos na rede e integração dos mesmos numa interface gráfica.

1.5 Método

Este projeto seguiu a metodologia de métodos de investigação de Hevner et al. 2004, que tem as seguintes *guidelines*:

- **Guideline 1 - Desenho da solução como artefacto**: O artefacto deste projeto, como referido nos objetivos do mesmo (secção 1.4), é, em suma, a criação de software capaz de gerar uma interface gráfica dinâmica através de eventos/mensagens que são recebidos provenientes de outros dispositivos na rede;
- **Guideline 2 - Relevância do problema**: O problema que serve de base à realização deste projeto foi descrito na secção 1.2. Podem ainda ser encontrados no Capítulo 2 o detalhe sobre abordagens e tecnologias da utilizadas na área em que se enquadra o projeto;

[18] Traduzido do termo inglês *Domain-Specific Language (DSL)*
[19] Traduzido do termo inglês *Graphical User Interface (GUI)*

- **Guideline 3 - Avaliação do desenho**: As etapas de avaliação e experimentação do artefacto encontram-se expostas no capítulo 6;
- **Guideline 4 - Contribuições da investigação**: Este projeto tem as suas contribuições apresentadas na secção 7.1;
- **Guideline 5 - Rigor da investigação**: As pesquisas realizadas no âmbito deste documento basearam-se na aplicação de métodos rigorosos tanto na construção, como na avaliação dos próprios desenhos dos artefactos;
- **Guideline 6 - Design como processo de investigação**: Este documento reporta apenas a versão final do Desenho da solução e avaliação, no entanto, foi feita a realização de várias iterações de pesquisa, desenho e avaliação. No capítulo 4 é possível ver mais detalhes sobre a análise efetuada do desenho da solução;
- **Guideline 7 - Comunicação da investigação**: É necessário que o resultado deste documento seja transmitido ao público alvo em fóruns dedicados de forma a que estes tenham acesso a este conhecimento.

1.6 Estrutura

Este documento encontra-se estruturado em diversos capítulos, os quais seguem a seguinte enumeração:

- **Capítulo 1 - Introdução**: Apresenta o contexto que serve de base ao projeto, assim como o problema que se prende resolver. São ainda definidos os objetivos e a metodologia do mesmo;
- **Capítulo 2 - Estado de Arte**: Apresenta uma análise sintetizada e sistematizada dos diferentes conceitos associados à comunicação entre dispositivos, gestão de eventos, geração de interfaces gráficas, linguagens de domínio e outras tecnologias ou *frameworks* relacionadas e consideradas relevantes. São ainda apresentadas e discutidas outras soluções semelhantes aquela que se pretende desenvolver;
- **Capítulo 3 - Análise de Valor**: Descreve os métodos e técnicas utilizadas com o intuito de definir o *Front-End of Innovation*. Indica o valor que a solução traz ao seu público alvo, assim como o quadro do modelo de negócios associado;
- **Capítulo 4 - Especificação da Solução**: Enumera e detalha os requisitos para este projeto, bem como descreve e ilustra o design da solução desenvolvida, detalhando as mais importantes decisões tomadas;
- **Capítulo 5 - Construção da Solução**: Detalha todos os passos referentes à implementação para cada fase de desenvolvimento da solução;

- **Capítulo 6 - Avaliação e Experimentação**: Expõe os detalhes e a metodologia utilizada na fase de experimentação e posterior avaliação dos resultados obtidos;
- **Capítulo 7 - Conclusão**: Lista os objetivos cumpridos, assim como limitações encontradas durante o desenvolvimento. Por último, é indicado o trabalho futuro possível de desenvolver com o sentido de melhorar o estado do projeto.

Capítulo 2

Enquadramento e Estado da Arte

Neste capítulo são apresentadas as tecnologias relevantes em todas as áreas da solução a produzir, passando pelas ferramentas para criação da Linguagem Específica de Domínio (DSL) e tecnologias para gestão de eventos e geração de interfaces gráficas, como definidos nos objetivos (secção 1.4). É feito inicialmente um levantamento e sistematização das principais características e necessidades requeridas ao desenvolvimento de interfaces gráficas no contexto deste projeto, bem como são identificados os possíveis problemas e limitações existentes que os sistemas computacionais alvo causados pelos seus recursos limitados.

2.1 Enquadramento

Na descrição do contexto e problema inerentes a este projeto (secção 1.1 e 1.2) foram descritas as limitações existentes por outras tecnologias já existentes tendo como referência as necessidades de determinados utilizadores da área. Estas limitações (e.g. necessidade personalização das interfaces gráficas, falta de standards) devem ser ultrapassadas para utilizadores que pretendem apenas uma solução simples de integração de vários dispositivos numa GUI personalizada. São cada vez mais comuns este tipo de projetos, devendo, por esta razão haver uma forma de simplificação dos seus processos de desenvolvimento (e.g. comunicação com dispositivos terminais, agregação de informação, geração de uma interface gráfica).

Neste sentido, é fundamental esclarecer as limitações existentes em cada um dos objetivos (ver secção 1.4) de forma a ser possível identificar e selecionar as ferramentas a utilizar na criação da solução final. Por este motivo e para sistematizar e justificar as ferramentas abordadas neste capítulo, foi desenvolvida a tabela 2.1 que visa levantar questões de implementação de cada um dos objetivos e sub-objetivos propostos.

Tabela 2.1: Questões por objetivo

Objetivo	**Sub-Objetivo**	**Áreas**	**Questões (Como/Qual?)**
O1	S1.1	Especificação dos dispositivos	Forma de especificação (DSL)
		Comunicação com dispositivos	Standard de comunicação a utilizar
	S1.2	Gestão das comunicações	Arquitetura das comunicações; Tecnologia para criação da ferramenta.
O2	-	Especificação dos dados recebidos/enviados	Forma de especificação (DSL)
O3	S3.1	Especificação de GUI	Linguagem base;
		Especificação das interações	Especificar ações e reações
	S3.2	Geração de GUI	Tecnologia para geração da GUI

Será assim sistematizadas as tecnologias existentes atualmente que sejam utilizadas para responder às questões colocadas, nomeadamente:

- Linguagem de Propósito Genérico (GPL) (e.g. Java, C, Go, python) para:
  - Desenvolvimento da Ferramenta de gestão de comunicações e tratamento de dados;
  - Geração final da GUI customizada pela utilizador para controlo dos seus dispositivos a partir de configuração;
- Protocolos a implementar para possibilitar a comunicação com os dispositivos na rede, permitindo o envio/receção de dados (e.g. Constrained Application Protocol (CoAP), Message Queuing Telemetry Transport (MQTT));
- Ferramenta para desenvolvimento da DSL de especificação da GUI pelo utilizador (e.g. Eclipse Modeling Framework (EMF)); e
- Linguagem que servirá de base à criação da DSL a criar (e.g. Extensible Application Markup Language (XAML), Extensible Markup Language (XML)) - Especificar mensagens a trocar com os dispositivos bem como especificar a interface gráfica a apresentar;

Na implementação das comunicações é necessário, inclusive, ter em conta a periodicidade da atualização de certos dados, sendo que pode ser necessário, em certos casos manter uma conexão sempre aberta ou pode dar-se o caso de ser apenas necessário abrir uma conexão com um dispositivo apenas quando necessário.

2.2 Enquadramento Teórico

Serão nesta secção enumeradas as ferramentas mais utilizadas para as três diferentes áreas deste projeto, segundo o que contexto descrito na secção 2.1: DSL para configuração da GUI, tecnologias e protocolos de comunicação utilizados para interligar dispositivos em rede e, por último, ferramentas e linguagens para a geração da GUI.

2.2.1 Linguagens de Domínio Específico

As DSL diferem das restantes Linguagem de Propósito Genérico (GPL) (Dart, C, C#, Python, Ruby ou Java), na medida em que estas últimas podem ser utilizadas para resolver problemas em qualquer domínio, quanto que as DSL são criadas para solucionar problemas específicos de um domínio particular. Existem ainda outras diferenças entre estes dois tipos de linguagem de programação, descritas na tabela 2.2.

Alguns exemplos destas linguagens específicas de um domínio são a *Structured Query Language* (SQL), utilizada para pesquisas e manipulação de base de dados, Hypertext Markup Language (HTML), para hipertexto e construção de páginas web. As DSL são produzidas para um domínio concreto de forma a que seja mais expressiva e de mais fácil utilização em comparação com as GPL (Strembeck e Zdun 2009; Voelter et al. 2013).

Uma linguagem, específica para um domínio ou não, tem os seguintes principais elementos (Voelter et al. 2013):

- **Sintaxe Concreta**: É a definição da notação a partir da qual os seus utilizadores podem expressar em programas, podendo ser gráfica, textual, tabular ou uma combinação delas.
- **Sintaxe Abstrata**: É a estruturação que pode conter informações de semântica relevantes expressas por um programa, sendo por normal uma árvore ou um gráfico. Esta não contem detalhes sobre a notação da linguagens.
- **Sintaxe Estática**: São as regras e/ou restrições de sistema com que os programas que se baseiam na linguagem se têm que conformar.
- **Sintaxe de Execução**: Refere-se ao significado de um programa assim que executado.

Existe ainda uma divisão das DSLs dependendo do seu tipo de implementação, que são Strembeck e Zdun 2009:

- **DSLs Internas (ou Embebidas)**: É definida como uma extensão a uma já existente Linguagem de Propósito Genérico (GPL) e utiliza os elementos sintáticos adjacentes a esta última. Desta forma, todas as funcionalidades da

Tabela 2.2: Linguagens Específicas de Domínio Vs. Linguagens de Uso Genérico (Voelter et al. 2013)

	GPL	**DSL**
Domínio	grande e complexo	pequeno e bem definido
Tamanho da Linguagem	grande	pequeno
Completude de Turing	sempre	nem sempre
Abstrações definidas por utilizador	sofisticadas	limitadas
Execução	via GPLs intermédias	nativas
Tempo de Vida	anos a décadas	meses a anos
Desenhado por	guru ou comité	poucos engenheiros ou experts do domínio
Comunidade de utilizadores	grande, anónima e dispersa	pequena, acessível e local
Evolução	lenta, geralmente padronizada	rápida
Alterações de Depreciação/Incompatibilidade	quase impossível	possível

linguagem estão disponíveis na própria DSL, *frameworks*, componentes, bibliotecas, compilador ou interpretador, debbuger, sendo que por esta razão se tornam desnecessárias transformações e geração de código adicional.

- **DSLs Externas**: É definida com um formato diferente da linguagem em que assenta, podendo utilizar vários tipos de elementos sintáticos. Para o utilizador não é possível a utilização de fucionalidades que não estejam disponíveis a partir da sintaxe concreta, podendo apenas ser utilizados aqueles que são diretamente expostos pela mesma. Para desenhar este tipo de DSL são utilizados sintaxes gráficas ou textuais que não se encontram limitadas pela linguagem da plataforma alvo.

Modelação e Desenvolvimento Guiado por Modelos

Existem dois tipos de modelação, a modelação descritiva que representa o sistema e foca apenas determinados aspetos e é utilizada com o intuito de ser matéria de discussão, comunicação e análise, e a modelação prescritiva, que pode ser utilizada para construir o sistema alvo, sendo mais rigorosa, formal, completa e consistente. A utilização de modelos a partir de modelação prescritiva é a base do Model Driven Engineering (MDE) (Voelter et al. 2013). Em MDE são criadas representações formais de partes concretas de um sistema que, por interpretação ou geração de código, são transformadas em código executável.

2.2.2 Ferramentas de DSL

Para desenvolver e manter DSLs, são necessárias ferramentas onde seja possível realizar a especificação da linguagem e os seus conceitos. São para este efeito utilizados projetos como o Xtext, que permite a criação de DSLs, o Xtend, que permite a geração de código e transformação de modelos, e outras ferramentas de funções similares como a Textual Editing Framework (TEF) ou a Meta Programming System (MPS), que serão posteriormente analisadas quanto à relevância para este projeto.

Xtext

O Xtext é um projeto de código aberto da *Eclipse.org* que torna possível a criação e desenvolvimento de DSLs de raíz, estando provida de um conjunto de APIs e DSLs para o efeito. Esta ferramenta é suportada por IDEs como o *Eclipse* [1] e outros que implementem a LSP, assim como navegadores Web e pode ser integrada com várias ferramentas de compilação, como o *Maven*, *Gradle* ou *Ant*. [2] O suporte de algumas plataformas para Xtext pode ser visto na figura 2.1.

O Xtext tem ainda várias outras funcionalidades [2]:

- É possível combinar os seus formatos baseados em texto com várias frameworks de edição gráfica (e.g. GEF[3], Sirius[4] ou Graphiti[5]);
- É escalável, isto é, o IDE consegue manter a mesma performance na reação a alterações textuais mesmo com centenas de ficheiros de código.
- Utiliza a *framework Google Guice* para Dependency Injection (DI) de forma a integrar toda a estrutura do IDE com a linguagem. É possível, também, construir editores para GPLs e DSLs, através da integração com o analisador LL(*) de ANother Tool for Language Recognition (ANTLR).
- A definição da gramática da linguagem de Xtext inclui um IDE que permite uma adaptação automática a cada linguagem criada.

Xtend

O Xtend é uma linguagem com uma sintaxe baseada na de Java e que compila para código legível e compatível com Java 8. O Xtext utiliza Xtend tanto na sua construção como para a transformação de modelos e gestão da geração de código. Pode ainda ser perfeitamente integrada qualquer biblioteca de Java nesta ferramenta. [6] Comparativamente com Java, o Xtend apresenta uma sintaxe melhorada em vários aspetos:[6]

[1] https://www.eclipse.org/ide/
[2] https://www.eclipse.org/Xtext/#feature-overview
[3] https://www.eclipse.org/gef/
[4] https://www.eclipse.org/sirius/
[5] https://www.eclipse.org/graphiti/
[6] http://www.eclipse.org/xtend/

	LSP	eclipse	
Syntax Coloring	✓	✓	✓
Semantic Coloring		✓	✓
Error Checking	✓	✓	✓
Auto-Completion	✓	✓	✓
Formatting	✓	✓	✓
Hover Information	✓	✓	✓
Mark Occurrences	✓	✓	✓
Go To Declaration	✓	✓	
Rename Refactoring	✓	✓	
Debugging		✓	
Toggle Comments	✓	✓	
Outline / Structure View	✓	✓	
Quick Fix Proposals	✓	✓	
Find References	✓	✓	
Call Hierarchy		✓	
Type Hierarchy		✓	
Folding		✓	

Figura 2.1: Xtext - Funcionalidades por plataforma [2]

- **Métodos de Extensão**: melhora tipos restritos com novas funcionalidades;
- **Expressões Lambda**: - tornam a sintaxe mais concisa para funções com literais anónimos;
- **Notações Ativas** - melhor processamento de anotações;
- **Sobrecarga de operadores** - torna as bibliotecas mais expressivas;
- **Poderosas expressões de troca** - trocas baseadas em tipos com *casts* implícitos;
- **Múltiplos encaminhamentos** - também conhecido como método de invocação polimórfica;
- **Expressões modelo** - com tratamento inteligente de espaços em branco;
- **Sem declarações** - tudo é considerado uma expressão;
- **Propriedades** - acesso rápido para aceder e definir *getters* e *setters*;
- **Inferência de tipo** - muito raramente se necessita, em Xtend, de escrever as assinaturas dos tipos de variáveis;
- **Suporte completo para genéricos de Java** - incluindo todas as regras de conformidade e conversão;
- **Traduz-se para Java e não para bytecode** - melhor perceção do que realmente acontece e possibilidade de utilização código para plataformas como Android ou Google Web Toolkit (GWT).

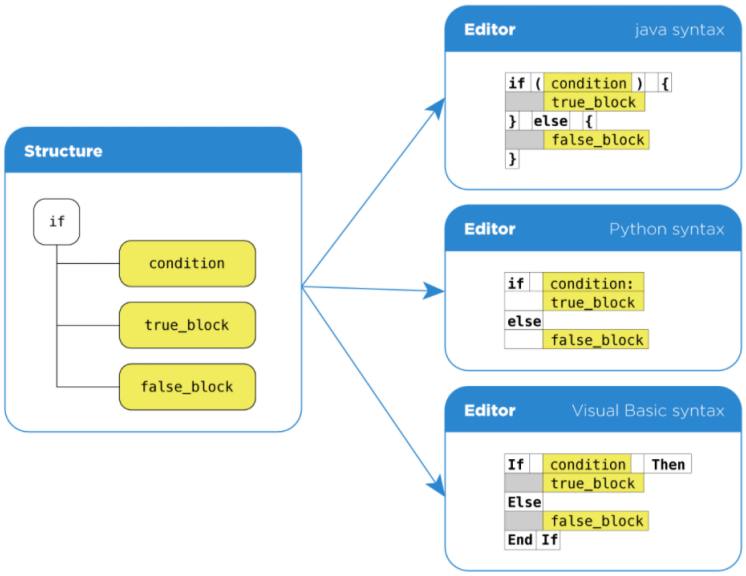

Figura 2.2: MPS - Editor para manipulação de AST [9]

Textual Editing Framework

A TEF permite a criação de editores de texto para as linguagens baseadas em Eclipse Modeling Framework (EMF). Esta framework fornece uma linguagem de definição de sintaxe chamada Textual Syntax Language (TSL) é possível descrever uma notação textual para dado metamodelo *Ecore*. Desta descrição, a TSL pode ser gerado automaticamente um editor TEF que fornece um conjunto vasto de recursos de editores de texto modernos, como: destaque de sintaxe, assistente de conteúdo (completação automática de código), navegação inteligente ou visualização de ocorrências. TEF é baseado no Eclipse, sendo inclusive fornecido como um conjunto de *plugins* do mesmo. [7]

Meta Programming System

O MPS é uma solução da *Jet Brains* [8] que se diferencia de outras *frameworks* de criação de linguagens por não utilizar formatos textuais na definição das mesmas, fazendo representações apenas em formato Abstract Syntax Tree (AST). O código é escrito, guardado e compilado em AST, o que permite evitar a criação de uma gramática e de um interpretador para as linguagens. [9] O editor de MPS manipula diretamente a AST e todo o código é sempre representado em formato de árvore. O editar apresenta a AST ao utilizador no formato pretendido pelo criador da linguagem, podendo inclusive escolher múltiplas e diferentes visualizações de código a qualquer momento (Figura 2.2).

A árvore é constituída por um conjunto de nós, onde cada um possui um nó-pai (exceptuando o nó raiz), nós-filho, propriedades e referências a outros nós. Os nós

[7] https://www.informatik.hu-berlin.de/sam/meta-tools/tef/tool.html
[8] https://www.jetbrains.com/
[9] https://www.jetbrains.com/help/mps/basic-notions.html

podem ser diferentes uns dos outros e por isso cada um armazena a referencia à sua declaração, que é entendido como sendo o conceito do nó. Este conceito define o tipo dos nós conectados, agrupando uma estrutura de nós e rotulando-a com esse tipo e formando uma hierarquia onde existe herança de propriedades e referências.[9]

Em MPS, a linguagem é precisamente o conjunto dos conceitos com mais alguma informação adicional associada à mesma - detalhes nos editores, menus de acabamento, intenções, sistema de tipos, fluxo de dados, etc. Linguagens em MPS formam componentes reutilizáveis, podendo uma linguagem estender outra e, desta forma, utilizar os conceitos desta última.[9]

O gerador de MPS realiza a conversão de modelo-para-modelo do modelo original para o modelo alvo que normalmente é definido numa linguagem diferente.[9]

Rascal

Rascal[10], criado em 2009, é uma linguagem extensível de meta-programação e IDE que faz análise e transformação de código-fonte. Combina e unifica recursos encontrados em várias outras ferramentas para manipulação de código-fonte e ambientes de desenvolvimento das linguagem. Rascal fornece uma interface programática que permite extender o Eclipse com suporte para as novas linguagens no mesmo IDE. Rascal é atualmente usado como um veículo de pesquisa para analisar software e implementação de DSLs (Klint, van der Storm e Vinju 2009).

EMFText

À semelhança de Xtext, pode ser utilizado para desenvolver sintaxes textuais para linguagens que sejam descritas por um meta-modelo Ecore. O EMFText oferece uma linguagem para especificação da sintaxe chamada CS e, na sua especificação, a sintaxe para os elementos do modelo é definida utilizando conceitos de *Extended Backups Naur Form (EBNF)* dado que conceitos da engenharia gramatical são bem conhecidos e são, portanto, uma ferramenta prática para refinamentos manuais da mesma. Para refinamentos que vão além da engenharia gramatical, como o mapeamento de identificadores para referências cruzadas, o EMFText gera uma API Java que pode ser utilizada para especificar como a implementação é feito mapeamento de alguns identificadores para os seus respetivos elementos (Heidenreich et al. 2009).

O EMFText é implementado como um conjunto de *plugins* para o Eclipse[1], fortemente integrado com o Eclipse Modeling Framework (EMF). Baseado em CS, vários artefactos são gerados pelo EMFText para fornecer ferramentas de *runtime* (Heidenreich et al. 2009), como pode ser verificado na Figura 2.3.

[10] `http://www.rascal-mpl.org/`

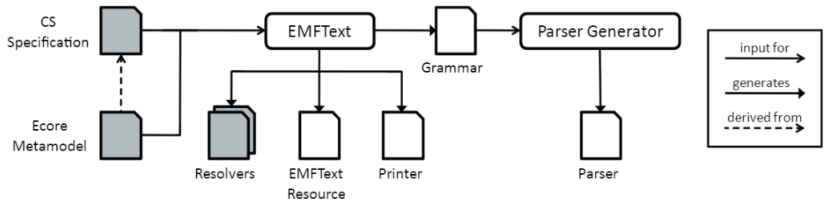

Figura 2.3: EMFText - Visão geral dos artefactos produzidos (Heidenreich et al. 2009)

Ferramentas Microsoft DSL

As ferramentas Microsoft DSL, integradas através do *Modeling SDK* no *Visual Studio*[11] (VMSDK) permitem a criação de modelos para representar conceitos de negócio, em formato de uma DSL.[12]

É utilizado um editor especializado para a definição de abstrações da sintaxe através de uma notação gráfica, através da qual o *Modeling SDK* gera: [12]

- Um implementação do modelo com uma API fortemente tipada que é executada num armazenamento com comunicação baseada em transações;
- Um explorador em formato de árvore;
- Um editor gráfico, a parir do qual é possível ver o modelo ou partes definidas do modelo.
- Métodos de persistência dos modelos para formato XML.
- Locais para geração de código de programas e outros artefactos, através de templates textuais.

Todas as funcionalidades acima expostas são passíveis de ser estendidas e posteriormente integradas de forma a atualizar a definição da DSL e re-gerar as mesmas funcionalidades sem perder as extensões feitas.[12]

Hospedado no Visual Studio, em termos de ferramentas DSL, já existem: [13]

- Assistentes de projetos que utilizam diferentes soluções template para dar apoio no início da aprendizagem sobre o desenvolvimento de DSLs;
- Um *designer* gráfico para criação e edição de DSL;
- Um motor de validações que garante que as definições das DSLs se encontram bem formatadas, mostrando erros, avisos e problemas, caso haja.
- Um gerador de código que recebe a definição de uma DSL e produz código fonte como resultado.

[11] https://visualstudio.microsoft.com/

[12] https://docs.microsoft.com/en-us/visualstudio/modeling/modeling-sdk-for-visual-studio-domain-specific-languages?view=vs-2019

[13] https://docs.microsoft.com/en-us/visualstudio/modeling/overview-of-domain-specific-language-tools?view=vs-2019

Eclipse Modeling Framework

O EMF é uma framework de modelação baseada em Eclipse que serve também como uma plataforma que permite a geração de código e construção de aplicações através de um modelo de dados estruturado (Gronback 2009). Com esta framework, através da modelação da aplicação, pode ser gerado o código (em vez de manualmente escrito) que especifica o comportamento desejado.

Tipicamente, as DSLs a ser desenvolvidas partem do desenvolvimento das suas sintaxes abstratas e, para isto pode ser utilizado o EMF, pela definição do modelo Ecore (que é uma implementação do meta-modelo de Meta Object Facility (MOF)). Esta framework permite criar, editar, validar, pesquisar e comparar os modelos criados, com suporte ainda para a persistência dos mesmos que pode, também, ser customizada (Gronback 2009).

Permite ainda transformação de modelo-para-modelo[14] e modelo-para-texto[15] (Gronback 2009).

2.2.3 DSL existentes

Atualmente já existem diversas DSLs que têm como objetivo especificar interfaces gráficas ou apenas configurações para sistemas. Serão nesta secção mencionadas as linguagens existentes mais relevantes no âmbito deste projeto.

HTML

HTML é uma DSL, com uma estrutura semelhante a Extensible Markup Language (XML) que serve de base à criação de páginas Web. Nesta linguagem é possível declarar vários elementos de UI e os navegadores de Internet já interpretam e renderizam graficamente estes elementos de forma nativa dado ser um padrão para a apresentação de páginas Web e é normalmente necessária a presença de um navegador de Internet para visualização das mesmas. Nesta linguagem é descrito o ecrã a desenhar através da construção textual de uma árvore hierárquica de elementos (Document Object Model (DOM)) e atributos da UI, como pode ser visto na figura 2.4, que corresponde ao código descrito na lista 2.1. [16]

[14] Do inglês *Model-to-Model*

[15] Do inglês Model-to-Text

[16] `https://www.html5rocks.com/en/tutorials/internals/howbrowserswork`

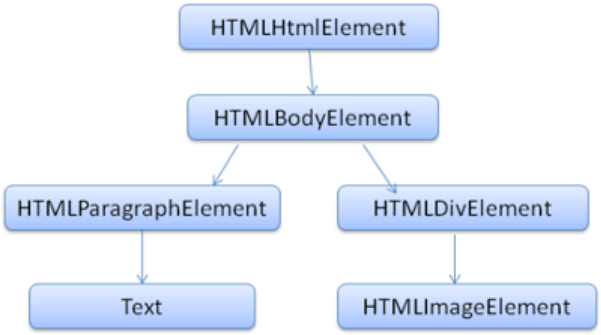

Figura 2.4: Exemplo de uma árvore DOM do processamento de HTML [17]

```
<html>
  <body>
    <p>
      Hello World
    </p>
    <div> <img src="example.png"/></div>
  </body>
</html>
```

Listing 2.1: Exemplo de uma página Web especificada em HTML

XAML

O Extensible Application Markup Language (XAML) é uma DSL de estrutura baseada em XML, que tem por objetivo simplificar a especificação de GUIs em ambientes aplicacionais. É utilizada por, por exemplo, aplicações *.NET Core* onde são especificados elementos da interface gráfica de forma declarativa através da linguagem, o que ajuda na separação da lógica aplicacional da definição da GUI. Neste exemplo, XAML representa a instanciação de objetos diretamente a partir de conjuntos específicos de tipos definidos, o que demonstra ter uma abordagem diferente relativamente a outras linguagens que são tipicamente linguagens interpretadas que não estão diretamente ligadas ao sistema de tipos. [18]

[18] https://docs.microsoft.com/en-us/dotnet/desktop/wpf/fundamentals/xaml

```
<Button>
  <Button.Background>
    <SolidColorBrush Color="Blue"/>
  </Button.Background>
  <Button.Foreground>
    <SolidColorBrush Color="Red"/>
  </Button.Foreground>
  <Button.Content>
    This is a button
  </Button.Content>
</Button>
```

Listing 2.2: Exemplo de um elemento de UI especificado em XAML

2.2.4 Comunicação e Gestão de Eventos

Nesta secção estarão descritas algumas tecnologias existentes para comunicação em rede, fazendo referências entre estas tecnologias e a sua aplicabilidade para a gestão ou comunicação de eventos.

É de notar que nesta área de IoT existe preocupações específicas dado que existem limitações a vários níveis, desde os próprios dispositivos que comunicam até à forma como a informação é transmitida.

Por este motivo são reunidos abaixo as principais características a considerar na escolha de um protocolo de comunicação nesta área (Naik 2017):

- Tamanho e overhead das mensagens:
- Energia consumida e necessidade de recursos
- Largura de banda necessária e latência:
- Fiabilidade/Qualidade do Serviço
- Interoperabilidade

Esta lista de características será posteriormente utilizada para classificar os diversos protocolos abordados.

HTTP

O Hypertext Transfer Protocol (HTTP) é um protocolo cliente-servidor baseado em conexões com Transmission Control Protocol (TCP) com possibilidade de serem encriptadas com recurso à Transport Layer Security (TLS). HTTP permite a aquisição de recursos na rede, como por exemplo documentos HTML, normalmente pedidos pelo cliente (que poderá ser um navegador *web*) ao servidor (Figura 2.5). Para adquirir um documentos é necessário que haja troca de várias mensagens entre cliente e servidor, dado que dependendo do tamanho deste, pode ser necessário reconstruí-lo a partir de várias partes do mesmo. [19]

[19] https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview

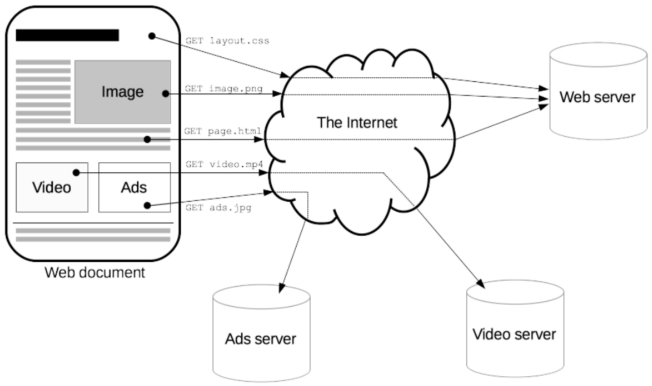

Figura 2.5: HTTP - Visão geral da comunicação entre cliente e servidor [14]

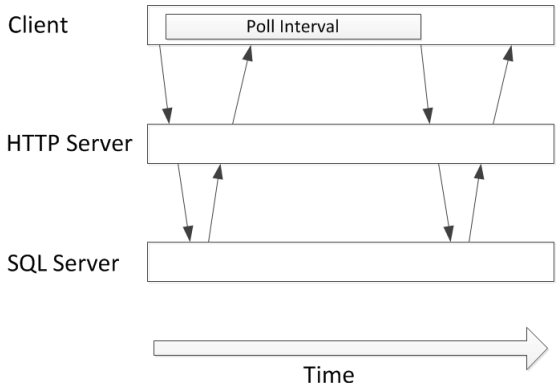

Figura 2.6: HTTP (Short) Polling (Cutting 2015)

O protocolo HTTP baseia-se em requisições, pelo que não tem como objetivo ser capaz de suportar a comunicação em tempo real de alterações do lado do servidor para o cliente. Para isso foram desenvolvidos alguns métodos, que estarão descritos abaixo, que visam atingir este objetivo: *polling*, *long polling* e *streaming*.

HTTP Polling

O *HTTP Polling* é o mecanismo através do qual um cliente (navegador *web*, aplicação mobile ou outros) envia periodicamente requisições ao servidor. Esta é um solução que apenas é indicada para casos em que o tempo de atualização dos dados do lado do servidor é conhecida e, por isso, conseguimos emular uma comunicação em tempo real, fazendo um pedido após o tempo de atualização do servidor, e repetir esta ação periodicamente (Figura 2.6) (Cutting 2015;Loreto et al. 2011).

Desta forma vemos que a limitação deste método é a necessidade de conhecer estes tempos de atualização, que normalmente não são conhecidos ou são imprevisíveis. Neste caso, o servidor poderia responder com informação que já não é a correta, ou que só se encontra atualizada por um período de tempo muito curto, ficando o

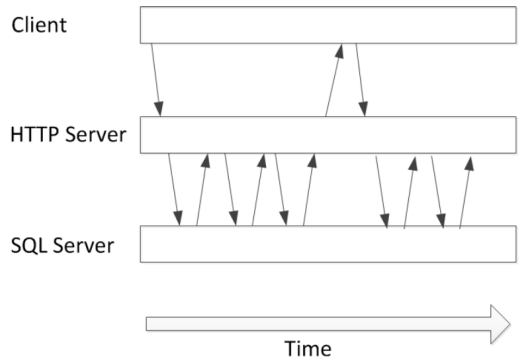

Figura 2.7: HTTP Long Polling (Cutting 2015)

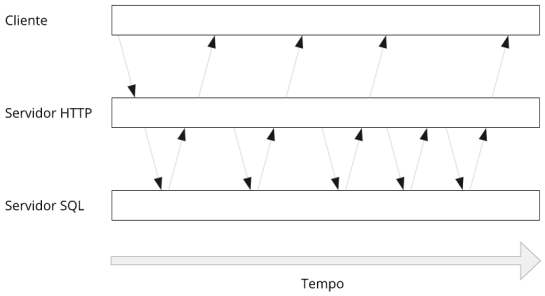

Figura 2.8: HTTP Streaming

lado cliente com informação não real até à próxima requisição escalonada (Cutting 2015; Loreto et al. 2011).

HTTP Long Polling

Ao contrário de *HTTP Polling* (secção 2.2.4), o método de *HTTP Long Polling* "[...]tenta minimizar tanto a latência na entrega das mensagens servidor-cliente como a utilização de recursos de processamento/*networking*"(traduzido de Loreto et al. 2011). Isto é alcançado pela implementação de uma requisição por parte do cliente para o servidor, em que o servidor mantem a comunicação aberta até ter dados atualizados ou que aconteça um *timeout*. Após a conclusão da comunicação, o cliente inicia imediatamente uma nova requisição a pedir novamente uma atualização de dados (Figura 2.7).

HTTP Streaming

Este método de *HTTP Streaming* é, como os restantes, iniciado pelo cliente através de uma requisição de um determinado recurso ao servidor. A diferença está no facto de que esta conexão que é criada para a transferência do recurso nunca é terminada, sendo que o servidor manda várias respostas ao longo do tempo (Figura 2.8) (Loreto et al. 2011).

Server-Side Events

Os *Server-Side Events* são uma tecnologia, baseada em HTTP, que permitem que informação seja recebida do servidor sob a forma de eventos com Document Object Model (DOM) (Loreto et al. 2011). Para fazer uso desta tecnologia, o cliente tem que fazer uma requisição ao servidor para receber um determinado recurso, sendo que após configurar o pedido, pode definir *listeners* para eventos específicos que a qualquer momento podem chegar do servidor e podem conter nova informação, atualização de informação ou até informação de qualquer erro que possa acontecer. [20]

CoAP

É um protocolo que serve se proposta de standard - RFC 725223 - pela Internet Engineering Task Force (IETF) CoRE (Constrained RESTful Environments), em Junho de 2014. Constrained Application Protocol (CoAP) é um protocolo concebido para dispositivos e redes com recursos limitados para transferência de dados. É construído tendo por base User Datagram Protocol (UDP) e é baseado no paradigma arquitetural Representational State Transfer (REST), sendo, por este motivo, semelhante (funcionalmente) ao HTTP, utilizando inclusive métodos HTTP como o GET, POST, PUT e DELETE (Shelby, Hartke e Bormann 2014).

Dado que se trata de um protocolo baseado em UDP (Shelby, Hartke e Bormann 2014):

- Não consegue garantir a entrega dos pacotes ao destino mas, por este motivo, tem dois tipos de mensagem podem ser utilizados e que diferem no facto de necessitarem de validação de recebimento da mesma pelo destinatário: Mensagem Confirmável[21] e Não-confirmável[22].
- Não consegue utilizar TLS - dado que este protocolo de segurança requer a garantia de entrega de mensagens -, havendo a substituição deste protocolo de segurança por Datagram Transport Layer Security (DTLS). A utilização deste protocolo para segurança faz com que se perca uma das vantagens do UDP de não exigir uma conexão aberta para a troca de mensagens.

AMQP

O Advanced Message Queuing Protocol (AMQP) é um padrão para envio assíncrono de mensagens entre aplicações. Foi desenhado para suportar o envio de mensagens para praticamente qualquer aplicação distribuída, compreendendo tanto uma camada de rede - que especifica entidades como o produtor, consumidor ou o

[20] https://nordicapis.com/stop-polling-and-consider-using-rest-hooks/
[21] Traduzido do termo em inglês *Confirmable*
[22] Traduzido do termo em inglês *Non-confirmable*

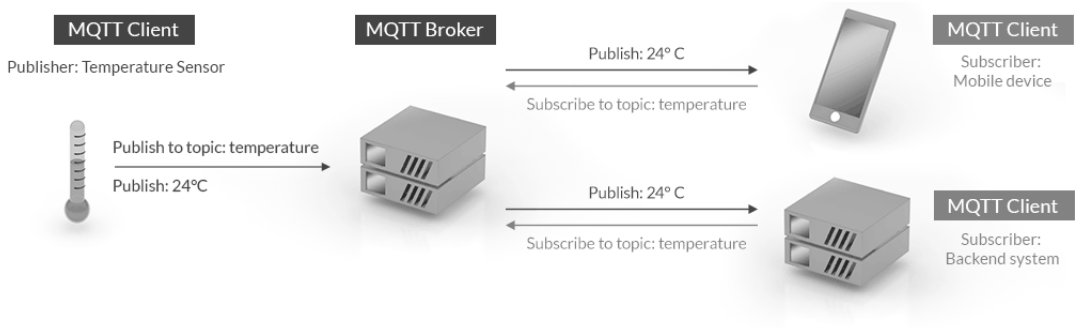

Figura 2.9: MQTT Visão geral do protocolo de subscrição/publicação

corretor de mensagens -, como o modelo do protocolo - que especifica a estrutura das mensagens a enviar. [23]

MQTT

O Message Queuing Telemetry Transport (MQTT), é um protocolo de envio (assíncrono) de mensagens padrão OASIS na área da Internet das Coisas, desenhado para conectar dispositivos em rede com reduzidas linhas de código e uma camada de transporte de mensagens leve para os próprios dispositivos (uma visão geral da comunicação pode ser vista na Figura 2.9). Este protocolo permite:[24]

- Baixa utilização de recursos de rede e suporte para redes instáveis, com redução do tempo para reestabelecer as conexões perdidas;
- Comunicação bidirecional entre o dispositivo e o servidor e entre o servidor e o dispositivo, ou entre dispositivo e dispositivo;
- Permite encriptação com utilização de TLS;
- Maior confiança na entrega de mensagens enviadas, com três níveis de qualidade de serviço pretendido para a recepção de mensagens: 0 - no máximo uma vez, 1 - pelo menos uma vez, 2 - exatamente uma vez;
- Escalável horizontalmente, possibilitando conexões com milhões de dispositivos.

Intermediários de Mensagens

Intermediários de Mensagens[25] são plataformas intermediárias na comunicação entre duas (ou mais) aplicações, sendo responsáveis pelo correto processamento e reencaminhamento das mensagens dos produtores para os respetivos consumidores - são

[23] `https://www.amqp.org/sites/amqp.org/files/amqp.pdf`https://www.amqp.org/sites/amqp.org/files/amqp.pdf
[24] `https://mqtt.org/`
[25] Traduzido do termo em inglês *Message brokers*

exemplo o *RabbitMQ*[26], *Apache Kafka*[27], *IBM MQ*[28], entre outros. Estas plataformas possuem integrações com vários protocolos de mensagens como o AMQP ou MQTT.

RabbitMQ

O RabbitMQ é um dos mais populares intermediários de mensagens que se encontra disponível para diversos sistema operativos bem como na nuvem, sendo que oferece um grande conjunto de ferramentas para desenvolvimento para as linguagens mais populares.[29]

Esta ferramenta, entre outras funcionalidades, Suporta[31]:

- Envio assíncrono de mensagens;
- Vários protocolos de comunicação (e.g. AMQP, MQTT, HTTP);
- Enfileiramento de mensagens;
- Confirmação de Entrega de mensagens;
- Utilização por parte de diversas linguagens de programação (Java, .NET, PHP, Python, JavaScript, Ruby, Go, etc.);
- Possibilidade de implantação na cloud com comunicação segura;
- Interface gráfica e API-HTTP que possibilitam a monitorização do estado da própria plataforma;

Apache Kafka

O Kafka é uma plataforma intermediária de mensagens de código aberto que foi desenhado como um registador de transações (*commit log*) distribuído e persistente para suportar microsserviços de comunicação baseada em eventos.[30] Esta plataforma é desenvolvida em Java e pode apenas ser implantada em equipamentos que sejam capazes de executar a Java Virtual Machine (JVM). No entanto, o desenvolvimento de clientes para comunicação pode ser construído em diversas linguagens de programação, havendo várias bibliotecas desenvolvidas para Go, Python, Node.js, etc. Kafka não opera diretamente com MQTT devendo as comunicações através deste protocolo passar por outro intermediário de mensagens que o suporte e, a partir deste, será possível comunicar com o Kafka. Este intermediário também não suporta ingestão de mensagens via HTTP.[31]

[26] https://www.rabbitmq.com/
[27] https://kafka.apache.org/
[28] https://www.ibm.com/products/mq
[29] `https://www.rabbitmq.com/`
[30] `https://www.confluent.io/blog/kafka-fastest-messaging-system`
[31] `https://thenewstack.io/apache-kafka-cornerstone-iot-data-platform/`

2.2.5 Desenho de Interfaces Gráficas

Para desenho de interfaces gráficas, nesta solução, uma das principais vantagens mais procuradas é a facilidade de implementação e conseguir executar e gerar interfaces gráficas num grande número de plataformas tentando não prejudicar a performance.

Para colocar dentro de alcance dispositivos com poucos recursos de processamento, pode ser necessária a utilização de soluções mais leves, que requeiram menos código e que tenham em consideração a performance em tempo de execução. Para isso foram selecionadas algumas linguagens que de alguma forma conseguem satisfazer estes requisitos, tendo compatibilidade para gerar interfaces gráficas em diversos sistemas operativos e sendo linguagens relativamente leves e muito utilizadas nestes dispositivos.

Golang

Golang (ou apenas Go) é uma linguagem criada pela *Google* em 2009, dirigida para programação de sistemas, sendo ainda muito semelhante a *C++*. É ainda uma GPL compilada e *multi-threaded* e que é "(...)expressiva, concisa, limpa e eficaz."(Brimzhanova et al. 2019).

Tem ainda mecanismos de controlo de concorrência que simplificam a escrita de programas que façam uso de múltiplos núcleos de processamento ou que utilizem recursos em rede. Go compila rapidamente para código de máquina conseguindo ainda comportar *trash collection* e *reflection* em tempo de execução (Brimzhanova et al. 2019).

Para comunicação na área de IoT, existem várias bibliotecas que facilitam a implementação de clientes para comunicação HTTP, MQTT, ou até com a comunicação com intermediários de mensagens como o Kafka ou o RabbitMQ.

Em termos de desenho de interfaces gráficas, Go tem vários projetos com implementação de várias interfaces para diversas plataformas[32]. Existem projetos para, utilizando a linguagem Go, criar interfaces gráficas:

- Para Windows (gform[33] ou winc[34]);
- Para Android e iOS (go-mobile[35]);
- Para Linux com GTK (GTK2 com go-gtk[36] e GTK3 com gotk3[37] ou gobbi[38]);

[32] https://github.com/go-graphics/go-gui-projects
[33] https://github.com/AllenDang/gform
[34] https://github.com/tadvi/winc
[35] https://github.com/golang/mobile
[36] https://github.com/mattn/go-gtk
[37] https://github.com/gotk3/gotk3
[38] https://github.com/pekim/gobbi

- Com suporte a vários sistemas operativos (*Cross-Platform*) (shiny[39], ui[40], webview[41] etc.).

Python

Python é uma GPL interpretada, de alto nível e orientada a objetos com uma semântica dinâmica. É uma linguagem muito utilizada na área da Internet das Coisas, sendo também um linguagem usual para a escrita de *scripts* ou como linguagem para conectar componentes existentes.[42]

Esta linguagem possui várias bibliotecas para criação de interfaces gráficas, sendo as mais relevantes (e que não são específicas a nenhuma plataforma) Tkinter[43], wxWidgets[44] e Qt[45]. Possui ainda bibliotecas para execução de código multi-processo (multiprocessing[46]). Python não permite que *threads* executem em paralelo devido ao Global Interpreter Lock (GIL) que assegura que duas *threads* não acedem simultâneamente aos mesmos recursos (útil muitas vezes para envio de mensagens na rede). No entanto, é possível ultrapassar o GIL com a utilização de multiplos processos dado que cada processo terá o seu próprio interpretador e espaço de memória[47].

Node.js

Node.js é uma plataforma de código aberto desenvolvida em 2009 por Ryan Dahl para criação de uma aplicação de servidor e é construída através do motor de JavaScript do Google Chrome[48]. Tem por objetivo a construção de aplicações web de forma rápida e facilmente escaláveis, utilizando um mecanismo baseado em eventos que não bloqueia operações I/O[49], tornando-se uma plataforma eficiente e leve, ideal para aplicações que contenham a necessidade de comunicação em tempo real com uma quantidade significativa de dados.[50]

Node.js tem como linguagem o JavaScript e tem a possibilidade de ser executada em múltiplos sistemas operativos como macOS[51], Microsoft Windows[52] e Linux[53].

Esta plataforma de desenvolvimento tem várias funcionalidades[50]:

[39] https://github.com/golang/exp/tree/master/shiny
[40] https://github.com/andlabs/ui
[41] https://github.com/zserge/webview
[42] https://www.python.org/doc/essays/blurb
[43] https://docs.python.org/3/faq/gui.html#id5
[44] https://docs.python.org/3/faq/gui.html#id6
[45] https://docs.python.org/3/faq/gui.html#id7
[46] https://docs.python.org/3/library/multiprocessing.html
[47] https://realpython.com/python-gil/
[48] https://www.google.com/chrome
[49] Input / Output
[50] https://www.tutorialspoint.com/nodejs/nodejs_introduction
[51] https://www.apple.com/macos
[52] https://www.microsoft.com/en-us/windows
[53] https://www.linux.org/

- Baseada em eventos assíncronos – O servidor não espera pelo retorno de dados, continuando a execução de outros pedidos, mas tem por base um mecanismo de gestão de eventos e notificações que garante que todos os pedidos tenham uma resposta;
- Rápida e eficiente - É considerada rápida em termos de execução de código dada a utilização de javaScript e baseada na engine da mesma para Google Chrome;
- Facilmente escalável - Embora Node.js utilize um modelo de apenas uma *thread* com *event looping*, que torna a as respostas assíncronas, o que torna esta plataforma altamente escalável, ao contrário dos servidores tradicionais que utilizam *threads* limitadas para responder aos pedidos recebidos;
- Sem carregamento contínuo de dados - Aplicações em Node.js simplesmente enviam dados em blocos;

Existe ainda a framework NodeGui[54] que permite a criação de interfaces gráficas sem restrição de sistema operativo (disponível para Windows, MacOS e mais populares distribuições de Linux).

2.3 Análise de Ferramentas

Nesta secção são analisadas, comparadas e selecionadas as ferramentas mencionadas na secção 2.2, tendo em conta as necessidades e requisitos observados para este projeto descritos em maior detalhe no capítulo 4.

2.3.1 Especificação de interfaces gráficas

Para a especificação de interfaces gráficas é necessário ter em conta a definição da DSL. Como ferramenta de desenvolvimento desta DSL será utilizado o MPS dado que permite a criação de editores específicos para a especificação da linguagem, bem como permite a criação da mesma de uma forma mais visual, mantendo toda a personalização necessária para a sua criação. o MPS permite ainda a geração dos conceitos criados para ficheiros de texto, podendo transformar os conceitos, no limite, em código de qualquer linguagem.

2.3.2 Protocolo de Comunicação

Para um projeto em IoT, motivado pelo público alvo deste projeto e pela falta de standards, deve ser possível implementar mais do que um protocolo de comunicação e, por isso, não será aqui selecionado exclusivamente um protocolo, mas será feita uma comparação dos vários existentes e mencionados.

[54] `https://github.com/nodegui/nodegui`

HTTP é um protocolo standard de comunicação utilizado na *web* no entanto, a sua utilização em sistemas IoT não é adequada na medida em que, comparativamente com os outros protocolos enumerados:

- Apenas permite comunicações de um-para-um, isto é, é útil na comunicação entre dois intervenientes, mas em redes dificulta a manutenção de tantos canais TCP abertos - faz inclusive aumentar os custos de energia dos dispositivos que enviam/recebem pedidos;
- É um protocolo em que as mensagens transportadas são consideradas grandes no meio, sendo que apenas o *overhead*, assumindo HTTP sobre TCP/IPv4, é de pelo menos 40bytes[55].

A alternativa ao HTTP para IoT é o CoAP. Em termos funcionais é muito semelhante, no entanto, o CoAP foi desenhado para dispositivos limitados e redes limitadas, tendo um *overhead* estático reduzido de 4 bytes apenas. A comunicação entre dispositivos pode acontecer sem ser estabelecido nenhuma conexão entre ambos dado que é utilizado UDP para o envio das mensagens. É baseado no paradigma arquitetural REST, o que permite o acesso a recursos pelos clientes através de funções como GET, PUT, POST e DELETE (à semelhança, mais uma vez, do HTTP).

AMQP e MQTT são, à semlhança do CoAP, dois protocolos muito utilizados em IoT. No entanto, este dois necessitam sempre da integração de um intermediário de mensagens compatível na solução. São protocolos com menor *overhead* quando comparados com HTTP, embora sejam, ainda assim, maiores do que CoAP.

Podemos ainda dizer (Naik 2017):

- **Tamanho e overhead das mensagens**: CoAP é o que tem o menor de ambos os pontos, seguido de MQTT, depois AMQP e, por último, HTTP.
- **Energia consumida e necessidade de recursos**: CoAP é o que tem o menor de ambos os pontos, seguido de MQTT, depois AMQP e, por último, HTTP.
- **Largura de banda necessária e latência**: CoAP é o que tem o menor de ambos os pontos, seguido de MQTT, depois AMQP e, por último, HTTP.
- **Fiabilidade/Qualidade do Serviço**: MQTT é o que a maior fiabilidade, seguido de AMQP, depois CoAP e, por último, HTTP.
- **Interoperabilidade**: HTTP é o que a maior interoperabilidade, seguido de CoAP, depois AMQP e, por último, MQTT.

Desta forma, serão pensadas as implementações dos protocolos, neste projeto, com se seguinte ordem de relevância:

1. CoAP;
2. MQTT;

[55] https://datatracker.ietf.org/doc/html/rfc791

3. AMQP;
4. HTTP;

Caso não seja possível em tempo útil a implementação de MQTT ou AMQP a par do CoAP, dada a semelhança desta última com HTTP, esta poderá ser também considerada para implementação no protótipo.

2.3.3 Linguagem de negócio e geração de GUI

A linguagem responsável pela geração da interface gráfica e que vai suportar as regras de negócio da aplicação, incluindo a gestão das comunicações e processamento das mensagens recebidas deve, em primeiro lugar, suportar a execução em vários sistemas operativos de forma a tornar a execução deste solução o menos limitada possível. Deve ainda ter ainda especial foco na programação assíncrona para melhor processamento de várias mensagens em tempo real. Nesta nota, *Python* será a linguagem ideal para o desenvolvimento desta solução por se adequar aos pontos indicados anteriormente e possuir, como as restantes, bibliotecas que permitem a integração com diversos protocolos de comunicação e/ou intermediários de mensagens.

2.4 Escolha de Métodos e Tecnologias

De modo a concluir sobre o enquadramento e estado de arte apresentado neste capítulo, respondemos às questões expostas na secção 2.1:

- As arquiteturas definidas para o módulo de gestão dos dispositivos é apresentada e detalhada no Capítulo 4.
- As DSL serão criadas com auxílio do MPS - a sua especificação pode ser vista com mais detalhe no Capítulo 4. Dado que é pretendido mais do que apenas a especificação da interface gráfica por via da DSL, esta deve ser criada com uma estrutura adequada, podendo não estar completamente estruturada como XAML ou XML;
- A GPL com a qual vai ser desenvolvida a solução e, ao mesmo tempo, utilizada para renderizar a GUI especificada pelo utilizador, é o *Python*. Esta escolha foi justificada na Secção 2.3.3;
- Dado que toda a solução será desenvolvida com recurso a *Python*, a framework a utilizar para o desenvolvimento da GUI a partir de especificação será *PyQt*. Este escolha deve-se ao facto de possui documentação organizada online[56] e ao mesmo tempo ser popular para desenvolvimento de GUI em *Python*;

[56] https://doc.qt.io/qtforpython

Capítulo 3

Análise de Valor

Lawrence Miles criou a análise de valor e, mais tarde, Peter Koen desenvolveu um conjunto de processos que ajudam a aferir os objetivos de inovação referentes a novos produtos ou melhorias feitas aos mesmos (P. Koen et al. 2001). Serve esta análise para validar a oportunidade identificada no contexto do problema e garantir que é gerada uma ideia que corresponda às necessidades dos seus clientes e se é possível criar valor reduzindo custos. Este capítulo descreve o processo da análise de valor seguindo o Novo Modelo de Desenvolvimento de Conceito[1] (NMDC) de Peter Koen.

3.1 Desenvolvimento do Conceito

Neste projeto, foi utilizado o NDCM que é divido em três partes chave (P. Koen et al. 2001):

- Os cinco elementos chave;
  - Identificação da oportunidade;
  - Análise da oportunidade;
  - Génese e enriquecimento da ideia;
  - Seleção da ideia;
  - Definição do conceito.
- O motor que alimenta os elementos (cultura, liderança e estratégia de negócio);
- Os fatores externos que afetam o processo de inovação e que não podem ser controlados pela empresa.

A Figura 3.1 ilustra as relações neste modelo, mostrando não ser um processo linear, o que quer dizer que as ideias e conceitos não são estáticos, podendo alternar por todos os cinco elementos chave.

[1] Traduzido do inglês *New Concept Development Model* (NCD)

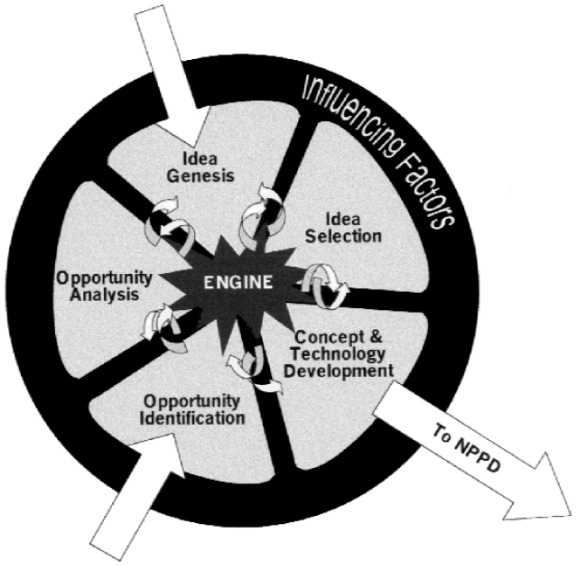

Figura 3.1: O novo modelo de desenvolvimento de conceito (NDCM) (P. Koen et al. 2001)

3.1.1 Identificação da Oportunidade

Este elemento tem por objetivo promover as oportunidades que podem vir a ser perseguidas. Estas podem ser a possibilidade de obter vantagem competitiva, ou um meio de simplificar as operações, acelerá-las ou reduzir os seus custos. A identificação de oportunidades pode ser realizada a partir do reconhecimento um problema ou uma necessidade de um cliente que pode ser resolvida (P. Koen et al. 2001).

Neste sentido, a oportunidade pode ser, por exemplo, uma possibilidade de captura de uma vantagem competitiva, uma resposta de curto prazo a uma ameaça competitiva, ou ainda um meio para reduzir, simplificar e/ou acelerar o custo de operações numa área. Pode ser tomada uma direção totalmente nova para os negócios ou apenas uma pequena melhoria num produto já existente, bem como pode ser um novo processo de fabrico, um novo produto, serviço, oferta, ou uma nova abordagem de marketing ou vendas (P. Koen et al. 2001).

Assim sendo, de forma a identificar oportunidades, podem ser feitas análises de tendências tecnológicas e definir oportunidades de melhoria de processos ou produtos que podem surgir através destas. Esta foi a técnica utilizada neste projeto.

Internet de Tudo[2]

Como já referido no capítulo anterior, existe um crescimento constante do número de dispositivos com capacidade de comunicação em rede. Este crescimento assenta, principalmente, no conceito de Internet das Coisas[3], onde existe uma comunicação direta entre estes dispositivos e as pessoas. No entanto, ao explorar esta área, cada vez mais se expande o leque de opções para a utilização de dispositivos na rede,

[2]Traduzido do inglês *"Internet of Everything (IoE)"*
[3]Traduzido do inglês *"Internet of Things (IoT)"*

conseguindo inclusive começar a criar processos de comunicação mais complexos, onde vários destes dispositivos interagem entre si para ajudar o seu utilizador em múltiplas tarefas. Aos poucos é desbloqueado um novo conceito, passando da Internet das Coisas para algo mais abrangente como a Internet de Tudo (Bhardwaj e Kole 2016).

Neste mercado emergente verificam-se oportunidades dado que, em termos tecnológicos, ainda há muitas áreas e aplicações que dispositivos comunicantes conseguem ter e que podem ser exploradas. E, como apontado também na secção 1.1 de definição do problema, existe uma cada vez maior comunidade DIY que cresce também com este aumento da oferta a preços reduzidos de dispositivos que podem utilizar nos seus projetos (Bajracharya e Hua 2020; Kuznetsov e Paulos 2010).

3.1.2 Análise da Oportunidade

Este passo do NDCM tem por objetivo a análise da oportunidade identificada no ponto anterior, de forma a validar a sua validade. Para esse efeito, foi utilizada a mesma técnica do ponto anterior, mas desta vez com o propósito de detalhar melhor a oportunidade, verificando a sua adequação e atratividade (P. Koen et al. 2001).

A comunicação e gestão de dispositivos em rede enquadra-se numa área de grande crescimento com um grande número de aplicações, como por exemplo (Chen et al. 2014):

- Aplicações do domínio industrial;
- Agricultura inteligente;
- Redes inteligentes (*Smart Grids*);
- Logísticas inteligentes;
- Transportação inteligente;
- Segurança inteligente;

Como foi mencionado na análise do problema (secção 1.2), atualmente existem no mercado diversas soluções que ajudam a gerir e/ou interagir com dispositivos na rede. No entanto, também como referido, estas tendem a ter uma ou mais desvantagens das seguintes:

- Direcionadas para produtos já existentes e com determinados protocolos de comunicação, o que dificulta a integração com novos produtos com novos protocolos (e.g. *Alexa*);
- Complexas de mais para uma utilização não profissional, isto é, são, por norma, ferramentas demasiado pesadas para projetos de pequena dimensão (e.g. *ThingsBoard*);
- Difíceis de utilizar ou pouco personalizáveis dado que por vezes o objetivo é a gestão de grandes quantidades de dispositivos, o que torna, normalmente, as

interfaces com os utilizadores mais ligadas a métricas e estados dos dispositivos, não tendo como objetivo fornecer grande experiências de utilização ao público mais leigo (e.g. *ThingsBoard*);

- Não possuem uma GUI, pois algumas das ferramentas apenas gerem os dispositivos, o que as torna apenas em ferramentas de *back-end* (e.g. *AWS IoT Events*, *Hub IoT do Azure*).

Denota-se assim a falta de uma ferramenta que consiga cobrir estes casos, ou seja, algo que permita facilitar a comunicação entre utilizadores e os dispositivos na rede, mas ao mesmo ser capaz de oferecer interfaces gráficas com a orientação que os utilizadores pretendem, seja apenas uma GUI para visualização de dados, ou que possibilite algum nível de interação através de uma comunicação bidirecional. Esta seria uma solução particularmente útil para autodidatas e comunidades DIY que pretendem explorar e mais facilmente entrar na área de IoT.

Note-se que para a criação de soluções na área de IoT é, normalmente, requerido desenvolvimento a vários níveis obrigando assim a que exista conhecimento multidisciplinar.

A oportunidade nesta área encontra-se na possibilidade de reduzir a importância dos conhecimentos necessários para as áreas de gestão de comunicações e criação de interfaces gráficas. Continua, ainda assim, a ser necessário o conhecimento na área de desenvolvimento dos próprios dispositivos terminais (dependendo do projeto) dado não ser âmbito deste projeto.

Esta é a área/mercado emergente em que este projeto se insere, tendo por objetivo a criação de uma ferramenta que permita:

- Rapidez no desenvolvimento de uma plataforma para controlo de vários dispositivos conectados;
- Oferecer a possibilidade de centralizar e gerir as informações recebidas pelos vários dispositivos terminais;
- Desenhar uma interface gráfica sem necessidade de conhecimentos nesta vertente;

3.1.3 Génese da Ideia

Criar, desenvolver e refinar uma oportunidade transformando-a numa ideia concreta passa por um processo iterativo e evolutivo onde ideias são criadas, combinadas, modificadas, atualizadas e rejeitadas (P. A. Koen, Bertels e Kleinschmidt 2014). Esta definição implica que nesta fase sejam geradas, algumas refinadas e outras rejeitadas.

Para que a criação de um ecossistema de dispositivos conectados seja criado, gerível e monitorizável, é necessário que, além da programação dos dispositivos, sejam criados os meios para gestão dos mesmos, coleta de informação e monitorização.

Hoje em dia já existem soluções no mercado que oferecem estas funcionalidades, estas são, porém, muito difícil de personalizar a um sistema específico, permitindo algum controlo mas não permitindo, por exemplo, que exista uma interface gráfica para monitorização ou controlo deste ecossistema, por exemplo, em redes fechadas. Outras desvantagens são as limitações em termos de interfaces gráficas com as quais se interage para monitorizar os dispositivos que são muito pouco ou nada personalizáveis. Em ambientes onde a versatilidade é fundamental para construir novas soluções ou soluções próprias, como acontece com autodidatas e comunidades DIY, este tipo de ferramentas pode não proporcionar as melhores integrações com projetos mais pequeno e/ou mais personalizados/específicos.

Desta forma, tendo em conta o problema e oportunidade existentes, surge a ideia, ainda que genérica, de criar uma ferramenta que permita e facilite a geração de interfaces gráficas, suportando ainda a possibilidade de estabelecer comunicações com os vários dispositivos numa rede e atualizar a interface gráfico mediante eventos/-mensagens recebidas. Após a criação desta ideia, entramos numa fase de seleção de funcionalidades de forma a melhor responder aos requisitos de futuros utilizadores.

3.1.4 Seleção da Ideia

Este elemento consiste na seleção de ideias a partir da lista de ideias geradas no ponto anterior (P. Koen et al. 2001). Neste caso apenas foram enumeradas funcionalidades desejadas para uma plataforma dado que apenas foi pensada numa ideia inicial. Desta forma, serão aqui selecionadas as funcionalidades mais importantes que fazem sentido para este projeto, tendo como principal objetivo a possibilidade de criação de um protótipo que sirva para o suprimir as necessidades existentes. Dado que foram listadas um grade número de funcionalidades, é possível que nem todas sejam exequíveis no espaço de tempo previsto para a execução deste projeto, pelo que devem ser selecionadas as mais importantes e que mais valor trazem.

Após discussão com profissionais da área de IoT e algumas sessões de *brainstorming*, foram extraídas algumas funcionalidades e comportamentos que seriam desejáveis de um produto para resolver alguns dos problemas acima expostos. São descritos na tabela 3.1 estas funcionalidades apontadas com a quantidade de pessoas, em 12, que atribuiu uma certa importância, de 1 (mais baixa) a 5 (mais alta), e a média de importância obtida.

Com base nos requisitos recolhidos e respetiva importância, foi construída uma House of Quality (HOQ), com base na Quality Function Deployment (QFD), de forma a conseguir priorizar os requisitos (Figura 3.2). Através desta foi possível determinar uma ordem de prioridades de desenvolvimento com base na sua importância calculada para o projeto, que tem por base o quão bem e quantas funcionalidades pedidas pelos experientes na área serão satisfeitas pelas várias tarefas do projeto:

1. Comunicação Segura;

Tabela 3.1: Resultados do levantamento de requisitos e respetiva importância

Funcionalidade	**1**	**2**	**3**	**4**	**5**	**Média**
Interface gráfica personalizável	0	2	2	5	3	3.75
Gestão da comunicação com diversos dispositivos simultaneamente	0	1	2	4	5	4.08
Possibilidade de seleção de protocolos para a comunicação com dispositivos distintos	0	1	4	5	2	3.67
Interface Gráfica atualizada através de eventos/mensagens recebidos	3	3	5	1	0	2.33
Interface Gráfica interativa (e.g. clique com rato, *touch*)	4	4	3	1	0	2.08
Programação de respostas automáticas a determinados eventos	1	3	4	1	3	3.17
Facilidade na especificação da interface gráfica, dispositivos, protocolos e mensagens	0	1	5	4	2	3.58
Baixo consumo de recursos	4	3	2	3	0	2.33
Taxa de Atualização de ecrã rápida	4	6	2	0	0	1.83
Comunicação Segura	0	0	3	3	6	4.25

2. Gestão da procura e conexão a dispositivos na rede, mediante especificações fornecidas;
3. Implementação dos protocolos mais relevantes, tendo em conta a segurança da comunicação;
4. Utilização de uma Linguagem Específica de Domínio para possibilitar a escrita de configurações;
5. Especificação da interface gráfica a partir de ficheiros de configuração;
6. Gestão de mensagens e eventos recebidos, com atualização da GUI gerada ou envio de resposta automática;
7. Desenho e atualização da interface gráfica mediante eventos recebidos.

3.2 Valor

O valor de um produto tem diferentes interpretações por diferentes clientes, estando normalmente ligado à relação entre o seu custo e as suas funcionalidades e características, nomeadamente as relacionadas com a sua performance, capacidade, aparência, entre outras.

No contexto de IoT, o principal valor consta na melhoria de processos de produção, melhores formas de comunicação e, fundamentalmente, proporcionar um melhor estilo de vida às pessoas no geral (Bhardwaj e Kole 2016).

3.2.1 Valor para o Cliente

Consiste na perceção pessoal de um cliente do que é uma vantagem e do que é uma desvantagem quando se associam com um determinado produto ou serviço de uma organização. Saber se um produto ou serviço é vantajoso ou não para um determinado cliente tem que ser pesado tendo em conta a combinação dos seus benefícios e sacrifícios (Woodall 2003).

Os clientes, neste projeto, sendo desenvolvedores da área estão cientes das dificuldades existentes na criação de soluções e prototipagem, principalmente na construções de sistemas que integrem numa rede vários dispositivos comunicantes e, ao mesmo, tempo criar interfaces gráficas que lidem com estes fluxos de informação. Esta dificuldade existente resume-se, como já dito na secção 3.1, à necessidade da existência de conhecimentos multi-disciplinares para o desenvolvimento de uma solução completa. Existem, por este motivo, custos associados ao tempo despendido na aprendizagem sobre o funcionamento de novas ferramentas ou linguagens, ou um custo na qualidade final do produto desenvolvido dada a falta de experiência com as mesmas.

Neste sentido o cliente tem valor acrescido com esta solução dado que, sendo dirigida principalmente a autodidatas ou comunidades DIY, facilita todo o processo de

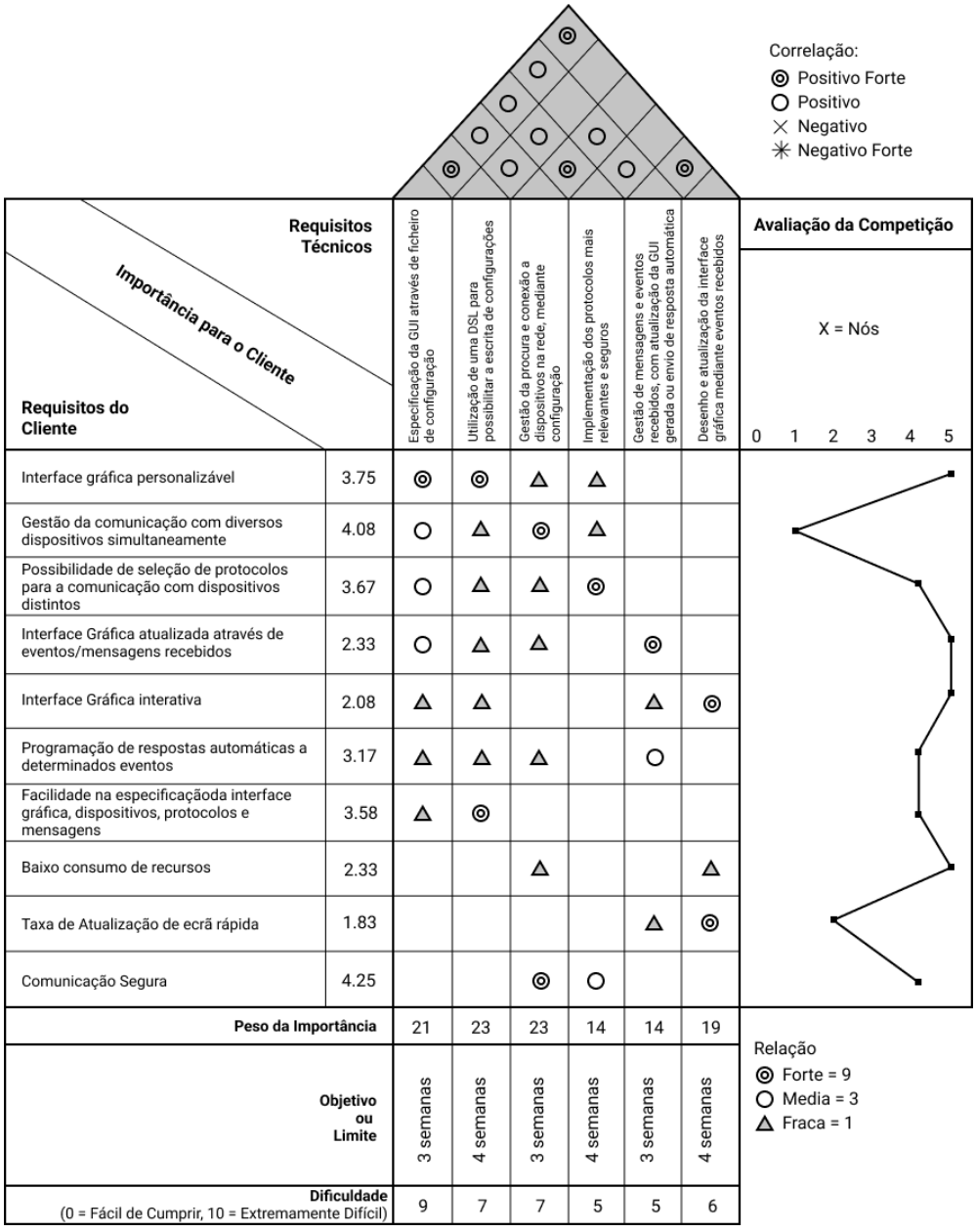

Figura 3.2: QFD - Casa de Qualidade

criação de um programa que gira comunicações e desenhe interfaces gráficas, tudo a partir de de uma configuração descrita com uma linguagem própria e acessível.

3.2.2 Valor Entendido

Diferentes clientes tem diferentes perceções de valor para os mesmos produtos ou serviços, assim como as organizações que estão envolvidas no processo de compra podem ter diferentes perceções do valor entregue pelo cliente (Ulaga e Eggert 2006). O Valor para o Cliente tem vários elementos associados, sendo que para cada um deles existem duas categorias que representam os seus benefícios e os sacrifícios associados. O benefícios dividem-se em atributos e resultados, na medida em que quando se pretende adicionar um novo atributo ao elemento, faz-se com o intuito de obter um determinado resultado. E esta adição só é possível com um sacrifício, que está normalmente associada a esta relação que implica que a geração de valor tem sempre um custo associado (Woodall 2003) (ver Tabela 3.3).

Face ao que foi referido na secção anterior (3.2.1), esta solução proporciona ao seu público alvo vários benefícios, no entanto, para criar valor são normalmente necessários sacrifícios. Os benefícios desta solução são os que se seguem:

- **Diminuição do risco** - Dado que a maior parte da lógica de gestão da informação é abstraída;
- **Poupança de tempo** - Dado que é necessário menos desenvolvimento para construção de uma solução completa;
- **Conveniência** - Pois não requer conhecimentos multi-disciplinares, permitindo que haja foco apenas nas áreas não cobertas pela solução (p.e. desenvolvimento dos próprios dispositivos terminais);
- **Personalização** - Através de um ficheiro de configuração é possível customizar a interface gráfica, dispositivos aos quais conectar, onde ir buscar determinada informação e como fazê-lo.

No entanto, existem sacrifícios que são necessários fazer de forma a proporcionar os pontos acima na solução. O facto desta ferramenta abstrair os utilizadores das problemáticas associadas ao desenho de interfaces gráficas ou à implementação da gestão das comunicações, faz com que o utilizador fique limitado pelas interfaces do programa. Isto é, em termos de GUI ou de protocolos de comunicação, o utilizador apenas pode utilizar os elementos ou protocolos (respetivamente) disponibilizados pela plataforma, estando limitados aos existentes. Embora o objetivo desta solução seja a criação de uma ferramenta versátil, a aplicabilidade a todos os casos de uso será à partida impossível por motivos vários: não serão suportados todos os protocolos de comunicação; a interface gráfica personalizável pode, ainda assim, não ser personalizável o suficiente para responder a algum caso específico do utilizador. Outro sacrifício existente é que o número de ligações a dispositivos e o a fluidez da GUI ficam limitados pelos próprios recursos disponíveis do sistema alvo.

BENEFITS		SACRIFICES
Attributes	Outcomes	
Perceived quality	Functional benefits	Price
Product quality	Utility	Market price
Quality	Use function	Monetary costs
Service quality	Aesthetic function	Financial
Technical quality	Operational benefits	Costs
Functional quality	Economy	Costs of use
Performance quality	Logistical benefits	Perceived costs
Service performance	Product benefits	Search costs
Service	Strategic benefits	Acquisition costs
Service support	Financial benefits	Opportunity costs
Special service aspects	Results for the customer	Delivery and installation costs
Additional services	Social benefits	Costs of repair
Core solution	Security	Training and maintenance costs
Customisation	Convenience	Non-monetary costs
Reliability	Enjoyment	Non-financial costs
Product characteristics	Appreciation from users	Relationship costs
Product attributes	Knowledge, humour	Psychological costs
Features	Self-expression	Time
Performance	Personal benefits	Human energy
	Association with social groups	Effort
	Affective arousal	

Figura 3.3: Benefícios e Sacrifícios (Woodall 2003)

3.2.3 Proposta de Valor

A proposta de valor deste projeto é a criação de uma ferramenta que, a partir de uma configuração, gere uma interface gráfica, estabeleça comunicação com os dispositivos na rede e permita a interação com os mesmos a partir da primeira.

Desta forma, é possível a redução da complexidade no desenvolvimento de protótipos ou soluções completas nesta área de gestão de dispositivos em rede, permitindo reduzir custos de desenvolvimento.

Esta solução distingue-se das restantes pelo nível de customização visual da interface gráfica e possibilidade de conectar dispositivos em redes locais, sendo possível fazer a atualização dinâmica da mesma consoante eventos/mensagens recebidas provenientes dos mesmos. Mantém desta forma, ainda, a possibilidade tanto de construção de uma GUI amiga do utilizador que facilite a interação com os dispositivos, mas também, dependendo do caso de uso e da preferência do utilizador, uma GUI mais orientada à monitorização de valores ou apenas com *dashboards* de monitorização de dados.

De forma a facilitar a configuração dos dispositivos na rede a este software, será ainda criado uma ferramenta para definição e configuração destes.

3.3 Modelo de Negócio CANVAS

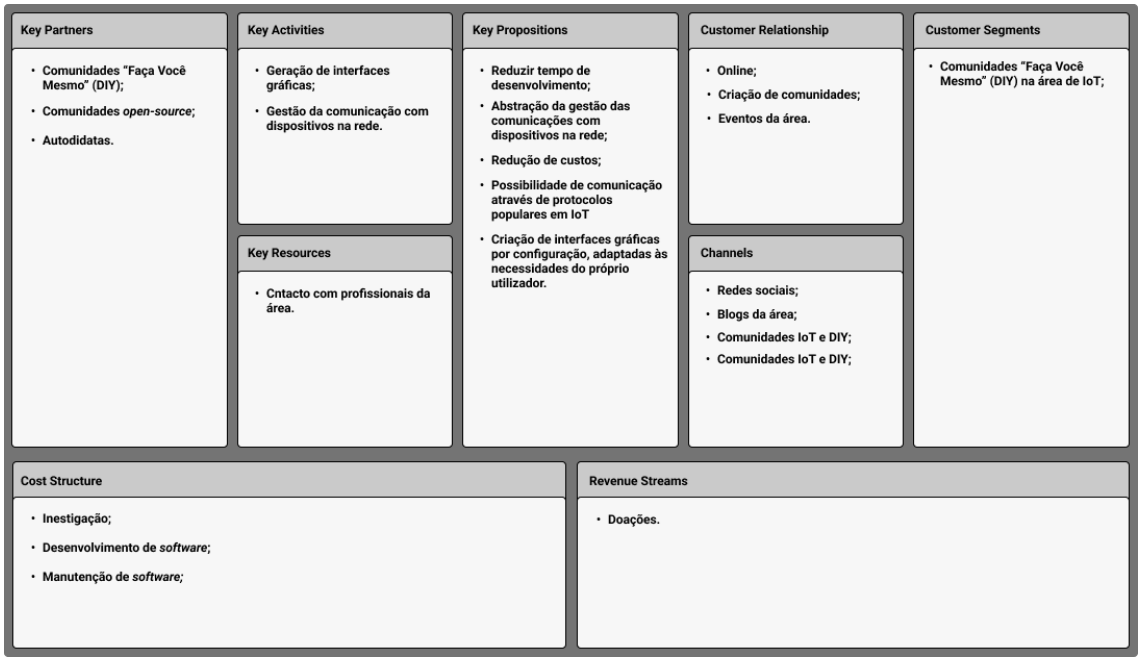

Figura 3.4: Modelo de Negócio CANVAS

Capítulo 4

Especificação da Solução

Serve este capítulo para enumerar os requisitos encontrados em fase de elicitação de requisitos enumerados na secção 3.1.4, bem como outros requisitos funcionais e não funcionais necessários.

4.1 Elicitação de Requisitos

Dado que comunidades DIY podem não ter conhecimentos na construção deste tipo de solução, também poderão não ser capazes, em princípio de definir como este deve funcionar. O foco destas comunidades é arranjar soluções para problemas que têm e, por falta de conhecimento, podem ser tomadas decisões de implementação erradas. Por esta razão, o levantamento dos requisitos foi feito em contacto com profissionais da área em conjunto com ativistas DIY, de forma a que a ferramenta que se pretende construir tenha tanto a visão como o desenho e implementação corretos e necessários. Os profissionais da área estavam responsáveis por sugerir funcionalidades e ideias de casos de uso, por outro lado, os pertencentes ao público alvo acrescentavam à discussão com os problemas que tinham e queriam resolver.

Os participantes (que perfazem um total de 12 pessoas) das sessões de levantamento de requisitos realizadas são apresentados com a sua área de conhecimento na tabela 4.1.

Tabela 4.1: Participantes das sessões de levantamento de requisitos

Área de conhecimento	**Número total**	**Número de praticantes DIY**
Sistemas embebidos	6	5
Backend e/ou *frontend* em plataformas System as a Service (SaaS)	3	3
UI/UX Designer	1	1
Aplicações Mobile	1	0
QA Tester	1	1

A enumeração destes requisitos e respetiva priorização por parte dos intervenientes foi descrita na secção 3.1.4 e na tabela 3.1, respetivamente.

Outros requisitos foram extraídos através da análise do problema e das limitações existentes noutras tecnologias e ainda da análise das ferramentas existentes na área e a sua utilização. Desta forma, foram enumeradas as funcionalidades identificadas anteriormente, iniciando este processo pela criação dos casos de uso do sistema, e expandindo para os restantes requisitos funcionais e não-funcionais. De forma a estruturar os requisitos da solução, foi utilizado o modelo *FURPS+*, que permite categorizar os requisitos recolhidos dispondo-os nas seguintes categorias (Eeles 2005):

- **Funcionalidade** - Requisitos funcionais do sistema;
- **Usabilidade** - Requisitos relativos à usabilidade da solução (e.g. prevenção de erros de *input* do utilizador);
- **Confiabilidade**[1] - Requisitos relacionados com a integridade do sistema (e.g. prevenção de falhas) ou com a conformidade de dados apresentados;
- **Performance** - Requisitos relacionados com a integridade do sistema;
- **Suportabilidade** - Requisitos que avaliam o sistema quanto à sua adaptabilidade, manutibilidade, instalabilidade, escabilidade, entre outros;
- **+** - Outras limitações impostas ao sistema como restrições ao nível do design, implementação, interfaces, ou até mesmo restrições físicas, legais ou de documentação.

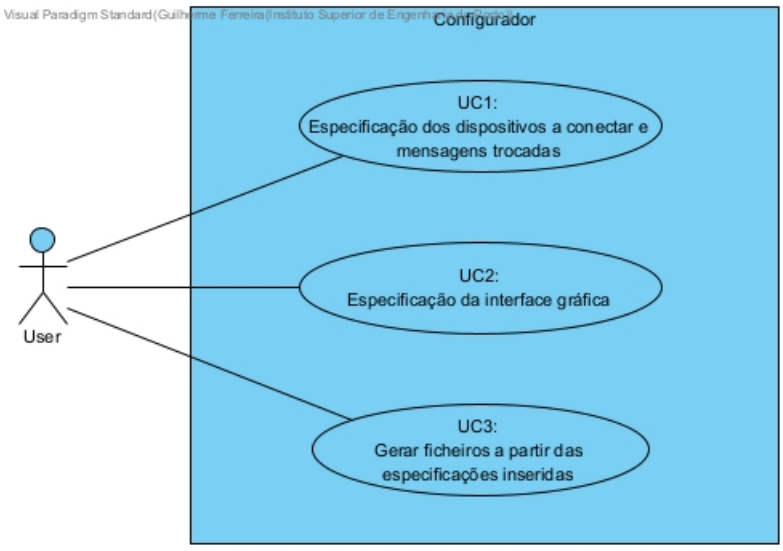

Figura 4.1: Diagrama de Casos de Uso

Foram assim encontrados, numa primeira etapa, os casos de uso da solução representados na Figura 4.1. Casos de uso representam, uma parte dos requisitos de

[1] Traduzido do inglês *Reliability*

Funcionalidade de *FURPS+* (Larman 2004). Estes representam ainda as ações que podem ser realizadas na solução. Neste caso o utilizador terá apenas interação com o sistema através da interação com o seu *Configurador*, isto é, a ferramenta utilizada (MPS) para especificar tanto as mensagens a trocar com os dispositivos, como os componentes da interface gráfica. Além destes requisitos funcionais dependentes do utilizador, existem ainda os restantes requisitos representados na Tabela 4.2, associando estes à categoria respetiva do modelo *FURPS+*. Nesta tabela foi ainda disposta a prioridade atribuída para cada item, tendo em conta as principais funcionalidades para realizar um primeiro protótipo da solução final. Os casos de uso são representados com o identificador "UC"associado ao número do requisito, enquanto que os restantes requisitos são apresentados como "RF"(requisitos funcionais, que não são casos de uso) e "RNF"(requisitos não funcionais). Os requisitos serão sempre referidos ao longo do documento dos seus identificadores (coluna "ID"nas tabela).

Tabela 4.2: Listagem de Requisitos do Sistema

ID	Categoria	Descrição	Prioridade
UC1	Funcionalidade	Especificar a informação dos dispositivos a conectar e mensagens trocadas	2
UC2	Funcionalidade	Especificar interface gráfica	1
UC3	Funcionalidade	Gerar programa a partir das especificações inseridas	1
RF4	Funcionalidade	Enviar mensagens para os dispositivos especificados	1
RF5	Funcionalidade	Extrair variáveis das mensagens recebidas dos dispositivos	1
RF6	Funcionalidade	Iniciar servidor para receção de mensagens	3
RF7	Funcionalidade	Apresentar a Interface Gráfica especificada	2
RF8	Funcionalidade	Enviar resposta automática a uma mensagem recebida	3
RF9	Funcionalidade	Persistir valores recebido na comunicação com os dispositivos	3
RF10	Funcionalidade	Atualizar automaticamente a interface gráfica gerada sempre que novos dados chegam ao sistema	1
RF11	Funcionalidade	Armazenar logs do programa localmente	2
RNF12	Confiabilidade	Implementar protocolos de comunicação que garantam a segurança da mesma	2
RNF13	Performance	Garantir uma alta disponibilidade do sistema	2
RNF14	Performance	A atualização da GUI não deve demorar mais de 2 millisegundos a executar para 25 mensagens configuradas	2
RNF15	Suportabilidade	Garantir a facilidade no acréscimo de novos protocolos de comunicação	1

São se seguida apresentadas as descrições dos diversos requisitos, sendo todos os

casos de uso descritos na forma simplificada (Larman 2004), não tendo sido adotada nenhuma nomenclatura. São também descritos os restantes requisitos funcionais seguindo uma mesma estrutura. Os requisitos não funcionais são mencionados nas funcionalidades que serão afetadas pelos mesmos. É necessário ter em conta que toda a descrição dos casos de uso que se segue se baseia nos dois componentes da solução final:

- *Configurador* - onde o utilizador realiza as especificações em UC1 e UC2, bem como realiza a geração de código a partir das mesmas no UC3.
- *Programa* - este será o nome pelo qual será identificada a ferramenta que ficará responsável por gerir as comunicações com os dispositivos, envio de mensagens, tratamento das mensagens, entre outros, representados pelos requisitos funcionais: RF4, RF5, RF6, RF7, RF8, RF9, RF10, RF11.

Os ficheiros gerados em UC3 são o *input* necessário para que o *Programa* seja executado.

Tabela 4.2: Descrição do Caso de Uso - UC1

UC1	
Caso de Uso:	Especificar a informação os dispositivos a conectar e mensagens trocadas
Dependência:	Nenhuma
Descrição:	O utilizador da solução deve conseguir especificar os dispositivos existentes na rede as mensagens que pretende que sejam trocadas entre os mesmos e o programa da solução.

Dado que pretendemos que esta solução seja utilizada em projetos DIY, a especificação não deverá ser demasiado técnica e, preferencialmente, não deverá ser necessário editar ficheiros do programa final para realizar alguma das especificações necessárias. Na especificação de dispositivos, deve ser possível inserir toda a informação necessária à realização da comunicação com o mesmo, nomeadamente, protocolo de comunicação, endereço de Internet Protocol (IP) na rede, porta de acesso, etc.. Para a especificação da Mensagem a ser trocada, deverá ser possível indicar o tipo do conteúdo da mensagem (texto, objeto *Json* ou XML), bem como uma forma para extrair um determinado valor da mesma (JsonPath para objetos *Json* ou *XPath* para XML).

A forma/estrutura da especificação deve ter em conta o requisito RNF15, devendo ser desenhado de forma a facilitar a adição de outros protocolos de comunicação.

Também deve ser tido em conta o RNF12 caso seja necessário especificar para os protocolos implementados algumas opções de segurança dos mesmos.

Tabela 4.3: Descrição do Caso de Uso - UC2

UC2	
Caso de Uso:	Especificar interface gráfica
Dependência:	UC1
Descrição:	O utilizador da solução deve conseguir especificar como quer que seja desenhada a interface gráfica do programa, bem como onde devem aparecer os valores extraídos das mensagens trocadas com os dispositivos

Numa linguagem simples, deverá ser possível especificar a disposição de componentes gráficos numa interface gráfica. Ainda, dado que o programa irá recolher valores provenientes dos vários dispositivos (através das mensagens especificadas), o utilizador deverá conseguir especificar em que componentes gráficos quer que essa informação seja apresentada.

Tabela 4.4: Descrição do Caso de Uso - UC3

UC3	
Caso de Uso:	Gerar programa a partir das especificações inseridas
Dependência:	UC1, UC2
Descrição:	O utilizador da solução deve conseguir realizar a geração dos ficheiros nas linguagens alvo a partir das especificações realizadas em UC1 e UC2.

Deve ser possível, através do *Configurador*, realizar a geração dos ficheiros finais numa linguagem alvo, a partir das especificações inseridas em UC1 e UC2. Esta geração deve resultar em dois ficheiros de configuração, um da especificação das mensagens (UC1) e outro da especificação da interface gráfica (UC2). Estes dois

ficheiros são o *input* que a solução desenvolver precisa obrigatoriamente para conseguir realizar as ações derivadas de todos os requisitos RF4, RF5, RF6, RF7, RF8, RF9 e RF10.

Tabela 4.5: Descrição do Requisito - RF4

RF4	
Requisito:	Enviar mensagens para os dispositivos especificados
Dependência:	-
Descrição:	O programa, em tempo de execução, deverá ser capaz de enviar mensagens automaticamente para os dispositivos conectados

O programa deverá ser capaz de seguir o protocolo especificado para cada mensagem para fazer o envio da mesma para o dispositivo respetivo. O tratamento da resposta e a extração de valores da mesma é da responsabilidade do RF5. O envio de mensagens deverá ser periódico segundo, sendo esta periodicidade (em segundos) especificada pelo utilizador no UC1.

A forma/estrutura da especificação deve ter em conta o requisito RNF15, devendo ser desenhado de forma a facilitar a adição de outros protocolos de comunicação e, nomeadamente, a adição da forma de envio de mensagens segundo estes novos protocolos. Deve ser tido em conta, também, o RNF12 dado que este exige que as comunicações sejam seguras, devendo ser aplicadas as configurações necessárias para todos os protocolos, de forma a que o enviodas mensagens seja seguro.

Tabela 4.6: Descrição do Requisito - RF5

RF5	
Requisito:	Extrair variáveis das mensagens recebidas dos dispositivos
Dependência:	RF4
Descrição:	O programa deverá ser capaz de, a partir da resposta a mensagens recebidas/enviadas, fazer o tratamento da mesma e extrair os valores necessários.

Como mencionado no UC1 o utilizador indica como extrair um valor da mensagem a receber de um dispotivo, incluindo, por exemplo, um XPath (no caso de XML)

ou um JSONPath (no caso de Json). Deverá ser possível utilizar esta especificação para efetuar a extração desse valor da mensagem. Todas as mensagem têm um identificador específico que deverá constar também na especificação da mesma e este irá identificar este valor lido em todo o programa. Por esta razão, o valor deverá ser guardado com esta referência ao identificador da extração da mensagem. Uma mensagem pode conter mais do que uma extração, pelo que cada extração terá o seu próprio identificador.

Tabela 4.7: Descrição do Requisito - RF6

RF6	
Requisito:	Iniciar servidor para receção de mensagens
Dependência:	-
Descrição:	O programa deverá ser capaz de iniciar um servidor que permita, segundo um protocolo definido, permitir a obtenção de um valor ou a atualização do mesmo.

O RF4 permite o envio de mensagens do dispositivo central para os dispositivos terminais, mas poderá haver casos em que seja necessário que sejam os próprios dispositivos terminais a definir quando querem que um determinado valor seja atualizado no dispositivo central. Este método pode inclusive ser útil na redução do número de pedidos na rede, diminuindo a congestão da mesma. Deve ser definido um contrato fixo para a comunicação da atualização de valores no dispositivo central.

A forma/estrutura da especificação deve ter em conta o requisito RNF15, devendo ser desenhado de forma a facilitar a adição de outros protocolos de comunicação e, nomeadamente, a adição de novas formas de receção de mensagens segundo estes novos protocolos.

Tabela 4.8: Descrição do Requisito - RF7

RF7	
Requisito:	Apresentar a Interface Gráfica especificada
Dependência:	UC2
Descrição:	O programa deverá ser capaz de apresentar uma interface gráfica que siga a especificação realizada pelo utilizador.

A interface gráfica a apresentar deve seguir as especificações realizadas no UC2. Este requisito não inclui a injeção dos valores extraídos das mensagens na interface gráfica (RF10).

Tabela 4.9: Descrição do Requisito - RF8

RF8	
Requisito:	Envio de resposta automática a uma mensagem recebida
Dependência:	-
Descrição:	O programa deverá ser capaz de realizar uma ação após receção de uma mensagem, através de especificação do utilizador.

De forma a criar uma maior interação entre os dispositivos na rede, centralizando a gestão dos mesmos no dispositivo central, deverá ser possível a configuração de ações/reações específicas após a receção de uma mensagem de um dispositivo. Por exemplo, após receção de uma mensagem que tenha uma valor extraível igual a um valor especificado, o programa deve enviar uma outra mensagem, também especificada pelo utilizador no UC1, para um outro dispositivo. Utilizando um caso real para demonstrar este exemplo, após a receção de uma temperatura acima de 28ºC, a partir de um sensor, enviar uma mensagem ao controlador do ar condicionado para o ligar e arrefecer a temperatura. A importância dada a este requisito foi baixa dado que não contribui diretamente para os principais objetivos desta fase de agregação de informação e geração de interface gráfica.

Tabela 4.10: Descrição do Requisito - RF9

RF9	
Requisito:	Persistir valores recebidos na comunicação com os dispositivos
Dependência:	RF5
Descrição:	O programa deverá ser capaz de persistir os valores extraídos das mensagens recebidas provenientes dos dispositivos conectados.

Este requisito depende diretamente do RF5 dado que os valores a persistir são os que deverão ser extraídos das mensagens. A persistência dos valores extraídos num

dado momento garante ao utilizador um histórico que pode ser consultado de todos os valores extraíveis das mensagens ao longo do tempo.

Tabela 4.11: Descrição do Requisito - RF10

RF10	
Requisito:	Atualização automática da Interface Gráfica após receção de mensagens
Dependência:	RF7
Descrição:	O programa deverá ser capaz de atualizar a interface gráfica sempre que receber uma atualização de um valor.

No RF7 é desenhada a interface gráfica a partir de componentes especificados pelo utilizador. Nessa especificação, o utilizador indica também em que componentes pretende que seja injetado o valor extraído de uma mensagem. Sempre que o valor de uma destas mensagem for atualizado, apenas esse componente deve atualizar.

Este requisito deve ter em conta do requisito não funcional RNF14, tentando que a atualização da interface gráfica cumpra o requisito estabelecido.

Tabela 4.12: Descrição do Requisito - RF11

RF11	
Requisito:	Armazenar logs do programa localmente
Dependência:	RF7
Descrição:	O programa deverá ser capaz de armazenar localmente todos os registos de logs do mesmo.

De forma a que seja possível encontrar problemas mais facilmente, todos os logs do programa devem ser persistidos localmente num ficheiro para consulta futura em casos de erros, problemas de comunicação, problemas na extração de um valor de alguma mensagem, ou outros registos que sejam efetuados pela ferramenta.

4.2 Análise e Design

Nesta secção é descrita a análise realizada ao problema abordado, bem como é apresentado o desenho da solução elaborada. De forma a representar a solução

final, foram criados diagramas com diferentes níveis de granularidade de forma a ser possível analisar gradualmente a implementação necessária.

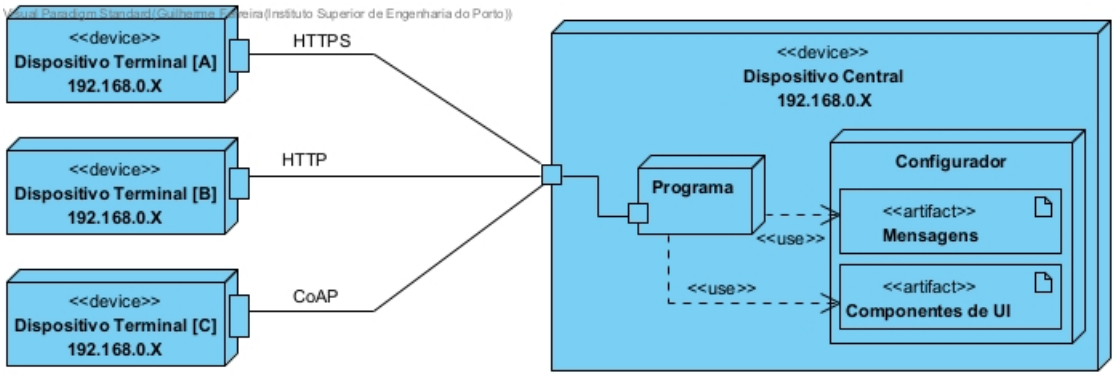

Figura 4.13: Diagrama de Implantação de alto nível

A solução a desenvolver possui dois componentes implantáveis, o *Programa* e o *Configurador*. O *Programa* tem um conjunto de sub-componentes (4.14) responsáveis que interpretam um conjunto de ficheiros estruturados, gerados pelo segundo componente, denominado como *Configurador*. O *Programa* tem como responsabilidade, com base nestes ficheiros de especificação, gerir o envio e receção de mensagens, tratar as respostas às mesmas e apresentar a interface gráfica ao utilizador. O *Configurador* tem apenas a função de ajudar o utilizador, com base na DSL criada, na especificação dos dois ficheiros necessários para que o *Programa* seja executado com sucesso.

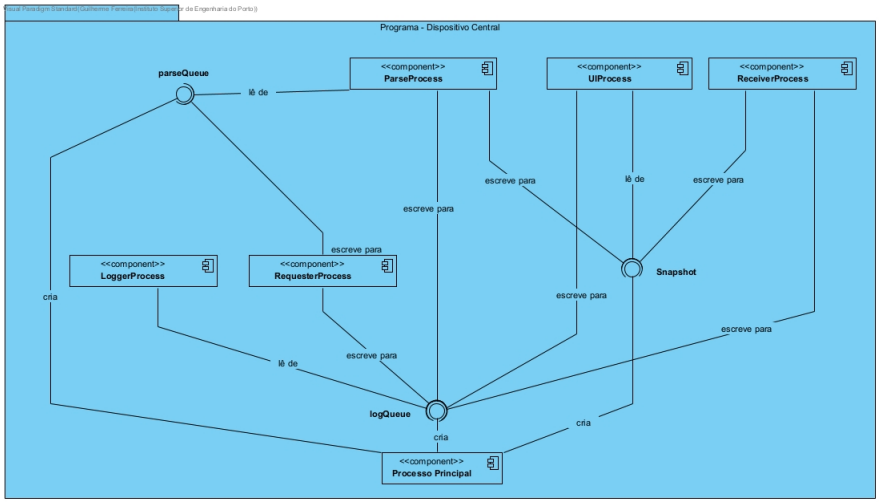

Figura 4.14: Diagrama de Componentes do *Programa*

Tendo em conta o público alvo e o objetivo deste trabalho, é esperado que este primeiro não necessite de saber desenvolver o software de nenhum dos componentes ou sub-componentes do projeto, ficando apenas encarregue de, com a ajuda do

Configurador, especificar os dados necessários para executar o *Programa*. O *Programa* é composto por diferentes sub-componentes dado que pretendemos, segundo os requisitos definidos anteriormente:

- *Requester* - Enviar mensagens aos dispositivos conectados (RF4 e RF8)
- *Parser* - Extrair valores das respostas às mensagens trocadas (RF5)
- *Receiver* / *ReceiverAPI* - Permitir receção de mensagens (RF6)
- *UI* - Apresentação de uma interface gráfica (RF7 e RF10)
- *Logger* - Persistir localmente eventos/*logs* do programa (RF11)

Todos os componentes necessitam de *inputs* comuns, isto é, os ficheiros de especificação (UC1 e UC2), no entanto, os requisitos funcionais exigem todos comportamentos que não estão dependentes entre si. A única dependência que existe entre os componentes é a passagem de dados entre o processo receção de mensagens e o processo de envio de mensagens para a interface gráfica. Que, segundo o RF5, deveria passar por um processo de tratamento para extrair variáveis dessas mensagens para apresentar na interface gráfica apenas valores tratados. Dado que os processos estão todos muito bem definidos, foram todos isolados em diferentes processos, comunicando comunicando entre si apenas os dados necessários para seguir este processamento. Ainda, todos estes requisitos representam ações que devem ser executadas em simultâneo, pelo que, à partida, já deveriam ser separados em diferentes componentes. Assim, foi feita a separação dos sub-componentes através do princípio de Separação de Preocupações[2], que tem por objetivo a separação das funcionalidades em tarefas (ou neste caso sub-componentes distintos), seguindo princípios de Responsabilidade Única e de Separação de Interfaces (2 dos 5 princípios SOLID[3]). A aplicação destes princípios tem envolve os processos de redução de acoplamento do código e o aumento da coesão.

Desta forma, dado que a execução do *Programa* depende das especificações das mensagens e da interface gráfica, é abordado na secção 4.2 a criação da DSL para depois ser analisado o desenho arquitetural do *Programa* na secção 4.2.

[2] `https://www.oreilly.com/library/view/programming-javascript-applications/9781491950289/ch05.html`

[3] `https://www.digitalocean.com/community/conceptual-articles/s-o-l-i-d-the-first-five-principles-of-object-oriented-design#dependency-inversion-principle`

Desenho da DSL

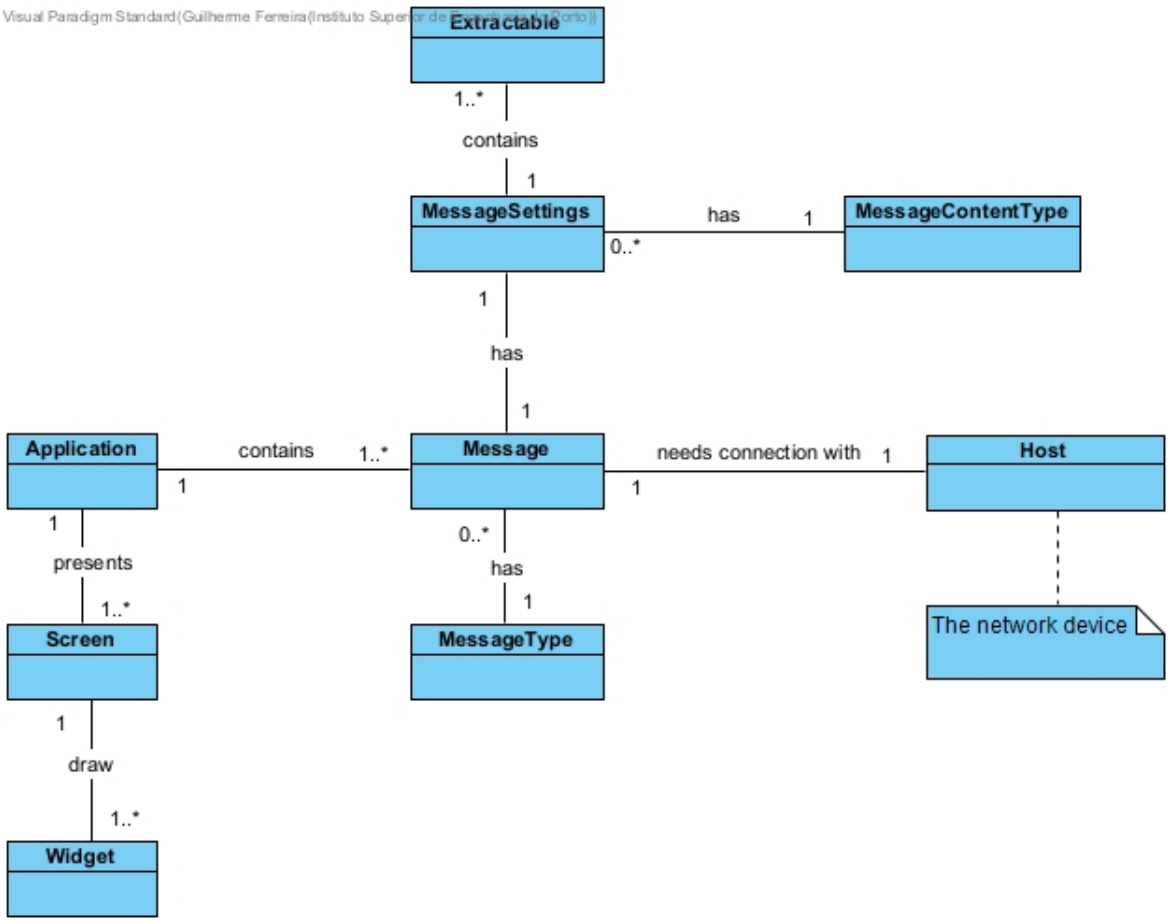

Figura 4.15: Conceitos da gramática para especificações do UC1 e UC2

Foi entendido e utilizado como linha guia para o desenho da DSL e a geração dos ficheiros de especificação que não seriam especificados mais do que um ficheiro tanto para a especificação da interface gráfica como para a especificação das mensagens a enviar para os dispositivos. Isto deveu-se ao facto de, para maior versatilidade no futuro, estas especificações serem independentes do programa em si. Isto é, os ficheiros gerados devem ter em conta o programa existente ao invés do oposto, tendo o programa que se adaptar à especificação. Desta forma, ao manter as especificações em ficheiros separados do projeto final, garantimos:

- Uma mais fácil manutenção do projeto como um todo - Dado que se trata de um projeto separado, novas funcionalidades podem ser adicionadas sem alterar os geradores da *genui*. Apenas em caso de adição de novos parâmetros às mensagens ou aos componentes de ui é que será necessário alterar a mesma.
- Que uma alteração num ficheiro de especificação nunca irá impactar o funcionamento da aplicação dado que são apenas dados de *input* ao projeto e, estando no formato correto, serão sempre interpretados corretamente.

- Que se torna mais facilitada a alteração de uma especificação, dado que basta alterar a mesma no editor (MPS) e voltar a gerar o ficheiro, substituindo-o posteriormente no projeto.

Desta forma, foi criado o modelo de domínio para uma linguagem que deve ser refletido na gramática a criar - ver 4.15. Nesta Figura temos a *Application* que contém tanto as mensagens configuradas, bem como necessita de saber como é a interface gráfica para que a consiga renderizar. Para isso, é necessário especificar as mensagens (*Message*) a enviar para os dispositivo (UC1), bem como a disposição dos componentes gráficos (*Widget*) no ecrã (*Screen*) (UC2). Serve o modelo *Application* apenas como agregador destes dois outros modelos para simbolizar a aplicação final que irá executar com base nestes (no fundo, o *Programa*). Cada *Message* da aplicação deve ter um dispositivo (*Host*) associado, sendo necessário definir toda a informação necessária para realizar a conexão. A *Message* deve ainda conter informação sobre o conteúdo esperado da resposta, incluindo o *MessageContentType* (tipo do conteúdo recebido), bem como a lista das variáveis a extrair desse conteúdo (*Extractable*). Cada mensagem tem que ter associado um *MessageType* e este será o valor que irá definir se esta será uma mensagem a enviar, receber ou que será espoletada através de um evento.

Foi criado um diagrama de classes para representar a criação dos Conceitos da DSL a criar, que teve por base o modelo de domínio da gramática da Figura 4.13. o diagrama encontra-se na versão completa e detalhada no Apêndice B e será, ao longo desta secção, fracionado e explicado.

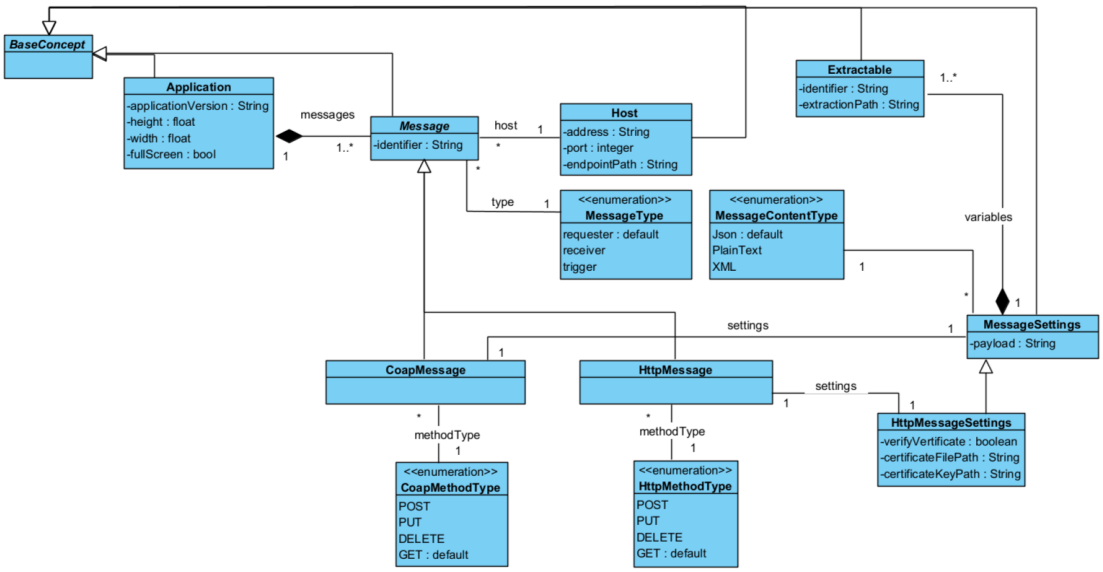

Figura 4.16: Excerto do Diagrama de Classes da gramática - focado no UC1

Na Figura 4.16 observamos as classes (conceitos) da DSL a criar para representação das mensagens a trocar com os dispositivos. A *Application* é, à semelhança do

modelo de domínio apresentado, o agregador de informação e será a partir deste que será deita a geração dos ficheiros (UC3). Neste excerto do diagrama, No excerto do diagrama apresentado, vemos que as Mensagens são um conceito abstrato que deve ser estendido para representar uma Mensagem num determinado protocolo, tendo, dependendo da implementação, diferentes propriedades possíveis ou configurações. Algo que é inerenente à mensagem e não ao protocolo em si é o dispositivo com o qual se irá comunicar (*Host*) pelo que está este associado à *Message*. O *MessageType* indica o tipo de mensagem, isto é, a função para a qual ela foi criada. Uma mensagem do tipo *requester* será enviada para o dispositivo configurado (RF4), o tipo *receiver* indica que será recebida (RF6) e o tipo *trigger* indica que será espoletada mediante uma ação ou reação (RF8 e, por exemplo, o clique de um botão). Ao criarmos esta abstração de *CoapMessage* e *HttpMessage* através da *Message* estamos a facilitar a utilização padrões *strategy* para geração e interpretação de mensagens, mantendo bons princípios de desenvolvimento de software através da abstração. Novos protocolos, futuramente, terão que seguir este mesmo padrão e estender a classe de *Message*.

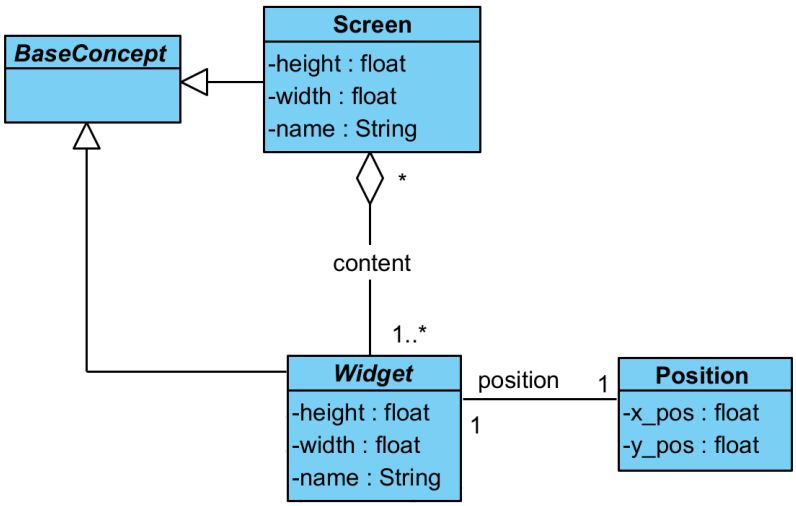

Figura 4.17: Excerto do Diagrama de Classes da gramática - focado no UC2 (especificação de um ecrã)

Na Figura 4.17 está representado o excerto do Diagrama de Classes da gramática que foca na criação de ecrãs para a GUI. A *Application*, que é o elemento comum ao UC1 como visto anteriormente, além de agregar as mensagens irá agregar também os ecrãs (*Screen*) a renderizar. Para que seja possível o utilizador especificar os componentes do ecrã, estes devem ser o mais simples possíveis, de forma a que seja possível combiná-los para obter componentes mais complexos. Para isso criamos uma classe abstrata *Widget* que representa todo e qualquer componente da interface gráfica possível de criar. Este *Widget* pode ser estendido para representar qualquer componente, como é apresentado na Figura 4.18

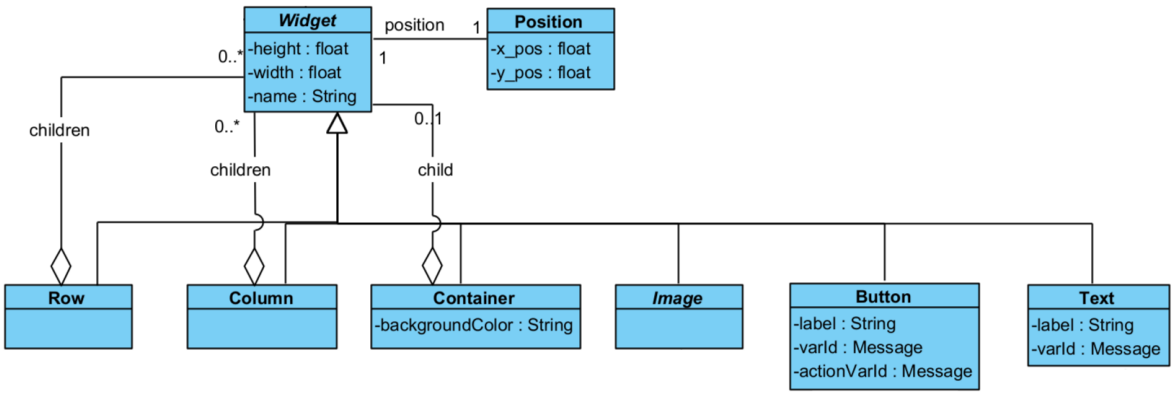

Figura 4.18: Excerto do Diagrama de Classes da gramática - focado no UC2 (Widgets)

Neste excerto da Figura 4.18 podemos ver os componentes principais a ser criados para que o utilizador consiga ter o máximo de personalização sobre a disposição da interface gráfica. Um *Widget* pode ser desde uma lista de elementos dispostos horizontalmente (*Row*) ou verticalmente (*Column*), um elemento textual (*Text*), uma imagem (*LocalImage*) ou até um botão (*Button*). O objetivo é que novos componentes a adicionar estendam sempre o *Widget* de forma a que seja sempre possível reutilizar os mesmos para qualquer parte do ecrã. Cada *Widget* tem as suas próprias propriedades e decorações que podem ser observados na versão completa do diagrama de classes.

Sub-componentes do Programa

Um dos problemas que pode existir num sistema de centralização de dados provenientes de vários dispositivos é a capacidade de processamento das mensagens recebidas. Este problema pode ser solucionado pela utilização de um *hardware* com maior capacidade de processamento (p.e. mais rápido, mais *cores*, mais eficiente), mas a construção da aplicação em si pode também ser construida tendo em conta estas possíveis limitações do ambiente de execução. Desta forma, na construção do Programa do Dispositivo Central devemos conseguir utilizar ao máximo soluções que melhorem a performance do mesmo e que tenham em conta a utilização eficiente da capacidade de processamento do sistema.

Uma das formas encontradas para melhorar a performance de uma aplicação foi arquitetar a mesma com foco no processamento paralelo, através da utilização de *Processos* e *Threads*. Processos são programas em execução na Unidade Central de Processamento (UCP) que, aquando da sua criação, lhes são alocados recursos individuais, isto é, não há partilha de *stack*, dados ou código. Por este motivo, processos não partilham, por defeito, memória com outros Processos do sistema. *Threads*, por outro lado, são parte de um processo, partilhando a sua memória com o processo pai e restantes *threads* desse mesmo processo (Stevens 1999), como apresentado na Figura 4.19.

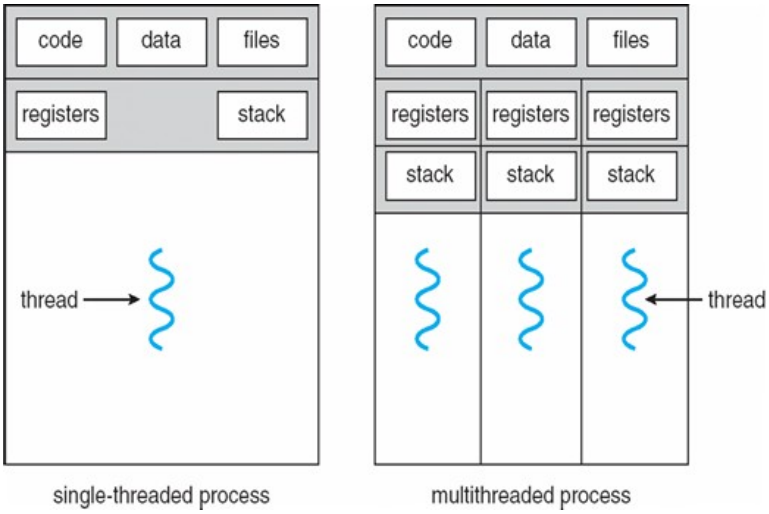

Figura 4.19: Processo de *thread* única Vs. Processo de multiplas *threads*[4]

O grande benefício de aplicações multi-processo e/ou com multi-*threads* é que o processamento de uma *thread* ou processo não impacta as restantes dado que o seu processamento é paralelo. Esta vantagem é útil na nossa solução dado que esta pretende implementar várias componentes em simultâneo, nomeadamente: receção de mensagens; envio de mensagens; processamento de respostas; renderização e atualização de uma interface gráfica. Cada um destes componentes a implementar deve ser criado em processos separados de forma a garantir que cada um tem apenas a informação que necessita para realizar as suas tarefas, bem como para garantir que cada um tem os seus recursos individualmente alocados. Para operações repetitivas como o envio periódico de mensagens (RF4), o envio de cada mensagem deve estar separado em diferentes *threads* de forma a que um pedido não bloqueie um outro. Foi representado na Figura 4.20 o diagrama de classes completo para o *Programa*, onde os sub-componentes do mesmo são representados como *Workers*. Um *Worker*, nesta solução é entendido como uma classe que executa apenas uma tarefa, sendo criado para representar cada sub-componente do *Programa*. Cada *Worker* deve correr num processo distinto das restantes de forma a que sejam executados assincronamente.

[4] `https://slideplayer.com/slide/4402210`, acedido a 14 Agosto 2022

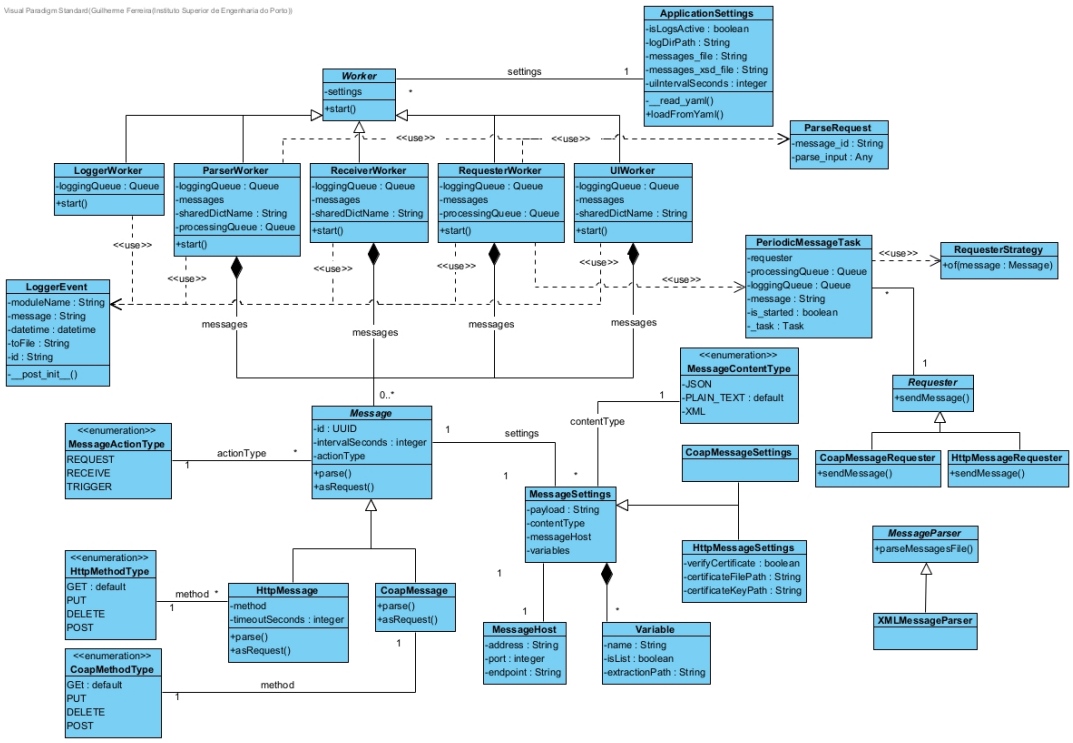

Figura 4.20: *Programa* - Diagrama de Classes

Desta forma, foi representada na Figura 4.21 a sequências de ações esperadas do processo principal do Programa para a criação dos diferentes processos e *Workers* para cada componente da solução.

Cada um dos sub-componentes do *Programa* tem responsabilidades concretas, devendo apenas ser partilhado entre todos uma forma de transmitir dados aos restantes. Esta comunicação pode ser feita através da utilização de *Queues* de eventos, partilhadas pelos componentes, sendo que, desta forma, todos os processos que tenham acesso às mesmas podem:

- Fazer um pedido de registo de *log* enviando um evento para a *Queue* de logs;
- Fazer um pedido de processamento de uma resposta recebida a uma mensagem enviada através de uma *Queue* de eventos de processamento;

A utilização de *Queues* para estes casos é feita com a intenção de que um evento seja processado em modo "Primeiro a entrar, Primeiro a sair"[5] (que garante que serão processados na ordem em que foram adicionados à *queue*) e porque cada evento deve ser processado apenas uma vez, sendo removido da *Queue* assim que iniciar o seu processamento. Desta forma garantimos que será futuramente possível ter várias instâncias de *Logger* ou *Parser* se for necessário que existam mais agentes a tratar destas tarefas - poderá ser útil com o aumento do número de mensagens a enviar/receber e será analisado na secção de Avaliação e Experimentação esta

[5]Traduzido do inglês *First In, First Out* (FIFO)

necessidade. Estas *Queues* para os dois casos acima estão representadas como *logQueue* e *parseQueue* na Figura 4.21.

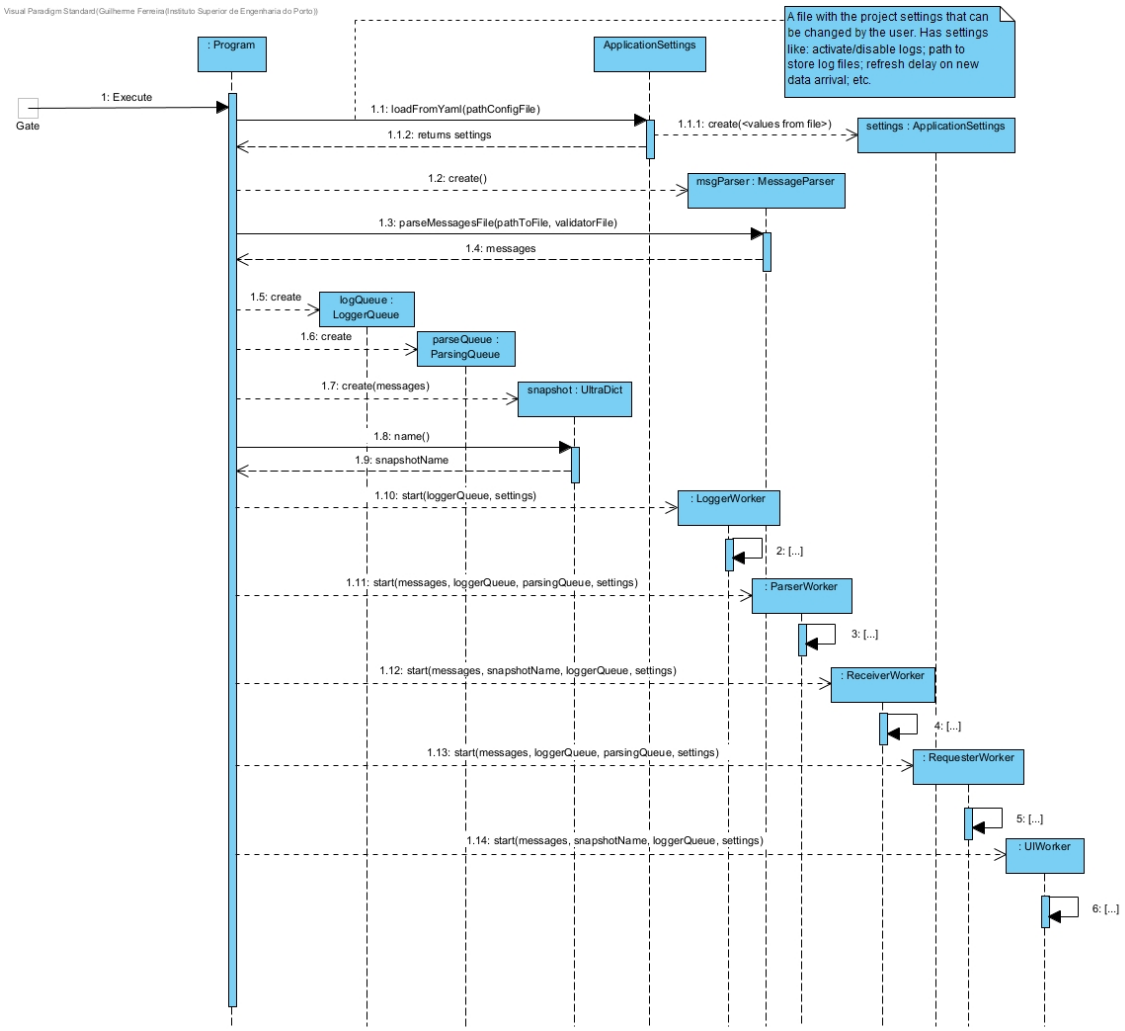

Figura 4.21: Diagrama de Sequência do processo principal da solução

Ainda, para a realização da RF7 e RF10, de forma a apresentar dados na interface gráfica e atualizar a mesma sempre que esses dados forem atualizados, é necessário que estes mesmos sejam armazenados em memória. Para tal, deve haver memória partilhada com a informação de todas as variáveis (especificadas na UC1) e os seus valores mais atuais (atualizados através da RF5 e RF6), que a interface gráfica consiga aceder para apresentar os mesmos ao utilizador. A utilização de uma *Queue* para este efeito implica que a interface gráfica, na RF10, atualize os valores sequencialmente, não conseguindo atualizar todos os valores apresentados de uma vez (a menos que exista apenas um valor a apresentar). Dado que a solução deve ter em vista a receção de mensagens de vários dispositivos, deve também ter em conta a atualizar se múltiplos valores em simultâneo. Assim, sempre que é recebido um novo valor de um dispositivo conectado, deve ser feita a criação de uma *snapshot*

com todos os valores atualizados. Posteriormente, a interface gráfica poderá subscrever esta *snapshot* e atualizar-se mediante as alterações da mesma. Para diminuir a sobrecarga de atualizações de toda a interface gráfica sempre que a *snapshot* é alterada, apenas o componente que com o valor desatualizado deve ser atualizado (e não todos os componentes do ecrã). Este último ponto pode ser resolvido através da implementação de um padrão publicação-subscrição[6] (ver Figura 4.22), onde cada componente subscreve apenas o valor que apresenta, forçando a sua própria atualização. A *snapshot* partilhada está representada como *snapshot* na Figura 4.21

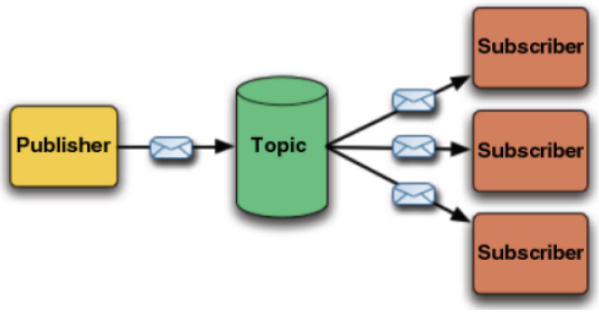

Figura 4.22: Exemplo do padrão de publicação-subscrição[7]

A representação dos componentes da solução e a sua ligação a estes endereços de memória partilhados (sejam *queues* ou a *snapshot* de dados), estão representados na Figura 4.14

Envio de mensagens (RF4)

De forma a realizar periodicamente o envio de mensagens segundo a especificação do utilizador (UC1), é necessário que esse processo seja criado e que seja passada a informação das mensagens a enviar para o mesmo. Com base na informação das mensagens a enviar, deve ser criado um processo de envio periódico de cada uma tendo em conta que um atraso na resposta ou no processamento de uma delas não impacte o envio das restantes. Neste sentido, dado que utilizamos Python para desenvolver esta solução, podemos utilizar a biblioteca *asyncio*[8] para executar um conjunto de tarefas (envio de mensagem) de forma assíncrona. O método de envio de mensagem, isto é, a tarefa em si, pode não ser igual para todas as mensagens dado que depende do protocolo a utilizar para realizar o envio. A escolha do método de envio é feita a partir do tipo de mensagem especificada pelo utilizador no UC1. O desenho desta sequência de ações necessárias para que o processo de envio de mensagens crie as tarefas de envio e as execute, está representado na Figura 4.23.

[6] Traduzido do inglês *Publish-Subscribe*

[7] `https://vitolavecchia.altervista.org/caratteristiche-di-un-sistema-publish-subscribe-in-informat` acedido a 16 Agosto 2022

[8] `https://docs.python.org/3/library/asyncio.html`

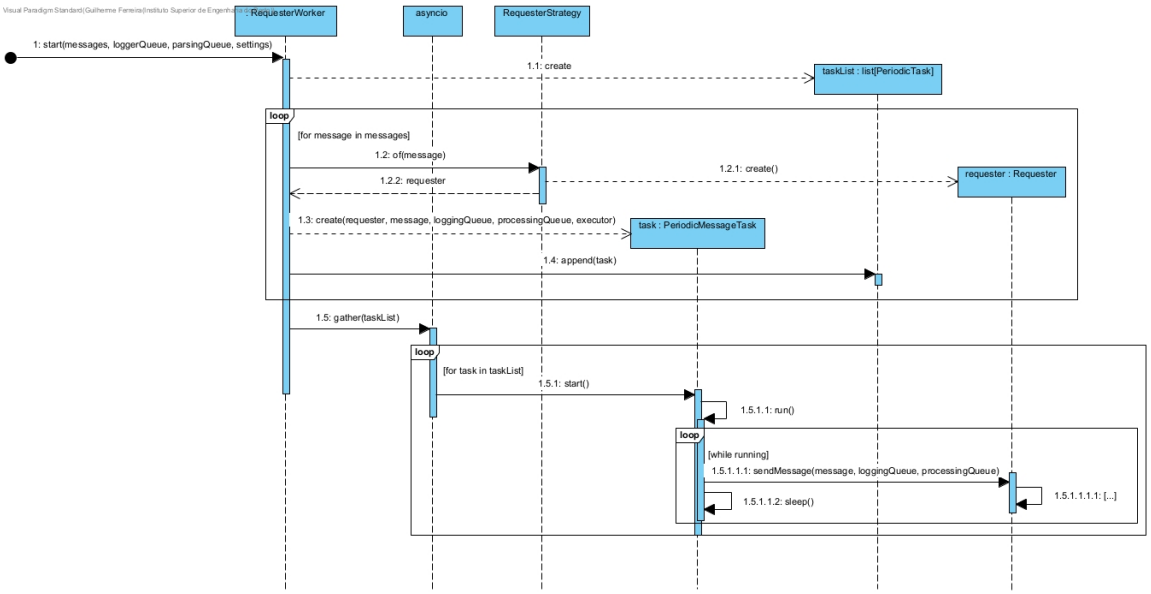

Figura 4.23: Diagrama de Sequência do *RequesterProcess*

Extrair valores a partir de mensagens enviadas (RF5)

O processo de extração de valores de mensagens enviadas, ParseProcess (Figura 4.14), deve escutar constantemente a *parseQueue*, isto é, a *queue* utilizada pelo componente de envio de mensagens sempre que recebe a resposta à mesma. Sempre que um novo evento for adicionado a esta *queue*, este deve ser processado, por ordem de chegada, extraindo os valores segunda a especificação inicial do utilizador (UC1). Cada valor especificado pelo utilizador deve ser extraído da mensagem recebida. Dado que a mensagem pode ser de vários tipos (texto livre, Json, XML, ou outros), na altura da especificação, esta deve ser uma das informações definidas pelo utilizador, sendo que este deve:

- Json - Deve ser especificado adicionalmente um caminho JsonPath[9] para a variável ou lista de variáveis a extrair.
- XML - Deve ser especificado adicionalmente um caminho XPath[10] para a variável ou lista de variáveis a extrair.
- Texto livre (*PlainText*) - Todo o conteúdo será considerado como sendo já o valor processado, sendo este adicionado diretamente como sendo o valor final.

No caso de Json e XML, apenas após o processamento do conteúdo segundo o JsonPath ou XPath especificado é que o resultado da extração será adicionado à variável associada a esse valor. Toda esta sequência de ações está representada no diagrama da Figura 4.24.

[9] `https://github.com/json-path/JsonPath`
[10] `https://developer.mozilla.org/en-US/docs/Web/XPath`

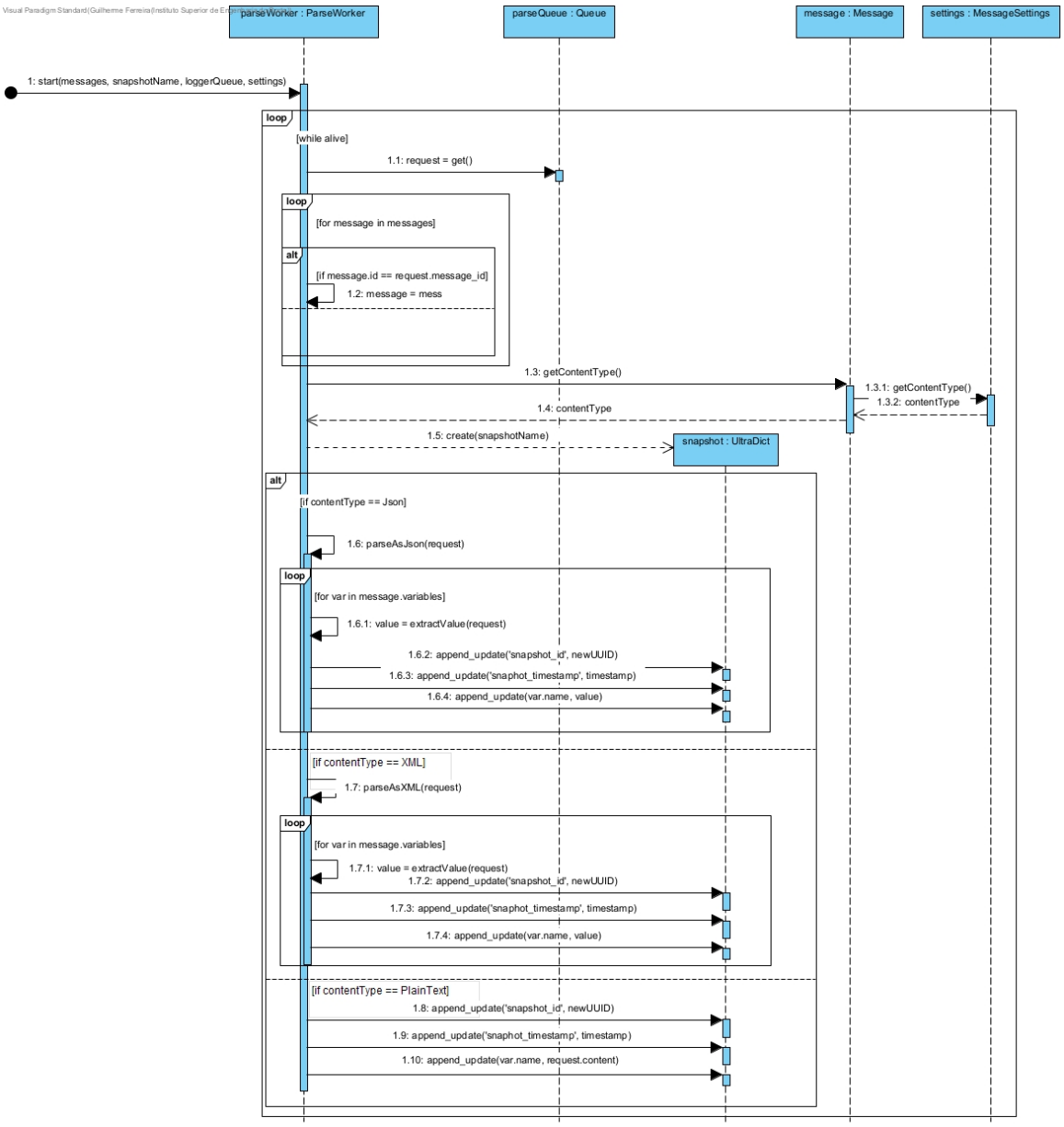

Figura 4.24: Diagrama de Sequência do *ParseProcess*

Interface Gráfica (RF7 e RF10)

Através de especificação do utilizador, deve ser apresentada uma interface gráfica (UC2) e, para isso, é criado um processo *UIProcess* (Figura 4.14) que deve renderizar os componentes gráficos especificados pelo utilizador. Como apresentado na Figura B.1, um ecrã (*Screen*) da *Application* é constituido por *Widgets* que podem ser de vários tipos:

- *Row* - Alinha os componentes que o constituem horizontalmente;
- *Column* - Alinha os componentes que o constituem verticalmente;
- *Label* - Representa um componente textual;

- *Button* - Representa um botão;
- *Image* - Apresenta uma imagem armazenada localmente;
- *Container* - Representa um contentor de um outro objeto, pode ser utilizado para dimensionar, positionar ou estilizar o espaço em volta do objeto que o compõe.

Estes componentes são os mais basicos e podem, quando utilizados em conjunto, criar diversos outros *Widgets* mais complexos. Por exemplo, uma tabela poderá ser uma *Row* constituída por *Columns*, por sua vez constituida por *Labels*.

A interface gráfica, ao contrário da especificação das mensagens, estará sempre mais ligada ao programa final, no sentido em que existe uma *framework* de GUI (e.g. PyGUI[11], Tkinter[12], PyQt[13]) na linguagem alvo que deve ser utilizada. Neste sentido, para que a *Ide* seja o mais independente possível da *framework* a utilizar, foi criada uma *framework* genérica e adaptada à DSL, com a qual seja possível implementar uma *framework* de GUI. Deve ser criada a mesma estrutura de Conceitos da *Ide* em classes Python, onde todas possuem um método de *asComponent()*, onde deverá constar a lógica de construção da própria classe (conceito) no componente de GUI respetivo. Desta forma, seria possível adaptar a solução a funcionar com outras *frameworks* no futuro, desde que utilizadas a mesma estrutura de classes com uma implementação diferente da função *asComponent()* - ver Figura 4.25. Como *framework* de GUI para desenvolvimento da solução foi utilizado o PyQt5 dado que foi a ferramenta que mostrou ser de mais fácil utilização, mantendo compatibilidade em várias plataformas (*Windows*, *MacOS* e *Linux*).

[11] `https://www.csse.canterbury.ac.nz/greg.ewing/python_gui/version/Doc/index.html`

[12] `https://docs.python.org/3/library/tkinter.html`

[13] `https://pypi.org/project/PyQt5/`

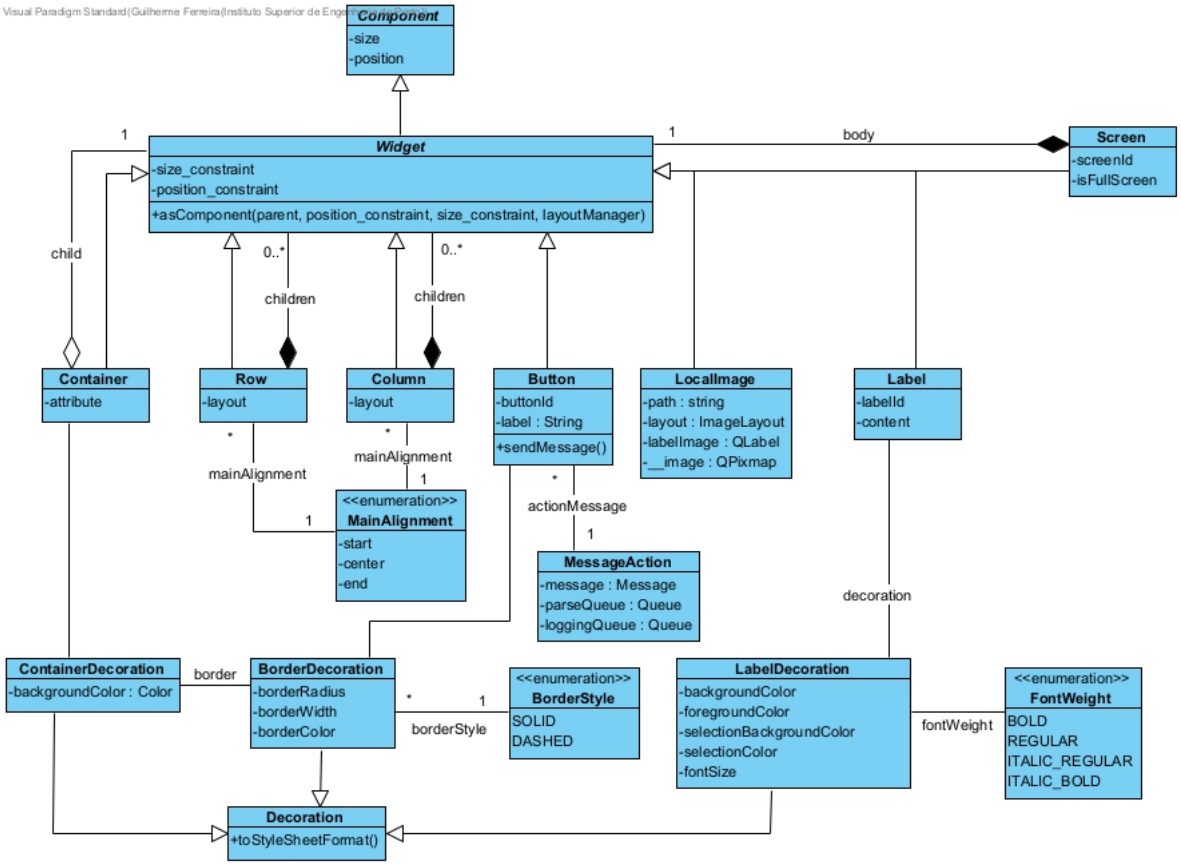

Figura 4.25: Diagrama de classes dos componentes da interface gráfica a gerar

Dado que pretendemos ainda que os componentes do ecrã sejam atualizados sempre que novos valores sejam obtidos (derivados dos RF4, RF5 e RF6), é necessário que estes subscrevam alterações aos valores que pretendem apresentar. Por exemplo, uma *Label* que pretenda apresentar uma variável "VarX", deve subscrever este tópico e atualizar-se automaticamente quando um novo valor é recebido nesse tópico (RF7). Para tal, cada *Widget* que contenha um texto (*Label* e *Button* numa primeira fase), devem conseguir, na sua inicialização, subscrever um tópico e atualizar-se com novos eventos.

Toda esta sequência de ações deste processo está representada no diagrama da Figura 4.26.

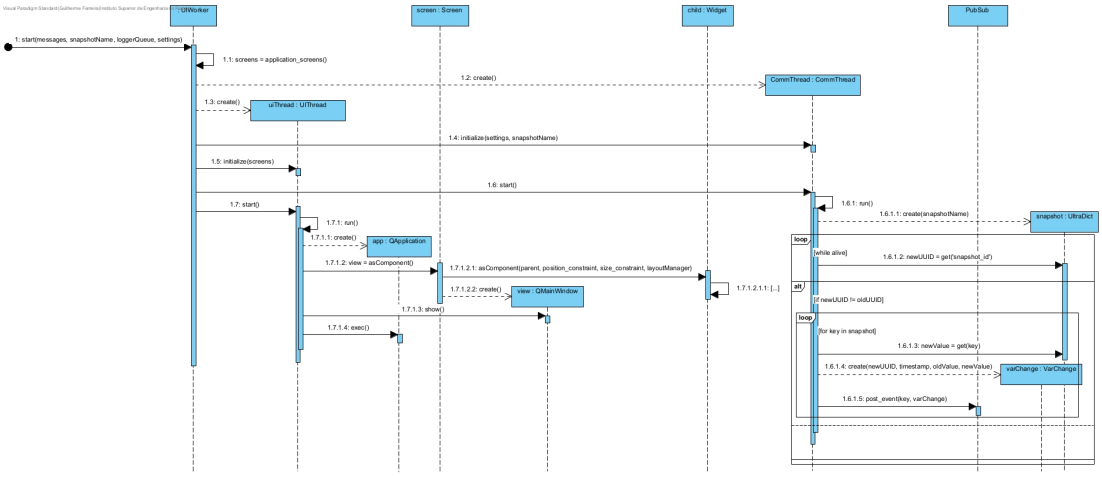

Figura 4.26: Diagrama de Sequência do *UIProcess*

Como apresentado na Figura 4.26, de forma a evitar que a escuta de novos valores possa influenciar o desempenho da interface gráfica, foi separado esse processamento numa *Thread* à parte. Desta forma, enquanto que a *CommThread* trata da escuta de novos valores na *snapshot*, a *UIThread* tem a responsabilidade apenas de renderizar os componentes gráficos.

Registo de logs (RF11)

O processo de registo de *logs* da solução deve escutar constantemente a *logQueue* (Figura 4.14), isto é, a *queue* utilizada pelos diversos componentes para pedir a persistência de uma linha de *log* em ficheiro. Sempre que um novo evento for adicionado a esta *queue*, este deve ser processado, por ordem de chegada, pelo processo de registo de *logs*, adicionando esse registo ao ficheiro de *logs*. Esta sequência de ações está representada no diagrama da Figura 4.27.

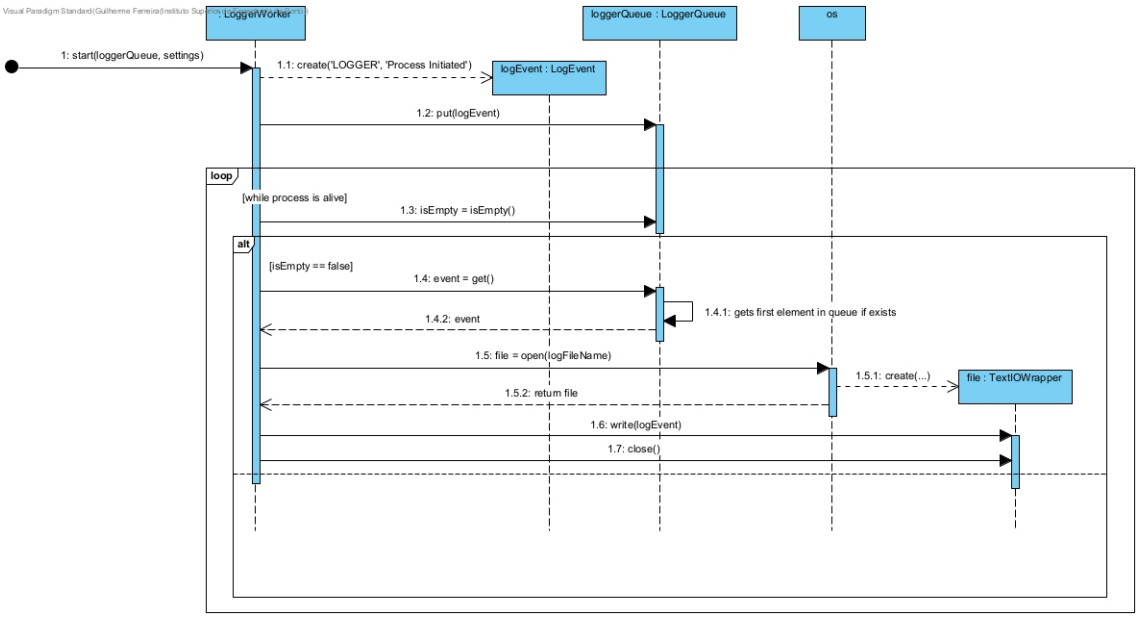

Figura 4.27: Diagrama de Sequência do *LoggerProcess*

Capítulo 5

Construção da Solução

Este capítulo descreve o processo de desenvolvimento da solução analisada na secção anterior. Inclui o desenvolvimento da DSL, a geração do código, e o desenvolvimento do *Programa* (Figura 4.14). O editor de código utilizado para desenvolvimento do *Programa* foi o Visual Code, e a DSL foi criada utilizando o MPS. O nome da DSL criada é "*genui*", sendo este o nome, daqui em diante no documento, que irá ser utilizado para referenciar a mesma. As secções dentro deste capítulo organizam-se da seguinte forma:

5.1 Criação da DSL - Apresenta os conceitos criados da *genui*, incluindo exemplos de especificações possíveis através desta linguagem.

5.2 Geradores de Código - Apresenta a implementação dos geradores de código de conceitos da *genui* para os ficheiros de especificação a ser lidos pelo programa do dispositivo central.

5.3 Programa do dispositivo central - Apresenta os desenvolvimentos feitos para o programa que é executado no dispositivo central, organizado por Processos.

5.4 Resultados da Implementação - Relaciona os desenvolvimentos apresentados nas secções anteriores com os requisitos apresentados no Capítulo 4.

Toda a implementação teve como ordem de desenvolvimento as prioridades definidas na Tabela 4.2, iniciando o mesmo pelos requisitos com prioridade 1, deixando os de prioridade 3 para o final do desenvolvimento.

5.1 Criação da DSL

Através do MPS foi criado um novo projeto para definição da nova DSL (Figura 5.1).

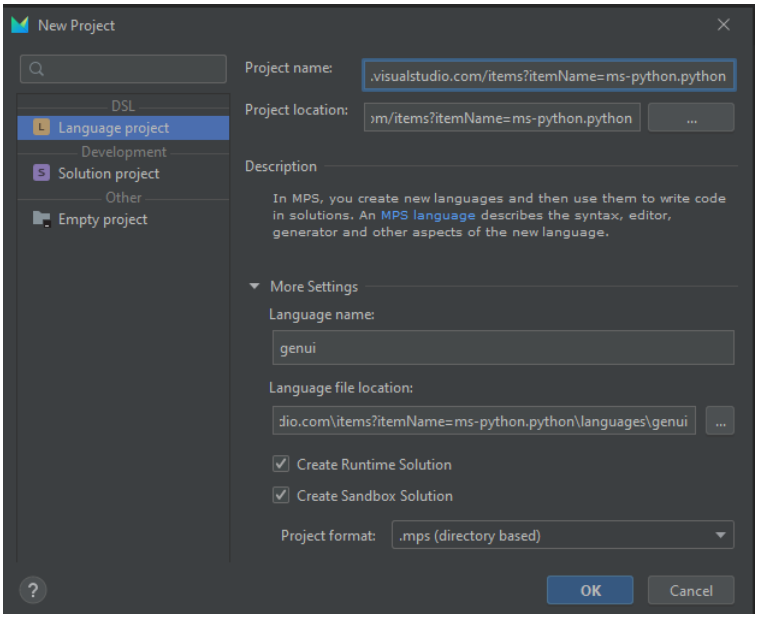

Figura 5.1: MPS - Criação de Novo Projeto

Num projeto em MPS, a organização dos ficheiros é a que se encontra na Figura 5.2, onde[1]:

- *genui.structure* - Contem as descrições dos nós (Conceitos) que compõe a estrutura da linguagem. São aqui colocados os ficheiros que declaram os Conceitos de, por exemplo, *Application*, *Screen*, *Message*, *Widget*, entre outros;
- *genui.editor* - Contem a forma como a linguagem será apresentada ao utilizador;
- *genui.constraints* - Contem as restrições da AST, sito é, onde um nó (conceito) é aplicável, propriedades e referências que são permitidas, etc.;
- *genui.behavior* - Contem a descrição dos aspetos comportamentais da AST;
- *genui.feedback* - Contem a definição das customizações de *feedback* do editor, incluindo *autocomplete* da linguagem;
- *genui.typesystem* - Contem a definição das regras que são utilizadas para calcular os tipos na linguagem.
- *genui.generator* - Contem a definição dos geradores para a linguagem (*genui* neste caso).

[1] `https://www.jetbrains.com/help/mps/mps-project-structure.html#modules`

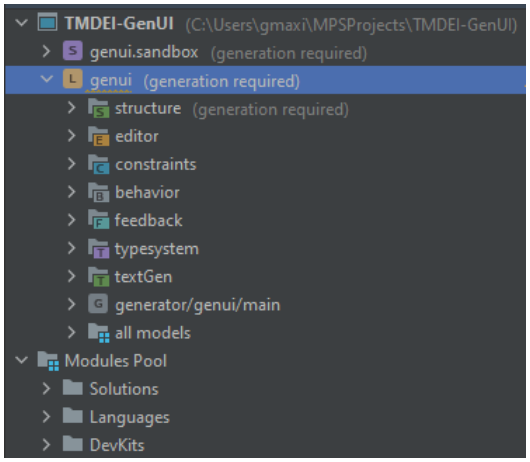

Figura 5.2: MPS - Estrutura do Projeto

Neste sentido, foram criados os Conceitos da linguagem no diretório *structure*, segundo o diagrama já exposto dos conceitos da *genui* na Figura B.1. Nas secções de 5.1.1 a 5.1.4, serão expostos alguns dos conceitos mais relevantes da *genui*.

5.1.1 Conceito Application

A *Application* tem que ser inicializada obrigatoriamente para que seja possível associar à mesma toda a informação relativa às *Message* (mensagens a trocar com os dispositivos) e *Screen* (ecrãs e componentes da interface gráfica). Para isso, é necessário criar este conceito, adicionando como *children* (adicionar uma relação com) as *Message* e *Screen*, como apresentado na Figura 5.3

```
Application
concept Application extends    BaseConcept
                    implements INamedConcept

  instance can be root: true
  alias: <no alias>
  short description: The application to be generated.

  properties:
  @doc  The application string version. Must have a "<number>.<number>.<number>" format.
  applicationVersion : string
  @doc  If application should open in full screen. If not, `height` and `width` will be used instead.
  fullScreen : boolean
  @doc  The application screen height. Ignored if `fullScreen` option is True.
  height : _FPNumber_String
  @doc  The application screen width. Ignored if `fullScreen` option is True.
  width : _FPNumber_String

  children:
  @doc  The messages specification.
  messages : Message[1..n]
  @doc  The user interface specification.
  screens : Screen[1..n]

  references:
  << ... >>
```

Figura 5.3: *genui* - Conceito *Application*

Foi ainda adicionado um editor para este conceito de forma a personalizar a apresentação do mesmo em editor para quando o utilizador o estiver a introduzir (Figura 5.4).

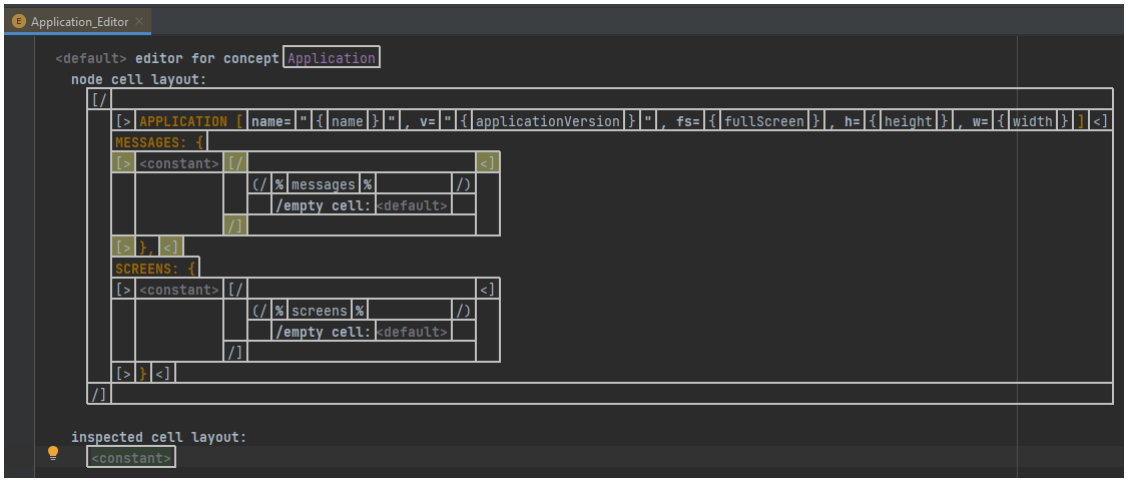

Figura 5.4: *genui* - Editor para conceito *Application*

Este editor pode ser visualizando através da criação de uma instância de *Application* num projeto de *sandbox*, e este é apresentado, sem nenhum dado introduzido, como se encontra na Figura 5.5.

Figura 5.5: *genui* - Especificação do conceito *Application*

Para validar um formato standard de versionamento, a *version* (Versão) da aplicação tem uma *constraint* (restrição) associada que indica o formato obrigatório da mesma (Figura 5.6).

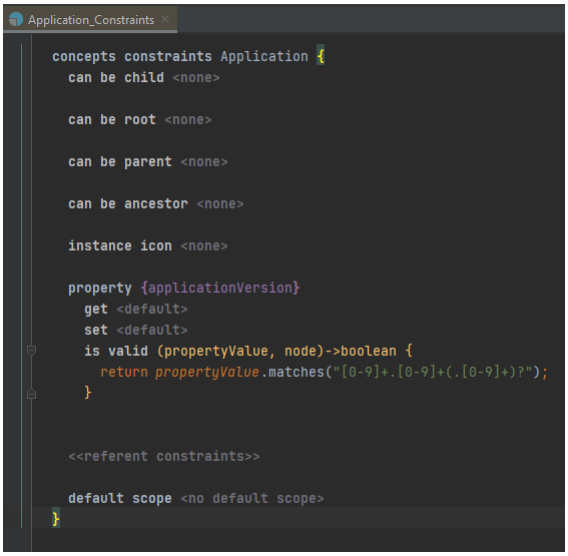

Figura 5.6: *genui* - Restrição do formato da versão da *Application*

5.1.2 Conceito Screen

Um *Screen* representa um ecrã da interface gráfica, e, por esta razão, deve conter todos os componentes gráficos que serão representados no mesmo. Na Figura 5.7 está exposto o conceito *Screen* da *genui*. O parâmetro *child* do *Screen* é um componente gráfico genérico (*Widget*) que será detalhado na secção 5.1.3.

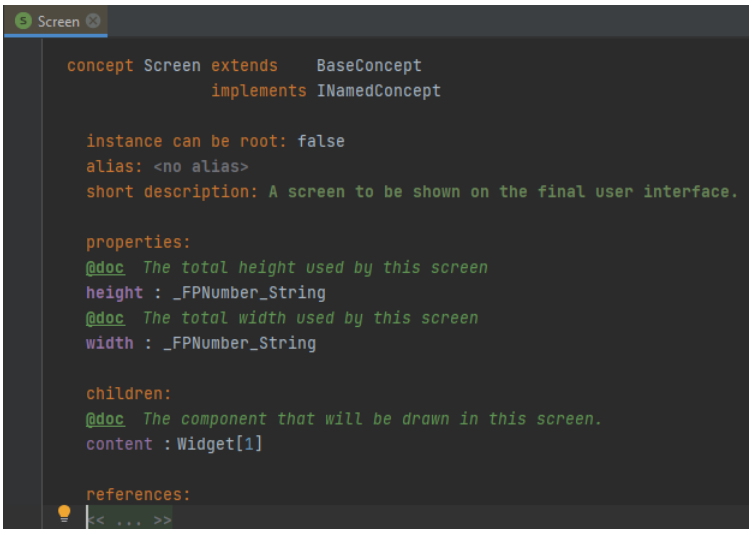

Figura 5.7: *genui* - Conceito *Screen*

O objetivo deste conceito é funcionar como separador de um componente ou de um grupo de componentes em diferentes ecrãs, pela atribuição deste conjunto a diferentes *Screen*. Um *Widget* é um qualquer componente gráfico, seja simples, como uma imagem, ou mais complexo, sendo uma composição de vários elementos (p.e. conjunto de imagens ou textos).

5.1.3 Conceito Widget

Um *Widget* é um conceito abstrato e não é possível criar instâncias do mesmo. Cada componente da interface gráfica deve estender este conceito (p.e. Textos, Imagens, botões). O conceito *Widget* está representado na Figura 5.8.

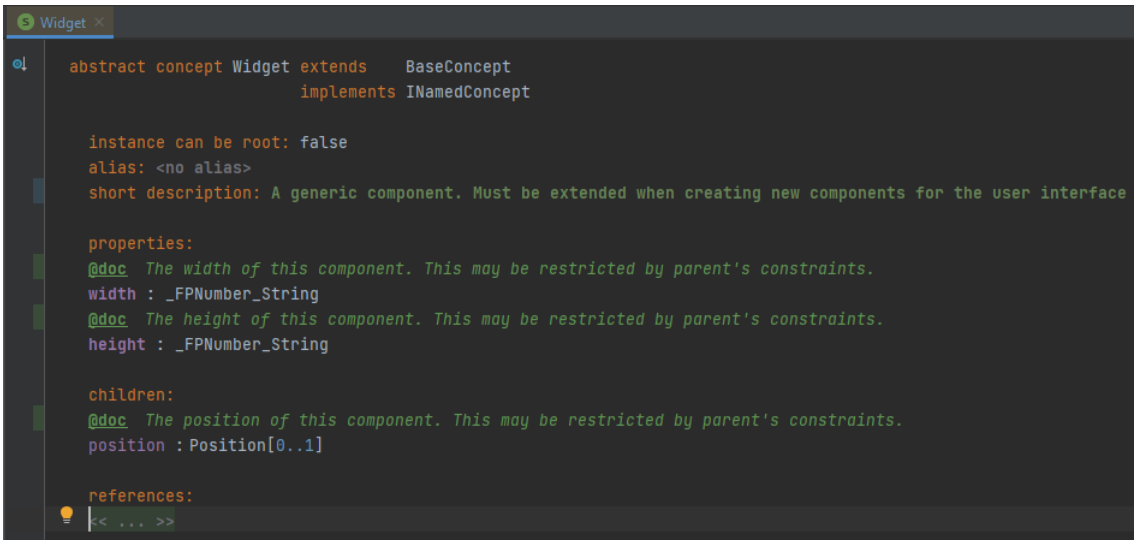

Figura 5.8: *genui* - Conceito *Widget*

Dado que um *Widget* representa um conceito da interface gráfica, deve possuir, sempre, um tamanho (*size*) e uma posição (*position*) que deverão ser utilizados para desenhar este componente no local correto do ecrã e com as dimensões corretas.

Os componentes criados foram (ver Figura B.1):

- *Container* - Representa um contentor de um outro widget. Pode ser utilizado para forçar um determinado tamanho e posição ao *Widget* nele contido, bem como um determinado estilo (cor de fundo, limites, etc.);
- *Text* - Representa um elemento textual no ecrã. Este componente apresenta um texto, definido pelo utilizador, no ecrã. Pode ser decorado com tamanho da fonte, cor de fundo, cor do texto, etc.;
- *Row* - Representa um conjunto de componentes (*Widgets*) alinhados horizontalmente no ecrã. Este componente gere o alinhamento dos componentes nele contidos automaticamente de forma a que todos se alinhem horizontamente;
- *Column* - Representa um conjunto de componentes (*Widgets*) alinhados verticalmente no ecrã. Este componente gere o alinhamento dos componentes nele contidos automaticamente de forma a que todos se alinhem verticalmente;
- *Image* - Representa uma imagem. Este conceito é abstrato, não sendo possível especificá-lo diretamente. O utilizador terá que indicar o tipo de imagem que quer apresentar, sendo que atualmente o único tipo suportado é o *LocalImage*

(carrega imagem a partir de um ficheiro local). No futuro poderá ser implementado, por exemplo, o carregamento de imagens a partir de algum local na rede (p.e. *NetworkImage*);

- *Button* - Representa um botão que será desenhado no ecrã. O utilizador pode definir o texto apresentado no botão, bem como um estilo para o mesmo (cor de fundo, cor do texto, etc.).

Um exemplo de uma especificação de uma interface gráfica simples, pode ser vista na Figura 5.9. Nesta, está representado um ecrã de 500 pixeis de altura por 700 pixeis de largura, que contem três quadrados pretos de 50 pixeis de altura por 50 pixeis de largura, alinhados verticalmente, dentro de um contentor branco, posicionado 200 pixeis à esquerda e 100 pixeis para abaixo no ecrã. Vemos que é possível, através desta estrutura fazer estruturas de informação simples, por exemplo, uma tabela de dados, poderia ser descrita como uma lista de *Column*s composta por uma lista de *Row*s, sendo que cada *Row* teria um conjunto de *Text*. Poderiam ainda ser colocadas cada *Row* dentro de um *Container* para colorir o fundo, ou adicionar limites.

```
SCREENS: {
 Screen [ name=" myScreen ", height= 500.0 , width= 700.0 ]
  content:  Container [ height= <no height> , width= <no width> , position= (x= 200.0 ,y= 100.0 ) ]
             > Decoration:
                backgroundColor= #FFFFFF
                border= <no borderDecoration>
             child:  Column [ mainAlignment= start , crossAlignment= center ]
                      children:  {
                       Container [ height= 50.0 , width= 50.0 , position= <no position> ]
                        > Decoration:
                           backgroundColor= #000000
                           border= <no borderDecoration>
                        child:  <no child>
                       Container [ height= 50.0 , width= 50.0 , position= <no position> ]
                        > Decoration:
                           backgroundColor= #000000
                           border= <no borderDecoration>
                        child:  <no child>
                       Container [ height= 50.0 , width= 50.0 , position= <no position> ]
                        > Decoration:
                           backgroundColor= #000000
                           border= <no borderDecoration>
                        child:  <no child>
                      }

}
```

Figura 5.9: *genui* - Exemplo de especificação de componentes gráficos na UI

5.1.4 Conceito Message

Uma *Message* (Figura 5.10) representa uma mensagem a ser enviada para ou recebida de um dispositivo. Por essa razão, deve ter associado à mesma o tipo de comportamento esperado: tipo *requester* para mensagens a enviar para um dispositivo; tipo *receiver* para mensagens a receber de um dispositivo; tipo *trigger* para mensagens a enviar mediante uma determinada ação.

Uma *Message* do tipo *requester*, que precisa de ser enviada, deve precisar de especificar um intervalo de tempo de espera entre envios. Uma *Message* do tipo *trigger* deve obrigar à introdução de um identificador, de forma a que seja possível atribuir a mesma a componentes da interface gráfica para espoletar envios (p.e. ao clicar num botão, enviar uma mensagem).

Figura 5.10: *genui* - Conceito *Message*

Para separar as mensagens por protocolo e mais facilmente ser possível separar os métodos de envio/receção das mesmas, este conceito *Message* é estendido pelos conceitos *CoapMessage* e *HttpMessage*. Como os próprios nomes indicam, *CoapMessage* representa uma mensagem a enviar segundo o protocolo CoAP, enquanto que a *HttpMessage* segue o protocolo HTTP.

5.2 Geradores de Código

Nesta secção é apresentada a implementação os geradores de código a partir de uma instância da *genui*, para o código fonte alvo. A geração de código em MPS é normalmente feita considerando que um Conceito será traduzido num único ficheiro final. No entanto, como a especificação das mensagens e da interface gráfica corresponde é constituída por vários Conceitos e queremos que todos sejam traduzidos em código alvo no mesmo ficheiro, foram criados geradores textuais em classes Java no MPS que definem como é que os vários conceitos são traduzida para a linguagem alvo.

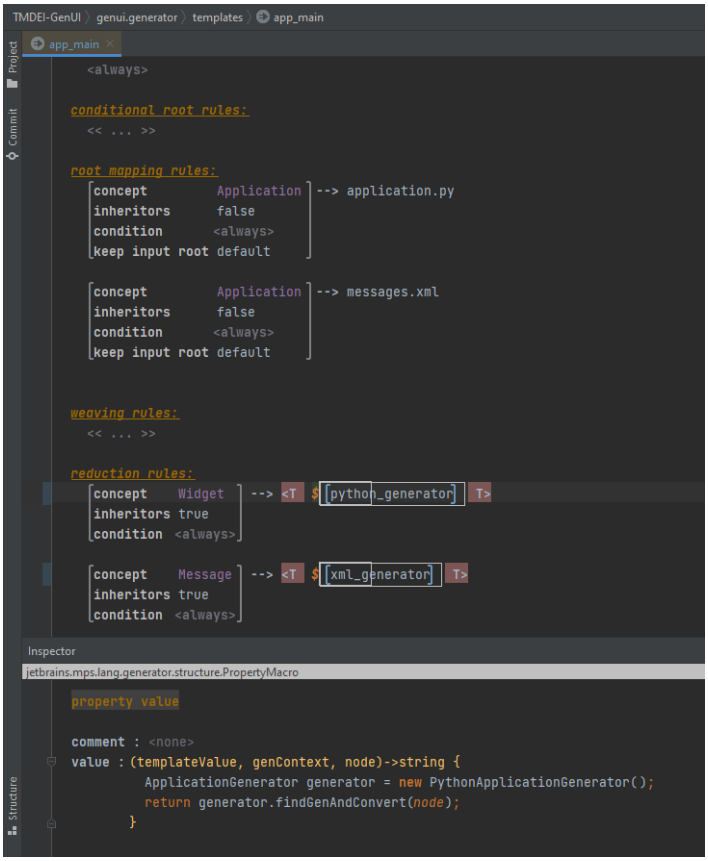

Figura 5.11: MPS - Configuração de Mapeamento para a *Application*

Como pode ser observado na Figura 5.11, nas configurações de mapeamento dos conceitos da *Application* foram adicionadas duas regras de mapeamento para ficheiro, uma para a interface gráfica (application.py - gera o código Python que renderiza a mesma) e a outra para a geração do ficheiro de mensagens (messages.xml - gera o XML com a estrutura das *Message*s). São ainda adicionadas duas regras de redução, isto é, regras que traduzem um conceito num único valor. Desta forma, o conceito *Widget* e *Message* serão interpretado e traduzido para uma string que contempla todas as suas propriedades e dependências. Esta interpretação e tradução do conceito num texto é feita pelas duas classes de geração criadas, a *PythonApplicationGenerator* e a *XMLMessageSpecGenerator*, descritos com maior detalhe na Figura 5.12.

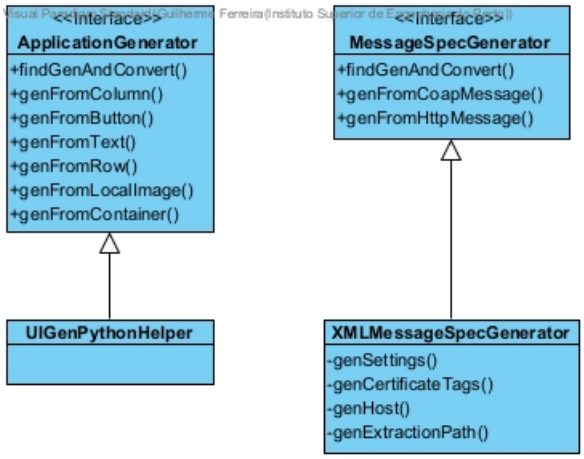

Figura 5.12: Diagrama de classes dos geradores de código

O método *ApplicationGenerator.findGenAndConvert* (Figura 5.12) é o responsável por determinar o tipo de *Widget* (seja uma *Row*, *Column* ou outro conceito que extenda a classe *Widget*) e definir qual o método a utilizar para o traduzir para texto (ver Figura 5.13. A mesma lógica de geração é aplicada ao conceito *Message*, através da função *XMLMessageSpecGenerator.findGenAndConvert*.

```
public class PythonApplicationGenerator implements ApplicationGenerator {

  @Override
  public string findGenAndConvert(node<Widget> widget) {
    concept switch (widget.concept) {
      exactly LocalImage :
        return genFromLocalImage((node<LocalImage>) widget);
      exactly Container :
        return genFromContainer((node<Container>) widget);
      exactly Row :
        return genFromRow((node<Row>) widget);
      exactly Text :
        return genFromText((node<Text>) widget);
      exactly Column :
        return genFromColumn((node<Column>) widget);
      exactly Button :
        return genFromButton((node<Button>) widget);
      default :
        return "No generator available for this concept [" + widget.concept.getClass() + "]\n";
    }
  }
```

Figura 5.13: Gerador - Tradução dos conceitos *Widget* para texto

```
public class XMLMessageSpecGenerator implements MessageSpecGenerator {
  @Override
  public string findGenAndConvert(node<Message> widget) {
    concept switch (widget.concept) {
      exactly CoapMessage :
        return genFromCoapMessage((node<CoapMessage>) widget);
      exactly HttpMessage :
        return genFromHttpMessage((node<HttpMessage>) widget);
      default :
        return "No generator available for this concept [" + widget.concept.getClass() + "]\n";
    }
  }
```

Figura 5.14: Gerador - Tradução dos conceitos *Message* para texto

5.2.1 Geração da especificação das Mensagens

A especificação das mensagens foi realizada tendo como alvo construir uma estrutura XML com o detalhe de cada mensagem, de forma a que seja possível validar a estrutura facilmente (através de um *XSD* de validação). Esta validação foi considerada importante dado que é sempre possível, depois de gerado o ficheiro final, que este seja manualmente editado.

5.2.2 Geração da especificação da interface gráfica

A geração da especificação da interface gráfica é descrita na secção 4.2. A implementação desta, passou pela geração de código Python que contem já a tradução dos conceitos da *genui* para as classes da *framework* de abstração criada.

```python
# file modules/ui/components/base/common.py
## extending abc.ABC means the class is abstract
@dataclass
class Component(ABC):
    position: Position = None
    size: Size = None
  ...

@dataclass
class Widget(Component, ABC):
  ...

# modules/ui/components/base/screen.py

@dataclass
class Screen():
    screenId: str = ""
    body: Widget = None
    ...
    def asComponent(

# modules/ui/components/base/row.py
@dataclass
class Row(Widget):
    children: list[Widget] = None
  ...

# modules/ui/components/base/text.py
@dataclass
class Label(Widget):
    decoration: LabelDecoration = None
  ...
    def asComponent(...):
        ...
        self._label: QLabel = QLabel(self.content, parent=parent)
        ...
        if self.decoration:
            self._label.setStyleSheet(self.decoration.
   toStyleSheetFormat())
        ...
        return self._label
```

Listing 5.1: *framework* de abstração para renderização da GUI

No excerto de código 5.1 vemos implementados os componentes principais deste conjunto de classes que serve de abstração à utilização de uma *framework* como o PyQt. Vemos ainda dois exemplos de implementação de *Widgets* (*Row* e *Label*), sendo que os restantes seguem esta mesma estrutura. No *Label* podemos ver um exemplo do método *asComponent()* descrito secção 4.2, responsável por retornar o elemento respetivo da *framework* de GUI utilizada, *PyQt* (ou seja, neste caso, retorna uma *QLabel*). Embora não constem todos os *Widget*s neste excerto, todos

seguem esta estrutura e todos, sem exceção, implementam o método asComponent() da mesma forma que o *Label* (mas retornando os elementos em *PyQt* respetivos). Podemos ver no excerto de código 5.2 um exemplo do ficheiro resultante da geração de código a partir da especificação da interface gráfica pelo utilizador. Este excerto representa a geração a partir da *Application* descrita na Figura 5.9.

```
1
2  application_name = "Sample Application"
3  application_version = "0.0.1"
4
5  def application_screens(__messageFromId) -> list[Screen]:
6      return [
7          Screen(
8              screenId="myScreen",
9              size=Size(500, 700),
10             isFullScreen=False,
11             body=Container(
12                 position=Position(200, 100),
13                 child=Column(
14                     children=[
15                         Container(
16                             height=50.0, width=50.0,
17                             decoration=ContainerDecoration(
18                                 backgroundColor='#000000',
19                             ),
20                         ),
21                         Container(
22                             height=50.0, width=50.0,
23                             decoration=ContainerDecoration(
24                                 backgroundColor='#000000',
25                             ),
26                         ), Container(
27                             height=50.0, width=50.0,
28                             decoration=ContainerDecoration(
29                                 backgroundColor='#000000',
30                             ),
31                         )
32                     ]
33                 )
34             )
35         )
36     ]
```

Listing 5.2: Ficheiro de especificação de interface gráfica - Exemplo

5.3 Programa do dispositivo central

A solução final, que estará em execução no dispositivo central foi dividida em vários processos com funcionalidades distintas e únicas. O objetivo deste software é que consiga comunicar com os dispositivos na rede e apresentar uma interface gráfica para apresentar informação proveniente destes. Para isso, o utilizador especifica,

como descrito na secção anterior, tanto as mensagens a trocar com os dispositivos da rede, bem como a disposição e quais componentes gráficos que devem ser apresentados na interface gráfica.

A Figura 4.20 (secção 4.2) apresenta o diagrama de classes de todo o Programa. Nas sub-secções que se seguem serão referidas classes sempre referentes às indicadas nesta figura.

5.3.1 Leitura das especificações das mensagens

A criação do Programa parte da instanciação dos componentes descritos na Figura 4.14, incluindo Processos, Queues de comunicação e *snapshot* inicial das variáveis a recolher. Para executar o programa é obrigatório que exista uma especificação das mensagens num ficheiro *messages.xml*, de forma a que esta informação consiga ser passada para todos os Processos que dela necessitam. Assim, devem estar descritas neste ficheiro as mensagens a receber e enviar de/para os dispositivos na rede. Foi implementada a estrutura (com detalhes omissos) representada em 5.3.

```
<?xml version="1.0"?>
<Messages>
    <CoapMessages>
        <CoapMessage>...</CoapMessage>
        ...
    </CoapMessages>
    <HttpMessages>
        <HttpMessage>...</HttpMessage>
        ...
    </HttpMessages>
</Messages>
```

Listing 5.3: Especificação genérica em XML do ficheiro *messages.xml*

O objetivo desta especificação é tornar-se reflexo da estrutura da *genui* para mais fácil geração posteriormente. Foi criado o elemento *<Messages>* que vai agregar todas as mensagens, separadas por protocolo de comunicação (*<HttpMessages>* e *<CoapMessages>*). Estes, embora sejam ambos mensagens, têm propriedades diferentes para que sejam enviadas ou recebidas, dado que seguem protocolos diferentes, e, por esta razão foram criados como elementos de tipos distintos. Podemos ver algumas diferenças quando comparamos um elemento <HttpMessage> (excerto de código 5.4) e um elemento <CoapMessage> (excerto de código 5.5). Estas diferenças são também reflexo das distinções feitas nos conceitos *HttpMessage* e *CoapMessage* da *genui*.

```
<?xml version="1.0"?>
<Messages>
    ...
    <HttpMessages>
        <HttpMessage action="request" method="GET" interval-seconds
    ="10" timeout-seconds="2">
            <MessageSettings>
                <VerifyCertificate>false</VerifyCertificate>
                <ContentType>Json</ContentType>
                <Variables>
                    <ExtractionPath id="Var4">$</ExtractionPath>
                </Variables>
            </MessageSettings>
            <Host>
                <Address>http://127.0.0.1</Address>
                <Port>8881</Port>
                <Endpoint>/api/v1/getrecord/Test7</Endpoint>
            </Host>
        </HttpMessage>
        ...
    </HttpMessages>
</Messages>
```

Listing 5.4: Especificação de uma HttpMessage no ficheiro *messages.xml*

```
<?xml version="1.0"?>
<Messages>
    <CoapMessages>
        <CoapMessage action="request" method="PUT" interval-seconds
    ="10">
            <MessageSettings>
                <MessagePayload></MessagePayload>
                <ContentType>Json</ContentType>
                <Variables>
                    <ExtractionPath id="temperature" isList="false"
    >$.temperature.celsius</ExtractionPath>
                </Variables>
            </MessageSettings>
            <Host>
                <Address>127.0.0.1</Address>
                <Port>8081</Port>
                <Endpoint>/v1/termperature</Endpoint>
            </Host>
        </CoapMessage>
        ...
    </CoapMessages>
...
</Messages>
```

Listing 5.5: Especificação de uma CoapMessage no ficheiro *messages.xml*

A primeira ação a ser feita na execução deste software é a leitura de um ficheiro de configurações da aplicação (como descrito na fase de análise na Figura 4.21), sendo apenas posteriormente feita a leitura do ficheiro *messages.xml* - ver excerto 5.6.

```
1
2  def main():
3
4      settings : ApplicationSettings = ApplicationSettings.
       loadFromYaml('settings.yaml')
5      print(f"Using messages file at: '{settings.messages_file}'")
6
7      msgParser : MessageParser = MessageParserXMLImpl()
8      messages = msgParser.parseMessagesFile(settings.messages_file,
       settings.messages_xsd_file)
9
10     if len(messages) == 0:
11         print("No Messages specified in the `messages.xml` file")
12         return
13
14     ...
```

Listing 5.6: Leitura do ficheiro *messages.xml*

Esta fase de desenvolvimento, quando terminada, teve os protocolo CoAP e HTTP implementados. Inicialmente foi apenas implementado CoAP, e, no final, para demonstrar a extensibilidade da solução, foi acrescentado HTTP. Para a implementação de novas mensagens que sigam novos protocolos, os únicos desenvolvimentos necessários são, dando o exemplo da implementação realizada de HTTP:

- A Criação de uma nova classe que estenda a classe abstrata *Message* (*HttpMessage*) e que implemente os métodos *parse()* e, se necessário, *asRequest()*;
- A Criação de uma nova classe que estenda a classe abstrata *Requester* (*HttpMessageRequester*) e que implemente o método *sendMessage()*;
- Adicionar na classe *HttpMessage RequesterStrategy* criando a ligação da mesma ao *HttpMessageRequester*.
- Adicionar os novos campos ao XSD que valida a estrutura do ficheiro *messages.xml*.

5.3.2 Criação dos Workers e Processos

Uma vez feita a leitura de quais a mensagens especificadas pelo utilizador, é necessário que esta informação chegue a todas as funcionalidades que, como descrito no Capítulo 4, serão separadas em diferentes Processos.

Inicialmente é feita a criação dos meios de comunicação entre Processos, através da instanciação de *Queues* e uma snapshot (dicionário em memória partilhada) para cada finalidade (excerto de código 5.7):

- *loggingQueue* - Para fazer pedido de escrita de *logs*;

- *parseQueue* - Para fazer pedido de processamento de mensagens recebidas e extração de valores das mesmas;
- *snapshot* - Onde constarão todos os valores processados das mensagens que correspondam a variáveis na especificação das mensagens feita pelo utilizador;

```
1  def main()
2
3      ...
4
5      # Initializing Queues and Snapshot
6      parseQueue = mp.Queue()
7      loggingQueue = mp.Queue()
8      terminateAllQueue = mp.Queue()
9      snapshot: DataSnapshot = DataSnapshot.create(messages)
10     sharedDict = UltraDict(snapshot.fetch())
11
12     ...
```

Listing 5.7: Inicialização das Queues e snapshot para comunicação entre Processos

Depois de termos as *queues* inicializadas, são criados os *Workers* para cada funcionalidade. Um *Worker* contem a execução a ser realizada por um processo, ou seja, casa *Worker* possui uma funcionalidade concreta (excerto de código 5.8). Assim, cada processo irá executar um destes *Workers*.

```
def main()

    ...

    # Initializing Workers that will run in a Process each
    uiWorker: Worker = UIWorker(messages=messages, sharedDictName=
sharedDict.name, settings=settings,
                    loggingQueue=loggingQueue, terminateAllQueue=
terminateAllQueue)

    loggingWorker: Worker = LoggerWorker(settings=settings,
loggingQueue=loggingQueue)

    parserWorker: Worker = ParserWorker(messages=messages, settings
=settings, loggingQueue=loggingQueue,
                        parseQueue=parseQueue, sharedDictName=
sharedDict.name)

    requesterWorker: Worker = RequesterWorker(messages=messages,
settings=settings, loggingQueue=loggingQueue,
                            parseQueue=parseQueue, terminateAllQueue=
terminateAllQueue)

    receiverWorker: Worker = ReceiverWorker(messages=messages,
settings=settings, loggingQueue=loggingQueue,
                                                sharedDictName=
sharedDict.name, terminateAllQueue=terminateAllQueue)

    ...
```

Listing 5.8: Criação dos *Workers*

Os *Workers* criados foram apresentados na secção de análise deste documento e têm os seguintes objetivos:

- *LoggerWorker*
  - Criado para realizar todas as funcionalidades descritas no RF11;
  - Aguarda *LogEvent*s enviados por outros processos para a *loggerQueue* e faz o registo desse evento num ficheiro local.
- *ParserWorker*
  - Criado para realizar todas as funcionalidades descritas no RF5 e RF9;
  - Aguarda *ParseRequest*s enviados por outros processos para a *parseQueue*, executa o tratamento da Mensagem dessde *ParseRequest* e atualiza os valores extraídos na *snapshot*.
- *ReceiverWorker*
  - Criado para realizar todas as funcionalidades descritas no RF6;

  - Cria um servidor HTTP que fica a aguardar o envio de pedidos de atualização de valores, provenientes de dispositivos na rede.

- *RequesterWorker*
  - Criado para realizar todas as funcionalidades descritas no RF4;
  - Cria eventos periódicos de envio de mensagens para dispositivos, segundo a especificação feita no ficheiro *messages.xml* pelo utilizador; e
  - Assim que recebe a resposta a uma mensagem enviada a um dispositivo, passa o processamento da mesma para o *ParserWorker*, através da submissão de um *ParseRequest* (que contem o conteúdo recebido) para a *parseQueue*.
- *UIWorker*
  - Criado para realizar todas as funcionalidades descritas no RF7 e RF10;
  - Renderiza uma interface gráfica a partir de uma especificação realizada pelo utilizador;
  - Aguarda alterações à *snapshot* para atualizar elementos da interface gráfica; sendo que
  - Atualiza a interface gráfica quando existe uma atualização de um valor apresentado no ecrã.

A separação do processamento das mensagens recebidas (*ParserWorker*) num *Worker*/Processo separado deve-se ao facto de tanto o *ReceiverWorker* como o *RequesterWorker* precisarem de fazer o tratamento das mensagens recebidas. Desta forma o processamento das mensagens não afeta o tempo de execução do envio periódico de mensagens (*RequesterWorker*), nem prejudica a execução do servidor do *ReceiverWorker*.

Assim, após a criação dos diversos *Workers*, passamos à sua atribuição a processos individuais, e o arranque dos mesmos (excerto de código 5.9

```
def main()

    ...

        # Instantiating processes with related Workers
        loggingProcess = mp.Process(target=loggingWorker.start, daemon=False)
        parserProcess = mp.Process(target=parserWorker.start, daemon=False)
        requesterProcess = mp.Process(target=requesterWorker.start, daemon=False)
        receiverProcess = mp.Process(target=receiverWorker.start, daemon=False)
        uiProcess = mp.Process(target=uiWorker.start, daemon=False)

    ...

    # Start running processes (executes 'Worker'.start() method)
        if settings.isLogActive:
            loggingProcess.start()
        parserProcess.start()
        requesterProcess.start()
        receiverProcess.start()
        uiProcess.start()


    ...
```

Listing 5.9: Arranque dos Processos (*Workers*)

5.3.3 LoggerWorker

O *LoggerWorker* é responsável pelo registo de eventos de *logs* num ficheiro local para persistir a sequência de ações feitas pelos vários *Worker*s e eventuais erros.

Esta escuta de eventos na *loggingQueue* é feita enquanto o próprio *Worker* estiver em execução. Os *logs*, para maior facilidade na pesquisa futura nos mesmos, são armazenados em ficheiros diários, isto é, os registos são efetuados num ficheiro cujo nome é a data do dia em que o evento foi lançado. O funcionamento do *LoggerWorker* pode ser visto no excerto de código 5.10.

```
@dataclass
class LoggerWorker(Worker):
    loggingQueue: mp.Queue
    def start(self):
        if not os.path.exists(self.settings.logDirPath):
            os.makedirs(self.settings.logDirPath)
        current_file = None

        while True:

            if not self.loggingQueue.empty():
                message: LoggerEvent = self.loggingQueue.get()

                if current_file is None or current_file != message.
    toFile:
                    current_file = message.toFile

                with open(f"{self.settings.logDirPath}{current_file
    }", 'a') as fopen:
                    fopen.write(f"{message}\n")
```

Listing 5.10: Classe *LoggerWorker*

A submissão e um evento *LoggerEvent* é feita através da *loggingQueue* por qualquer outro processo, como pode ser visto no excerto de código 5.11 que apresenta uma submissão de um evento que sinaliza o início da execução do *ParserWorker* (este evento de início de execução é enviado por todos os processos).

```
@dataclass
class ParserWorker(Worker):
    loggingQueue: mp.Queue
    ...

    def start(self):
        self.loggingQueue.put(LoggerEvent(moduleName='PARSER',
    message='Process initiated'))

        ...
```

Listing 5.11: Submit *LoggerEvent*

5.3.4 RequesterWorker

O *RequesterWorker* tem como responsabilidade o envio periódico das mensagens configuradas pelo utilizador, no âmbito dos requisitos referentes ao RF4. Para a realização do envio de cada mensagem é necessário ter em conta o protocolo de comunicação com que a mesma foi especificada no UC1. Foi assim criada uma camada de abstração à lógica de envio das mensagens (*Requester* - ver Figura 4.20). Ao estender esta classe é possível especificar uma lógica de envio de um tipo de mensagem, sendo possível criar *CoapMessageRequester* (que possui a lógica necessária ao envio de mensagens via protocolo CoAP). Foi posteriormente adicionada

ainda um *HttpMessageRequester* de forma a permitir ainda o envio de mensagens através de HTTP, como forma de demonstração da extensibilidade da solução como um todo.

O envio periódico das mensagens é feito através de *PeriodicMessageTask*s que são declaradas pelo *RequesterWorker* (uma instância por cada mensagem do tipo *requester* a enviar), sendo a este primeiro atribuída uma *Message* aquando da sua criação. O *PeriodicMessageTask* faz uso, previamente, da função *RequesterStrategy.of(message: Message)* que tem como retorno a instância correta da classe *Requester* a utilizar (*CoapMessageRequester* para *CoapMessage* e *HttpMessageRequester* para *HttpMessage*) - ver excerto de código 5.12.

A criação de uma nova implementação de um protocolo precisa, assim, que seja: criado uma classe (que extenda a) *Requester* que indique a forma de envio da mesma por esse mesmo protocolo; adicionado no *RequesterStrategy* esta nova implementação do *Requester*.

```
@dataclass
class RequesterWorker(Worker):
    ...
    def start(self):
        ...
        taskList: list[PeriodicMessageTask] = []
        loop = asyncio.get_event_loop()
        asyncio.set_event_loop(loop)
        try:
            # Create a periodic task for each message
            # allowedMessages is the list of Messages with type 'requester'
            for message in allowedMessages:
                # Get strategy for each message
                requester: Requester = RequesterStrategy.of(message)
                # Create the periodic task
                task: PeriodicMessageTask = PeriodicMessageTask(
                    requester=requester, message=message,
                    loggingQueue=self.loggingQueue,
                    parseQueue=self.parseQueue,
                    loop=loop)
                # Add to the task list
                taskList.append(task)

            # execute all tasks in the task list in parallel
            asyncio.gather(*[task.start() for task in taskList])
            # Don't stop Process untill the loop is stopped
            loop.run_forever()
        except (KeyboardInterrupt or Exception) as e:
            asyncio.gather(*[task.stop() for task in taskList])
            loop.stop()
```

Listing 5.12: Classe *RequesterWorker*

Para o envio de uma mensagem é necessário que seja passada a *parserQueue* para a *PeriodicMessageTask* dado que após um envio bem sucedido de uma mensagem, é sempre necessário processar a resposta recebida. Esta função de processamento das mensagens é feita por um outro *Worker* (*ParserWorker*) dado que a fonte destes pedidos de processamento de mensagem não é unicamente realizada pelo *ReceiverWorker*, podendo ser feita também por outros *Workers*. Ao mesmo tempo, ao separar o processamento da resposta num outro processo, garantimos que o *ReceiverWorker* nunca irá atrasar as mensagens periódicas derivado de processamentos possivelmente demorados das respostas, mantendo assim o tempo entre mensagens o mais constante possível. Este valor do período de espera entre envios de mensagens é especificado no UC1. Assim que é recebida a resposta a uma mensagem, e esta é redirecionada para a *parseQueue*, *PeriodicMessageTask* entra na fase de espera onde aguarda o período especificado antes de realizar um novo envio.

5.3.5 ParserWorker

O *ParserWorker* faz o tratamento de todas as respostas recebidas que cheguem à *parseQueue*. Serve para separar o processamento das respostas do envio das Mensagens, evitando possíveis atrasos no escalonamento das mensagens do *RequesterWorker* ou que haja respostas demoradas nos servidores do *ReceiverWorker*.

Dado que na especificação em UC1 o utilizador deve conseguir especificar o tipo do conteúdo das mensagens/respostas a receber, o processamento das mensagens deve variar dependendo deste tipo de conteudo. Isto é, Um conteúdo *Json* deve ser processado com JsonPath, XML com xPath e, em caso de texto simples, todo o conteúdo deve ser assumido como sendo o valor a guardar. Este JsonPath ou xPath faz parte da especificação da *Message* em UC1.

Em termos de funcionamento, o *ParserWorker* aguarda permanentemente a chegada de eventos à *parseQueue* que é a *Queue* para onde devem ser enviados todos os pedidos de processamento de mensagem. Um pedido de processamento envolve a introdução do conteúdo a processar, bem como o identificador da mensagem a que corresponde.

Depois de obtido o valor ou lista de valores, o mesmo é guardado na variável respetiva ao mesmo, na *snapshot* partilhada com os restantes processos (apenas utilizada pelo *UIWorker* para atualização da interface gráfica com a chegada de novos valores).

5.3.6 ReceiverWorker

O *ReceiverWorker* inicia, e pode alterar mediante configuração, um servidor com um determinado protocolo (implementado CoAP e HTTP). Este servidor, para ambos os protocolos implementados implementa:

- Um endpoint de pedido de atualização de um valor - para que possam ser atualizados valores de variáveis através do envio de mensagens por outros dispositivos, ao invés da utilização de mensagens periódicas do *RequesterWorker*;

- Um endpoint de pesquisa do valor atual de uma variável - Para que seja possível um outro sistema poder consultar o valor atual de uma variável, sem ter que saber como (ou conseguir) comunicar com os dispositivos terminais.

5.3.7 UIWorker

Este *UIWorker* serve para renderizar a interface gráfica especificada no UC2. Esta funcionalidade é separada num processo distinto de forma a que a atualização da interface gráfica com novos valores não fique afetada por demoras no processamento em qualquer outra funcionalidade. Ainda para prevenir atrasos na atualização do ecrã, permitindo uma melhor experiência de utilização, este *Worker* executa duas *threads* em simultâneo: renderizar a interface gráfica (daqui em diante designada por *UIThread*); estar atento às alterações da *snapshot* das variáveis e pedir atualização de um componente da interface gráfica sempre que existir uma nova alteração no valor associado ao mesmo (daqui em diante designada por *CommThread*).

Para a atualização da interface gráfica foi escolhida a utilização de uma *snapshot* ao invés de uma *Queue* de novos valores, de forma a evitar que fossem feitos vários pedidos de atualização do ecrã seguidos, para a mesma *Message*. Desta forma, o *ParserWorker* guarda na *snapshot* todas as alterações ao mesmo tempo e altera o identificador único da mesma, de forma a que seja possível garantir que uma *snapshot* foi alterada pela comparação dos seus identificadores.

Sempre que é necessário atualizar um valor apresentado na interface gráfico, apenas esse componente é atualizado e isto é conseguido através da utilização de um padrão *Publisher-Subscriber*. Os componentes, na sua especificação, precisam de saber qual a variável que está associada ao mesmo. Com esta informação, durante a renderização inicial da interface gráfica, todos os componentes subscrevem o tópico (com o nome dessa variável que lhes foi associada) do *Publisher* que neste caso será a *CommThread*. A subscrição de um tópico foi simplificada neste projeto, correspondendo apenas à definição de um *callback* de atualização do componente respetivo e, a *CommThread* que verifica uma alteração numa determinada variável da *snapshot*, faz um pedido de execução da função de *callback* respetivo - ver excerto de código 5.13 que apresenta a classe que atua como sistema *Publisher-Subscriber*.

```
subscribers = dict()

class PubSub:
    def subscribe(topic: str, fn):
        if not topic in subscribers:
            subscribers[topic] = []

        subscribers[topic].append(fn)

    def post_event(topic: str, data: VarChange):
        if not topic in subscribers:
            return
        for fn in subscribers[topic]:
            fn(data)
```

Listing 5.13: Publish-Subscribe - Classe *PubSub*

A *CommThread* executa a função *PubSub.post_event* com o tópico (nome da variável) e a alteração da variável (objeto que contem a data de alteração, novo valor e valor antigo da variável). Os componentes da interface gráfica registam os seus *callbacks* através do *PubSub.subscribe*, passando como argumentos o nome do tópico (nome da variável) que querem subscrever e a função que atualiza o próprio conteúdo do componente. Com a utilização deste tipo de padrão, evitamos que todos os componentes da interface gráfica sejam atualizados sempre que um precisa de ser alterado, o que provoca melhorias de desempenho da interface gráfica através da utilização de menos recursos.

Por último, é de notar que em tempo útil de projeto foi impossível completar todo o desenvolvimento necessário pretendido a geração de uma interface gráfica personalizável. Embora seja possível especificar vários componentes e representar muitas estruturas gráficas, atualmente não podemos considerar como finalizada a implementação dado que falta que sejam especificáveis pontos como: animações, reconhecimento de gestos, navegação entre ecrãs, etc.. Estes pontos devem ser iterativamente acrescentados à solução.

5.4 Resultados da Implementação

De forma associar a implementação desenvolvida e apresentada nas secções anteriores, é reunida a lista dos requisitos funcionais detalhados no Capítulo 4, expondo o os resultados obtidos, para cada um destes, no final da fase de implementação:

- UC1 - São especificadas as mensagens via Configurador através da DSL criada (*genui*);
- UC2 - São especificados os componentes a renderizar da interface gráfica via editor (MPS) através da DSL criada (*genui*);
- UC3 - A partir das especificações em UC1 e UC2, são gerados os ficheiros na linguagem alvo que alimentam a execução da solução;

- RF4 - O envio de mensagens para os dispositivos especificados em UC1 é realizada via *RequesterWorker* que envia periodicamente as mensagens especificadas para os dispositivos respetivos;
- RF5 - A extração dos valores das mensagens é realizado pelo *ParserWorker* que trata todas as respostas recebidas a mensagens enviadas no RF4;
- RF6 - O *RequesterWorker* inicia o servidor com o protocolo desejado (CoAP ou HTTP), permitindo, através do mesmo, atualizar valores de variáveis ou obter o valor de variáveis existentes;
- RF7 - A apresentação da interface gráfica é feita pelo *UIWorker* através da *UIThread* que, com base na especificação em UC2, renderiza uma GUI em *PyQT*.
- RF10 - A atualização automática a interface gráfica é realizada no *UIWorker*, onde a *CommThread* ao verificar atualizações na *snapshot* de variáveis, publica a alteração as mesmas na classe de *Publish-Subscribe* que espoleta, por sua vez, atualização dos componentes da GUI;
- RF11 - O armazenamento de ficheiros de *logs* do sistema é feito através do *LoggerWorker* que armazena em ficheiros diários todos os *logs* submetidos para o efeito pelos restantes processos da solução.

Não foi possível, por aproximação do fim do período de entrega do documento, a realização dos seguintes use cases RF8 e RF9. Estes foram propositadamente colocados no final do *backlog* de desenvolvimento dada a baixa prioridade atribuída - ver tabela 4.2.

Capítulo 6

Avaliação e Experimentação

Este capítulo tem por objetivo a avaliação das soluções implementadas, definindo as métricas a ser utilizadas, a metodologia utilizada e apresenta os resultados objetivos.

6.1 Indicadores

Para avaliar a solução é necessário definir os indicadores de avaliação, isto é, indicar os pedaços de informação de um determinado aspeto do programa que são documentáveis e/ou mensuráveis. Desta forma, os indicadores foram definidos com base nos requisitos e objetivos do projeto. Os principais indicadores são:

1. **Cumpre os requisitos funcionais**: Qualquer software deveria cumprir e executar de acordo com os requisitos funcionais e casos de uso definidos;
2. **Usabilidade e Satisfação de Uso**: Dado que a razão do desenvolvimento desta solução é facilitar o desenvolvimento de outras soluções de IoT, deve ser confirmada a facilidade de uso da mesma e o quão bem satisfaz as preocupações dos utilizadores/clientes.
3. **Performance**: Verificar o tempo de resposta da aplicação mediante:
    - O número de atualizações de ecrã por segundo;
    - O número de mensagens/eventos recebidos por segundo;

6.2 Especificação de Hipóteses

Para que seja feita a medição dos indicadores definidos em 6.1, é necessário definir as hipóteses que pretendemos testar, de forma a validar, para cada indicador, se este foi cumprido. Para tal, para cada indicador, são definidas duas hipóteses opostas, a hipótese nula (H0) e a hipótese alternativa (H1), sendo esta última a negação da primeira (Carvalho Pedrosa e Marques A. Gama 2016). Assim, para cada indicador introduzido, temos as hipóteses que constam na tabela 6.1.

Tabela 6.1: Resultados do levantamento de requisitos e respetiva importância

Indicador	**H0**	**H1**
Cumpre os requisitos funcionais	A solução não cumpre os requisitos necessários	A solução cumpre os requisitos necessários
Usabilidade	A solução é fácil de utilizar	A solução não é fácil de utilizar
Satisfação de Uso	A solução facilita o desenvolvimento	A solução não facilita o desenvolvimento
Performance em relação à atualização da GUI	Até 25 mensagens em processamento, o tempo de atualização do ecrã é inferior a 2 milissegundos	Até 25 mensagens em processamento, o tempo de atualização do ecrã é maior ou igual a 2 milissegundos

É de notar que as hipóteses relacionadas com a performance da solução (RNF14 da Tabela 4.2) poderiam ter como limite de atualização do ecrã o valor de 0.1 segundos dado que este valor é o considerado como sendo o limite para que o utilizador sinta que o sistema está a reagir instantaneamente (Nielsen 1994). Embora 0.1 segundos seja o suficiente para o utilizador considerar como uma resposta instantânea, foi considerado, arbitrariamente, que o processamento de uma atualização da GUI deve ser inferior a 2 milissegundos. Nestas hipóteses foi ainda utilizado o limite de 25 mensagens especificadas pois dado que este projeto é dirigido a praticantes de DIY que não constroem à partida projetos de grandes dimensões - 25 mensagens configuradas significa até 25 dispositivos conectados.

As hipóteses definidas relativamente ao cumprimento dos requisitos funcionais é baseada no grau de completude dos requisitos definidos na secção 4. As hipóteses relacionadas com a satisfação do cliente e com a facilidade de uso serão baseadas num inquérito realizado a ao público alvo deste projeto após uma sessão de experimentação da solução realizada no final do desenvolvimento. Será tida em conta uma satisfação global de 80%, valor definido como sendo o mínimo para declarar satisfação por parte dos utilizadores, segundo o Customer Satisfaction Score (CSAT) (CFI 2021).

6.3 Métodos de Avaliação

Serve esta secção para apresentar os métodos de avaliação dos indicadores identificados. Serão utilizados três métodos diferentes para realizar a avaliação, que, tal como demonstrado na tabela 6.2: testes de software, avaliação de performance e questionário de usabilidade e satisfação.

Tabela 6.2: Alocação dos Métodos de Avaliação

Indicador	**Método**
Cumpre os requisitos funcionais	Testes de aceitação
Performance	Testes de performance
Usabilidade e Satisfação de Uso	Questionário de usabilidade e satisfação

6.3.1 Testes de Sistema e Aceitação

O desenvolvimento aplicacional deve passar sempre por processos de teste manuais e automáticos, sendo comum a utilização de testes unitários, testes de integração e testes de aceitação (Basak e Shazzad Hosain 2014). Neste sentido serão realizados testes unitários e de aceitação da solução final de forma a avaliar o cumprimento, ou não, dos requisitos funcionais identificados (Tabela 4.2).

Os testes unitários focaram-se na leitura correta dos ficheiros de especificação das mensagens dado que ainda tem que ser traduzido de XML para objetos de *runtime*, assim como foram realizados testes ao *parse* dos valores das mensagens recebidos para os formatos especificados (*Json*, *XML* e texto).

Foram assim finalizados os testes unitários com os seguintes casos (ver Figura 6.1):

- Leitura do ficheiro de configurações:
  - Apresenta erro se ficheiro não existir
  - Apresenta erro se ficheiro for inválido
- Leitura do ficheiro de especificação de mensagens:
  - Apresenta erro se ficheiro não existir
  - Apresenta erro se ficheiro for inválido (em validação com o *XSD*)
  - Apresenta erro no ficheiro faltarem componentes obrigatórios (em validação com o *XSD*)
- Extração de valores das variáveis de mensagens recebidas:
  - Apresenta um erro caso o formato seja diferente do especificado (e.g. especificado Json, mas conteúdo é XML);
  - Faz a leitura correta a partir de um objeto *Json*;
  - Faz a leitura correta a partir de um ficheiro *XML*;
  - Faz a leitura correta a partir de conteúdo de texto simples (*plaintext*);

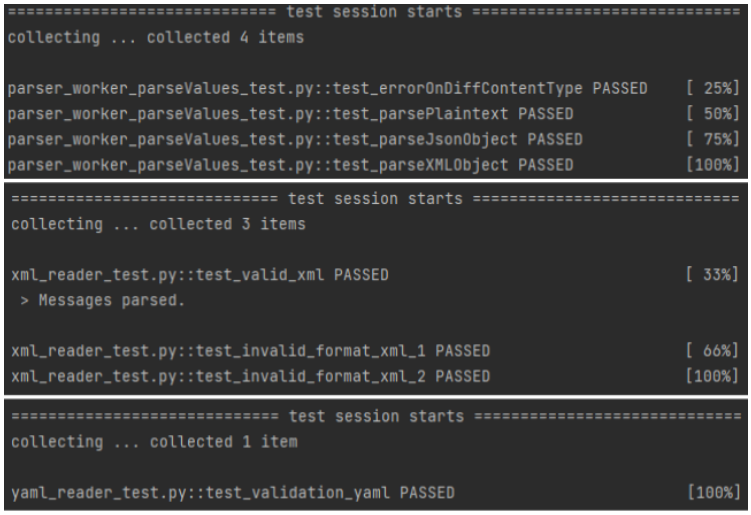

Figura 6.1: Execução dos Testes Unitários

Todos os testes unitários estão a passar no código. Ficou por fazer uns casos de testes unitários para a escrita de logs no sistema dado que existem alguns componentes unitários testáveis, mas que por falta de tempo até à conclusão do projeto não foi possível efetuar.

Os testes de aceitação foram feitos percorrendo os casos de uso e restantes requisitos funcionais na Tabela 4.2, criando alguns cenários de teste. Todos os testes foram efetuados com sucesso, estando o sumário dos mesmos descrito abaixo:

1. **Configurador (UC1 a UC3)**
   (a) É possível criar uma *Application*;
   (b) É possível adicionar uma Mensagem à *Application* de um determinado protocolo;
   (c) É possível adicionar um *Screen* à *Application*;
   (d) É possível adicionar componentes de interface gráfica a um *Screen*;
   (e) Ao adicionar um texto como componente de GUI, são sugeridas a lista de mensagens a apresentar (campo opcional);
   (f) Ao adicionar um botão como componente de GUI, são sugeridas a lista de mensagens a enviar com o clicar do mesmo (campo opcional);
   (g) Não deve ser possível gerar o código alvo se houver erros na especificação (e.g. faltam campos obrigatórios, *input* com formato inválido);
   (h) Tendo todos os dados da *Application* corretamente inseridos, é possível realizar a geração dos ficheiros finais.
2. **Programa (RF4 a RF11)**

(a) O ficheiro de mensagens é lido corretamente (validado via *logs* do sistema);

(b) O ficheiro de configurações do projeto é lido corretamente (validado via *logs* do sistema);

(c) Os *logs* do sistema são armazenados localmente em ficheiro;

(d) O envio de mensagens é iniciado assim que o programa arranca e todas as mensagens são enviadas;

(e) No envio de mensagens, em caso de especificação errada (e.g. dispositivo não existe ou não há acesso à rede) é apresentado o erro (via *logs*);

(f) No envio de mensagens, a resposta é corretamente processada independentemente do formato (Json, XML ou texto simples);

(g) a interface gráfica apresenta os componentes com o posicionamento e estilização definida no ficheiro de especificação dos mesmos.

6.3.2 Testes de Performance

Em sistemas de monitorização de dados em tempo real, é muito importante ter em conta a performance tanto das comunicações como da interface gráfica que vai apresentar os dados. Serão desta forma analisados os tempos de resposta nas comunicações com outros dispositivos e os tempos de atualização da GUI, assim como será feita a relação entre estes valores obtidos com o número de mensagens/eventos recebidos/enviados.

Para a recolha dos valores de performance foi adicionado um processo (*Worker*) extra à solução, ativado apenas por configuração que terá como função coletar eventos de performance enviados por outros processos. A arquitetura e funcionamento deste *Worker* é muito semelhante ao *LoggerWorker* e por isso será ocultada neste documento, apenas a função de ambos diverge dado armazenarem objetos de diferentes tipo de ficheiros locais. Desta forma é possível centralizar os dados de performance num único *Worker* responsável pelo seu armazenamento.

Para analisar os pontos expostos na tabela 6.1 e retirar as conclusões quanto às hipoteses sugeridas, foram recolhidas as seguintes informações em diferentes ficheiros:

1. Registo do momento do envio ou recebimento de mensagens - permite saber num dado instante quantas mensagens estariam em execução;
2. Registo do momento de todas as atualizações de componentes da GUI - com este dado podemos aferir possíveis variações ao relacionar o mesmo com outros fatores, por exemplo, número de mensagens recebidas nesse instante.

3. Registo do número de pedidos de processamento de resposta a cada instante - permite saber num dado instante quantos pedidos de processamento estão pendentes e verificar quanto tempo demoram a processar.
4. Registo do tempo de execução processamento do envio de uma mensagem - para correlacionar com outros parâmetros recolhidos e verificar se existe relação.
5. Registo do número de alterações à *snapshot* de variáveis - uma alteração à *snapshot*, no limite, pode fazer atualizar toda a interface gráfica por isso, este parâmetro vai ajudar a verificar relações entre tempos de processamento e o número real de componentes da GUI atualizados.

Tempo de processamento em relação do número de mensagens

As hipóteses para este teste são:

H0 Até 25 mensagens em processamento, o tempo de atualização do ecrã é inferior a 2 milissegundos;

H1 Até 25 mensagens em processamento, o tempo de atualização do ecrã é maior ou igual a 2 milissegundos;

Neste teste queremos verificar se existe uma correção entre o numero de mensagens trocadas e o tempo de processamento das mesmas na solução desenvolvida, para isso foram recolhidos os dados no formato exposto no excerto 6.2. Estes dados foram recolhidos ao longo de 13 minutos e 30 segundos de execução do programa, com 25 mensagens especificadas a serem chamadas entre 2, 5, 10 e 15 segundos de espaço. Desta forma conseguimos ter vários momentos em que todas coincidem no envio, podendo ter momentos em que todas as mensagens estão em processamento ao mesmo tempo e, ao mesmo tempo, ter momentos que apenas algumas estão em processamento - podemos assim ter dados para fazer comparações entre estes momentos distintos. Esta amostra de mais de 13 minutos conseguiu recolher o tempo de processamento de 275 atualizações da GUI e registar o número de mensagens em processamento em cada um destes momentos. Este ficheiro em formato CSV possui os seguintes dados:

- **start_milli** - O momento de início (em milissegundos desde 1970) da execução do processamento das alterações na *snapshot* e pedido da atualização do ecrã;
- **end_milli** - O momento de conclusão (em milissegundos desde 1970) da execução do processamento das alterações na *snapshot* e pedido da atualização do ecrã;
- **start_formatted_sec** - O momento de início da execução do processamento das alterações na *snapshot* e pedido da atualização do ecrã, em segundos, tendo como referência o momento de arranque do programa;

- **end_formatted_sec** - O momento de conclusão da execução do processamento das alterações na *snapshot* e pedido da atualização do ecrã, em segundos, tendo como referência o momento de arranque do programa;
- **duration_milli** - O tempo de duração do processamento, em milissegundos (intervalo entre *start_milli* e *end_milli*;
- **calls_in_exec** - O número de mensagens a serem enviadas no momento *start_milli*.

	A	B	C	D	E	F
1	start_milli	end_milli	start_formatted_sec	end_formatted_sec	duration_milli	calls_in_exec
2	1665016355324.445	1665016355326.945	3.257501220703125	3.260001220703125	2.5	5
3	1665016358330.445	1665016358330.944	6.263501220703125	6.264000244140625	0.4990234375	1
4	1665016361337.445	1665016361337.945	9.270501220703125	9.271001220703125	0.5	1
5	1665016364344.446	1665016364346.445	12.277502197265624	12.279501220703125	1.9990234375	14
6	1665016367351.445	1665016367351.9448	15.284501220703126	15.2850009765625	0.499755859375	5
7	1665016370358.4458	1665016370358.4458	18.291501953125	18.291501953125	0.0	5
8	1665016373365.447	1665016373367.446	21.298503173828124	21.300502197265626	1.9990234375	13
9	1665016376372.446	1665016376374.947	24.305502197265625	24.308003173828126	2.5009765625	19
10	1665016379379.447	1665016379380.447	27.312503173828127	27.313503173828124	1.0	5
11	1665016382386.447	1665016382388.447	30.319503173828124	30.321503173828123	2.0	18
12	1665016385392.948	1665016385393.448	33.326004150390624	33.326504150390626	0.5	7
13	1665016388400.448	1665016388402.948	36.333504150390624	36.33600415039062	2.5	22
14	1665016391407.448	1665016391407.948	39.34050415039062	39.34100415039062	0.5	8
15	1665016394413.949	1665016394415.9492	42.34700512695313	42.34900537109375	2.000244140625	14
16	1665016397420.95	1665016397422.45	45.35400610351562	45.35550610351562	1.5	5
17	1665016400427.45	1665016400427.95	48.360506103515625	48.36100610351563	0.5	2
18	1665016403434.9512	1665016403436.951	51.36800732421875	51.370007080078125	1.999755859375	17
19	1665016406441.4502	1665016406441.9502	54.37450634765625	54.37500634765625	0.5	8
20	1665016409448.951	1665016409448.951	57.382007080078125	57.382007080078125	0.0	1
21	1665016412455.951	1665016412457.4512	60.38900708007812	60.39050732421875	1.500244140625	1
22	1665016415462.951	1665016415463.451	63.39600708007813	63.39650708007812	0.5	3
23	1665016418469.9521	1665016418470.9521	66.40300830078125	66.40400830078126	1.0	3
24	1665016421476.9521	1665016421476.9521	69.41000830078124	69.41000830078124	0.0	0

Figura 6.2: Excerto dos valores extraídos de tempo de processamento com número de mensagens trocadas

Para testar as hipóteses H0 e H1, precisamos de validar num primeiro passo que estamos perante distribuições normais. Para este efeito, dado que estamos perante uma amostra com tamanho $n > 50$, realizamos o teste Kolmogorov–Smirnov, assumindo H0 "A variável segue uma distribuição normal"e H1 "A variável segue uma outra distribuição". Poderia ter sido utilizado o teste Shapiro-Wilk mas não é o mais apropriado para amostras em que o tamanho n < 50. Estes testes servem para quantificar os desvios que existem entre os diferentes valores de uma distribuição normal. Para isso, com base na amostra coletada calculamos os parâmetros na Tabelas 6.3 (e na Figura 6.3), para aplicar no cálculo do teste[1]:

$$D = \max_{1 \leq i \leq N} \left(F(Y_i) - \frac{i-1}{N}, \frac{i}{N} - F(Y_i) \right) \tag{6.1}$$

[1]https://www.itl.nist.gov/div898/handbook/eda/section3/eda35g.htm

Tabela 6.3: Dados sobre a amostra para aplicação do teste Kolmogorov–Smirnov - parâmetro **duration_milli**

Parâmetro	**Valor Obtido**
Tamanho da Amostra	275
Média	1.2978
Mediana	1.4170
Desvio Padrão	0.8425

Figura 6.3: Diagrama de caixa para verificar a variação de dados da variável *duration_milli*

Foi assim realizado este teste, com estas hipóteses, para ambas as variáveis, obtendo assim *D(275) = 0.1665* e *p-valor < 0.00001*. Dado que o *p-valor* é inferior à significância utilizada (0.05), temos assim evidência estatística que nos permite rejeitar H0 utilizando uma significância de 5%. Concluímos desta forma que esta variável não segue uma distribuição normal.

Por este motivo, de forma a testar a amostra quanto à hipótese nula ("Até 25 mensagens em processamento, o tempo de atualização do ecrã é inferior a 2 milissegundos"), utilizamos o teste de *Wilcoxon* para uma amostra que é aplicado para este efeito a variáveis que não seguem a distribuição normal - dados não-paramétricos. Para testar a hipótese nula são contabilizadas na amostra os casos de durações acima e abaixo de 2 milissegundos, calculando a média de cada uma desta classificações - Tabela 6.4.

Tabela 6.4: Classificações para parâmetro **duration_milli**

	N (Quantidade)	**Média da Classificação**	**Soma da Classificação**
Classificação abaixo de 2 milissegundos	209	159.88	33.42
Classificação acima de 2 milissegundos	66	68.71	4.54
Classificação igual a 2 milissegundos	0		
Total	275		

Na aplicação do teste de *Wilcoxon* foi obtido um *pvalor* = 1, portanto, dado um nível de significância de 5%, sendo *1 > 0.05*, não rejeitamos H0. Por este motivo podemos concluir que a solução cumpre, tendo por base a amostra recolhida, o requisito de performance.

6.3.3 Questionário de Usabilidade e Satisfação

O questionário para avaliação da usabilidade da solução e grau de satisfação de uso deve ser composto por questões simples e que evitem a ambiguidade e respostas incertas. As respostas às questões devem enquadrar-se numa de três escalas diferentes de resposta:

- Escala Dicotómica - Escolha de uma resposta a partir de duas alternativas;
- Escala de Classificação - Utilização da escala de Likert[2], dando hipótese de escolha de uma alternativa entre cinco níveis de classificação (Discordo completamente, Discordo, Nem concordo nem discordo, Concordo, Concordo Completamente);
- Escala de Diferencial Semântico - Permite a escolha de classificação de proximidade a um elemento de entre dois com significado oposto (e.g. Simples e Complexo).

Dado que, como descrito no contexto do problema (secção 1.2), esta solução tem por objetivo melhorar a forma de desenvolvimento de soluções de gestão de eventos e criação de interfaces gráficas na área de IoT, é fundamental avaliar a quão bem a própria conseguiu mitigar o problema existente. Para este efeito, consta no questionário um pedido de avaliação da implementação de cada requisito implementado. O questionário segue, desta forma, duas etapas: questionar o utilizador quanto à usabilidade da solução desde a instalação de ferramentas ao desenvolvimento de um projeto; e questionar o utilizador quando à eficácia da solução na resolução do problema inicial.

[2]`https://www.britannica.com/topic/Likert-Scale`

Planeamento e Experimentação

Para que estes utilizadores tivessem experiência a utilizar a solução antes de responder ao questionário, foi realizada uma sessão de experimentação que contou com doze participantes de áreas relacionadas com informática e desenvolvimento de software - os mesmos que se encontram expostos Tabela 4.1 da secção 4.1.

Numa primeira parte na sessão foi explicado o problema da área a todos os participantes e demonstrada a solução com um exemplo simples. Numa segunda fase, os participantes tiveram que instalar as ferramentas necessárias e desenvolver um projeto utilizando a solução desenvolvida. Os dispositivos para efetuar a comunicação e obter determinados parâmetros foram já disponibilizados na rede e não foram alvo de análise na sessão. Os passos realizados pelos participantes na segunda fase da sessão foram, por esta ordem:

1. Fazer download do MPS;
2. Fazer download do projeto;
3. Criar uma "Application"
4. Especificar mensagens;
5. Especificar interface gráfica - adicionar componentes gráficos;
6. Gerar código a partir das especificações;
7. Atualizar código do projeto (passo 2) com os ficheiros gerados;
8. Executar a aplicação.

Foi dada alguma liberdade aos utilizadores para configurarem as mensagens e a interface gráfica como pretendessem. Os dispositivos a que se podiam conectar foi preparado previamente e tinha em execução dois servidores, um de CoAP e outro de HTTP, ambos a disponibilizar um *endpoint* para obter um valor aleatório (e.g. número ou texto aleatório). Foi disponibilizada uma lista de *endpoints* que poderiam ser utilizados para o efeito, com a estrutura de dados retornada de cada um destes. Todos os dados retornados mantinham a estrutura mas variavam aleatoriamente os seus valores. Foi pedido que cada participante criasse entre 3 a 5 mensagens de cada tipo (*requester*, *receiver* e *trigger*) para que os utilizadores se habituassem à criação das mesmas e pudessem experimentar diferentes formas de as configurar. Em termos de interface gráfico, foi dada liberdade total aos utilizadores, desde que o ecrã criado apresentasse pelo menos 5 das variáveis recebidas nas mensagens *requester*, 1 botão com um envio de mensagem associado (o valor desta mensagem deveria também ser apresentado no ecrã) e 2 variáveis recebidas por via de mensagem do tipo *receiver*. A experimentação teve por objetivo que os utilizadores com base nestas regras e na especificação dada das estruturas de dados disponíveis na rede, agregassem um conjunto de dados variados, à sua escolha, e os apresentasse no ecrã.

No final da experimentação, foi pedido que todos os participantes preenchessem o Questionário de Usabilidade e Satisfação de Uso tendo em conta a utilização que

tiveram da ferramenta durante a sessão - ver questionário no Apêndice A. A divisão do questionário teve como objetivo:

- Determinar se os participantes pertencem ao público alvo;
- Avaliar a solução quanto à facilidade instalação e utilização do editor MPS para utilização da *genui*;
- Avaliar a satisfação de utilização de cada funcionalidade da aplicação;
- Avaliar se a solução resolve o problema proposto;
- Avaliar se o utilizador utilizaria esta solução no desenvolvimento de outros projetos.

Público Alvo

Os doze participantes (Tabela 4.1) encontram-se todos entre os 25 e os 29 anos de idade, entre os 5 e os 7 anos de experiência profissional e realizaram todos apenas entre 0 a 2 projetos DIY - estes dados foram recolhidos à parte do questionário.

Na Figura 6.4 são apresentadas as respostas dos participantes da sessão de experimentação relativas às perguntas sobre a sua inserção no público alvo. Podemos ver que a maioria dos participantes, 83.3%, já tentou realizar projetos segundos princípios DIY e 58.3% dos participantes já tem experiência no desenvolvimento de soluções com base em redes de dispositivos. O público alvo da ferramenta desenvolvida pode ou não ter conhecimentos da área, pelo que os utilizadores não necessitam de ter experiência no desenvolvimento destas soluções em IoT.

Na "Pergunta 3", é questionado se o participante tem um mínimo de experiência numa das três áreas listadas. Estas áreas são precisamente as bases da ferramenta desenvolvida e o facto de a maioria dos utilizadores ter experiência no desenvolvimento de uma destas áreas ajuda a que a solução seja melhor avaliada pelos mesmos, já que todos saberão à partida como deve funcionar algumas das funcionalidades da solução.

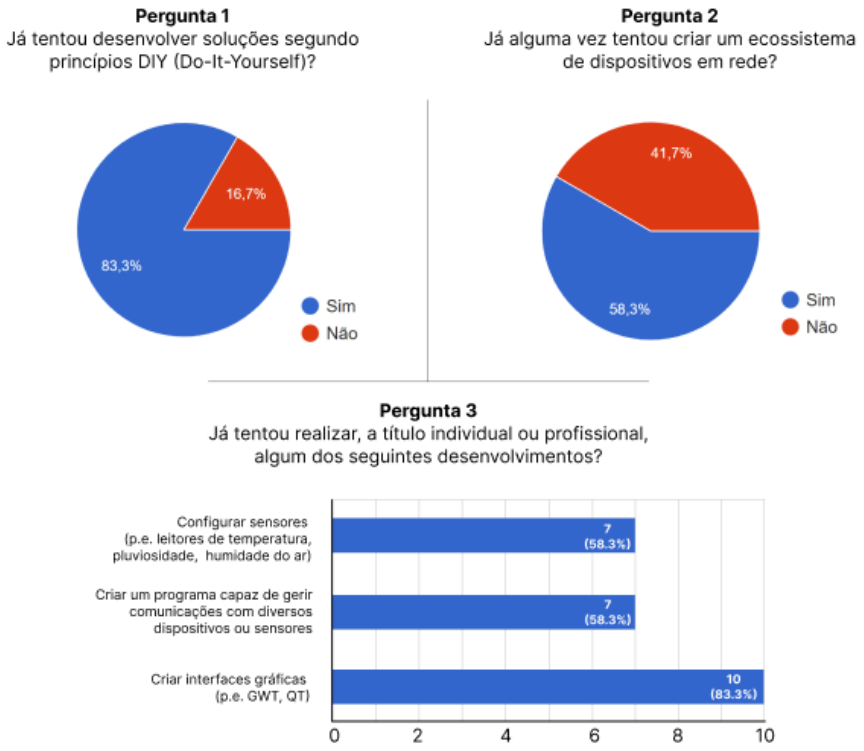

Figura 6.4: Questionário - Análise dos participantes

Resultados da Usabilidade da Solução

Passamos assim para a avaliação da facilidade de utilização das ferramentas necessárias para fazer a especificação. Foi calculada para cada alínea da pergunta (facilidade de utilização das ferramentas) a média do valor das respostas dadas, de 1 a 5 como é apresentado na Figura 6.5. Nesta Figura encontram-se ainda anotadas as pontuações mínimas e máximas para cada alínea. Com as médias de cada alínea foi feita a média aritmética total somando estas e dividindo pelo total de alíneas. A média total obtida foi de 4.57 pontos (como apresentado na Figura acima), o que indica que é "Simples"instalar e utilizar as ferramentas de especificação (ver implementações do UC1 e UC2). No entanto, podemos ver que temos o menor valor médio, 4.25 pontos, foi obtido na alínea "Usar o MPS para realizar as especificações da interface Gráfica". Este ponto pode dever-se ao facto de poder não ser claro para os participantes como é feita a criação de componentes da interface gráfica ou de como é criada a *Application*.

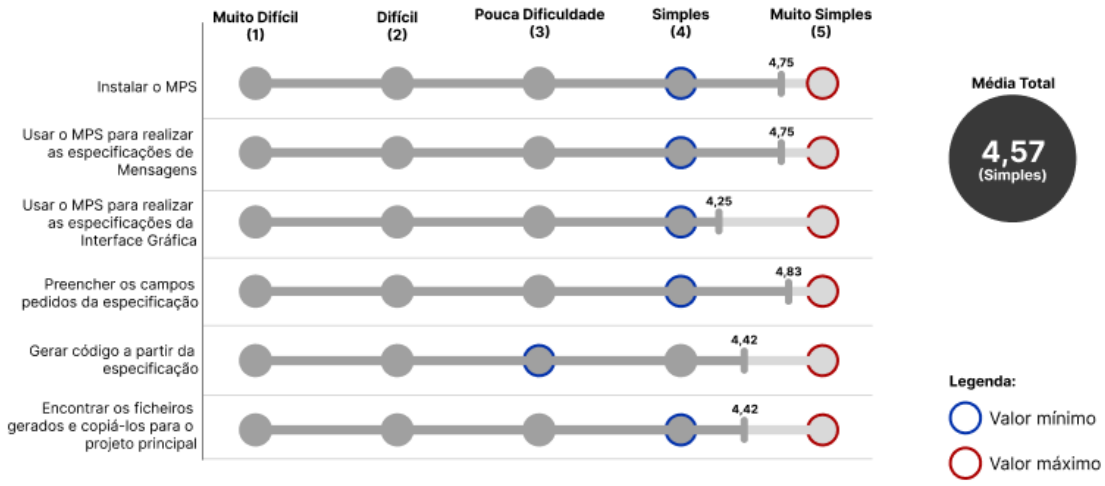

Figura 6.5: Questionário - Médias de pontuações nas respostas à usabilidade das ferramentas

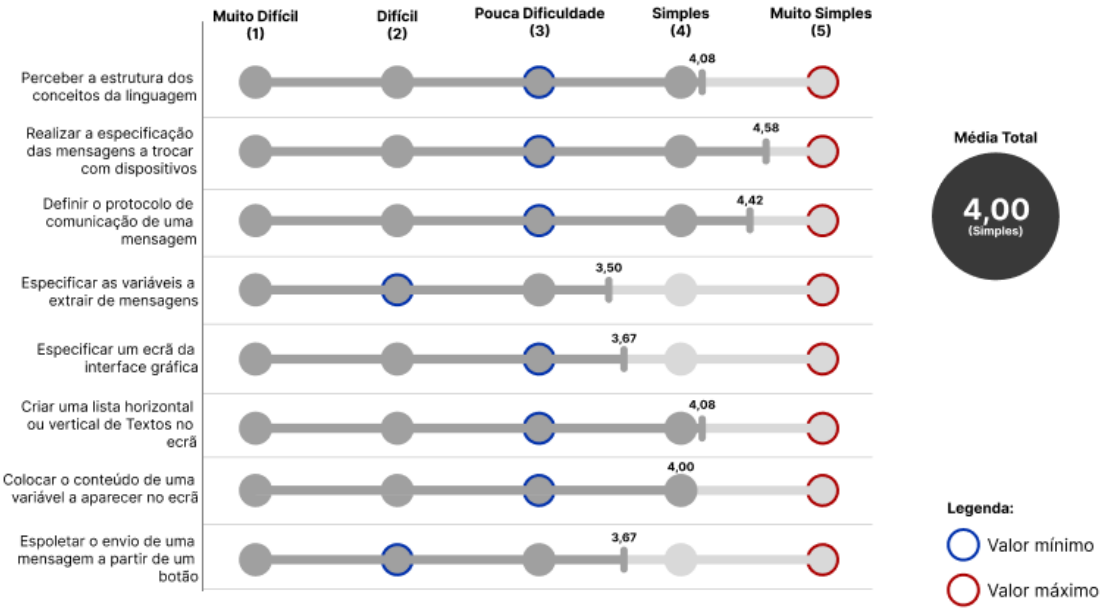

Figura 6.6: Questionário - Médias de pontuações nas respostas à usabilidade da *genui*

As repostas às perguntas sobre a usabilidade da *genui* como linguagem para especificar as mensagens a trocar com os dispositivos e os componentes da interface gráfica, estão representadas na Figura 6.6. Foi calculada para cada alínea da pergunta (facilidade de utilização da *genui*) a média do valor das respostas dadas, de 1 a 5 como é apresentado na Figura. Nesta, foram ainda anotados os valores mínimos e máximos de pontuações dadas pelos utilizadores. Nesta secção de usabilidade da *genui* vemos algumas oscilações nas pontuações dadas a diferentes pontos:

- A especificação da extração das variáveis na *genui* aparenta não ser óbvia para os utilizadores dado que, em média, obtém uma classificação "Pouca Dificuldade". Embora este ponto possa estar relacionado com uma curva de aprendizagem de uma nova linguagem, uma forma de mitigar a dificuldade será melhorar a documentação dos conceitos da *genui* com apresentação de exemplos de como extrair valores nos vários formatos. Este ponto pode estar ainda pior cotado dado que é necessário que o utilizador conheça XPath ou JsonPath caso queira especificar uma extração de um destes tipos de conteúdo;
- A especificação da interface gráfica volta nesta secção a ser uma das alíneas com pontuação baixa (comparativamente). Este caso pode dever-se ao facto de os participantes não estarem habituados à estrutura da especificação dos componentes como é feita na *genui*, devendo por isso existir uma curva de aprendizagem que não foi possível ultrapassar apenas com uma sessão de experimentação. Deve ainda assim ser melhora a documentação de como devem ser especificados os componentes, com apresentação de exemplos concretos do que é possível fazer. É ainda de notar que a *genui* não permite todos os *designs* possíveis dado que não está completamente implementada.
- A dificuldade no espoletar envios de mensagens a partir de botões pode dever-se à conjugação de ambos os pontos anteriores, dado que é necessário, por um lado, especificar o botão (que é um componente da GUI) como, por outro lado, é necessário associar o botão a uma extração de variável especificada.

Resultados da Satisfação de Uso da Solução

Por último, dado que os participantes da experimentação estiveram também presentes da construção dos requisitos da solução (Tabela 3.1 no Capítulo 3), foi questionado se a solução respondia aos mesmos. Desta forma, dado que os requisitos foram pensados para que fosse possível mitigar um problema que existia na área, caso haja uma boa satisfação da implementação dos mesmos, podemos dizer que existe uma igual satisfação na resolução do problema inicial através desta solução desenvolvida. Pode não existir a relação entre estes dois pontos dado que os requisitos de "Definição das respostas automáticas a determinados eventos/mensagens recebidas"e "Possibilitar o armazenamento do histórico de valores recebidos por mensagens"não chegaram a ser implementados. Deste modo podemos apenas concluir face aos que puderam de facto ser experimentados. As respostas à satisfação da utilização da solução, tendo por base os requisitos finalizados, estão presentes na Figura 6.7. Foi cálculada a média, bem como anotado o valor mínimo e máximo das pontuações de cada alínea da pergunta (satisfação de utilização da solução), de 1 a 5 como é apresentado na Figura. É de notar que a satisfação de utilização, neste caso, está diretamente relacionada com a satisfação da implementação feita para desenvolver o requisito.

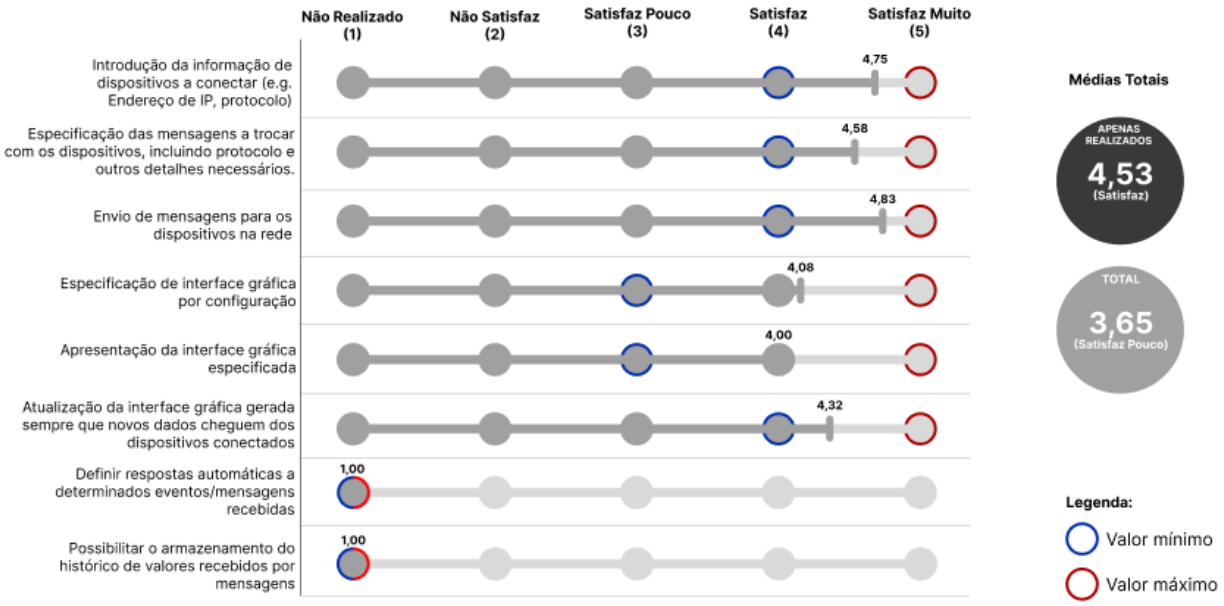

Figura 6.7: Questionário - Médias de pontuações nas respostas à satisfação de uso da solução

O total das respostas oferece uma pontuação de 3.65 ("Satisfaz Pouco"), assumindo todos os requisitos. Porém, ignorando os requisitos não implementados, mostrando apenas aqueles foi possível experimentar durante a sessão, obtemos uma pontuação de 4.53 ("Satisfaz"). Podemos assim dizer que esta solução "Satisfaz"também quanto à mitigação do problema inicial.

Todas as afirmações nesta secção assumem que a amostra de 12 participantes é suficiente, no entanto, é necessário realizar mais sessões de experimentação e recolher mais respostas ao questionário.

Capítulo 7

Conclusão e trabalho futuro

Este capítulo contém as principais conclusões a retirar deste projeto acompanhadas da apresentação dos objetivos cumpridos. Serão ainda expostas algumas ideias para trabalho futuro de modo a dar continuidade ao desenvolvimento da ferramenta desenvolvida.

7.1 Objetivos Cumpridos e Contribuições

A solução desenvolvida teve por base os objetivos propostos na Secção 1.4. Era assim esperado, de uma forma genérica, que no final do projeto existisse uma solução que permitisse realizar a gestão das comunicações com dispositivos, centralizando os dados obtidos e apresentando-os numa interface gráfica. Esta ferramenta deveria ainda personalizável ao nível das mensagens a trocar com os dispositivos, bem como na configuração da interface gráfica.

Tabela 7.1: Listagem de Objetivos e Sub-Objetivos Cumpridos

ID	**Breve descrição**	**Estado Final**
O1	Gerir comunicações com os dispositivos na rede	Parcial
S1.1	Criar DSL para especificação de dispositivos e mensagens	Finalizado
S1.2	Desenvolver módulo de comunicações	Parcial
O2	Criar camada de tratamento de dados recebidos	Finalizado
O3	Gerar Interface Gráfica	Finalizado
O3.1	Criar DSL para especificação da GUI	Finalizado
O3.2	Renderizar GUI com base em especificação	Finalizado

Na Tabela 7.1 são apresentados os objetivos e sub-objetivos com o estado final de conclusão dos mesmos. O objetivo O1 agrega os sub-objetivos S1.1 e S1.2 estando com estado "Parcial"dado que S1.2 está "Parcial". O S1.2 finalizou com este estado dado que, embora se possa dizer que a solução final consegue comunicar

com dispositivos na rede através da especificação em S1.1, existiram dois requisitos funcionais que não foram realizados - RF8 e RF9. Estes dois casos de uso tiveram atribuída uma baixa prioridade e por falta de tempo para conclusão dos mesmos, não foram desenvolvidos.

Podemos assim concluir que obtivemos no final do projeto uma solução que resolve os problemas descritos na Secção 1.5, mas que não contém algumas das funcionalidades pedidas, nomeadamente a de persistir os valores recebidos e de espoletar respostas automáticas a mensagens recebidas.

7.2 Trabalho Futuro

Para que a solução possa servir a mais cenários, é necessário, numa primeira instância, fechar a versão inicial com a conclusão das funcionalidades em falta - UC8 e UC9. Depois de concluídos estes requisitos, podemos passar para outras funcionalidades sugeridas nas secções que se seguem.

7.2.1 Melhoria contínua da genui

O desenvolvimento da *genui* para especificação da interface gráfica deve continuar em melhorias iterativas de modo a que cada vez mais customizações sejam possíveis ou que haja origem de novos componentes gráficos (e.g componente de tabela, carregar imagens a partir de um sítio rede).

7.2.2 Novos protocolos de comunicação

Em IoT não existe nenhum standard de protocolo comunicação, pelo que quanto maior for o número de protoclos suportados pela aplicação, a mais situações se vai conseguir adaptar a solução. Devem ser implementado o AMQP e MQTT que foram os restantes protocolos revistos na análise do Estado da Arte. Para este último protocolo deverá ser possível ainda suportar diferentes intermediários de mensagem se possível.

7.2.3 Grupos de Workers

Deveria ser possível gerar mais do que apenas *Worker* de cada tipo. Atualmente existe apenas um *LoggerWorker*, *ReceiverWorker*, *RequesterWorker*, entre outros, mas no futuro pode ser necessário, por exemplo:

- Executar dois servidores com protocolos diferentes para receber mensagens (i.e. dois *ReceiverWorker* em paralelo);
- Adicionar mais *ParserWorker*s para casos em que o número de mensagens a enviar ou receber ao mesmo tempo seja muito grande;

- Adicionar mais *RequesterWorker*s para melhor dividir o envio das mensagens em processos diferentes.

Para que isto seja feito, os *Worker*s têm que deixar de ser declarados individualmente como exposto na Figura 5.8 (Secção 5.3.2), passando a poder haver uma lista de cada Worker. No caso do segundo exemplo, deve ser analisado o possível impacto na escrita à *snapshot* pois dado que poderia haver várias escritas ao mesmo tempo na mesma por *Workers* diferentes, poderá ser necessária a implementação de mecanismos de semáforo.

7.2.4 Ferramenta de drag-and-drop

A realização da especificação da GUI e renderização da mesma foi das pontuações mais baixas obtidas a partir do questionário de usabilidade realizado - Secção 6.3.3. Por este motivo e de forma a reduzir a eventual curva de aprendizagem da *genui*, deve ser construida uma ferramenta *drag-and-drop* para gerar a especificação da interface gráfica, com ou sem recurso à *genui*. A vantagem desta abordagem seria o facto de permitir que o utilizador consiga visualizar a especificação numa linguagem visual ao invés de textual. A especificação textual atual apresenta e pede demasiada informação ao utilizador e por isso torna-se difícil prever o resultado final, especialmente em especificações longas. Com uma ferramenta *drag-and-drop* poderiam ser ocultadas muitas informações ao utilizador como as relações entre *Widgets* ou o posicionamento e tamanho dos *Widgets*.

Bibliografia

Babun, Leonardo et al. (2021). «A survey on IoT platforms: Communication, security, and privacy perspectives». Em: *Computer Networks* 192 (April 2020), p. 108040. issn: 13891286. doi: 10.1016/j.comnet.2021.108040. url: https://doi.org/10.1016/j.comnet.2021.108040.

Bajracharya, Biju e David Hua (2020). «IoT Education using Learning Kits of IoT Devices». Em: pp. 1–5.

Basak, Sukanta e Md. Shazzad Hosain (2014). «Software Testing Process Model from Requirement Analysis to Maintenance». Em: *International Journal of Computer Applications* 107.11, pp. 14–22. doi: 10.5120/18795-0147.

Bhardwaj, S e A Kole (2016). «Review and study of internet of things: It's the future». Em: *2016 International Conference on Intelligent Control Power and Instrumentation (ICICPI)*. IEEE, pp. 47–50. doi: 10.1109/ICICPI.2016.7859671.

Brimzhanova, S. S. et al. (2019). «Cross-platform compilation of programming language Golang for Raspberry Pi». Em: *ACM International Conference Proceeding Series*. doi: 10.1145/3330431.3330441.

Carvalho Pedrosa, António e Sílvio Marques A. Gama (2016). *Introdução Computacional à probabilidade Estatística com Excel*. 3ª ed. Porto: Porto Editora. Cap. 8, p. 589. isbn: 978-972-0-01990-5.

CFI (2021). *Customer Satisfaction*. url: https://corporatefinanceinstitute.com/resources/knowledge/other/measuring-customer-satisfaction/ (acedido em 06/02/2021).

Chen, Shanzhi et al. (2014). «A Vision of IoT: Applications, Challenges, and Opportunities With China Perspective». Em: *IEEE INTERNET OF THINGS* 1 (4), pp. 349–359. url: https://ieeexplore.ieee.org/abstract/document/6851114.

Chilipirea, Cristian et al. (jun. de 2016). «Presumably Simple: Monitoring Crowds Using WiFi». Em: IEEE, pp. 220–225. isbn: 978-1-5090-0883-4. doi: 10.1109/MDM.2016.42.

Cutting, David (2015). «Evaluation of Long-Held HTTP Polling for PHP / MySQL Architecture Evaluation of Long-Held HTTP Polling for PHP / MySQL Architecture». Em: doi: 10.13140/RG.2.2.26264.80648.

Eeles, Peter (nov. de 2005). «Capturing Architectural Requirements». Em: *IBM - developerWorks*. url: https://web.archive.org/web/20210315183751/http://www.ibm.com/developerworks/rational/library/4706-pdf.pdf.

Gronback, Richard C. (2009). *Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit*. Pearson, p. 736. isbn: 9780321534071. url: http://dns.uls.cl/%7B~%7Dej/web%7B%5C_%7Dtalf%7B%5C_%7D2018/AW,.Eclipse.

Modeling.Project.A.Domain-Specific.Language.Toolkit.(2009),.1Ed.%7B%5C%%7D5B0321534077%7B%5C%%7D5D.pdf.

Heidenreich, Florian et al. (2009). «Derivation and refinement of textual syntax for models». Em: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 5562 LNCS, pp. 114–129. issn: 03029743. doi: 10.1007/978-3-642-02674-4_9.

Hevner, Alan R. et al. (2004). «Design science in information systems research». Em: *MIS Quarterly: Management Information Systems* 28.1, pp. 75–105. issn: 02767783. doi: 10.2307/25148625.

Klint, P., T. van der Storm e J. Vinju (2009). «RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation». Em: *2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation*, pp. 168–177. doi: 10.1109/SCAM.2009.28.

Koen, Peter et al. (mar. de 2001). «Providing Clarity and A Common Language to the "Fuzzy Front End"». Em: *Research-Technology Management* 44.2, pp. 46–55. issn: 0895-6308. doi: 10.1080/08956308.2001.11671418. url: https://www.tandfonline.com/doi/full/10.1080/08956308.2001.11671418.

Koen, Peter A., Heidi M.J. Bertels e Elko J. Kleinschmidt (2014). «Managing the front end of innovation-part II: Results from a three-year study». Em: *Research Technology Management* 57.3, pp. 25–35. issn: 19300166. doi: 10.5437/08956308X5703199.

Kuznetsov, Stacey e Eric Paulos (2010). «Rise of the expert amateur: DIY projects, communities, and cultures». Em: *NordiCHI 2010: Extending Boundaries - Proceedings of the 6th Nordic Conference on Human-Computer Interaction* (Figure 1), pp. 295–304. doi: 10.1145/1868914.1868950.

Larman, Craig (2004). *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development (3rd Edition)*. Upper Saddle River, NJ, USA: Prentice Hall PTR. isbn: 0131489062.

Loreto, S et al. (2011). «Known Issues and Best Practices for the Use of Long Polling and Streaming in Bidirectional HTTP». Em: *Internet Engineering Task Force (IETF)*, pp. 1–21.

Naik, Nitin (2017). «Choice of effective messaging protocols for IoT systems: MQTT, CoAP, AMQP and HTTP». Em: *2017 IEEE International Systems Engineering Symposium (ISSE)*, pp. 1–7. doi: 10.1109/SysEng.2017.8088251.

Nielsen, Jakob (1994). *Usability Engineering*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc. isbn: 9780080520292.

Ojo, Mike O. et al. (2018). «A Review of Low-End, Middle-End, and High-End Iot Devices». Em: *IEEE Access* 6.710583, pp. 70528–70554. issn: 21693536. doi: 10.1109/ACCESS.2018.2879615.

Okano, Marcelo Tsuguio (2017). «IOT and Industry 4.0: The Industrial New Revolution». Em: *ICMIS-17 - International Conference on Management and Information Systems* (September), pp. 75–82.

Shelby, Zach, Klaus Hartke e Carsten Bormann (jun. de 2014). *The Constrained Application Protocol (CoAP)*. RFC 7252. doi: 10.17487/RFC7252. url: https://www.rfc-editor.org/info/rfc7252.

Stevens, W. Richard (1999). *UNIX Network Programming*. 2ª ed. Vol. 2. Prentice Hall PTR.

Strembeck, Mark e Uwe Zdun (2009). «An approach for the systematic development of domain-specific languages». Em: *Software - Practice and Experience* 39.15, pp. 1253–1292. issn: 00380644. doi: 10.1002/spe.936.

Ulaga, Wolfgang e Andreas Eggert (jan. de 2006). «Value-Based Differentiation in Business Relationships: Gaining and Sustaining Key Supplier Status». Em: *Journal of Marketing - J MARKETING* 70, pp. 119–136. doi: 10.1509/jmkg.2006.70.1.119.

Voelter, Markus et al. (2013). *DSL book*. Ed. por Dslbook.org, p. 560.

Woodall, T (2003). «Conceptualising 'value for the customer': an attributional, structural and dispositional analysis». Em: *Academy of Marketing Science Review* 2003.12. issn: 1526-1794.

Apêndice A

Questionário de Usabilidade e Satisfação de Uso

Geração de UI para controlo e monitorização de dispositivos na rede | TMDEI 2022

Estimativa de duração do questionário: 5 min.
Este questionário enquadra-se no âmbito da Tese de Mestrado para obtenção do grau de Mestre em Engenharia Informática, na especialização em Engenharia de Software.
O objetivo deste formulário é obter, de uma forma estruturada, o feedback de utilizadores quanto à usabilidade da solução criada.

Resumo da solução:

A solução criada permite:

1. Gerir as comunicações com dispositivos na rede (especificar dispositivos e mensagens a trocar);
2. Criar uma interface gráfica para apresentação de dados recebidos dos dispositivos;

Ambos este processos devem ser feitos através da utilização da ferramenta MPS, com a qual é possível criar as especificações tanto para configurar os dispositivos na rede, como para definir quais os dispositivos a conectar e com qual protocolo.

Experimentação:

Para responder ao questionário abaixo, deverá ter conseguido (ou tentado) executar os seguintes passos durante a sessão de experimentação:

1. Fazer download do MPS;
2. Fazer download do projeto;
3. Criar uma "Application"
4. Definir mensagens;
5. Definir interface gráfica - adicionar componentes gráficos;
6. Gerar código a partir das especificações;
7. Atualizar código do projeto (passo 2) com os ficheiros gerados;
8. Executar a aplicação.

Nota: Este questionário deve ser respondido no final da sessão de experimentação organizada.

*Obrigatório

Informação sobre o	Esta secção serve para reunir informações sobre os utilizadores que permitam validar sua relação com o público alvo desta

1. Já tentou desenvolver soluções segundo princípios DIY (Do-It-Yourself)*?

   * DIY (Do-It-Yourself) é o método de construir, alterar, ou arranjar coisas por si próprio sem a ajuda directa de profissionais ou peritos certificados da área.

   Marcar apenas uma oval.

   ◯ Sim
   ◯ Não

2. Já alguma vez tentou criar um ecossistema de dispositivos em rede?

   Por exemplo:
   - a criação de uma rede de sensores para criação de uma casa inteligente;
   - um sistema de rega inteligente.

   Marcar apenas uma oval.

   ◯ Sim
   ◯ Não

3. Já tentou realizar, a título individual ou profissional, algum dos seguintes desenvolvimentos?

   Nota: Coloque em "Outra opção", outros desenvolvimentos que ache relevantes.

   Marcar tudo o que for aplicável.

   ☐ Configurar sensores (p.e. leitores de temperatura, pluviosidade, humidade do ar)
   ☐ Criar um programa capaz de gerir comunicações com diversos dispositivos ou sensores
   ☐ Criar interfaces gráficas (p.e. GWT, QT)
   ☐ Outra: _____

Ferramentas - MPS

Deverá responder às questões desta secção tendo apenas como referência as ferramentas utilizadas para definir as especificações criadas.

4. Utilização do MPS *

Marcar apenas uma oval por linha.

	Muito Difícil	Difícil	Normal	Simples	Muito Simples
Instalar o MPS	◯	◯	◯	◯	◯
Usar o MPS para realizar as especificações de Mensagens	◯	◯	◯	◯	◯
Usar o MPS para realizar as especificações da Interface Gráfica	◯	◯	◯	◯	◯
Preencher os campos pedidos da especificação	◯	◯	◯	◯	◯
Gerar código a partir da especificação	◯	◯	◯	◯	◯
Encontrar os ficheiros gerados e copiá-los para o projeto principal	◯	◯	◯	◯	◯

Realizar as Especificações

As perguntas nesta secção são apenas relativas à fase de especificação de mensagens e componentes da interface gráfica.

5. Utilização da Linguagem - *genui* *

Marcar apenas uma oval por linha.

	Muito Difícil	Difícil	Pouca Dificuldade	Simples	Muito Simples
Percebi a estrutura dos conceitos da linguagem	◯	◯	◯	◯	◯
Realizar a especificação das mensagens a trocar com dispositivos	◯	◯	◯	◯	◯
Definir o protocolo de comunicação de uma mensagem	◯	◯	◯	◯	◯
Especificar as variáveis a extrair de mensagens	◯	◯	◯	◯	◯
Especificar um ecrã da interface gráfica	◯	◯	◯	◯	◯
Criar uma lista horizontal ou vertical de Textos no ecrã	◯	◯	◯	◯	◯
Colocar o conteúdo de uma variável a aparecer no ecrã	◯	◯	◯	◯	◯
Espoletar o envio de uma mensagem a partir de um botão	◯	◯	◯	◯	◯

Satisfação de Uso

Esta secção tem como objetivo perceber a eficácia com que foram implementados os requisitos da solução.
As respostas são dadas numa escala de 1 a 5 onde:
1 - Significa o não cumprimento do requisito ou este possui uma implementação pouco eficaz na resolução do problema;
2 - Significa que é cumprida uma pequena parte do requisito proposto ou a implementação não é a mais eficaz para o problema;
3 - Significa que é cumprido o requisito mas a implementação é pouco eficaz;
4 - Significa que é cumprido o requisito e a implementação é eficaz na solução do mesmo;
5 - que é cumprido o requisito e a implementação é considerada muito eficaz na solução do mesmo.

6. Implementação dos Requisitos

Marcar apenas uma oval por linha.

	1	2	3	4	5
Introdução da informação de dispositivos a conectar (e.g. Endereço de IP, protocolo)	◯	◯	◯	◯	◯
Especificação das mensagens a trocar com os dispositivos, incluindo protocolo e outros detalhes necessários.	◯	◯	◯	◯	◯
Envio de mensagens para os dispositivos na rede	◯	◯	◯	◯	◯
Especificação de interface gráfica por configuração	◯	◯	◯	◯	◯
Apresentação da interface gráfica especificada	◯	◯	◯	◯	◯
Atualização da interface gráfica gerada sempre que novos dados cheguem dos dispositivos conectados	◯	◯	◯	◯	◯
Definir respostas automáticas a determinados eventos/mensagens recebidas	◯	◯	◯	◯	◯
Possibilitar o armazenamento do histórico de valores recebidos por mensagens	◯	◯	◯	◯	◯

7. Durante a execução do programa sentiu perdas de performance ou teve uma performance pior do que a esperada em alguma das seguintes situações?

Marcar tudo o que for aplicável.

- [ ] O arranque do programa é lento
- [ ] A interface gráfica demorava a renderizar
- [ ] As mensagens demoravam mais do que o esperado a ser enviadas
- [ ] As mensagens eram enviadas mas a interface gráfica demorava a atualizar os valores recebidos
- [ ] A geração dos ficheiros de especificação demora muito tempo
- [ ] O computador onde foi executada a solução ficou lento ou deixou de responder
- [ ] O computador onde foi executada a solução gastou muitos recursos de memória ou CPU enquanto corria a ferramenta desenvolvida

8. Quão provável é que utilize esta ferramenta para desenvolver projetos IoT no futuro? *

(1 - Não vou utilizar / 5 - Vou Utilizar)

Marcar apenas uma oval.

	1	2	3	4	5	
Não vou utilizar	◯	◯	◯	◯	◯	Vou utilizar

Figura A.1: Questionário de Usabilidade e Satisfação de Uso

Apêndice B

Diagrama de Classes da genui - UC1 e UC2

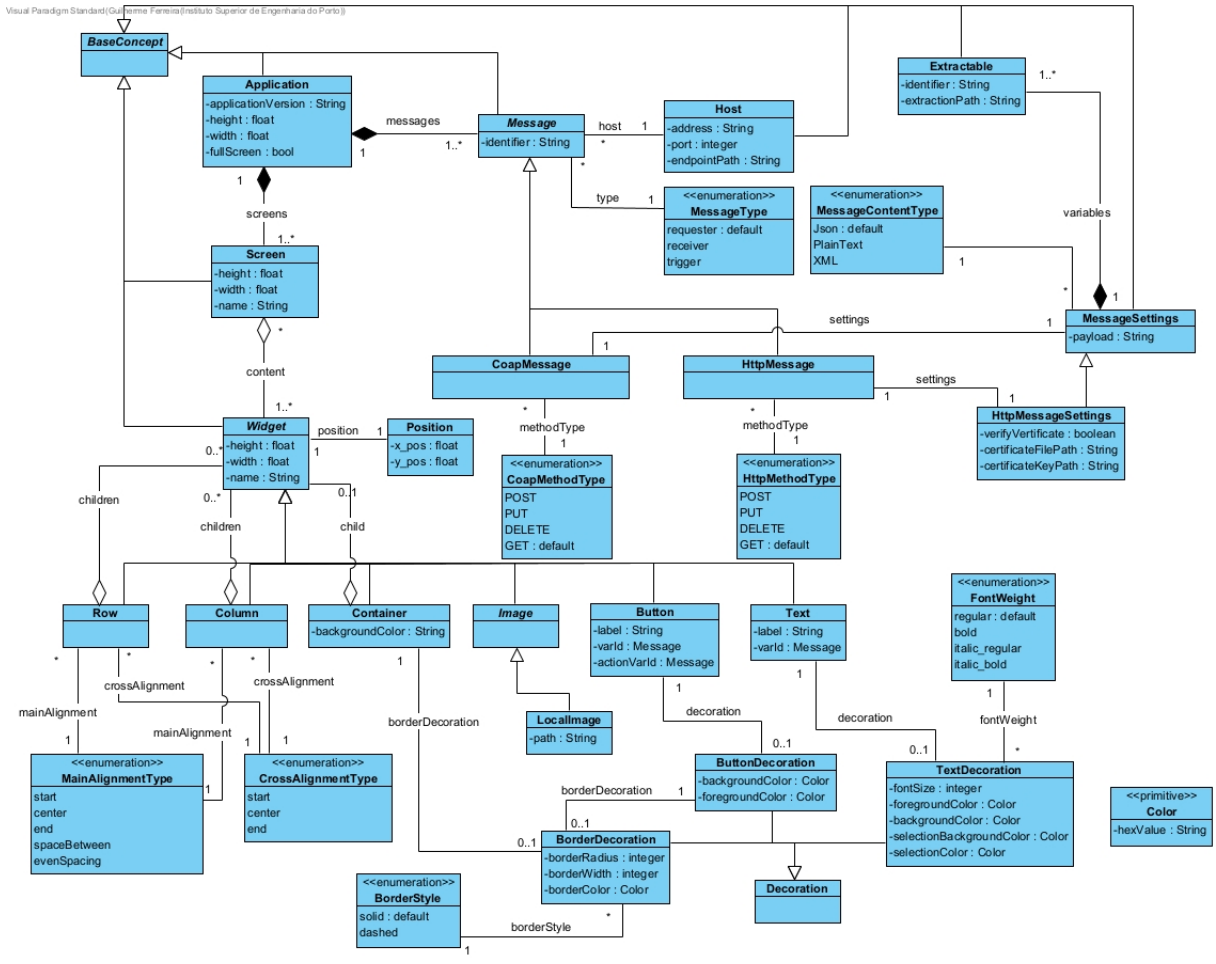

Figura B.1: Diagrama de Classes completo da gramática para especificações do UC1 e UC2