Pontifícia Universidade Católica
do Rio de Janeiro

Polyana Sampaio Ramos Barboza

On the application to the eHealth domain of a software framework that generates agent-based intelligent applications

Dissertação de Mestrado

Dissertation presented to the Programa de Pós–graduação em Informática of PUC-Rio in partial fulfillment of the requirements for the degree of Mestre em Informática.

Advisor: Prof. Carlos José Pereira de Lucena

Rio de Janeiro
April 2022

Pontifícia Universidade Católica
do Rio de Janeiro

Polyana Sampaio Ramos Barboza

On the application to the eHealth domain of a software framework that generates agent-based intelligent applications

Dissertation presented to the Programa de Pós–graduação em Informática of PUC-Rio in partial fulfillment of the requirements for the degree of Mestre em Informática. Approved by the Examination Committee.

Prof. Carlos José Pereira de Lucena
Advisor
Departamento de Informática – PUC-Rio

Prof. Andrew Diniz da Costa
Departamento de Informática – PUC-Rio

Prof. Alessandro Fabricio Garcia
Departamento de Informática – PUC-Rio

Rio de Janeiro, April 25th, 2022

Polyana Sampaio Ramos Barboza

Graduated in Appied Math by the Applied Math School of Getulio Vargas Foundation (FGV EMAp).

Bibliographic data

Sampaio Ramos Barboza, Polyana

On the application to the eHealth domain of a software framework that generates agent-based intelligent applications / Polyana Sampaio Ramos Barboza; advisor: Carlos José Pereira de Lucena. – Rio de janeiro: PUC-Rio , Departamento de Informática, 2022.

v., 67 f: il. color. ; 30 cm

Dissertação (mestrado) - Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática.

Inclui bibliografia

1. Informática – Teses. 2. eSaúde. 3. Monitoramento Remoto de Pacientes. 4. Framework. 5. Sistemas Multiagentes. 6. Swift. 7. Arquitetura BDI. I. José Pereira de Lucena, Carlos. II. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. III. Título.

CDD: 004

To my parents and sister, for their support
and encouragement.

Acknowledgments

I would like to thank God, for always knowing the best moments to realize our desires.

To my parents, for all the values they taught me and the resources they made available, especially their love and patience in my moments of great stress and distress.

To my twin sister, Natalie Barboza, for all tireless support and prayers, all trust placed in me, all words of encouragement, and all help in the domain research around the health field.

To the Carmelite sisters from Rio de Janeiro and Três Pontas, for all prayers and kindness.

To my godmother Marilena Barboza, for being my inspiration in academic life and dedication to studies.

To Thaís Freire, for all companionship and emotional support, especially in the last months, giving me the strength to reach the end.

To my soul sister, Fernanda Duarte, for always knowing the right words and for all support in my academic and personal life. And to Guga, for the illustrious presence on defense day.

To Pedro Amaro, for being the brother and tireless companion.

To my family and friends, for understanding my absence in many important moments.

To my Youth Team of Our Lady, for all the prayers and follow-up.

To FGV/DAPP, in the person of Marco Ruediger, for all support and for making it all possible.

To Juliana Mayrinck, for all understanding, support, and friendship since the beginning.

To Danilo Carvalho and Lucas Roberto, for understanding my absences, for reassuring me, and for their friendship.

To my advisor, Professor Lucena, for all generosity in teaching and patience guiding me to the best path. It was an honor to be your student.

To my co-advisor, Professor Andrew da Costa, for all the countless hours spent on meetings, patience in teaching and answering my questions, and dedication to walking with me to make this a good job. Without your support, it would not be possible.

To the empirical evaluation participants, for the voluntary participation, which was very helpful in completing and improving this work.

To CNPq and PUC-Rio, for the aids granted, without which this work does not could have been accomplished. This study was financed in part by

the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001.
This work was supported by PPI Softex program, Convênio nº 0200-10/2021/SOFTEX/PUCRio/Residência at TIC, financed by the Ministry of Science, Technology and Innovations with resources from Law nº 8.248, of October 23, 1991.

Abstract

Sampaio Ramos Barboza, Polyana; José Pereira de Lucena, Carlos (Advisor). **On the application to the eHealth domain of a software framework that generates agent-based intelligent applications**. Rio de Janeiro, 2022. 67p. Dissertação de Mestrado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

The healthcare area is emerging as a fertile ground for scientific research in Information Technology. Research activities in this field allow us to address several issues to promote technological development. In addition, regarding mobile device apps, the leading app stores registered significant growth in the number of available eHealth apps. One of the causes of this growth derives from the pandemic reality we have faced since the beginning of 2020.

In order to confront the different health challenges presented currently, the use of multi-agent systems has been considered a good approach, dealing with, for example, distribution, pro-activity and autonomy of systems. Although several known platforms that use software agents, some of them do not offer appropriate support to develop agents, such as the iOS platform. Thus, in this dissertation we have proposed a BDI framework for iOS that aims to support the development of health mobile apps with software agents. This is known as Swift Agent Development Framework for Health (SADE4Health) and it uses native iOS resources, such as Healthkit, comprising a central repository for health and fitness data to access and share data while maintaining the user's privacy and control. To show how the framework supports the development of new iOS apps with software agents, the minimum necessary steps to create an agent using health features offered by iOS are explained, as well as a modeled use scenario based on it. Furthermore, a use scenario related to remote monitoring of patients' vital signs that illustrates how to develop an instance of the proposed framework is presented. Finally, an empirical evaluation with iOS developers to measure the framework usability brought important findings.

Keywords

eHealth; Remote Patient Monitoring; Framework; Multiagent Systems; Swift; BDI Architecture.

Resumo

Sampaio Ramos Barboza, Polyana; José Pereira de Lucena, Carlos. **Aplicação ao domínio e-health de um framework que gera aplicações inteligentes baseadas em agentes**. Rio de Janeiro, 2022. 67p. Dissertação de Mestrado – Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

A área da saúde desponta como um terreno fértil para a pesquisa científica em Tecnologia da Informação. As atividades de pesquisa na área permitem abordar diversas questões para promover o desenvolvimento tecnológico. Além disso, em relação aos aplicativos para dispositivos móveis, as principais lojas de aplicativos registraram um crescimento significativo no número de aplicativos de saúde disponíveis. Esses crescimentos encontram uma de suas causas na realidade pandêmica que enfrentamos desde o início de 2020.

Para lidar com os diferentes desafios de saúde apresentados atualmente, o uso de sistemas multiagentes tem sido considerado uma boa abordagem para lidar, por exemplo, com distribuição, pró-atividade e autonomia dos sistemas. Embora várias plataformas conhecidas utilizem agentes de software, algumas delas não oferecem suporte adequado para o desenvolvimento de agentes, como a plataforma iOS. Assim, nesta dissertação propusemos um framework BDI para iOS que visa apoiar o desenvolvimento de aplicativos móveis de saúde com agentes de software. Esse framework é chamado de Swift Agent Development framework for health (SADE4Health) e usa recursos nativos do iOS, como o Healthkit, que é um repositório central de dados de saúde e condicionamento físico para acessar e compartilhar dados, mantendo a privacidade e o controle do usuário. Para mostrar como o framework suporta o desenvolvimento de novos aplicativos iOS com agentes de software, são explicados os passos mínimos necessários para criar um agente usando os recursos de saúde oferecidos pelo iOS, assim como um cenário de uso modelado a partir deles. Além disso, é apresentado um cenário de uso relacionado ao monitoramento remoto de sinais vitais de pacientes, ilustrando como desenvolver uma instância do framework proposto. Por fim, uma avaliação empírica com desenvolvedores iOS para medir a usabilidade do framework trouxe importantes achados.

Palavras-chave

eSaúde; Monitoramento Remoto de Pacientes; Framework; Sistemas Multiagentes; Swift; Arquitetura BDI.

Table of contents

List of figures

List of tables

List of Abreviations

ACL – Agent Communication Language

ACM – Association for Computing Machinery

AMS – Agent Management System

arXiv – e-Print archive

BDI – elief-Desire-Intention Reasoning

DAO – Database Access Object

ECG – Electrocardiogram

FIPA – Foundation of Intelligent Physical Agents

IEEE – Institute of Electrical and Electronics Engineers

IoT – Internet of Things

JADE – JAVA Agent Development Framework

MAS – Multi-Agent System

MVC – Model, View, and Controller

PubMed – Publisher Medical Literature Analysis and Retrieval System Online

RFID – Radio-Frequency Identification

SADE4Health – Swift Agent Development Framework for Health

UML – Unified Modeling Language

WWDC – Apple Worldwide Developers Conference

We think too much and feel too little
More than machinery, we need humanity
More than cleverness, we need kindness and gentleness

Charlie Chaplin, *The Great Dictator.*

1
Introduction

The multi-agent system paradigm [10] has been used in different domains due to features that handle the increasing complexity of software systems being developed. One of these domains is health, which has presented significant growth of offered solutions. The leading app stores have registered a rise in the number of mobile device apps available for eHealth apps. Until the first quarter of 2021, there were 53,054 healthcare apps available in Google Play [1] and 53,979 iOS healthcare apps available in the Apple Store [2], representing an 18.7 and 22.6 percent increase over the first quarter of 2020, respectively. One cause of this growth derives from the pandemic reality we have faced since the beginning of 2020 [3, 4]. Reflecting this context, we succeeded in designing and developing innovative and mobile technological solutions using the multi-agent system paradigm to help leverage the healthcare sector to a technological development level compatible with its importance and criticality levels.

1.1
Motivation and Objectives

Much of the current operational problems of medical care and hospitals can be solved through technological support. The application of computational solutions to hospital activities has the capacity to transform the present reality by, for instance, improving the work processes. As examples of improvements introduced by the use of innovative technological solutions, we can name: 1- Changes in the way the physician-patient relationship occurs, due to remote patient monitoring possibilities [5]; 2- Ease of information access and sharing among the medical team and the patients' relatives [6]; 3- More mobility for patients, whose health status can be monitored from home or work without being physically restricted to hospital facilities; 4- Possibility of collaborative work between the local team and external professionals; it allows a second opinion about patients' diagnoses and treatments, since patient information is already in a distributed database; 5- Possibility of automatic processes, such as vital patient data collection, by using sensors; 6- Remote and real-time monitoring of patient health conditions; 7- Alerts to healthcare professionals in emergency situations; 8- Decrease in elapsed time for detection of anomalies in

the vital signs of monitored patients, by using software agents; in this context, software agents consist of computational entities that perform activities in response to emergency situations.

Investments in technology permit attaining results that extrapolate the operational level. Since operational problems related to medical routines can be solved by technology solutions, it may allow hospital managers to focus on more relevant, strategic level, questions. However, to develop eHealth solutions, depending on the platform, especially those that use new programming languages, some challenges need to be dealt with, as follows.

- Offering reliable libraries/frameworks that support the development of software agents and offer a set of features that make it possible to maximize the use of agents;
- Indicating how native resources of some platform can be used and integrated to software agents;
- Offering health resources that help the development of health solutions.

Based on the aforementioned challenges, this dissertation aims to contribute to the development of mobile eHealth solutions from the Swift Agent Development Framework for Health (SADE4Health), an iOS framework developed in Swift considering the eHealth domain by using software agents reasoning through a BDI architecture.

The iOS platform was chosen to offer this new framework because it is one of the best known of the platforms that do not offer recognized solutions in support of the development of multi-agent systems. The platform uses Swift as the main programming language, presented in 2014 to create apps, and it offers several native resources used for available App Store apps, such as health kits. With regard to the health scenario, the Apple ecosystem has paid special attention to recognized, developed kits, such as HealthKit [18], which permits health data accessing and sharing, forming a collaborative and sharing network to benefit the user's health.

Thus, the objectives of our research are as follows.

- Mapping which works were proposed and have a relation to Swift and software agents;
- Offering a framework that follows good practices of software engineering and provides features defined for any software agent;
- Allowing integration of the iOS platform's native resources, such as health and notification kits;
- Guaranteeing that health applications instantiated from the proposed framework can encompass the eight improvements listed above.

1.2 Assumptions

To conduct our research, initially we considered two assumptions. The first was a gap between multi-agent systems and the iOS developers' community, i.e., there are no solutions in Swift bearing in mind the use of software agents; consequently, no frameworks support such use. In order to confirm this assumption, we carried out systematic mapping of works involving agents and the Swift language, besides analyzing some related works.

The second assumption was that the cost-benefit of constructing a more generic and complex system would be rewarded because we were not facing a sizable independent application but rather a family of related applications.. Thus, a framework aggregating Software Engineering concepts and Apple frameworks would make it easier to create an application with software agents in Swift, not requiring the need to recreate a complex and robust structure. To confirm that assumption, we verified which parts of SADE4Health would be reused in a health instance that has software agents for its main tasks, presented an empirical evaluation with iOS developers containing this hypothesis, and modeled a classic use scenario considering SADE4Health dependencies.

1.3 Contributions and Limitations

The main contributions of this research are the following:

1. Offering a multi-agent framework in Swift for the iOS platform and integrated with native resources used in different domains, such as Notification resources.

2. Offering a health module integrated with the multi-agent structure and the HealthKit, native framework offered by Apple and that allows dealing with health data in iOS apps.

3. Guaranteeing that the proposed framework is FIPA-compliant, i.e., follows the specifications of the Foundation of Intelligent Physical Agents (FIPA) that established requirements patterns for a multi-agent system, the FIPA-compliant middleware [28] - a collection of standards intended to promote the interoperation of heterogeneous agents and the services that they can represent.

4. Enabling the reuse of our research, including domain analysis, designs, and implementations.

Limitations of the proposed approach include: (1) SADE4Health was only tested on the iOS platform, but it probably can be used on other Apple platforms; (2) CareKit and ResearchKit are other health kits offered by Apple, that are not being used in the framework; (3) the Health module needs more generic aspects of the health domain.

1.4 Organization of the Document

The dissertation is organized as follows. Chapter 2 provides some important definitions used in the research. Chapter 3 presents the works related to multi-agent frameworks, health domain, and iOS. Chapter 4 describes the SADE4Health framework and its architecture. Chapter 5 discriminates which steps are necessary to create an instance from it, models a use scenario, describes some unit tests, and explains the use of the framework from a use scenario related to remote monitoring of vital signs' patients. Chapter 6 describes the execution of an empirical evaluation with iOS developers. Chapter 7 presents a discussion of the work, and Chapter 8 provides the conclusions and future works.

2
Theoretical Definitions

2.1
General Concepts about Software Frameworks

Frameworks are tools used to generate applications related to a specific domain; that is, to handle a family of related problems [7]. The choice of using existing frameworks or developing new generators of applications is justified by the ability of this kind of tool to offer design and code reuse. This fact allows a boost in software development productivity and shorter time-to-market compared to traditional approaches.

Frameworks contain fixed and flexible points known as frozen spots and hot spots, respectively. Hot spots are extension points that allow developers to create an application from the framework instantiation process. In this case, developers should create specific application code to each hot spot, through the implementation of abstract classes and methods defined in the framework. Frozen spots, in turn, consist of the framework's kernel, corresponding to its fixed parts, previously implemented and hard to change, which will call one or more of the application's hot spots and will be present in each framework's instance [7].

Frameworks' building activities comprise three main steps: (i) domain analysis; (ii) design; (iii) instantiation process. The domain analysis step includes elicitation activity requirements, along with hot and frozen spots' definitions. The design step is responsible for drawing the hot and frozen spots through a modeling language, by using, for example, UML [8] diagrams to show the framework's extensible and flexible points. Design patterns [9] are also used in this phase. The instantiation phase corresponds to the application generation phase, through hot spot implementation [7].

2.2
Software Agents and Multi-Agent Systems

An agent [10] is an element of a computational system that is situated in some environment where it can perform autonomous actions, in order to reach the goals that are delegated to it. Agents contain properties, such as autonomy

and learning. The learning process is related to its capability of learning from its experience. Autonomy has been acknowledged as a key characteristic for its actuation, which it uses to decide how to act to satisfy its goals [11]. In this context, autonomy means the possibility of actuation without the intervention of humans or other systems, although the set of possible actions should be previously defined.

Although agents control its behavior and internal state, they do not have full control of the environment. Agents contain a set of actions that can carry out tasks, whose execution can result in changes in the environments. For this reason, one can consider that an agent can have partial control over its environment, being able to influence it, depending on the action that the agent decides to perform [12].

An additional concept is a multi-agent system (MAS), which is a society of software agents. A MAS [10] is a computational system containing many software agents, which interact and cooperate to solve their tasks. Following this idea, they are inserted into an environment with objects, relationships, resources, and possible operations, to work together to achieve similar or different goals.

2.3 Belief-Desire-Intention Reasoning

A Belief-Desire-Intention (BDI) architecture is a philosophical theory of practical reasoning, introduced by Bratman in 1987, explaining human reasoning with some attitudes: beliefs, desires and intentions. In 1991, Rao and Georgeff transformed it into a formal theory to define the agents' behavior, defining the way they interact with the environment, perceiving it and acting into it. BDI is about the path taken from perceptions to actions, about how the agent decides to perform in the environment.

Belief is the state related to the information that each agent has about the environment into which they are inserted; Desire is the motivational state, when an agent has goals it would like to achieve; and the Intention state is the deliberation one, when some goals are selected to be tried through concrete actions [13]. In a BDI architecture system, agents are submitted to each of these states as life cycles, where the input is the perception of the environment through sensors and the output is the actions through effectors. In the middle, we have the brain, where the agent reasons about what it knows and chooses the next step. This brain is exactly the core of the BDI architecture, where important functions are modeled to make the cycle work (Figure 2.1).

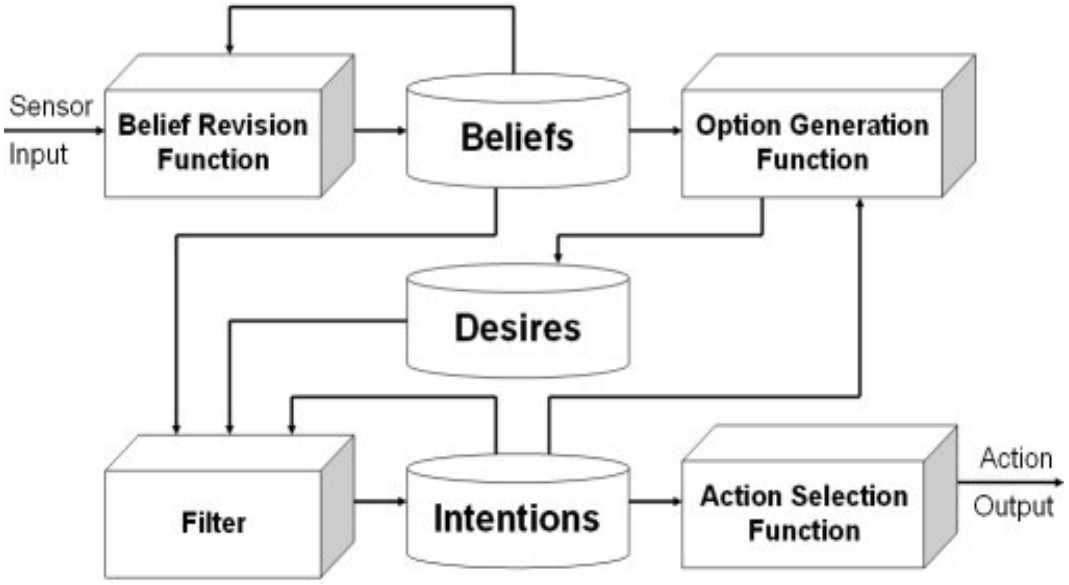

Figure 2.1: Generic BDI architecture [14]

In the BDI cycle represented by the figure above, four main functions are defined. The first function, called Belief Revision Function, receives the agent's beliefs and updates them, looking for inconsistencies, for example, and outputs the remaining beliefs. The second function, Option Generation Function, determines the agent's desires based on its beliefs and current intentions. The Filter serves as the deliberation function, selecting the desires that should be tried, converting them into intentions. At the end of the cycle, the fourth function, Action Selection Function, determines the actions to be executed and achieve the current intentions.

2.4 Apple Ecosystem

The programming language chosen to conduct this research was Swift [15], a modern language presented in 2014 by Apple. In 2018, Apple's CEO, Tim Cook, affirmed there were over 350,000 apps written in Swift in the App Store. These apps present a certain complexity that requires features to make them more powerful, such as distribution and autonomy, which, as seen, can be offered by a multi-agent system. On the other hand, the Swift language presents simplicity, clearly declared by its ambitious goal, "make programming simple things easy, and difficult things possible."

On June 4, 2018, at the Apple Worldwide Developers Conference (WWDC), Cook announced 20 million registered developers on iOS [16]. In 2020, these certainly more than 20 million developers attempted to reach peo-

ple behind the 1.5 billion active Apple devices [17] - according to the CEO at the beginning of 2020.

Focusing on the health domain, by the first quarter of 2021 there were 53,979 iOS healthcare apps available in the Apple Store [2]. These apps' developers have at their disposal several tools provided by Apple to support health data, features and research, such as HealthKit [18], CareKit [19] and ResearchKit [20].

Moreover, Apple has one crucial value for the health domain, considering the sensitivity of the data. User privacy and security are among their main concerns [21, 22]. Hence, iOS ecosystem health developers must consider this and be watchful about how data is being shared and stored in their apps, ensuring each user's protection through user permissions.

Also interesting is that most physicians who use mobile apps prefer iPhones [23, 24]. In the UK, for example, in 2017 the iPhone was the smartphone that doctors and nurses most commonly used, with 75.6% of the doctors owning an iPhone and 58.4% of the nurses [25].

2.5 Internet of Things (IoT)

IoT is a new field within Computer Science that has grown quickly in recent years. Kevin Ashton introduced the term Internet of Things in 1999 [26]. One can define IoT as a global network of smart devices that can sense and interact with their environment for communication with users and other things and systems alike. In this manner, things would be identified solely by using RFID (Radio-Frequency Identification) [27] tags in order to be connected to the Internet and publish their information. Things are physical objects such as refrigerators, cars, walking sticks, dog collars and whatever else one can think of.

The main motivation to expand the connectivity was the fact that the volume of data that people could publish on the network was much smaller than the available processing capability. Since devices with computational power were less dependent on human beings and able to capture the data that they would process, the costs of measuring, tracking, and controlling things could decrease.

Thereby, by using sensors, actuators and RFID-like technology, things of the environment can be identified, perceived and controlled in an autonomous manner. In such cases, the things themselves could inform when they need to be replaced, fixed or reported if they were appropriate for consumption [26].

One interesting view of IoT in the context of mobile devices is remote

health monitoring. For example, this type of monitoring has used vital sign monitors and smartwatches. This dissertation will address the former in a use scenario, capturing data from the use of agents. Furthermore, in the related works chapter, we will present examples of apps focused on patient monitoring.

3
Related Works

In this chapter, a set of works were analyzed and compared with our proposed framework. It is organized in sections depending on the type of contributions that each one offers.

3.1
Multi-Agent Frameworks

Considering some of the better known and most widely used frameworks with multi-agent systems, we analyzed frameworks that fit one or more of the following three requirements: i) support of software agents; ii) agents with BDI reasoning; iii) comply with the conventions established by FIPA [28] for a multi-agent system. The main frameworks studied were JADE [29], JADEX [30], and BDI4JADE [31].

JADE (JAVA Agent Development Framework) [29] is a framework to develop agent systems in Java. It simplifies multi-agent systems' implementation through FIPA-compliant middleware. The JADE's purpose is the development of distributed multi-agent applications based on the peer-to-peer communication architecture. Moreover, it is excellent in communication, making available white and yellow pages that allow publishing and discovering the features and the services offered by a peer (agent). In addition, it offers the JADE-LEAP, which is a modified version of the JADE platform that can run on Java-enabled mobile phones.

JADEX [30] and BDI4Jade [31] are layers to give JADE agents a BDI reasoning. However, the latter does not define agents through XML files, which benefits developers, especially considering integration with other technologies and searching for errors during compilation time.

Despite all the previously cited benefits of the frameworks, currently the iOS platform does not have a framework that is able to deliver these advantages. Considering that the platform is one of the world's most used, and offers apps related to a number of domains, software agents are able to deal with different situations as they have through other platforms, such as cases that require autonomous and pro-active software entities to make a decision. Furthermore, iOS provides many native resources (e.g., manners of

notifications and health kits) that can be used with software agents, becoming easier and more powerful for use in solutions that require these entities.

3.2 Health

There are some works that take similar approaches to ours, as is the case of the proposal from [32]. This effort presents the implementation of a distributed information infrastructure that uses the intelligent agents' paradigm for: i) automatically notifying the medical team responsible for patient's healthcare about abnormalities in their health status, ii) offering medical advice at a distance, and iii) enabling continuous monitoring of patient health status. This same work defends the adoption of ubiquitous [33] and mobile systems, that allow immediate analysis of physiological data of each patient in the form of personalized feedback of their condition in real-time through an alarms and remembers mechanism. In this solution, patients can be evaluated, diagnosed, and cared for in remote and ubiquitous mode. In the event of the rapid worsening of a patient's condition, the system automatically notifies the medical team through voice calls or SMS messages, supporting a first level medical response. Different from our proposal, it is a closed application, as opposed to an application generator, and it also does not implement software agents.

The approach presented by [34], in turn, focuses on the design and development of a distributed information system based on mobile agents to allow automatic and real-time fetal monitoring. The monitoring is conducted from devices such as PDAs, smartphones, laptops, or personal computers. The project was developed in JADE, just like [32]. Hence, both used a non-BDI multi-agent framework, different from our proposal, which offers a BDI architecture to the iOS platform.

In [35], mobile health applications are pointed out as solutions for: i) transposing personalized health service barriers, ii) providing opportune access to critical information of patients' health status, and iii) avoiding duplication of exams, delays and errors in patient treatment. From SADE4Health, we proposed a health module to help the manipulation of health data, in particular, vital sign data.

Finally, we found many health apps in the Apple Store, representing a subset focused on patient monitoring. Three examples are My-Vitals [36], Multi Vital Monitor [37], and Binah Team [38]. The My-Vitals app's intention is to be an easy way to manage health and medical information, in the form of a diary, and to link family health care information. Multi Vital Monitor

app is a Data Service for vital signs, as a monitor, which captures the vital signs of patients through VivaLNK's wearable sensors. The latter, Binah Team, captures vital signs through facial recognition. An interesting feature offered is the integration of measurements into the Apple HealthKit. Nevertheless, none of them use an intelligence system, and are focused on being a data repository to collect and display vital signs. Through SADE4Health, systems with intelligence that use software agents with HealthKit integrated can be developed.

3.3 Agents and iOS

Despite all of these outcomes, we found a gap in the relation between the iOS platform using the Swift language to multi-agent systems. Considering this context, we conducted a systematic survey - inspired by [39] - to understand how Swift developers and researchers combine their applications with Software Engineering essential concepts, particularly Multi-Agent Systems. Based on this review, we confirmed there was a research gap between Swift development and software agents. Hence, the iOS developers' community requires support and research into Software Agents.

The systematic mapping project analyzed papers published in journals or proceeding conferences after 2014 - the year Swift was introduced by Apple. We used the application Findpapers [40] to search scientific papers from five digital libraries, ACM [41], arXiv [42], IEEE [43], PubMed [44], and Scopus [45]. The search considered contributions that combined terms shown in Table 3.1 and were used in titles, abstracts, or keywords of papers.

Table 3.1: Terms used in research

Concepts keywords	iOS keywords	Mobile keywords
multi-agent system	iOS	mobile development
framework	swift	mobile application
agents	apple development platform	mobile agents
		mobile devices

This procedure detected 658 publications indexed in the five bibliographic databases mentioned, distributed by them as shown in Table 3.2.

Table 3.2: Results of the first search per digital library

Digital Library	**Number of Papers**	**Percentual (%)**
Scopus	526	79.9
IEEE	80	12.2
ACM	40	6.1
PubMed	7	1.1
arXiv	5	0.7
Total	**658**	**100**

However, some titles appear in more than one library, which required the removal of duplicate references in the research sample. By eliminating duplicates, we reached the number of 536 papers, distributed over time as shown in Figure 3.1.

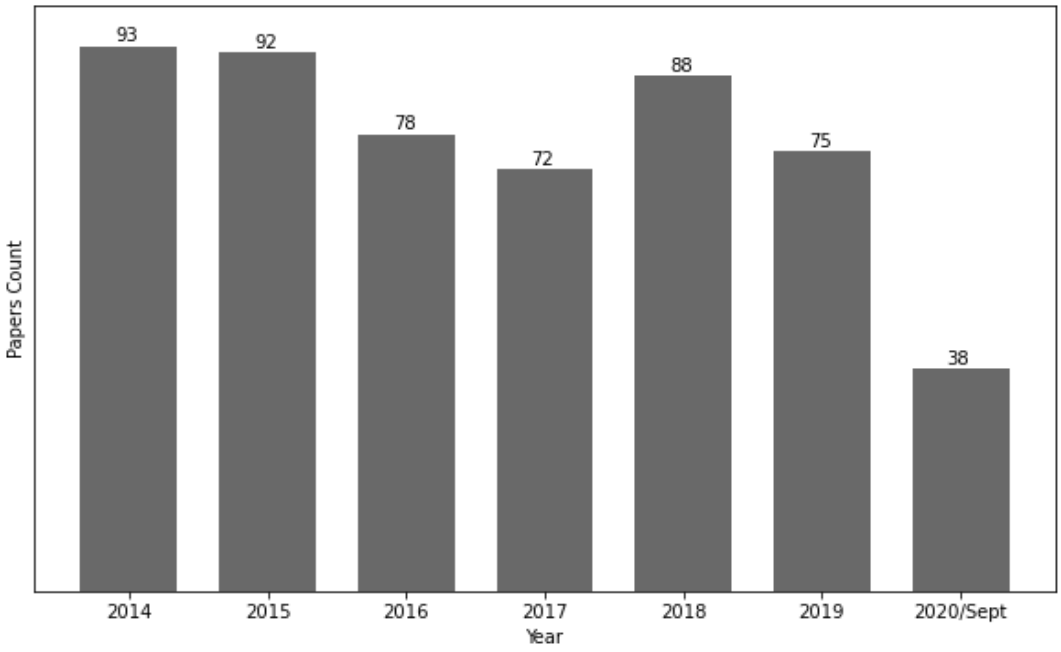

Figure 3.1: Number of papers over time

Next, we analyzed the titles and keywords of these papers, identifying those that in some way seemed to consider any of the inner research topics. We identified 79 papers that, by title and keywords, seemed to be related to Multi-Agent Systems, Frameworks, eHealth, or iOS development.

The selection process advanced through a new scheme, abstract analysis, applied to the 79 remaining papers. Their abstracts presented an overview that made it possible to understand what to expect from each complete paper. Therefore, it contributed to grouping them into predefined considered topics.

The purpose of this analysis was to eliminate papers that did not fit into any of the predefined topics and, consequently, would not help us with our mapping goal.

We chose the first set of topics based on research fields, while the second categorization intended to classify the papers by the contribution to be addressed. The results of these classifications and the predefined topics are shown in Figure 3.2.

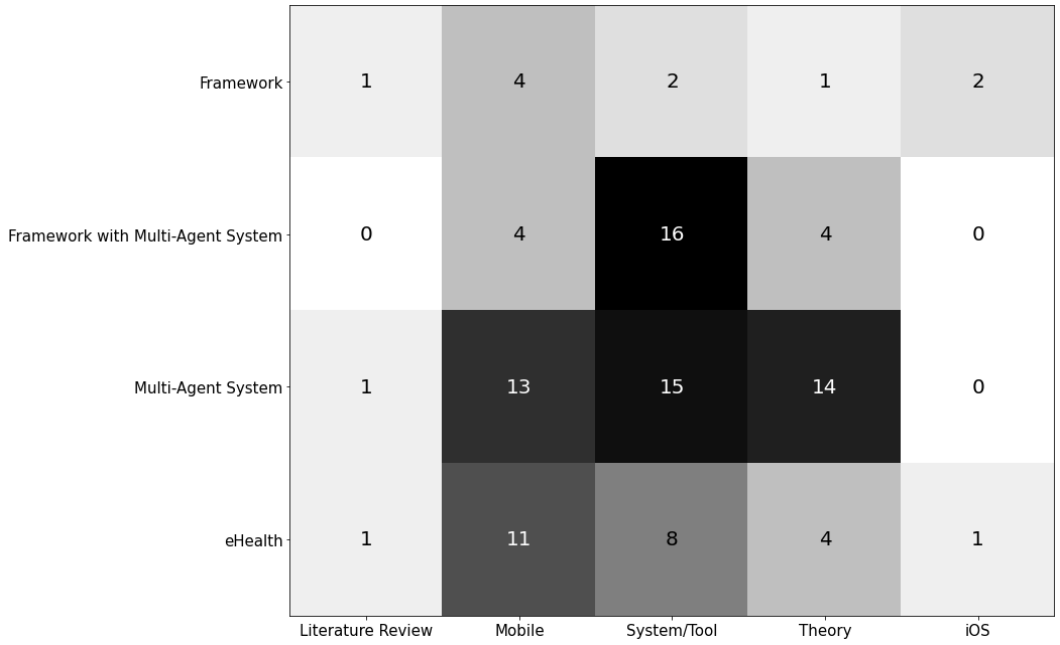

Figure 3.2: Number of papers combining field and contribution [13]

The most important categories for the systematic mapping were the two research field topics that evolve multi-agents systems and the contribution topic for the iOS development ecosystem. The combination of these two topics and the iOS research contribution is the only one that could address the proposed systematic mapping goal. From the heat map in Figure 3.2, we could understand how many papers from each research area topic were allocated into each research contribution topic. No papers were classified as joining our most important categories. It clearly showed the gap in developing solutions and research containing multi-agent systems in Swift. Thus, the SADE4Health framework represents our effort to bridge the gap regarding multi-agent systems development in the iOS platform from the Swift language.

4
The SADE4Health Framework Description

This chapter presents our FIPA compliant multi-agent framework, called SADE4Health, developed in Swift. This framework is structured in five interconnected modules, each with its classes, to ensure the interaction of agents with the environment through BDI reasoning, communication with other agents, and execution to achieve goals that solve issues related to the health domain.

The SADE4Health's architecture is structured in five modules - Core, BDIReasoning, Communication, Health, and Notification - as illustrated in Figure 4.1. Each module has a set of hotspots and frozen spots (see section 2.1) offered by the framework. Below, details of each module are presented.

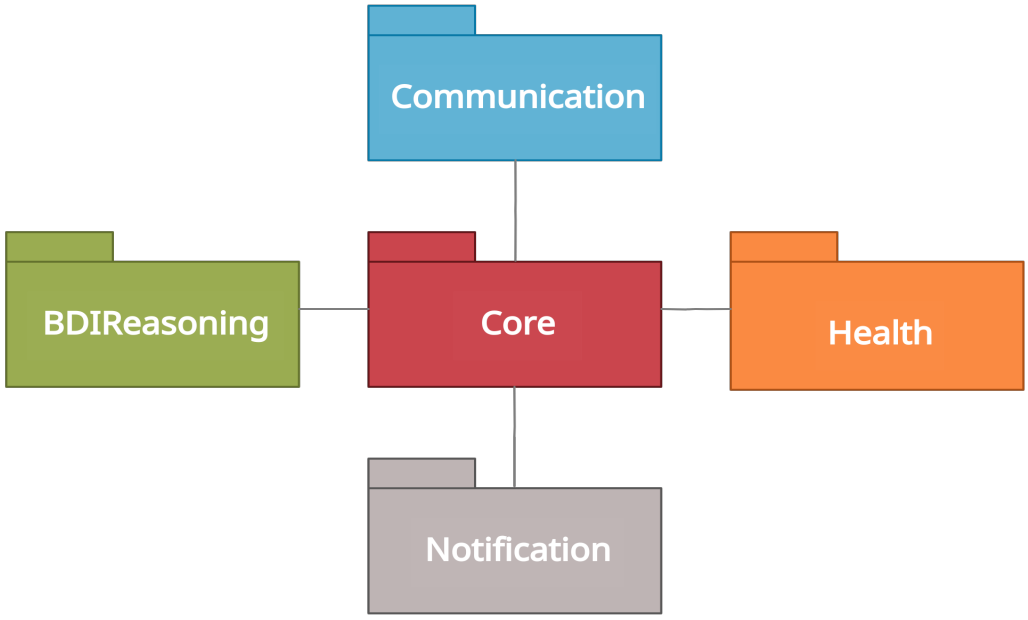

Figure 4.1: The SADE4Health's architecture with its five modules

4.1
Core

The Core module offers the main classes (Figure 4.2) that allow representing software agents, including important classes related to the BDI concept.

Below, details of the concepts represented in that module are explained.

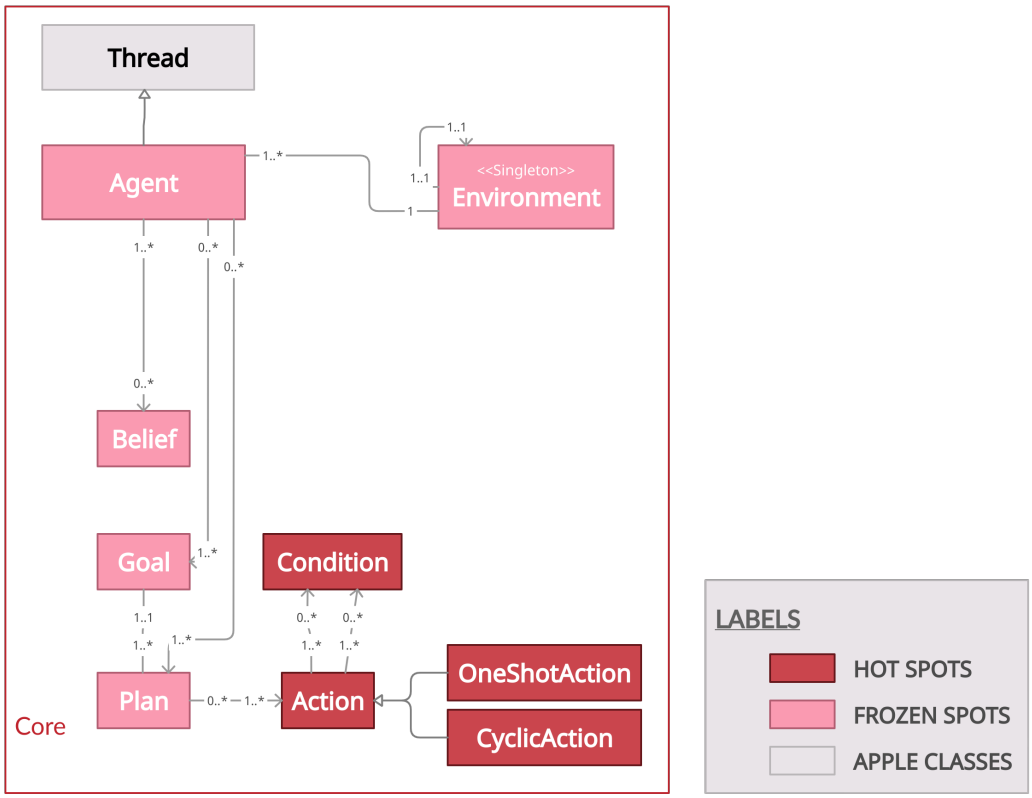

Figure 4.2: Classes from Core module

1. Software Agents

   We created the *Agent* class to represent the concept of a software agent, which has a set of properties, such as ID, goals, and beliefs. Our *Agent* is an extension of the *Thread* class [46], an Apple class from the Foundation framework that allows an application to run an execution thread without blocking the performance of the rest of the application. It is possible to instantiate an agent object from this frozen-spot class.

2. Environment containing agents

   Environment class represents the environment into which agents are inserted, beyond interacting, perceiving, and acting into it. The class contains every active object of the framework, so it was modeled as a Singleton design pattern to ensure a single point of access to all of them. As a Singleton, it creates itself in the application initialization; the objects only need to be registered into it. It is a frozen spot because it is ready to be used.

The *Environment* works as an Agent Management System (AMS), a FIPA requirement for managing a multi-agent system. Hence, it contains a white page service to allow an agent to find another agent. All system agents must be registered in order to become visible.

3. Agent's Beliefs

   The knowledge base of agents is represented by the *Belief* class. The objects instantiated through this class can carry any information the agent must know about the environment to execute their tasks. The *Belief* class is a frozen spot.

4. Agent's Desires

   The agent's goals in our framework represent the desires of the BDI reasoning, a motivational state into the BDI cycle. An object instantiated through the class *Goal* describes a goal that an agent should try to achieve, but, in this state, we do not yet know how they will achieve it. As it is a descriptive class, just linking agents with its goals, it is a frozen spot.

5. Agent's Intentions

   The deliberation state of a BDI reasoning cycle is represented by the intentions of an agent (see section 2.3), which happens when some agent attempts to achieve a goal through concrete actions. To represent that concept, the framework offers the following classes: *Plan* and *Action*. Each plan can have one or more actions, which are executed in sequence. Considering the *Plan* class is ready to be used, working as a set of actions to achieve a goal, it is a frozen spot. The *Action* class is a hot spot because there are functions that instances must implement considering their needs.

6. Actions' Behavior

   Actions can execute different tasks of agents. Currently, the framework offers two action types: *OneShotAction* and *CyclicAction*. The first type represents actions that execute just once. The latter represents actions in cycle, i.e., they can execute many times until they are stopped. Moreover, each action may be related to *Conditions*, pre- or post-condition, which must be verified before or after the execution of the action. All three classes are hot spots.

4.2 BDIReasoning

This module corresponds to the BDI reasoning modeling, containing strategies to allow agents' perceptions and interactions through BDI reasoning (Figure 4.3). It is considered that the Strategy design pattern was applied in all classes with the Strategy stereotype. Details of the concepts represented from that module are described in greater detail below. Every Strategy class is a hot spot because it works as a protocol to be extended and the Defaults classes are frozen spots because they contain default reasoning strategies.

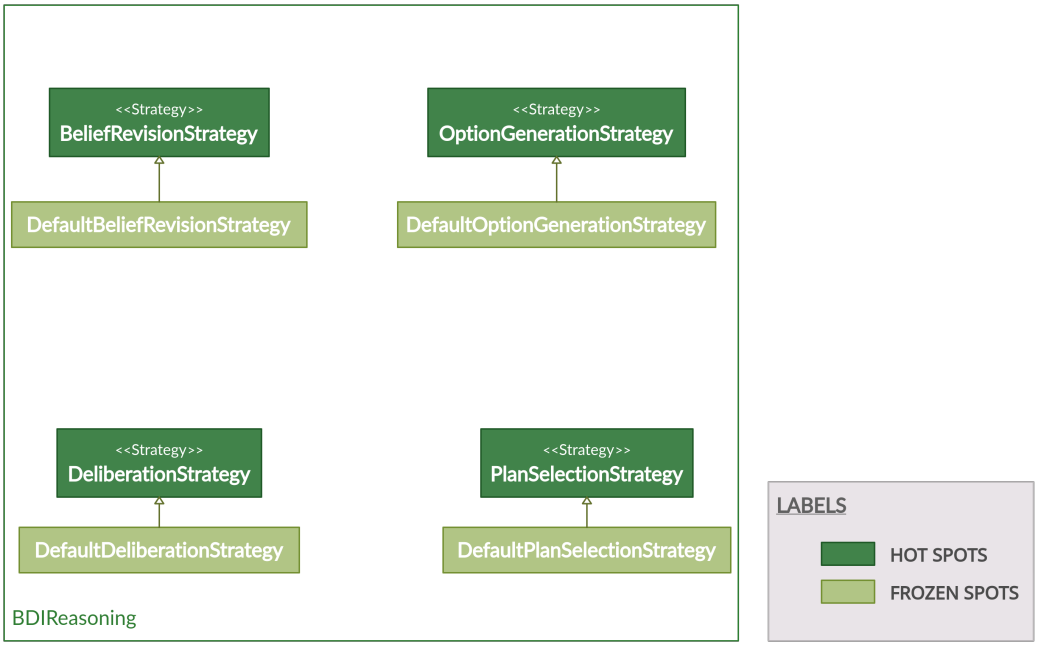

Figure 4.3: Classes from BDIReasoning module

1. Belief Revision

   From the *BeliefRevisionStrategy* class, we modeled the review of an agent's beliefs, aiming to check inconsistencies. It represents the first step of the BDI reasoning cycle (see section 2.3 for details). The framework already offers a default function of beliefs review from the *DefaultBeliefRevisionStrategy* class, which reviews the beliefs sequentially, walking through a set known by the correspondent agent.

2. Option Generation

   The next step of the BDI reasoning is to generate paths that can be taken by agents, through the availability of a set of goals. From the *OptionGenerationStrategy* class, the framework defines the need to create

a function that requests the review of agent's goals and the generation of new ones. Our default function that generates and reviews agent's options is offered from the *DefaultOptionGenerationStrategy* class, updating goals' statuses through the verification of its plans.

3. Filter

   An agent's set of goals passes through a filter that selects a subset to be tried and turned into intentions. It represents the deliberation step of the BDI reasoning (see section 2.3). Our default filter function, modeled from the *DefaultDeliberationStrategy* class, checks the goals with the status "waiting" and selects those the agent will be able to "execute" and, vice versa, which ones the agent must stop trying to execute and wait for it.

4. Plan Selection

   Here, the agent should execute plans to reach a specific goal, selected in the last step (Filter). The strategy to select these plans is executed in this step. Our default function, represented from the *DefaultPlanSelectionStrategy* class, chooses the plans to be executed one by one, sequentially, as they are organized into the set of goals' plans.

By default, the BDI cycle - as defined in section 2.3 - is executed by an agent in two moments: (i) when the agent starts; (ii) whenever the agent receives a message. Agent class contains a function, *runBDICycle*, which calls each function from the BDI cycle in sequence.

4.3 Communication

FIPA also established requirements patterns around agent communication. These requirements aim to ensure easy communication between different software agents. Therefore, this module takes charge of following these requirements through agent communication (Figure 4.4).

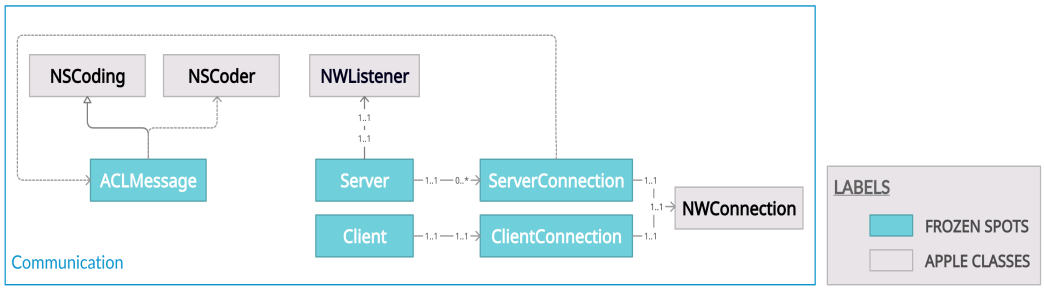

Figure 4.4: Classes from Communication module

Aiming to allow agents to interact with others, each agent can send and receive messages. Each agent is allocated in a host and a port as a *Server* to receive messages. The *Server* behaves as a listener, which is represented by *NWListener* class, an Apple class offered by *Network* framework [51] to create an object to listen for incoming network connections and that is ready to receive messages at any time. To send messages, an agent must create a *Client* object considering the recipient agent server's address (host and port).

Both *Server* and *Client* classes contain auxiliary classes, *ServerConnection* and *ClientConnection*, with functions to establish the connection in the receiver or sender, respectively. These connection classes contain objects from *NWConnection*, which is another class offered by the Apple Network framework and that is able to receive and send messages from or to an address.

ACLMessage class represents messages sent from one agent to another. Each message follows the Agent Communication Language (ACL), a set of one or more message parameters, which defines which information each message must contain. Among these parameters are, for example, the type of communicative act, sender, receiver and message content.

All the communication module classes are frozen spots because they are ready for use by the instance, considering FIPA requirements for good agent communication.

4.4 Health

This module makes it possible to represent important concepts related to remote monitoring of patients. Below, there are details of each concept represented. It works as a remote database of a patient's vital signs. It can store many signs, such as patient arterial pressure, ECG, heart rate, temperature, and others. It occurs through the relation to the *Patient* class, a hot spot which stores the monitored patient's key information, the *VitalSignType* class, a frozen spot to register which vital signs will be monitored, and the *PatientVitalSign* class, a frozen spot which stores the value of the vital sign. Related to patients' vital signs is the *AnomaliesAction* frozen spot, which extends the *OneShotAction* class offered in the Core module (see section 4.1) and detects anomalies considering typical values of each vital sign.

Moreover, that module supports the representation of healthcare workers (*HealthCareWorker* class) and hospitals (*Hospital* class), where healthcare staff work and where patients can be treated. *HealthCareWorker* class is a hotspot, but the framework also provides frozen spots representing two types of healthcare workers: doctors (*Doctor* class) and nurses (*Nurse* class).

Hospital class is a frozen spot that is related to the *HospitalSector* class, which indicates the sector(s) where patients are interned. Another important class is *DAONewData*, which applies DAO design pattern, responsible for communicating with databases and collecting new data. Since each instance must collect new data in different ways, *DAONewData* is a hotspot.

In addition to the classes described above, we integrated the framework with HealthKit [18] from two main classes, *HealthKitSetUpAssistant* and *HealthKitReaderWriter*. In the *HealthKitSetUpAssistant* class, the framework offers functions to request authorization to read and write health data. However, two steps are necessary before calling these functions: i) adding HealthKit as a capability in a development project; ii) describing why you need access to health data keys in the "Privacy – Health Share Usage Description" and "Privacy – Health Update Usage Description" of the Info.plist file.

From the *HealthKitReaderWriter* class, there are auxiliary functions to query health data (e.g., characteristic and quantity sample data) and save new health data in HealthKit [47] (e.g., quantity, category, and correlation samples). Every application generated through our framework is inserted in the collaborative network around the Apple Health app with these classes. As instances may implement new functions, both classes are hot spots. All classes in this module are represented in the figure below, including the dependencies with HealthKit classes (Figure 14).

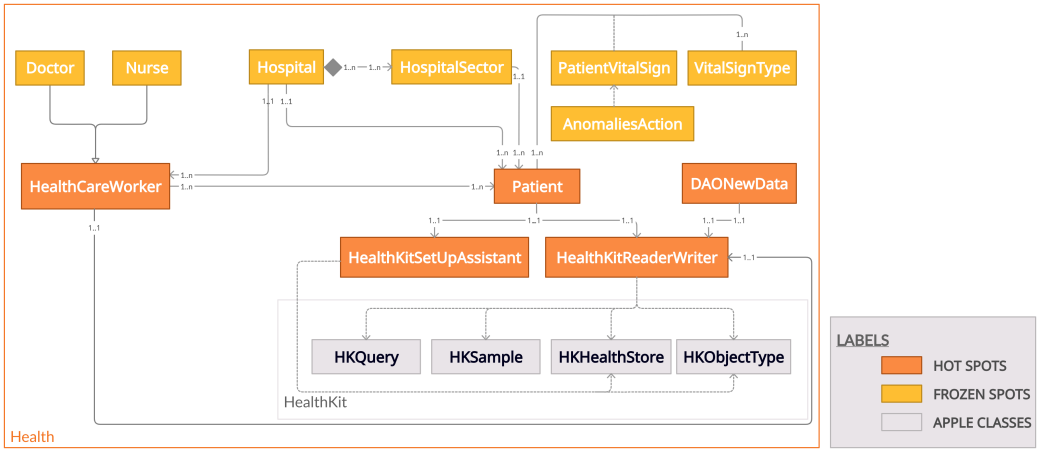

Figure 4.5: Classes from Health module

4.5 Notification

The Notification module allows creating different types of notifications, such as alerts to healthcare workers. Figure 15 illustrates two types of notifica-

tion offered by the framework, represented as two frozen spots: app notification (*AppNotification* class) and email (*EmailNotification* class).

If some instance of the framework desires to implement new ways of notification, it can just extend the Notification class, a hotspot class, and create a new correspondent class. *AppNotification* class depends on four main classes of the UserNotifications [48], a native Apple framework, which already has made available a way to request the user's permission to send notifications.

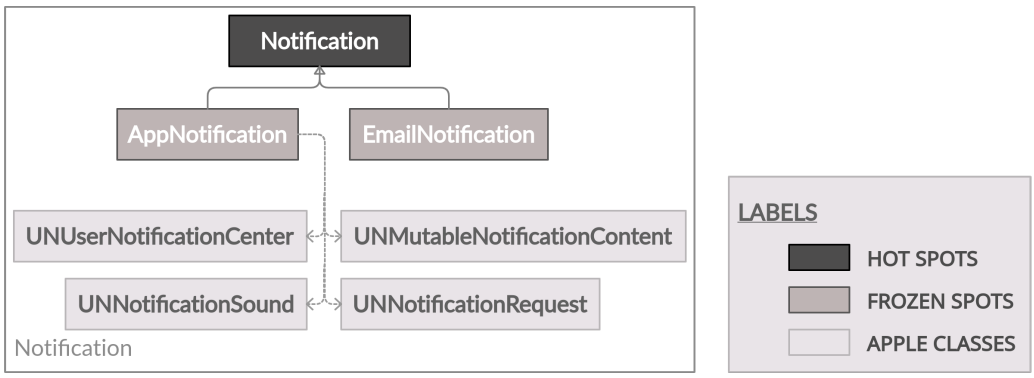

Figure 4.6: Classes from Notification module

5
Instantiating and Testing a SADE4Health Agent

This chapter describes step by step how to create a software agent through the SADE4Health framework. Furthermore, it presents sections with a simple modeled example of a use scenario and some SADE4Health unit tests. Finally, the last section implements a use scenario for remote monitoring using the SADE4Health.

The main steps to instantiate and start an agent are listed below:

1. Creating Action classes inheriting from *Action*, *OneShotAction*, or *CyclicAction*. The central point here is implementing the *start* function, which begins the execution of the action. On *OneShotAction* and *CyclicAction*, the *start* function is already implemented, so the function *runAction* should be implemented if inheriting from one of these classes. If one needs an action that uses health data, the action can be related to the instances of the Health module;

   (a) If some action desires to access health data in HealthKit, it should use the *HealthKitReaderWriter* class that provides the functions *getAgeSexAndBloodType* and *getSample* to read data from HealthKit and the functions *saveQuantitySample*, *saveCategorySample*, and *saveCorrelation* to write data to HealthKit.

   (b) If one needs to detect anomalies in health data, it should use the *AnomaliesAction* class.

   (c) If some agent needs to send messages to another agent, the *sendMessage* should be used informing the address (host and port) parameters of the recipient agent and an *ACLMessage* object.

2. Creating one or more *Plans* instances providing the priority of execution and a list of actions that should be concluded to reach each plan. Please note that these actions are those created in the previous step;

3. Creating Goal instances providing its name, a list of plan(s) associated (created in step 3), and status (by default, it starts as "waiting"). Thus, a set of actions compose a plan, and a set of plans compose a goal.

4. Creating the *Beliefs* instances that the agent must know to execute its tasks;

5. Creating the strategies of the BDI cycle inheriting from *Strategy* classes or instantiating strategies objects from the defaults strategies implementations;

6. Create the *Agent* instance providing:

   (a) its name;

   (b) goals it will help to achieve;

   (c) plans containing actions;

   (d) the environmental beliefs it must know to execute its tasks;

   (e) strategies to execute the BDI cycle;

   (f) a boolean to determine if the BDI cycle should be run for the first time on the startup;

   (g) and its address (host and port) to receive and send messages.

7. Starting the agent from the *start* function of the *Agent* instance. The agent is started as a *Thread* in this function, registered in the environment, and the BDI cycle is executed if desired.

5.1 Modeling a Classic Use Scenario

This section models a first application example of using SADE4Health. It is a generic example, not from the field of Health, and the same applied to the participants' tasks of the empirical evaluation reported in chapter 6.

There must be two agents in an environment in this classic example, Alarmed and Fireman. The Alarmed agent verifies if the environment is on fire, and if it perceives a fire, it notifies the Fireman agent. On the other hand, the Fireman agent, when notified, must act to extinguish the fire. Then, the agents should be created, and their goals, plans, actions, and BDI strategies. Described below are the creation pseudocodes of this situation.

Supposing the Alarmed agent keeps verifying the environment to find a fire, we should create a cyclic action for these verifications - called *CheckFire*, for example -, given a time interval. Then, this action extends the *CyclicAction* class and we should override function *runAction*, represented in pseudocode 1.

Algorithm 1: Creating a cyclic action to perceive a fire

```
Data: Belief isInFire
Function runAction():
    if Fire then
        isInFire ← true;
        send message to Fireman;
```

On the other hand, the Fireman waits until an alarm arrives to extinguish a fire. Thus, we should create an one-shot action - called *ExtinguishFire*, for example - extending the OneShotAction class and override function *runAction*, illustrated in pseudocode 2.

Algorithm 2: Creating an one-shot action to extinguish a fire

```
Data: Belief isInFire
Function runAction():
    if isInFire then
        throw water on the fire;
        isInFire ← false;
```

Now, we must instantiate plans and goals, as in pseudocode 3, to associate them with the agents.

Algorithm 3: Creating plans and goals

```
alarmedAction ← CheckFire(interval = 10);
alarmedPlan ← Plan(actions = [alarmedAction]);
alarmedGoal ← Goal(plans = [alarmedPlan]);
firemanAction ← ExtinguishFire();
firemanPlan ← Plan(actions = [firemanAction]);
firemanGoal ← Goal(plans = [firemanPlan]);
```

In the last step before finnaly instantiating the agents, we should create the objects for BDI cycle strategies - we could simply instantiate strategies from default classes. As strategies recognize the agents they are related to, each agent must contains its own stategies objects - pseudocode 4.

Algorithm 4: Creating plans and goals

```
1 alarmedBeliefRevision ← DefaultBeliefRevisionStrategy();
2 alarmedOptGeneration ← DefaultOptionGenerationStrategy();
3 alarmedDeliberation ← DefaultDeliberationStrategy();
4 alarmedPlanSelection ← DefaultPlanSelectionStrategy();
5 firemanBeliefRevision ← DefaultBeliefRevisionStrategy();
6 firemanOptGeneration ← DefaultOptionGenerationStrategy();
7 firemanDeliberation ← DefaultDeliberationStrategy();
8 firemanPlanSelection ← DefaultPlanSelectionStrategy();
```

Finally, we should instantiate the agents with the parameters we created in the previous steps. As the Alarmed agent must verify the fire from the beginning, we could set runBDICycleInStart as *true*. However, Fireman only extinguishes a fire when notified by Alarmed, so we could set *runBDICycleInStart* as *false* to be executed only when a message is received. Pseudocodes 5 and 6 show the agents inicialization.

Algorithm 5: Creating and starting Alarmed agent

```
1  alarmed ← Agent(
2                  agentName : "Alarmed",
3                  goals : [alarmedGoal],
4                  plans : [alarmedPlan],
5                  beliefs : [isInFire],
6                  beliefRevision : alarmaedBeliefRevision,
7                  optionGeneration : alarmedOptGeneration,
8                  filter : alarmedDeliberation,
9                  planSelection : alarmedPlanSelection,
10                 runBDICycleInStart : true,
11                 port : 8888)
12
13 alarmed.start()
```

Algorithm 6: Creating and starting Fireman agent

```
1  fireman ← Agent(
2                  agentName : "Fireman",
3                  goals : [firemanGoal],
4                  plans : [firemanPlan],
5                  beliefs : [isInFire],
6                  beliefRevision : firemanBeliefRevision,
7                  optionGeneration : firemanOptGeneration,
8                  filter : firemanDeliberation,
9                  planSelection : firemanPlanSelection,
10                 runBDICycleInStart : false,
11                 port : 9999)
12
13 fireman.start()
```


5.2 Unit Tests

This section describes some unit tests created on SADE4Health to evaluate its features. As a unit test, each one pretends to validate that a unit part of the framework code performs as expected. The tests were executed by mocking objects to simulate the agents modeled in the previous section. To realize them, we considered Apple's native framework *XCTest* [49] and the XCode environment that allows test execution by importing the entire SADE4Health as a *@testable* inside the test class - which extends from *XCTestCase*.

The topics below describe four of the realized unit tests in detail:

1. Init test

   In the first test, we mocked the Fireman agent with some properties in this test through the *Agent* initializer. After its initialization, we verified - through function *XCTAssertEqual* - if each property assigned to the Fireman agent object was equal to the expected one.

2. Registration in the environment

   In this second test, we mocked the Fireman agent with some properties again through the *Agent* initializer. In the *Agent* initializer, there is a self-registration in the *Environment*. So, we tested if the Fireman agent was already registered after its initialization. This test is essential

because of the white pages service, which allows finding any agent in the environment through its name. For this test, we verified - through function *XCTAssertNotNil* - if *whitePages* function would not return nil when we search for "Fireman" agent.

3. Cyclic action

   To test if a cyclic action is being correctly executed, we mocked the Alarmed agent, who verifies if there is a fire in the environment. Then, we should associate with the Alarmed agent a goal and a plan to be reached. Inside the plan, a cyclic action must be executed at time intervals. After started, the Alarmed agent must execute its BDI cycle for the first time, and we expect it will start to verify the fires. So, we tested - through function *XCTAssertEqual* - if the timer of the cyclic action is valid - then it is running - and if the Goal status and Plan status keep as "executing" while the cyclic action is running.

4. One-shot action

   To verify if a one-shot action was correctly executed, we mocked the Fireman agent by associating it with a goal, plan, and action to extinguish a fire. Then, we let Fireman agent knows a belief called "isInFire", assigned to *true*, and starts the agent considering the variable *runBDICycleInStart* as *true*. It is expected that the agent will execute its BDI cycle for the first time when started and will extinguish the fire, turning "isInFire" to *false*. So, we tested if, after starting the agent, the belief "isInFire" is false, meaning that the fire was extinguished. For that, we used the function *XCTAssertEqual* again.

Finally, Figure 5.1 shows that these four tests succeeded.

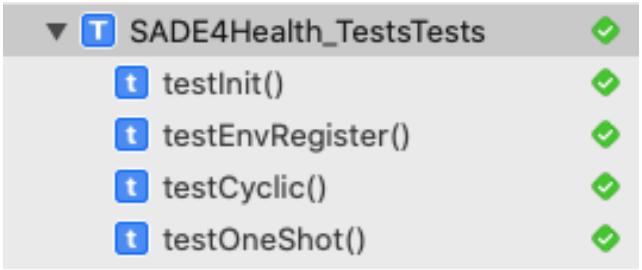

Figure 5.1: Tests results.

5.3 A Use Scenario for Remote Monitoring

This section presents a more complex and implemented - not only modeled - use scenario for remote monitoring using the SADE4Health framework presented in chapter 4. That instance, which is an iOS application (app), was developed using Swift language (version 5) and XCode (version 12.4).

5.3.1 Overview and Architecture

In this use scenario, we considered the COVID-19 disease to guide our choices and considerations, including the most important vital signs and the critical alerts to be used. The idea behind the application is to allow healthcare workers to monitor their patients wherever they are. It maximizes monitoring the patients' condition, allowing them to act based on the disease's evolution, and minimizes the contact between them and the patients, considering the ailment as extremely contagious. Thus, the proposed solution benefits both healthcare professionals and patients.

The application should receive data from patients' vital sign monitoring, display it to the final user, detect anomalies, and notify the medical team in the case of anomalies found. Software agents perform all of these tasks through BDI reasoning, as each is an extension of the Agent Class from SADE4Health. Finally, we structured it applying an MVC design pattern [50]. Thus, the following modules were created: Model, View, and Controller.

The Model module contains classes that inherit from the framework classes and two other classes to support the app's particular operations, *UsefulData* and *COVID19Monitor*. Figure 5.2 shows the instance's class diagram and how it is related to the classes offered by the framework.

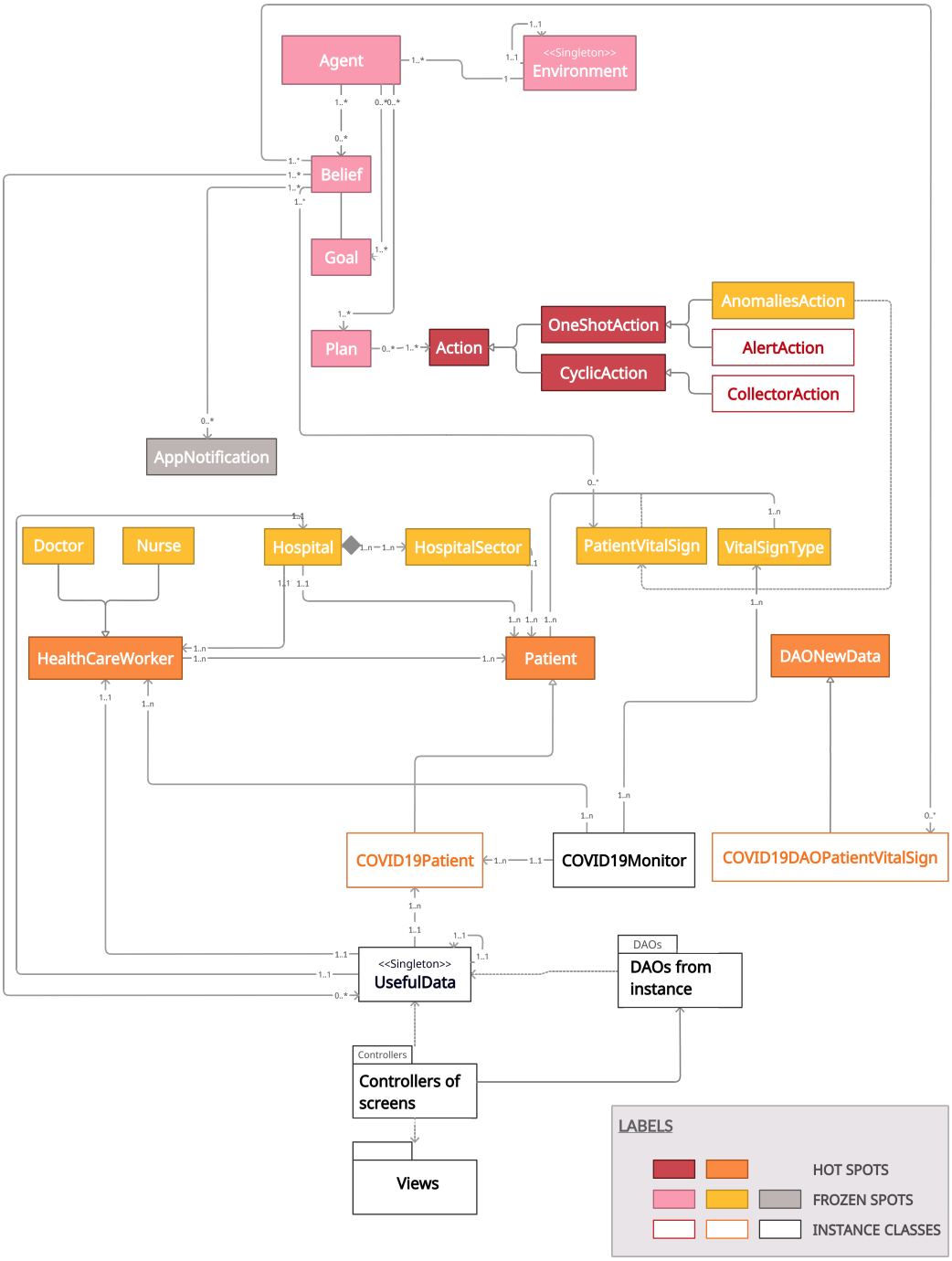

Figure 5.2: The use scenario class diagram

In our instance, we modeled three software agents, *CollectorAgent*, *AnomaliesAgent*, and *AlertAgent*, each with specific tasks configured through their beliefs about the environment and the goals they should try to achieve through plans and, consequently, concrete actions.

The *CollectorAgent*'s goal is to collect new vital signs of the patients monitored by the healthcare worker logged into the app and in-

clude it in the patients' monitor. Thus, this agent has, as beliefs, the *COVID19DAOPatientVitalSign* class to request new data and the *PatientVitalSign* class to register the data collected. However, it is carried out through plans and actions. Thus, we modeled the *CollectorAction* assuming a cyclic behavior to collect new data continuously. After each cycle, the *AnomaliesAgent* is contacted to detect anomalies in the newly collected data.

AnomaliesAgent, in turn, knows *PatientVitalSign* and the desired values interval for each vital sign to compare the vital sign collected with typical values to detect anomalies. For these actions, we used *AnomaliesAction*, which assumes a one-shot action because there is no predefined time to be executed. If any anomaly is detected, *AlertAgent* is contacted to alert the healthcare worker about it.

Finally, *AlertAgent* tasks *AppNotification* to send alerts from the app to the medical team, alerting them to anomalies. In order to execute that alert, the solution used the *AlertAction* class, which executes only on demand. All these possible flows are represented from the activity diagram illustrated in Figure 5.3.

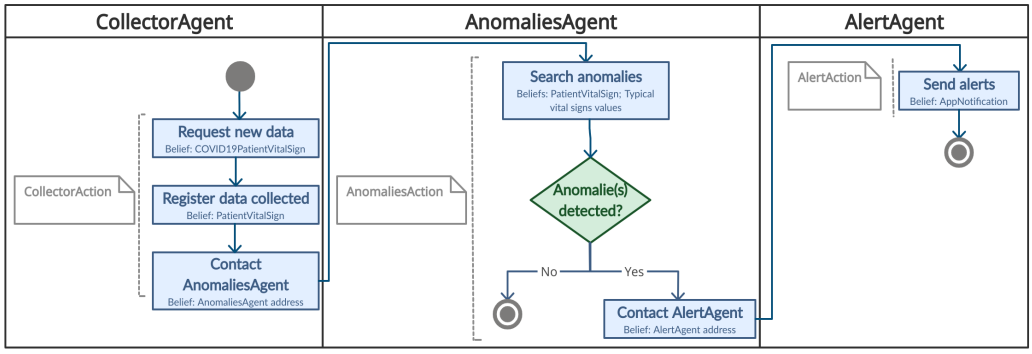

Figure 5.3: Instance activity diagram

In addition to the classes already mentioned, we modeled other particularities of our instance:

1. *COVID19Patient* class inherits from the Patient class offered by the framework, adding a new control property when a patient is using a respirator, an essential aspect of the COVID-19 disease;

2. *COVID19Monitor* controls the vital sign that will be displayed on the app's monitor screen;

3. *UsefulData* is a Singleton class that provides a single point of access to useful data for the app;

4. DAOs package contains useful DAOs to consult other information, different from vital signs.

In the *View* layer, there are classes that define per screen of the app which visual components (e.g., buttons, text fields, etc.) will be presented to the users. Besides, each screen of the app has a Controller class, which is responsible for making configurations that help each screen to be ready for use from the usage of classes of the View and Model layers when necessary.

Figure 5.4 illustrates three important app screens. Figure 5.4(a) lists patients being monitored by a healthcare worker. Figure 5.4(b) shows the screen that simulates a vital sign monitor containing the vital signs of a selected patient. And, Figure 5.4(c) shows examples of alert notifications of some detected anomalies.

5.4(a): Patients list 5.4(b): Vital signs monitor 5.4(c): Notifications

Figure 5.4: Instance main screens

5.3.2 Step-by-step Framework Instantiating

This section describes step-by-step how the agents of the app were created using the SADE4Health framework.

1. Creating *CollectorAction* class extending from *CyclicAction* class and *AlertAction* class extending from *OneShotAction* class - the *AnomaliesAction* is already being provided by the framework;

2. Creating plan objects to collect vital signs, detect anomalies and send alerts (*CollectPlan*, *AnomaliesPlan* and *AlertPlan*);

   (a) *CollectPlan* has a *CollectorAction*.

   (b) *AnomaliesPlan* has an *AnomaliesAction*.

   (c) *AlertPlan* has an *AlertAction*.

3. Creating goal objects - *CollectGoal*, *AnomaliesGoal* and *AlertGoal*;

4. Creating the agents beliefs - *CollectorAgent* must know *AnomaliesAgent*, and *AnomaliesAgent* must know *AlertAgent*;

5. Creating the BDI cycle strategies;

6. Creating the agent objects from the *Agent* class - *CollectorAgent*, *AnomaliesAgent* and *AlertAgent*;

   (a) *CollectorAgent* executes the *CollectPlan*.

   (b) *AnomaliesAgent* executes the *AnomaliesPlan*.

   (c) *AlertAgent* executes the *AlertPlan*.

7. Starting each agent. Into the *start* function, the agent is already registered in the Environment.

6
Empirical Evaluation

This chapter presents the design, execution, and analysis of an empirical evaluation of the use of SADE4Health for Swift developers. It was divided in two moments, one of them with tasks to perform freely and the other one with the same tasks using the SADE4Health framework. As not all the participants were previously aware of the software agent's paradigm, training was provided in order to give them a conceptual base around it. Moreover, before the participants used the framework, we presented its architecture and brief documentation. This chapter is organized in sections inspired by the reporting guidelines for experiments in software engineering presented by Jedlitschka and Pfahl [52].

6.1
Goal, Hypotheses and Variables

This experiment aims to evaluate the benefits of using the SADE4Health framework to create and learn how to use software agents in iOS application development. Thus, two hypotheses were considered:

1. SADE4Health reduces the effort in creating software agents in iOS apps compared to a scenario where the framework is not used.

2. SADE4Health helps to learn to apply the multi-agent systems paradigm in iOS application development compared to a scenario where the framework is not used.

The first hypothesis was elaborated regarding the utility of a framework, by definition, which allows the design and code reuse, as mentioned in section 2.1. Moreover, the second is considering the gap in developing solutions and research containing multi-agents in Swift, as shown in section 3.3. In order to analyze the hypotheses presented above, the independent variables considered were as follows:

– participants involved.
– training applied.
– activities requested from participants.

Dependent variables were as follows:

- time to carry out activities without the framework.
- time to carry out activities using the framework.
- difficulties encountered.
- facilities identified.

6.2 Design

This empirical evaluation was divided into seven stages. The first stage carried out the training of the participants around the software agent's paradigm. The training was necessary because not all the participants previously knew software agents in practice, and this knowledge was essential to evaluate the second hypothesis. In the second stage, participants should individually perform tasks freely, not using the framework. Then, a questionnaire about these tasks was applied. In the fourth stage, we introduced them to SADE4Health architecture and documentation. Finally, in order to not influence the evaluation results, the participants performed the same tasks and answered the same questionnaire, but this time using the framework to perform the tasks. The last stage was a final interview. Figure 6.1 shows a diagram of all the stages. An important observation is that for half of the participants, the order of the stages was inverted - they performed the tasks using the framework before the tasks that did not use it.

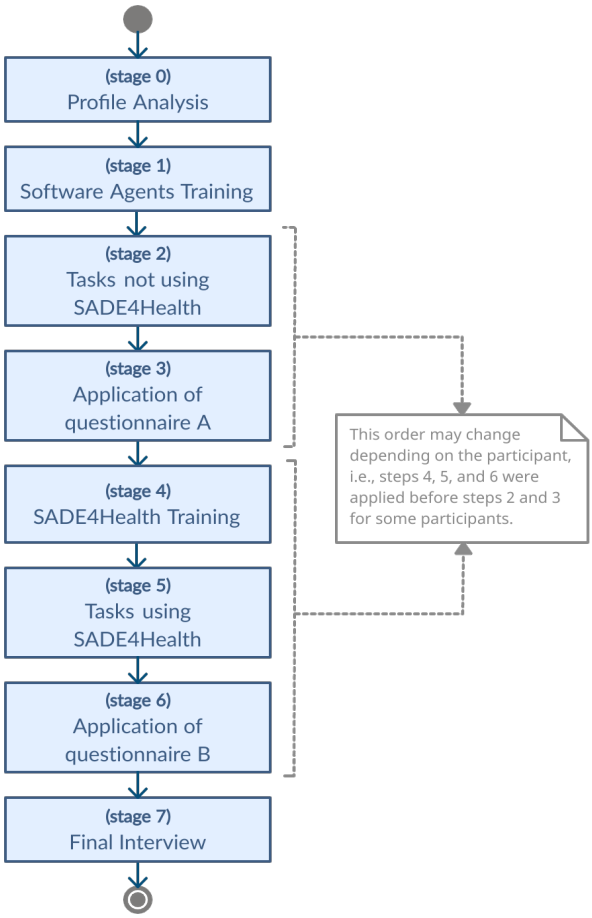

Figure 6.1: Empirical evaluation activity diagram.

6.3 Subjects

The participants were recruited by directly approaching to participate voluntarily, so there was no motivation other than knowledge. Only one inclusion criteria was considered: they needed to have satisfactory knowledge in development with Swift programming language. In this way, the limitation on the programming language would not be an extra obstacle to the elaboration of the requested tasks. To reach Swift developers, all the recruited participants had already been part of an Apple Development Academy.

We wanted to have around ten participants, but we were able to reach only six people. Despite that, we considered that the result of this empirical evaluation was very significant for the improvement and evolution of this research.

6.4 Objects and Instrumentation

To apply this empirical evaluation, we used online resources through virtual calls. The two training sessions were held through online presentations, and the questionnaires were answered through Google Forms, as well as its data collection.

There were two training sessions. In the first one regarding software agents, we conceptually described them, including their main properties, their relation with the environment they are inserted in, the BDI reasoning applied to software agents, and examples of existing frameworks in different programming languages to create them. The second training was about SADE4Health. We explained the motivation to develop it, showed its architecture, and gave step- by-step documentation to instantiate a software agent using the framework.

At the beginning of the evaluation, the participants should start the form by answering some personal questions for a profile analysis about:

- how many years they have developed using Swift;
- how they rate their experience programming in Swift (on a scale from one to five);
- for what purpose they use Swift;
- if they know the software agent's paradigm; and
- if they know the BDI reasoning applied to software agents.
- what is their degree of school education;
- what is their research field.

The tasks they should execute twice - stages two and five of Figure 6.1 -, using and not using the dependencies of SADE4Health, consisted in creating two software agents that communicate with each other. The first one would be the "Alarmed" that perceives a fire. If the "Alarmed" agent finds fire, it sends a message to the second agent, the "Fireman", that acts to extinguish the fire. For this purpose, we sent the participants two XCode projects containing a file called "Fire.swift" with a function that returns if there is a fire taking place in the environment. However, the SADE4Health modules were present in one of these projects, allowing them to use its architecture to perform the tasks using the framework.

For each group of tasks, using and not using SADE4Health, the participants answered a questionnaire containing the following aspects:

- how much time they took to realize each activity, noting the start and end time, and if it was totally executed, partially executed, or not executed;
- how was the execution of the tasks;
- how was the complexity of the activities (on a scale from one, not complex, to five, very complex), justifying their answer;
- if it was helpful to learn about software agents (on a scale from one, very difficult to learn, to five, very easy to learn), justifying their answer.

6.5 Data Collection

At each stage of the empirical evaluation, data collection was executed manually by the participants. The form from Google Forms contained eight sections, starting with the personal questions and following with one section per stage as shown in Figure 6.1. After participants submitted the form, the data were automatically registered into a spreadsheet. Stages number one, four, and seven were the only moments when they stopped following the form to participate in the online video call training or interview. To perform the tasks, the participants opened the XCode but filled the form in parallel with the completion time of each assignment.

For each group of participants, we spent from ten to 15 minutes in each training, with a maximum of 60 minutes in each group of tasks - stages two and five -, five minutes in each questionnaire - stages three and six -, and from five to ten minutes in the final interview. Data collection from the final interview was up to the interviewer, who took notes during the conversations.

6.6 Analysis Procedure and Evaluation of Validity

As we conducted the empirical evaluation with few participants, data analysis could be individually performed, collecting crucial information for the improvement and evolution of the project.

At the end of all stages, the participants sent us their XCode projects after their changes, so we could check if they reached the goal of each assignment in the tasks for the two cases, using and not using SADE4Health. Furthermore, the software agent training, the SADE4Health training, the questionnaires, and the final interview helped us guarantee this empirical evaluation's validity.

6.7 Execution and Analysis

To execute the empirical experiment, four meetings with the participants were carried out. In half of them, we applied it in the order shown in Figure 6.1, and for the other half, we changed this order as also observed in Figure 6.1. Therefore, the analysis of the results are splitted into these two groups. Table 6.1 expresses the participants' profile based on answers in stage zero[1], discriminating the order in which they executed the experiment.

Table 6.1: Description of the participants' profile

	Ordinary order	**Reverse order**
Number of participants	3	3
Swift experience in years	1 to 3 (1) 3 to 5 (1) >5 (1)	1 to 3 (1) 3 to 5 (2)
Swift experience rate (1 to 5)	3 (1) 4 (2)	3 (2) 5 (1)
Swift usage purpose	solutions in companies (2) research (1) personal projects (3)	solutions in companies (3) personal projects (3)
Knows MAS	yes (1) no (2)	yes (1) no (2)
Knows BDI	yes (1) no (2)	no (3)
Degree of school education	incomplete master's degree (2) complete master's degree (1)	incomplete master's degree (2) incomplete graduation (1)
Research Field	Software Engineering (1) Machine Learning (1) Optimization (1)	Software Engineering (2) Machine Learning (1)

6.7.1 Ordinary Order

Considering the ordinary order of the empirical experiment (e.g., following stages two, three, four, five, and six), we could observe that the participants considered that they executed the tasks in stage three well, even those who did not know the software agent' paradigm previously. Participants took 36 minutes on average to complete these activities. Indeed, when asked about the execution of the tasks in stage two, they answered that there were conceptual difficulties, especially for those who did not know software agents. However, they affirmed that the training in stage one was helpful and allowed them to execute the tasks. Regarding the answers about the complexity of the tasks, they

[1]Numbers in parentheses represent the number of participants who gave the described answer to the question.

considered that it had a medium complexity (2.6 on average). Nevertheless, they punctuated that the complexity was mostly related to the abstraction of the concepts. They also stated that even if the task itself was simple, the problem had enough structure to encourage the creation of more complex software agent's systems.

After the SADE4Health training in stage four, the participants started stage five. Now, they should execute the same tasks from stage two, but this time considering the whole structure from SADE4Health to create the Agents, their reasoning, and put one in contact with the other. This time, they took 65 minutes on average to conclude the tasks - and one of the participants did not conclude it. In general, participants found it more complex (3.6 on average) than stage two, and, according to them, it was because they spent time understanding how the framework worked, going back to the documentation many times. Nevertheless, they recognized that the framework seems to be a good solution for a more complex multi-agent system. They pointed out some improvements to the documentation, adding a concrete example of agents' instantiation, not only showing it at a high level.

Regarding the use of SADE4Health with the aim of learning about software agents, participants pointed out it was possible to deeply exercise agent systems' concepts when compared to developing a simple "freehand" agent system. Indeed, when we examined their XCode projects from stage two, we could notice that they developed a much simpler solution. They were not able to reach some crucial properties from a multi-agent system in many cases.

In the final interview, participants pointed out many improvements in objects instantiation, documentation, and the relation between the framework classes. On the other hand, they seemed very excited about an open source framework in Swift to work with software agents. Listed below are some phrases mentioned by the participants from the first group[2]:

- *"There are many parameters that can be optional"*;
- *"The instantiation flow could be more fluid. Better documentation could help"*;
- *"The only thing that was not 100% clear to me was some interactions between classes or how I could access a particular property from a specific place. Still, the examples in the documentation helped a lot"*;
- *"The experiment was carried out very well"*;

[2]As the interviews were conducted in Portuguese, we made a faithful translation of the participants' speeches.

- *"I believe that the initial experiment is critical to understanding the basics of agent systems' concepts. So, when using the framework, the tester already knows what to do, and the question becomes 'how'"*;
- *"On tasks in stage two, I had done the bare minimum. With the framework, it became more complete in the way an agent should be".*

6.7.2 Reverse Order

The empirical evaluation was applied in the reverse order to the second group (e.g., following stages four, five, six, two, and three). Considering that this group performed the tasks using the SADE4Health dependencies first, we observed different and interesting results. Beginning with stages four and five, we conducted the SADE4Health training and then asked them to perform the tasks using it. Participants took 52 minutes on average to execute stage five. Yet, not all the participants could conclude the activities. When asked about the execution of these tasks, they answered that maybe they could conclude it if they had additional time. However, they were a bit confused about classes and their relations. Thus, they punctuate that if some things were more straightforward and with clear steps, they could use them better, even considering the fact that they were not so familiar with software agents' concepts. Moreover, they classified this stage with a complexity of 3.3 on average, justifying that it would be easier if they were more aware of the framework. Despite the difficulties, they recognized that the software agents' concepts were evident after the training in addition to the documentation and coding. They also punctuated that the difficulties were natural because of using something new.

After executing the tasks with the framework's dependencies, the second group executed them freely. However, it was natural that they were inspired by the framework, and therefore, much more demanding with their solutions. Participants took 32 minutes to execute stage two on average, but they did not consider it concluded. For them, the complexity was the same as in the previous stage on average. Nevertheless, they punctuate that they implemented simpler solutions than the ones available on the framework, which is more robust, and that they were uncertain if they attempted all software agents' needs.

Their last notes in the final interview considered that SADE4Health could help develop multi-agent systems, saving considerable time for people who want to develop these types of systems. However, they considered that better documentation is crucial, that the framework is difficult to learn in such

short time, and it needs some improvements and simplifications. Listed below are some phrases mentioned by the participants from the second group:

- *"Framework idea is very good and complete"*;
- *"In more complex cases, it could be simpler to use the framework. But in this simple case, it was easier to execute it freely"*;
- *"I used a lot of framework inspiration"*;
- *"Hard to learn fast"*.

6.7.3 Empirical Evaluation Conclusions

Although the experiment had few participants, which was undoubtedly its main limitation, we could learn a lot from its application and the participants' opinions. They showed us points that must receive improvements, both in code and documentation. As participants were splitted into two groups in order to apply the experiment in two different orders, we could conclude two essential aspects:

1. When the tasks not using the framework are executed first, the participant thinks about the problem for more time before being in contact with the framework. Then, when the participant needs to execute it using the framework, the he/she worries only about how to do it using the framework;

2. On the other hand, the participants could be more faithful to the software agents' essential concepts in the free tasks when the framework tasks came first.

Backing the hypotheses, we cannot affirm that SADE4Health reduced the effort to create software agents considering the time participants spent in performing the tasks, comparing stages two and five. However, it was a straightforward application case, and not all the participants reached the software agents' concepts in tasks in which the framework was not used.

Considering the second hypothesis, we could observe that participants ended the empirical evaluation with a good notion of software agents. The SADE4Health helped, mainly because it is guaranteed to meet the essential agents' concepts.

Another promising finding from this empirical evaluation was that SADE4Health can easily be used not only for eHealth applications, but also

for applications from other domains, as it is a modular framework in which the developer could use only the necessary modules.

Finally, we intend to consider every improvement pointed out by the participants. We believe that we could be closer to saying yes more accurately to both hypotheses by making these improvements. Furthermore, this empirical evaluation generated experiment data that can be more deeply analyzed and fully explored in future work.

7
Discussion

At the beginning of this dissertation, we mentioned two assumptions, which are the following.

1. There is a gap between multi-agent systems and the iOS developers' community;

2. The cost-benefit of constructing a more generic and complex system would be rewarded because we were not facing a sizable independent application but rather a family of related applications.

To confirm the first assumption, we conducted a systematic survey of five digital libraries, and it presented us strong evidence of the gap. After a few steps into the mapping process, we could not classify any paper joining the multi-agent systems research area with iOS contributions. Hence, we could identify that there really is a dearth of studies of agents in the Apple ecosystem, and this is one of our pillars of support to conduct our research and develop our framework.

The second assumption could be simply confirmed through the step-by-step description of an agent instantiation, a modeled use scenario based on it, and the use scenario for health field in Chapter 5. Furthermore, in Chapter 6, iOS developers used for the first time the SADE4Health dependencies, which brought us future improvements but showed that SADE4Health can already be used and is promising.

In SADE4Health, we developed a large structure to accommodate the creation of software agents with BDI reasoning, and it became as simple as extending classes to make use of this structure. Thus, if one needs to develop a software agent in Swift language, it is no longer necessary to recreate all of the structure and redo all of the research. The only concerns are the tasks of the agents and the functionality of the application.

From these assumptions, we identified a set of objectives for our research, as follows.

– Mapping which works were proposed and have relation to Swift and software agents;

– Offering a framework that follows good software engineering practices and provides features defined for any software agents;
– Allowing integration of native iOS platform resources, such as health and notification kits;
– Guaranteeing that health applications instantiated from the proposed framework can achieve the eight improvements listed in the introduction.

The first objective was achieved with the first assumption, where we verified a gap between software agents and the Apple ecosystem. The Swift community has not developed anything using multi-agent systems, and there are no native libraries from Apple to support agents.

The second was reached through the construction of SADE4Health, which applies design patterns and follows object-oriented programming, and it is related to the second assumption. SADE4Health is composed of five integrated modules, each with its importance to form the framework and make the use of software agents easy to accomplish through the Swift language. The *Core* Module is the central one, where the main objects around the creation of an agent are located and where the environment that will contain the agent is defined. An agent's reasoning can obey the cycle of a BDI reasoning, and the structure to make it possible is in the *BDIReasoning* module. The *Communication* module guarantees the communication between software agents, including communicating to agents of other FIPA-compliant applications or platforms. In the *Notification* module, ways to send alerts to the user are available. Finally, essential health concepts are considered in the *Health* module, and classes can be used to instantiate it, which counts on integration with *HealthKit*. In all modules, native Apple frameworks and libraries were used, such as *Foundation*, *Thread*, *NWFramework*, *User Notifications*, and *HealthKit*. Following the few steps described in Chapter 5, this entire structure is available to the developer.

To achieve the third objective, the three Apple libraries offered to the health domain were studied, *HealthKit*, *CareKit*, and *ResearchKit*. We decided that the essential one to be considered in SADE4Health is *HealthKit* because it inserts all instances developed through our framework into a collaborative network around the native Health application. Thus, we integrated *HealthKit* into the Health module through two classes, *HealthKitSetUpAssistant* and *HealthKitReaderWriter*.

The latter objective was demonstrated from the remote monitoring use scenario which developed an iOS app that used the proposed framework. Each improvement of an innovative technological solution can be clearly observed through the agents usage and the *HealthKit* integration.

Indeed, we can conclude that it is easier to use the structure offered by SADE4Health than to implement everything from scratch, especially considering the absence of concepts of multi-agent systems in the Apple ecosystem. However, we can not deny that some points need to be improved. We should consider more generic aspects of the health domain in the health module, not so focused on remote monitoring. In the *Core* module, we should make the use of BDI reasoning more straightforward, showing its benefits more strongly compared to a non-BDI reasoning agent. Finally, we should make the instantiation of some objects more straightforward and consider better usage documentation, as pointed out by iOS developers in the empirical evaluation described in Chapter 6.

8
Conclusion and Future Works

Our main goal with this dissertation was to present the BDI multi-agent system framework called SADE4Health. It proposes a FIPA compliant framework created from the Swift language, which allows the development of agents for Apple platforms - although it has only been tested for iOS, we believe it can be used on iPadOS, watchOS, TvOS and macOS as well. Thus, it is a contribution to the Apple ecosystem, where there is a gap of resources that can support the creation of software agents. Furthermore, this document presented a modeled use scenario, an instance of the framework related to healthcare workers being able to remotely monitor vital signs of patients, and an empirical evaluation showing the use of SADE4Health in practice.

From the modeled use scenario, we could exemplify through a simple example how to instantiate a software agent through SADE4Health. Furthermore, in the same chapter, we developed some unit tests to evaluate the framework dependencies.

From the use scenario developed, we were able to validate the framework proposed. In that instance, we observed that its design contributed toward making the patient's environment more proactive. Through this experimental system, we have also been capable of detecting anomalies in real-time and sending alerts to the health providers instantly and in an autonomous manner, by using software agents.

We confirmed many framework benefits from the empirical evaluation, but we too clearly noticed required improvements. We could notice that the use of SADE4Health guarantees the creation of a more faithful application to the software agents' essential concepts. On the other hand, the empirical evaluation showed that the framework needs to be simplified in some aspects, such as the instantiation of some objects.

For future work, we are considering a security module that allows the developer to encrypt data, to have another layer that guarantees the security of sensitive data. Despite the security base ensured by Apple in a number of different ways - mainly by always asking for the user's permission - health data should receive a closer look. Thus, encryption seems to be a good solution, making our SADE4Health framework more reliable. Another future work under

consideration is to test the communication of agents with agents developed on other platforms. Currently, although the framework has created protocols for communication with agents developed on different platforms, only tests with agents created from the Swift language were conducted.

Finally, SADE4Health is already available in GitHub Repository[1] and will be in constant evolution.

[1] Available in `https://github.com/PolyanaSRB/SADE`

Bibliography

[1] STATISCA. **Number of mhealth apps available in the google play store from 1st quarter 2015 to 1st quarter 2021**. https://www.statista.com/statistics/779919/health-apps-available-google-play-worldwide/, 2021. Accessed: 2021-07-16.

[2] STATISCA. **Number of mhealth apps available in the apple app store from 1st quarter 2015 to 1st quarter 2021**. https://www.statista.com/statistics/779910/health-apps-available-ios-worldwide/, 2021. Accessed: 2021-07-16.

[3] VAN DER VLIST, F.; HELMOND, A.; CHAO, J.; DIETER, M.; TKACZ, N. ; WELTEVREDE, E.. **[covid-19]-related android (google play) and ios (app store) app ecosystems**. https://osf.io/wq3dr/, 2021-2022. Accessed: 2021-08-06.

[4] STATISCA. **Growth in the number of medical apps downloaded during the covid-19 pandemic by country in 2020**. https://www.statista.com/statistics/1181413/medical-app-downloads-growth-during-covid-pandemic-by-country/, 2020. Accessed: 2021-07-16.

[5] FERNANDES, C. O.; DE LUCENA, C. J. P.. **An internet of things application with an accessible interface for remote monitoring patients**. In: DESIGN, USER EXPERIENCE, AND USABILITY: INTERACTIVE EXPERIENCE DESIGN, p. 651–661. Springer International Publishing, 2015.

[6] FERNANDES, C. O.; DE LUCENA, C. J. P.; DE LUCENA, C. A. P. ; DE AZEVEDO, B. A.. **Enabling a smart and distributed communication infrastructure in healthcare**. In: INNOVATION IN MEDICINE AND HEALTHCARE 2015, p. 435–446. Springer International Publishing, 2016.

[7] MARKIEWICZ, M. E.; DE LUCENA, C. J. P.. **Object oriented framework development**. XRDS, 7(4):3–9, 2001.

[8] **Unified modeling language (UML)**. `https://www.uml.org/`. Accessed: 2021-08-20.

[9] GAMMA, E.; HELM, R.; JOHNSON, R. ; VLISSIDES, J.. **Design Patterns: Elements of Reusable Object-Oriented Software**. Addison-Wesley Professional, 1st edition, 1994.

[10] WEISS, G.. **Multiagent Systems – A Modern Approach to Distributed Artificial Intelligence**. MIT Press, 1st edition, 1999.

[11] WOOLDRIDGE, M. J.. **An Introduction to MultiAgent Systems**. John Wiley & Sons, 2nd edition, 2009.

[12] RUSSELL, S.; NORVIG, P.. **Artificial Intelligence: A Modern Approach**. Prentice Hall, 3rd edition, 2009.

[13] RAO, A. S.; GEORGEFF, M. P.. **BDI Agents: From theory to practice**. In: PROCEEDINGS OF THE FIRST INTERNATIONAL CONFERENCE ON MULTI-AGENT SYSTEMS (ICMAS-95), p. 312–319, 1995.

[14] WOOLDRIDGE, M.; JENNINGS, N. R. ; KINNY, D.. **A methodology for agent-oriented analysis and design**. In: PROCEEDINGS OF THE THIRD ANNUAL CONFERENCE ON AUTONOMOUS AGENTS, AGENTS '99, p. 69–76, 1999.

[15] **Swift**. `https://swift.org/`. Accessed: 2021-07-23.

[16] LUNDEN, I.. **App store hits 20m registered developers and $100b in revenues, 500m visitors per week**. `https://techcrunch.com/`, 2018. Accessed: 2021-07-23.

[17] MILLER, C.. **Apple announces record holiday q1 2020 earnings: revenue of $91.8 billion, more**. `https://9to5mac.com/`, 2020. Accessed: 2021-07-23.

[18] **Healthkit**. `https://developer.apple.com/documentation/healthkit/`. Accessed: 2021-07-23.

[19] **Carekit**. `https://developer.apple.com/documentation/carekit/`. Accessed: 2021-07-23.

[20] **Researchkit**. `https://www.researchandcare.org/researchkit/`. Accessed: 2021-07-23.

[21] **Privacy**. https://www.apple.com/privacy/. Accessed: 2021-07-23.

[22] **Protecting user privacy**. https://developer.apple.com/documentation/healthkit/protecting_user_privacy. Accessed: 2021-07-23.

[23] HEISLER, Y.. **Physicians who use mobile apps prefer iphones and ipads**. https://www.engadget.com/, 2013. Accessed: 2021-07-30.

[24] PERKINS, S.. **9 out of 10 doctors prefer the iphone**. https://www.slashgear.com/, 2011. Accessed: 2021-07-30.

[25] ASHLYN, R.; ET AL. **What percentage of doctors and nurses have android vs. ios devices in the uk?** https://askwonder.com/research/, 2017. Accessed: 2021-07-30.

[26] ASHTON, K.. **That 'internet of things' thing - in the real world, things matter more than ideas**. http://www.rfidjournal.com/articles/view?4986, 2009. Accessed: 2021-07-30.

[27] FINKENZELLER, K.. **RFID Handbook: Fundamentals and Applications in Contactless Smart Cards, Radio Frequency Identification and Near-Field Communication**. Wiley, 3rd edition, 2010.

[28] **FIPA**. http://www.fipa.org. Accessed: 2021-07-23.

[29] **JADE**. http://jade.tilab.com/. Accessed: 2021-07-23.

[30] **JADEX**. https://www.activecomponents.org/. Accessed: 2021-07-23.

[31] **BDI4JADE**. https://www.inf.ufrgs.br/prosoft/bdi4jade/. Accessed: 2021-07-23.

[32] SU, C.-J.; CHU, T.-W.. **A mobile multi-agent information system for ubiquitous fetal monitoring**. International journal of environmental research and public health, 11:600–625, 2014.

[33] WEISER, M.. **Some computer science issues in ubiquitous computing**. Commun. ACM, 36(7):75–84, 1993.

[34] SU, C.-J.; WU, C.-Y.. **Jade implemented mobile multi-agent based, distributed information platform for pervasive health care monitoring**. Applied Soft Computing, 11(1):315–325, 2011.

[35] MOHAMMADZADEH, N.; SAFDARI, R.. **Patient monitoring in mobile health: opportunities and challenges**. Medical archives, 68(1):57–60, 2014.

[36] LTD, D. S. L.. **My-vitals**. https://apps.apple.com/br/app/my-vitals/id1191476063. Accessed: 2021-07-23.

[37] VIVALNK. **Multi vital monitor**. https://apps.apple.com/br/app/multi-vital-monitor/id1445374809. Accessed: 2021-07-23.

[38] LTD, B. A.. **Binah team**. https://apps.apple.com/br/app/binah-team/id1528346500. Accessed: 2021-07-23.

[39] KITCHENHAM, B.. **Guidelines for performing systematic literature reviews in software engineering, version 2.3**. Technical Report EBSE-2007-01, Keele University and University of Durham, 2007.

[40] GROSMAN, J.. **Findpapers**. https://gitlab.com/jonatasgrosman/findpapers, 2020. Accessed: 2020-12-06.

[41] **ACM digital library**. https://dl.acm.org/. Accessed: 2020-12-06.

[42] **arXiv**. https://arxiv.org/. Accessed: 2020-12-06.

[43] **Institute of Electrical and Electronics Engineers, IEEE**. https://ieeexplore.ieee.org/Xplore/home.jsp. Accessed: 2020-12-06.

[44] **PubMed**. https://pubmed.ncbi.nlm.nih.gov/. Accessed: 2020-12-06.

[45] **Scopus**. https://www.scopus.com/. Accessed: 2020-12-06.

[46] **Thread class**. https://developer.apple.com/documentation/foundation/thread. Accessed: 2021-10-10.

[47] **Saving data to healthkit**. https://developer.apple.com/documentation/healthkit/saving_data_to_healthkit. Accessed: 2021-10-10.

[48] **User notifications**. https://developer.apple.com/documentation/usernotifications/. Accessed: 2021-10-10.

[49] **Xctest class**. https://developer.apple.com/documentation/xctest. Accessed: 2022-04-08.

[50] BUCANEK, J.. **Learn Objective-C for Java Developers**, chapter 20. Model-View-Controller Pattern, p. 353–402. Apress, 2009. Accessed: 2021-10-10.

[51] **Network framework**. `https://developer.apple.com/documentation/network`. Accessed: 2022-03-08.

[52] JEDLITSCHKA, A.; PFAHL, D.. **Reporting guidelines for controlled experiments in software engineering**. In: INTERNATIONAL SYMPOSIUM ON EMPIRICAL SOFTWARE ENGINEERING, p. 10, 2005.