

Iván Armando Alatorre Espinoza

Multi-Agent Systems in Mobile Application development

Final Master's Project

Directed by
Dr. Antonio Moreno & Prof. Mario Jimenez

Computer Security Engineering and Artificial Intelligence



UNIVERSITAT ROVIRA I VIRGILI



Tarragona
2020

Abstract

Native mobile application development is a task that is increasing among the industries. The implementation step in the software development life-cycle is a hard and time-consuming task performed by developers together with product owners and software architecture leaders. There is not a specific path to follow, due to the variability of how developers analyze, think and resolve different sorts of problems. However, there are guidelines that developers follow in order to write qualified, functional and clear software applications.

An architectural pattern, that refers to a design pattern, that deal mainly with the architecture itself and how responsibilities between different modules or components can be defined based on the requirements. As the case of VIPER (View Interactor Presenter Entity and Router) together with the idea of a design pattern that prevents high coupling and standardize a common way of writing code, provide a general solution for recurring situation when developing mobile applications.

On the other hand having the power of artificial intelligence that could be applied in such a way to improve the hard and time-consuming task in the server communication phase. Multi-Agents systems, autonomous entities, that collaborates to solve tasks, offering flexibility due to their inherent ability to learn and make autonomous decisions.

As a result, having in some structured manner the way to split all the set of services given by an API (Application Programming Interface) where all the data needed is explicit, as the basis of the service name, URL, headers, request parameters, request type, response parameters, etc. Giving to each agent the capability of read each piece and write little pieces of code that together as a collaborative program, the agents could build the necessary code files following the architectural and design pattern. Hence, the time taken is significantly reduced.

Acknowledgment

The completion of this undertaking could not have been possible without the support of the **Dr. Antonio Moreno**, I am extremely grateful for your patience, advice, guidance, and providing me with your knowledge and expertise in this trail.

In additional, I wish especially to thank **Prof. Mario Jimenez**, due to all the timeshare, knowledge, advice during social service at the university. That gave me one of the most important opportunities to learn, enjoy and be really passionate today.

I would also like to extend my gratitude to **Dr. Susana Álvarez** for giving me the opportunity to propose this TFM project and therefore combine two topics that I find really interesting such as Mobile Development and Multi-Agent Systems.

In addition, I would like to extend my gratitude and my thanks to **Prof. Ana Laura Sandoval**, due to her advice, guide and knowledge in my English skill, through this TFM.

Moreover, I would like to remind all the people to whom I owe being here, I know that it was not easy to get here since I decided to undertake this long journey. To my family, my father **Hector Alatorre Nuño** and my mother **Leticia Espinoza Martinez** who taught me how to be a good person, inspiring me day by day, my brother **Hector Alatorre** and my little sister **Priscila Alatorre**, giving me the courage to go ahead.

Thank you all, a constant source of inspiration in my life.

Index

Abstract	3
Acknowledgment	5
Index	7- 8
Figure Index	9-10
Table Index	11
Chapter 1: Introduction	12-16
1.1 Motivation	12-13
1.2 Objectives	13-15
1.3 Chapters overview	15-16
Chapter 2: Postman and web services	17-27
2.1 Postman	18-19
2.2 Guiding principles of REST	19-20
2.3 Web services	20-23
2.4 Collections of services and exportation	23-24
2.5 Examples of collections of services	24-27
Chapter 3: Architectural and Design patterns	28-40
3.1 Why pattern should be used?.....	29-30
3.2 Pattern Classification	30-32
3.3 How VIPER works?	32-33
3.4 Data Access Object (DAO)	34-35
3.5 Example of a service implementation using VIPER and Data Access Object	35-38
3.6 Service mobile architecture	38-40
Chapter 4: Multi-Agent Systems	41-52
4.1 Agents, behaviours and architectures	41-45
4.2 Multi-Agent systems in distributed problem solving	45-46
4.3 Agent communication mechanisms	46-48
4.4 Voting protocol	48-49
4.5 JADE platform	49-52

Chapter 5: Multi-Agent system to improve mobile application development	53-70
5.1 Software requirements	54
5.2 MAS with service configuration	54-55
5.3 Agent sorts and roles	55-56
5.4 MAS architecture	57-59
5.5 Shared resource files	59-61
5.6 Agent tasks	61-64
5.7 MAS communication (workflow)	64-70
Chapter 6: Conclusions and future related works	71-77
6.1 Results obtained	71-75
6.2 Conclusions	75-76
6.3 Future related works	76-77
Appendices	78-87
References	88

Figure index

Chapter 2: Postman

Fig. 1 – Postman's launchpad screen	19
Fig. 2 – Defining a service request URL and type	21
Fig. 3 – Defining a service request URL params	21
Fig. 4 – Defining a service request header params	22
Fig. 5 – Sending the request	22
Fig. 6 – Postman response obtained	22
Fig. 7 – Saving request in Postman	23
Fig. 8 – Saving response in Postman	23
Fig. 9 – Postman collection of services called HotelsAPI ready to be exported	24
Fig. 10 – Exporting Postman collection	24
Fig. 11 – Postman collection basic Info	25
Fig. 12 – Postman collection items or services definition	25
Fig. 13 – Postman collection item request structure definition	26
Fig. 14 – Postman collection item response structure definition	27

Chapter 3: Architectural and Design patterns

Fig. 15 – Patterns classification	30
Fig. 16 – Architectural pattern sorts	31
Fig. 17 – Interactive architecture patterns	31
Fig. 18 – Architectural and design pattern path to VIPER	32
Fig. 19 – VIPER architecture	33
Fig. 20 – Data Access Object (DAO) structure diagram	34
Fig. 21 – Data Access Object (DAO) applied in VIPER perspective	35

Chapter 4: Multi-Agent Systems

Fig. 22 – Agent in its environment	43
Fig. 23 – Agent behaviour visualization	44
Fig. 24 – Direct passing message	47
Fig. 25 – Blackboard indirect communication	48
Fig. 26 – Architecture of a FIPA agent platform	49
Fig. 27 – FIPA message structure	50
Fig. 28 – FIPA-ACL workflow among agents	52

Chapter 5: Multi-Agent System to improve mobile application development

Fig. 29 – XML configuration file for the HotelsAPI	55
Fig. 30 – Multi-Agent System architecture – first round general overview (VIPER Interactor & DAO design pattern)	57
Fig. 31 – Multi-Agent System architecture – final round general overview (Service mobile architecture)	58
Fig. 32 – Agent registration into the directory facilitator to handle direct communication....	65
Fig. 33 – Manager agent behaviour assigned to a waiting queue	66
Fig. 34 – Manager first round workflow)	66
Fig. 35 – Manager final round workflow (Configure Service Architecture)	67
Fig. 36 – Manager final round workflow (Configure Service Backend)	68
Fig. 37 – Worker first round workflow	68
Fig. 38 – Worker final round workflow	69
Fig. 39 – Worker workflow	69

Chapter 6: Conclusions and future related works

Fig. 40 – APIs tested in the TFM project	75
--	----

Table index

Chapter 3: Architectural and Design patterns

Tab. 1 – Advantages and disadvantages of using design patterns	29
--	----

Chapter 4: Multi-Agent Systems

Tab. 2 – FIPA-ACL message parameters and description	51
--	----

Chapter 5: Multi-Agent System to improve mobile application development

Tab. 3 – XML configuration file attributes	54
Tab. 4 – Service configuration, manager resource file	59
Tab. 5 – Generated code file, worker resource file	60
Tab. 6 – Generated code, writer resource file	60
Tab. 7 – Blackboard TASK structure	65

Chapter 6: Conclusions and future related works

Tab. 8 – Airlines API execution time through 10 rounds and its average, mapping 2 services	72
Tab. 9 – Hotels API execution time through 10 rounds and its average, mapping 5 services	72
Tab. 10 – RecargasTuCell API execution time through 10 rounds and its average, mapping 8 services	73
Tab. 11 – CockTail API execution time through 10 rounds and its average, mapping 12 services	73
Tab. 12 – All In One API execution time through 10 rounds and its average, mapping 63 services	74
Tab. 13 – APIs tested in the TFM project	74
Tab. 14 – Future MAS configuration attributes to increase the scope capabilities	77

Chapter 1: Introduction

With the arrival of smartphones and the new generation of touch devices, a new era begun, the App Store for iOS devices at July 10th, and the Google Play Store for Android at October 22th, Both of them launched in 2008. Developers were able of use these platforms to upload their own applications, a new market appeared in the software development. With the possibility of having a wide variety of applications, from maps searching, banking transactions, social networks, multimedia playback, file management, etc. New ideas arise every day for both new application concepts and transforming old tools into mobile applications. This market is continuously growing year by year, that is why industries and companies are constantly looking for developers capable to code mobile applications. To date, most of the companies have developed a product in the iOS and / or Android platform.

That opened many questions regarding to find a simple way to build a mobile application. Keeping in mind that there are a variety of factors involved in the development of a mobile application. Usually, a development takes 18 weeks to produce a first version or minimum viable product. The result based on 100 mobile architects show how long they had to expect to develop it, but it depends completely in the amount of requirements or functionality that the application will have, the resources that the customer is capable to pay and the time to invest in it, etc. [2, 3, 4].

It is a huge process that is continuously analyzed by the community of developers, industries and research engineers around the world, to more quickly achieve the common goal of basic construction of an architecture between different situations. Therefore, design patterns became very popular in the field of software development, trying to find better and cleaner processes, even knowing that mobile technology is quite recent.

On the other hand, artificial intelligence is a topic that appeared formally as an academic discipline in 1956, by John McCarthy, Marvin Minsky and Claude Shannon at the Dartmouth conference, there begun some discussions of the possibility of creating an artificial brain earlier years. Artificial intelligence has the power to transform the tedious and time-consuming practices. Design patterns can be analyzed and checked, in order to apply Multi-Agent systems to improve the time-consuming practices, automating the process following architectural and design patterns. As a result, this increases the profitability of businesses potential.

Multi-Agent Systems, or more precisely intelligent agents a novel paradigm for software applications development, supporting the simulation of complex individuals interactions.

These agents are capable of achieve one or more tasks depending on the final implementation or the requirements involved in the solution proposed.

Specifically, in this TFM the usage of Multi-Agents to improve the server communication in a mobile application development following architectural and design patterns. The first one called VIPER, followed by the DAO pattern, and finally applying a service mobile architecture, thereby making easy the implementation of the mobile application, automating in this way the creation of those code files needed in the server communication part.

1.1 Motivation

So well, in this TFM work, I would like to introduce you, how the use of a design pattern together with a collection of defined web services (Postman) can be merged with the paradigm of Multi-Agent Systems to gather a huge amount of agents that will be labeled with some role and hierarchy, making them collaborate, in order to read those defined services from the collection given, to analyze, split, and create micro code generation tasks. As a result of following, the architectural and design pattern a set of code files, the Multi-Agent system will automatically generate those files which will reduce the time-consuming that normally a developer takes to write.

The beauty of combining different technological paradigms in a simple, feasible, practical and useful platform that developers can use and save time to concentrate in other development phases.

“The art of combining different technologies, methodologies and paradigms, provide us with a new perspective to confront any obstacles”.

1.2 Objectives

To be able to accomplish with the purpose of this new platform some objectives will be established. First of all, the main goal of this TFM is to build a Multi-Agent System that will generate java android code files, in such a way, it can be easily added into an android project, providing them with the final service response data in plain old java object (POJO) which is only an instance of a class that does not follow any of the major java object models, conventions, or frameworks, its is just the class declaration with their own properties and encapsulation methods. *[Appendix A]*

Therefore, a developer can just take the final data model and use it in the mobile application to manage the data as the mobile application needs. As a result, the time-consuming will considerably decrease.

Secondly, provide with the enough knowledge of the architectural and design pattern paradigms, more specifically, with VIPER, DAO pattern and the service mobile architecture. Besides, in a more general way, other sorts of contexts and definitions involved. Moreover, how Postman is used, in order to create and export the collection of services as a file, that will be the input in the Multi-Agent System.

These agents should be divided by roles of the tasks that they will handle, That is, having one web service (a little piece of the input file), It will be assigned to a manager, the manager will read and extract the name, URL, headers, request parameters, request type, response parameters, etc. In this way, the manager will create something called **“Service configuration”**, eventually the manager will create 5 workers agents that will represent five code files, to the moment, I just will introduce their names, (Dao interface, Dao Implementation, the Interactor interface, the request and response).

Each one of the workers will take one of them to work with, each worker will create a bunch of writers together with micro code generation tasks, those agents will handle the code generation protocols following the architectural, design pattern and service architecture. Thereby, each service branch will be composed by 1 manager, 5 workers, and the set of writers needed in that specific code file. Full details regarding how many writers are needed by each file and how they will work, they are going to appear in the chapter 5 dedicated to the MAS platform.

Imagine the above process as the first round of the platform, when the manager parsed the web service and all the agents creation took place, dividing responsibilities among the workers and the writers through micro code generation tasks, then each worker has to wait until all his set of writers accomplish their corresponding tasks, in order to merge all the pieces of generated code, becoming in a final generated file, that will be sent to the manager back, in order to physically create it, inside a folder.

Furthermore, is going be a second round, that will handle the creation of other five files related to the server mobile architecture, such as backend information, error manipulation, API methods definition, service configuration, and the service request as master class. To do so, the first round should be completed by each manager's branch, in order to know which branch obtains the best execution time (which means the least time necessary to finish). In addition, a voting protocol among agents will be run, to determine the most popular agent. In this way, those two branches will be used in the second round.

Having those agent's branches, the final five files will be divided in such a way that one of them will build two of the files and the other the three missing files. Taking advantage of the existing agents by each branch. Thus, repeating more or less the same process of split the file into micro generated code tasks and returning the pieces to the worker, this last one merging them into a generated code file and sending it to the manager to physically write it.

The output of this new platform is going be a folder by each service containing the first five files, and lastly five more files regarding the backend manipulation, and service manipulation and configuration.

1.3 Chapters overview

➤ Chapter 2: Postman and web services

The purpose of this chapter is to explain the software called Postman, in addition, how it is used to export a collection of web services, how to write and save each web service request and response, as well as test it, to have a complete definition of them. Also, give some definitions about the different types of web services, how they are composed and their capabilities, what are the guidelines or guiding principles to build a RESTful API. In this way, identifying the web service as resources that a mobile application can use to communicate with a server, responding with the data that will later be used to be shown to the user or client.

➤ Chapter 3: Architectural and Design patterns

The purpose of this chapter is to explain how patterns should be used to have a clear, functional, and well-structured Android Java application. In addition, give some definitions about the classification of patterns, providing the basic concepts, trying to provide enough knowledge to understand its structure. There will be three main patterns, firstly one called the VIPER architectural pattern that implements the separation of concerns paradigm and makes the application structure more modular. Second, the DAO pattern whose main objective is to provide access to a certain requester through an interface and finally a mobile service architecture, this is the way in which a mobile application can use the previous patterns. In order to give access to the response model of a set of services in a module that uses reusable code, hierarchy classes, and configuration classes. Bringing software technology to a point where new alternatives and improvements are sought.

➤ **Chapter 4: Multi-Agent systems**

The purpose of this chapter is to introduce with more details regarding Multi-Agent Systems (MAS) paradigm, how JADE platform works (the one used to build this TFM work). Providing some basic definitions of what is an Agent, what is the behaviour of an agent in a certain environment where other agents may be or not collaborating each other to achieve their corresponding tasks. The basic architectures and properties that an agent can be labeled with. In addition of how agents can communicate each other sending messages and sharing resources making them the perfect paradigm, to be combined with Postman and architectural and design patterns to automatically generate java code files.

➤ **Chapter 5: Multi-Agent System to improve mobile application development**

The purpose of this chapter is to fully explain the integration of all the technologies, software tools and different paradigms that were mentioned in the chapters 2, 3 and 4, in order to achieve one platform which will take a Postman collection of web services as input, then the Multi-Agent System will split the huge file into small pieces (one piece correspond to one web service), and how each piece will be passed by a branch of agents that will be collaborating to code five java files corresponding to the architectural and design pattern. In addition, producing competitive agents evaluating the execution time that each branch needs to accomplish the global task. The full architecture of agents used in this TFM and all the tasks that were needed to generate each piece of code. Besides, how agents communicate each other internally in their respective branch to delegate the tasks. Exposing in this way the workflow of each agent role and how each one of them interact, generating and sharing file resources in every step of the MAS project.

➤ **Chapter 6: Conclusions and future related works**

The purpose of this chapter is to analyze the output of the tested MAS project, as well as the performance obtained through different APIs. In this way, to write the conclusions that were achieved, in order to see if using a Multi-Agent System really can improve in some way, a mobile development environment. Furthermore, discuss regarding possible future related works or improvements that this MAS project can get, to increase the scope, for example generating other sorts of patterns with different languages.

Chapter 2: Postman and web services

The purpose of this chapter is to explain how the software called Postman is used to export a collection of web services, how to write and save each tested web service request and response, in order to have a complete definition of them. So that, if you have enough knowledge to define your own API web service using this tool and export it, you could move on to the next chapter.

To being to understand, in mobile development world exists many paths to invoke a web service, this is how a mobile application communicates with a server, waiting for a response with certain data, in other words, the client (mobile app) – server architecture. There are two approaches that are most commonly used. One is called Simple Object Access Protocol (SOAP) and Representational State Transfer (REST).

SOAP defines a standard communication protocol or set of rules, specification to XML-based message exchange. SOAP uses different transport protocols, such as HTTP and SMTP. The standard protocol HTTP makes easier for SOAP model to tunnel across firewalls and proxies without any modifications to the SOAP protocol. SOAP can sometimes be slower than middleware technologies due to its verbose XML format.

REST describes a set of architectural principles by which data can be transmitted over a standardized interface such as HTTP. REST does not contain an additional messaging layer and focuses on design rules for creating stateless services. A client can access the resource using the unique URI (Uniform Resource Identifier) and representation of the resource is returned. With each new resource representation, the client is said to transfer state. While accessing RESTful resources with HTTP protocol, THE URL of the resource serves as the resource identifier and GET, PUT, DELETE, POST and HEAD are the standard HTTP operations to be performed on that resource. Commonly represented in JSON format

(JavaScript Object Notation) a standard text-based format for representing structured data based on JavaScript object syntax.

In order to understand the purpose of this chapter, firstly, some definitions should be explained. Even the reason of the usage of Postman as an automated tool, and its potential to be used by a Multi-Agent System, being capable of read the services defined by the tool and transform them into files codes, following the REST protocol, As a result those files can be easily used in an android mobile application.

2.1 Postman

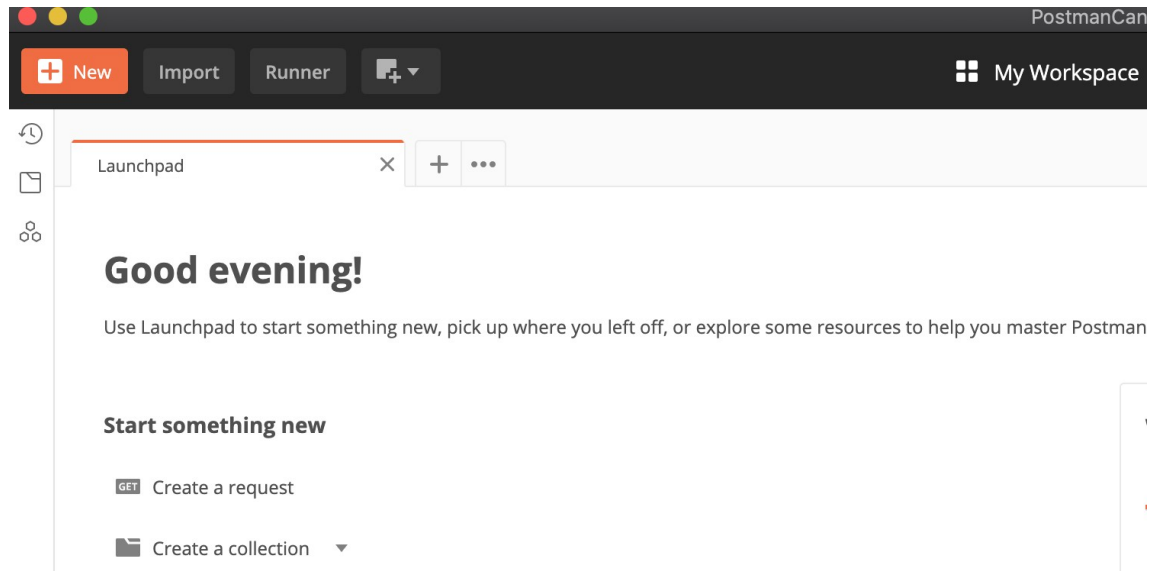
Postman is a tool that handle both SOAP and RESTful API services, It offers a clear and functional user interface with which to make HTML requests, without the hassle of writing a bunch of code just to test and API's functionality. You can download it here [18].

Send requests and view responses, defining complex requests, the user can save custom methods and send requests with the following body types:

- **URL-encoded:** The default content type for sending simple text data.
- **Multipart/form-data:** For sending large quantities of binary data or text containing non-ASCII characters.
- **Raw body:** For sending data without any encoding.
- **Binary data:** For sending image, audio, video or text files.

Some features such as inspect responses in a clear and readable format, view the status code, response time, and response size. Postman's automatic language detection, link and syntax highlighting, search, and text formatting make it easy to inspect the response body. It also has a feature that makes easily turn API data into charts and graphs with Postman visualizer. To illustrate how Postman looks like, the following screenshot represent the initial screen when Postman is launched the first time. Clearly, in that screen all the basic features

are listed. The most important in this case, are the features to create a service request and create a collection of requests.



(Fig. 1 – Postman's launchpad screen)

2.2 Guiding principles of REST

Like other architectural styles, REST also has its own 6 guiding principles which must be satisfied if an interface needs to be referred to as RESTful.

- **Client-Server:** By separating the user interface concerns from the data storage concerns, improving the portability of the user interface across multiple platforms and improve scalability by simplifying the server components.
- **Stateless:** Each request from the client to server must contain all of the information necessary to understand the request, and cannot take advantage of any stored content on the server. The session state is therefore kept entirely on the client.
- **Cacheable:** Cache constraints require that the data within a response to a request be implicitly labeled as cacheable or non-cacheable. If a response is cacheable, then a client cache is given the right to reuse that response data for later, equivalent requests.

- **Uniform interface:** By applying the software engineering principle of generality to the component interface, the overall system architecture is simplified and the visibility of interactions is improved. In order to obtain a uniform interface, multiple architectural constraints are needed to guide the behavior of components. REST is defined by four interface constraints: identification of resources; manipulation of resources through representations; self-descriptive messages; and hypermedia as the engine of application state.
- **Layered system:** The layered system style allows an architecture to be composed of hierarchical layers by constraining component behavior such that each component cannot see beyond the immediate layer with which they are interacting.
- **Code on demand (Optional):** REST allows client functionality to be extended by downloading and executing code in the form of applets or scripts. This simplifies clients by reducing the number of features required to be pre-implemented.

2.3 Web service

The term web service describes a standardized way of integrating web-based applications using the SOAP, REST, and other protocols, in this occasion writing regarding only the previously mentioned. A web service is a method of communication between two electronic devices over a network. It is a software function provided at a network address over the web with the service “always-on” as in the concept of utility computing.

Many organizations use multiple software systems for management. Different software systems often need to exchange data with each other, and web service is a method of communication that allows those systems to exchange this data over the internet. The software system that requests data is called a service requester, whereas the software system that would process the request and provide the data is called a server provider.

Imagine a mobile application whose objective is to locate hotels inside a certain city. So, a web service called “**SearchHotel**” may be defined there. Taking one of the public API at [10]. The following service is defined:

- **API endpoint:** <http://hotels4.p.rapidapi.com/locations/search>
- **Request Type:** GET

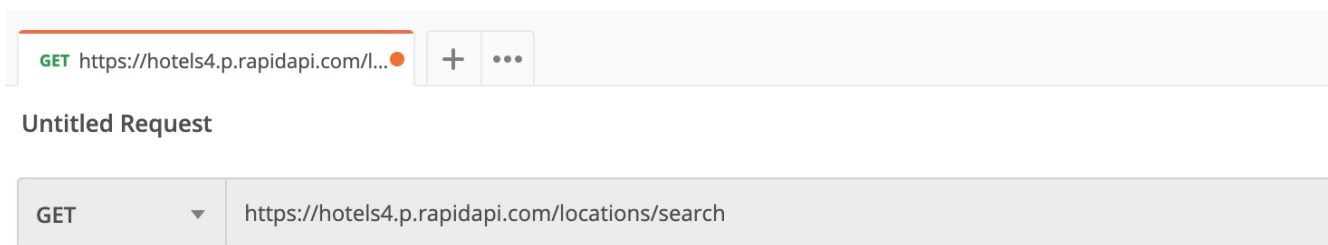
➤ **Headers:**

- **x-rapidapi-host:** hotels4.p.rapidapi.com
- **x-rapidapi-key:** a82156eb53msh83ccd3c830f58f5p10d463jsn6c5c06889085

➤ **URL Params:**

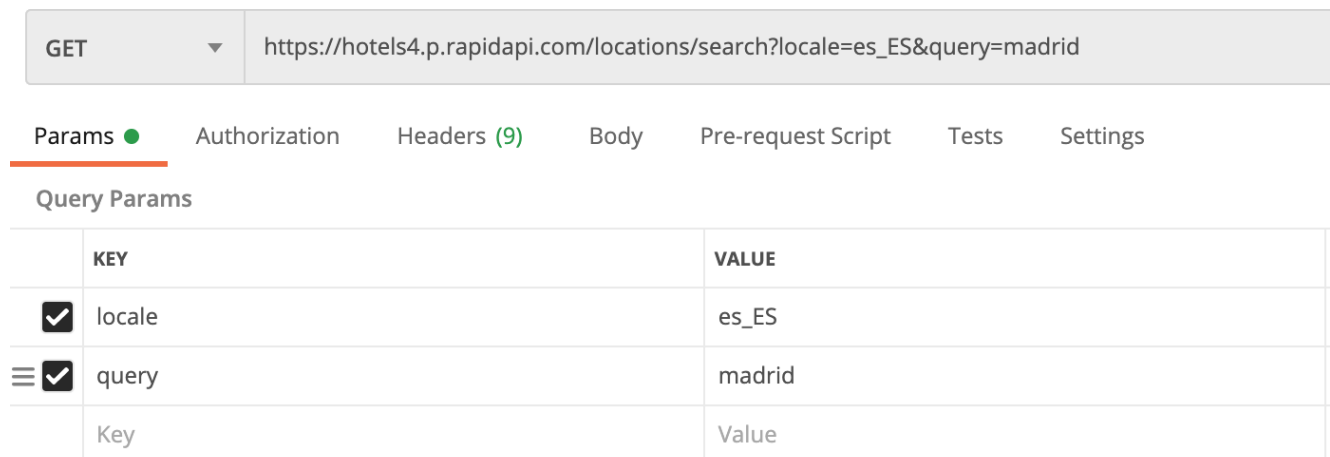
- **locale:** es_ES
- **query:** madrid

The above definition can be written in Postman in order to test it. First of all, write the URL in the space and select the request type.



(Fig. 2 – Defining a service request URL and type)

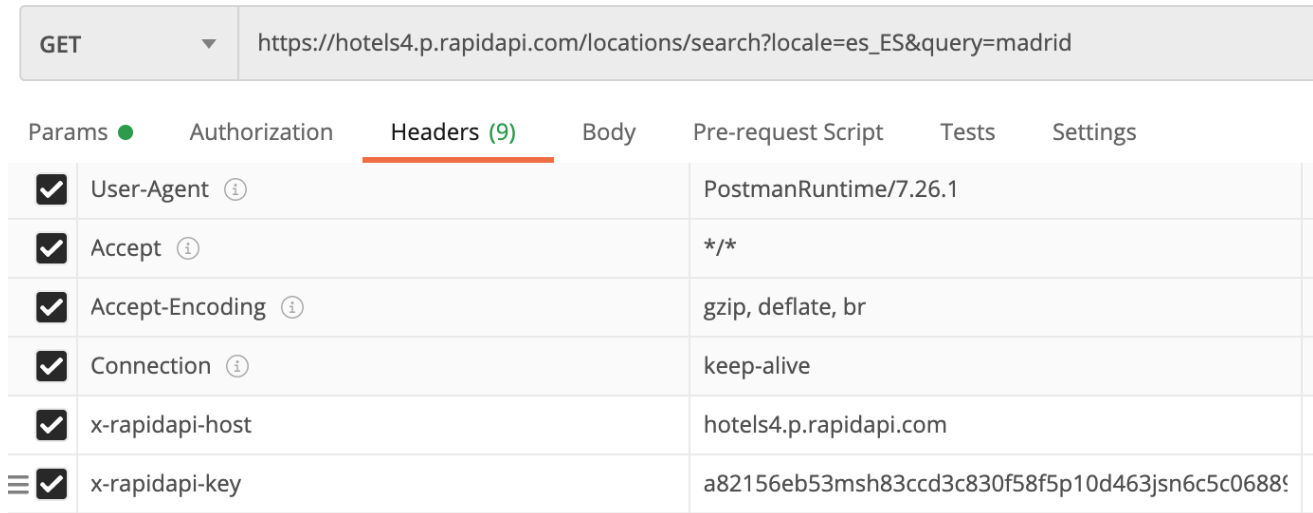
Following with the definition in the below side of the URL and the request type will be a bar of features in our request, this bar contains, the params, authorization, headers, the body, among others. Let's open the params bar by doing click in it. A table of query params will be opened because this request is of the type GET, which means those params should be passed through the URL. As follows:



(Fig. 3 – Defining a service request URL params)

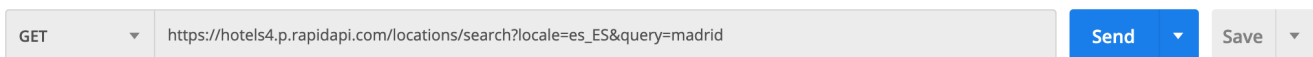
Adding these values into the text fields, automatically these values will be added into the URL as request params. This example does not require authentication methods or body

parameters, so the next is to add the header's value, automatically some values will be default added by Postman. However, our values should be manually added. By clicking in the header bar and then writing our values into the text fields the result will be.



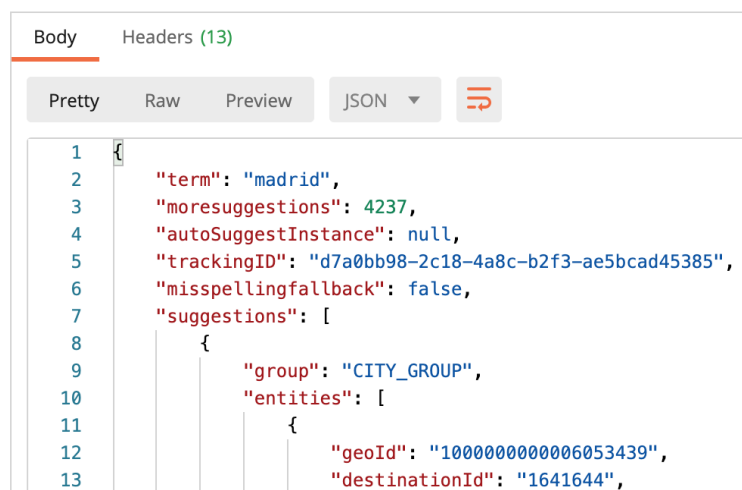
(Fig. 4 – Defining a service request header params)

Now, our first web service is ready to test. By pressing the send button the request will be launched to the API server.



(Fig. 5 – Postman sending the request)

Then if the final server is working fine, the response will appear in the below part of the software. Describing in JSON format the raw data for that service in this example as the parameter values were to looking by the location of Spain and precisely in the city of Madrid. Those hotels that are saved in the server will be displayed.



(Fig. 6 – Postman response sample obtained)

Finally, a functional web service that a mobile app can consume and display in some way for user interaction. But first, **the request and the response should be saved**, later on, merge it with other services to become them as a collection of services. To do so, in the save button you can select “save as”, following adding a service name, in this case, “SearchHotel”, optionally, add a description of what the service is for, and in the bottom side, you can store it into an existing collection or create a new one to add.

(Fig. 8 – Saving response in Postman)

(Fig. 7 – Saving request in Postman)

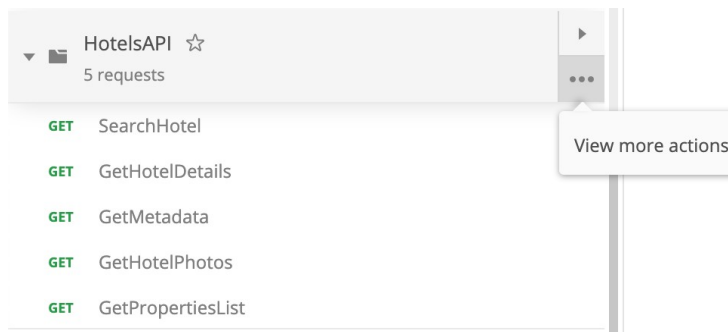
On the top side of the response window appear the button to save the response as an example, providing a name of how to identify it.

Once, the request and the response are saved, all is ready to merge it into other services.

2.4 Collections of services and exportation

Now that some definitions were explained, the purpose of these chapter has arrived. That is have a file with a bunch of service definitions grouped and saved to become the input of our Multi-Agent systems. This is a functionality in Postman that allows the user to keep API requests and elements organized.

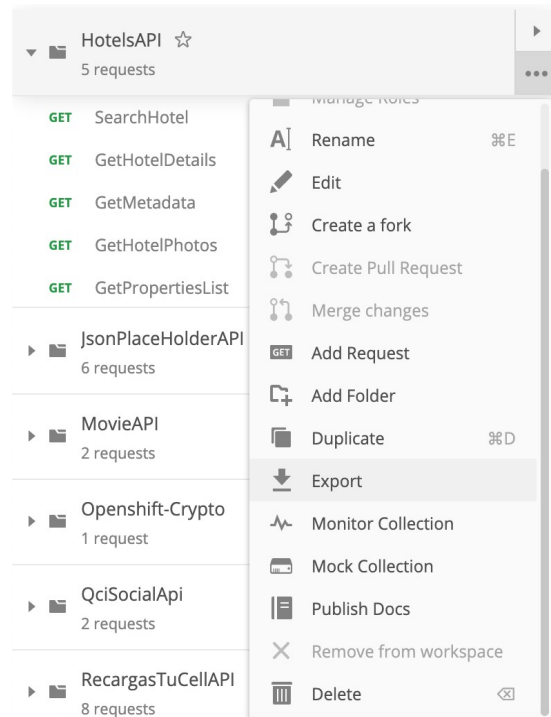
As was mentioned, in the creation of a simple web service call using Postman, the next is to repeat those steps adding the number of services that our mobile application requires. And export the final collection as a simple JSON file. By clicking in the left bar of Postman software, all the collections will be listed and inside each collection the set of web services defined. In this example, 5 services were added according to the API definition taken at [10].



(Fig. 9 – Postman collection of services called HotelsAPI ready to be exported)

The services are ready to be exported.

Each one of them has its own parameters and responses. In order to export them with all the information saved by clicking in the “view more options” button (represented with three points).



(Fig. 10 – Exporting Postman collection)

The button “export” will appear in the list, then just clicking it and follow the normal path as whatever export procedure, taking the recommended and most recent collection version of Postman JSON file. Finally, set the collection file name.

2.5 Examples of collections of services

How the previous collection looks like? and why those steps were followed?. Well, as simple as Postman is a quick software that allows us to obtain a JSON representation of a web service definition, while, is true that if someone wants to manually write the JSON representation, The MAS platform will perfectly work, but it will be more susceptible to writing errors. The collection will be divided into five parts, this TFM version only focuses on the first two: Full definition of the example used at [25].

The collection info and the items itself corresponding to the set of services defined.

- **Collection Info:** The first info part correspond just to the name of the collection, the schema of Postman that follows when exported, and an identity number. Just to track the API that will be processed by the Multi-Agent System.

```
"info": {
  "_postman_id": "48396593-3acf-44c3-8d6a-9f6d08e468e1",
  "name": "HotelsAPI",
  "schema": "https://schema.getpostman.com/json/collection/v2.1.0/collection.json"
},
```

(Fig. 11 – Postman collection basic info)

- **Collection items:** Secondly, the list of items or services that the collection has, in this case, all the previously mentioned. Basically, each item is conformed by three properties, the name, the request definition, and the response definition. Is important to notice that, those values should not be empty or null in order to maintain the full functionality of our Multi-Agent System.

```
"item": [
  {
    "name": "SearchHotel",
    "request": {
    },
    "response": [
    ]
  },
  {
    "name": "GetHotelDetails",
    "request": {
    },
    "response": [
    ]
  },
  {
    "name": "GetMetadata",
    "request": {
    },
    "response": [
    ]
  },
]
```

(Fig. 12 – Postman collection items or services definition)

Looking inside the request definition in the first item search hotel sample, all the definition needed to make the request is written, which method, what values of headers, the URL information such as protocol, host, path, and query if needed, etc.

```

"request": {
  "method": "GET",
  "header": [
    {
      "key": "x-rapidapi-host",
      "value": "hotels4.p.rapidapi.com",
      "type": "text"
    },
    {
      "key": "x-rapidapi-key",
      "value": "a82156eb53msh83ccd3c830f58f5p10d463jsn6c5c06889085",
      "type": "text"
    }
  ],
  "url": {
    "raw": "https://hotels4.p.rapidapi.com/locations/search?locale=es_ES&query=madrid",
    "protocol": "https",
    "host": [
      "hotels4",
      "p",
      "rapidapi",
      "com"
    ],
    "path": [
      "locations",
      "search"
    ],
    "query": [
      {
        "key": "locale",
        "value": "es_ES"
      },
      {
        "key": "query",
        "value": "madrid"
      }
    ]
  }
},

```

(Fig. 13 – Postman collection item request structure definition)

Then, the response structure contains how the response was saved, including the original request that is a copy of the previous illustration inside one property in the response, the response status, code, and the body the most important field in the response structure, due to that field will be used by the Multi-Agent system analyzing how is the response raw data in order to create the POJO models.

```

"response": [
  {
    "name": "SearchHotelResponse",
    "originalRequest": { },
    },
    "status": "OK",
    "code": 200,
    "_postman_previewlanguage": "json",
    "header": [ ],
    ],
    "cookie": [],
    "body": "{\n  \"term\": \"guadalajara\", \n  \"moresuggestions\": 981, \n  \"autoSuggestInstance\": null, \n
}
]

```

(Fig. 14 – Postman collection item response structure definition)

It is worth mentioning that, those does not all the params that a request and the response can have there are many other depending on how the request is structured which request method, if it is by post what kind of body data will have, etc. To have more information about it please visit [9, 11]. On this occasion, the previous illustration works as a first look.

Chapter 3: Architectural and Design pattern

The purpose of this chapter is to explain how the patterns have to be used, in order to have a clear, functional, and well-structured java android application. Besides, give some definitions regarding the patterns classification, explaining just the basic concepts, trying to provide you with enough knowledge to understand their structure. There will be three main patterns, first of all, one called VIPER architectural pattern, secondly, the DAO pattern, and finally the service mobile architecture. In order to set, a basic idea of all the files that will be needed in the Multi-Agent System. So, if you have enough knowledge, especially, in the usage of VIPER, DAO patterns, and mobile architecture you could move on to the next chapter.

Architecture and design are a fundamental aspect of the software development life-cycle. In software engineering, architectural pattern expresses a fundamental structural organization schema for a software system. It provides a set of predefined subsystems, specifies their responsibilities, and includes rules and guidelines for organizing the relationships between them. While a design pattern is a general repeatable solution to a commonly occurring problem in software design. A design pattern is not a finished design that can be transformed directly into code. It is a description or template for how to solve a problem that can be used in many different situations. Architectural and design patterns can speed up the development life-cycle process by providing tested, proven development paradigms. Effective software design requires considering issues that may not become visible until later in the implementation.

Reusing design patterns helps us to prevent subtle issues that can cause major problems and improves code readability for coders and architects familiar with the pattern. Often, people only understand how to apply certain software design techniques to certain problems. These techniques are difficult to apply to a broader range of problems. Design patterns provide general solutions, documented in a format that does not require specific tied to a particular problem. In addition, patterns allow developers to communicate using well-known, well-understood names for software interactions. Common design patterns can be improved over time, making them more robust than ad-hoc designs.

Therefore, the purpose of this chapter, beyond the explanation of what is it and how am architectural and design pattern works, it is about the implementation of a Multi-Agent System that would be capable to generate code of a certain kind. Thus, achieving the main goals and objectives of this TFM work.

3.1 Why pattern should be used?

Patterns are commonly defined as time-tested solutions to recurring design problems. Patterns are used to represent some of the best practices adopted by experienced object-oriented software developers, describing the problem, the solution, when to apply the solution, and its consequences. It also gives implementation hints and examples.

Over 26 years ago the iconic computer science “**Design patterns: Elements of Reusable Object-Oriented Software**” was the first published and still continues to be the best. The Four authors of the book: Eric Gamma, Richard Helm, Ralph Jonson, and John Vlissides have been dubbed: The Gang of Four. In technology circles, there will appear these nicknames shorted to GoF. [5].

The subsystems of software architecture, as well as the relationship between them, usually consists of several smaller architectural units. Those described by using design patterns. [6] Having an impact on the architecture of its subsystems. For this reason, the selection of an architectural pattern is a fundamental design decision when developing software.

Christopher Alexander is known for many books on the design and building process, is regarded as the father of the pattern language movement. He says, “**Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice**”. [7]

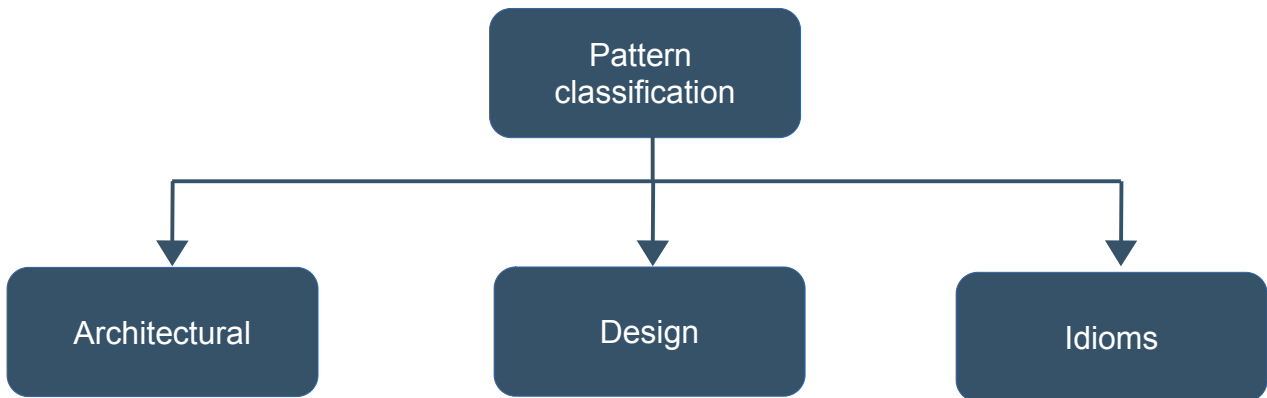
An important person in the software engineering world, which influenced the development of the term “**agile software development**”. Keeping all this in mind, it seems, like almost everything, that using design patterns have advantages and disadvantages.

Advantages	Disadvantages
They help to “start” in the design of a complex program.	Beginner engineers may not understand them.
They give an initial “well thought out” decomposition of objects. Designed to make the program scalable and easy to maintain.	Often times used improperly without a realistic understanding of how the software is likely to change.
They help reuse techniques on how to build, design, maintain, and write code.	Can add memory and processing overhead, sometimes not appropriate for applications such as low-level systems programming or certain embedded systems.
Many people know them and already know how to apply them.	
There is a high level of abstraction.	
The design can be applied to different situations.	

(Tab. 1 – Advantages and disadvantages of using design patterns)

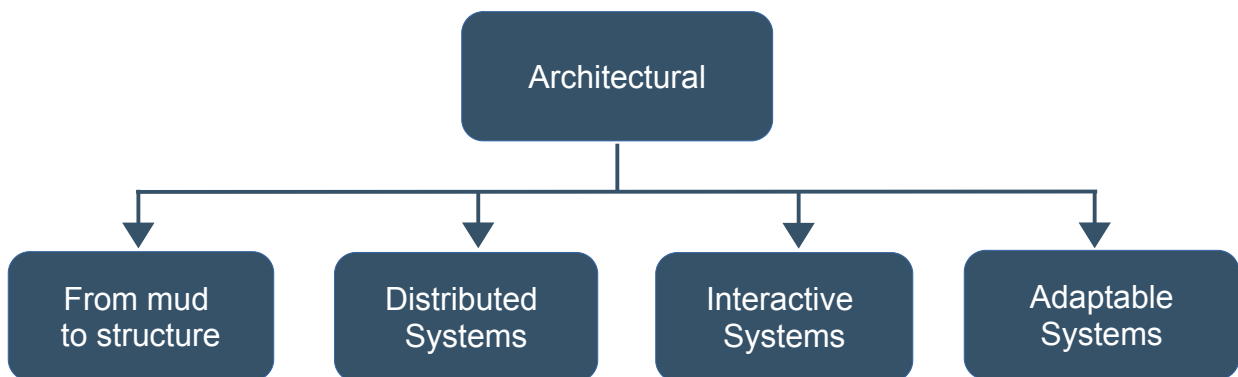
3.2 Pattern classification

Investigating among some books specially dedicated to design patterns [1, 5, 6] and on the internet. There are three categories, each one consists of patterns having a similar range of scale or abstraction.



(Fig. 15 – Patterns classification)

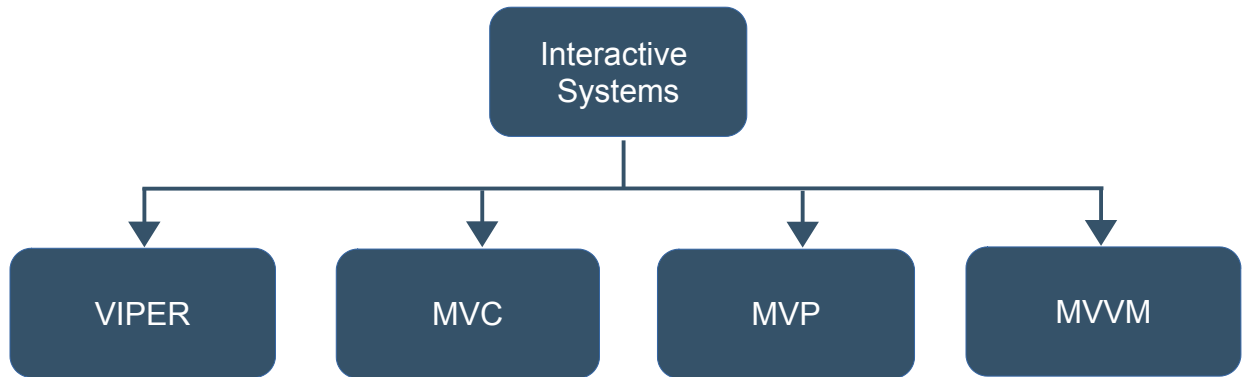
- **Architectural classification:** The architectural design patterns as its name suggests, deal with the architecture of the application or module. They represent the highest-level pattern in our pattern system. Helping to specify the fundamental structure of an application. The communication and collaboration between different parts of the system and its extensions. Each architectural pattern helps developers to achieve a specific goal system property, such as the composition, the adaptability, the interactions, and the distribution of the entire system. Giving way to four types.



(Fig. 16 – Architectural pattern sorts)

Now, going deeper, looking specifically in the sort of interactive systems, with the objective to enhance the usability of an application. Usable software systems provide

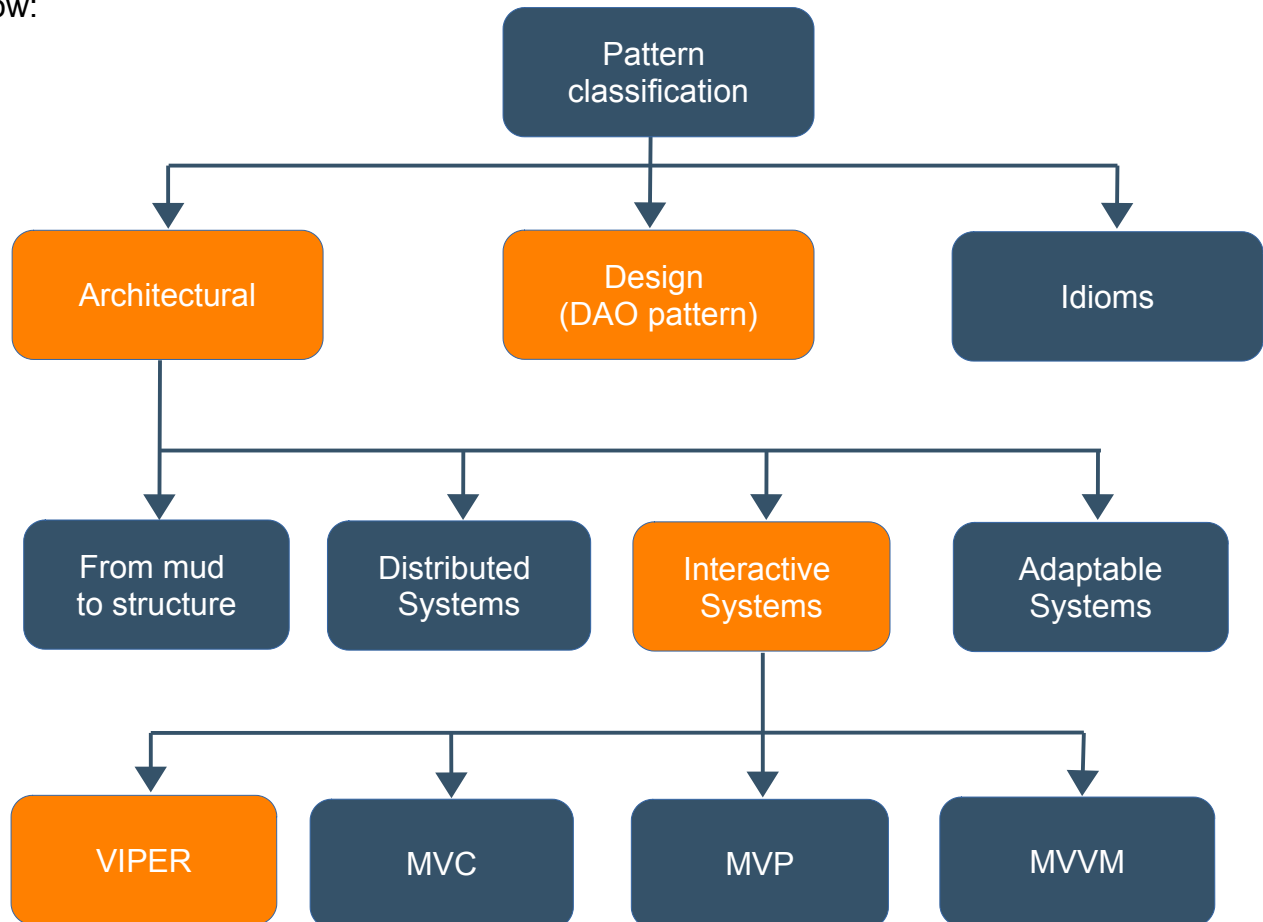
convenient access to their services and therefore allow users to learn the application and produce results quickly. When specifying the architecture of such systems, the challenge is to keep the functional core independent of the user interface. The core of interactive systems is based on the functional requirements for the systems and usually remains stable. It is not the case of the user interfaces (UI) that are often subject to change and adaptation. Some of the most frequent patterns that are used in the mobile development context are the following:



(Fig. 17 – Interactive architecture patterns)

- ◆ **View Interactor Presenter Entity and Router (VIPER):** This design pattern implements the separation of concern paradigm and makes the application structure more modular.
- ◆ **Model View Controller (MVC):** It divides the application into three separate entities: Data models, view or user interface (UI), and the controller.
- ◆ **Model View Presenter (MVP):** Unlike MVC, all the presentation logic moves to the presenter.
- ◆ **Model View View-Model (MVVM):** A view model is responsible for exposing the data objects from the model in such a way that objects are easily managed and presented.
- **Design patterns classification:** Design patterns are medium-scale patterns, they are smaller in scale than architectural patterns, but are at a higher level than the programming language-specific idioms. The application of a design pattern has no effect on the fundamental structure of a software system but may have a strong influence on the architecture of a subsystem [6].
- **Idiom classification:** Idioms are low-level patterns specific to a programming language. An idiom describes how to implement particular aspects of components or the relationship between them with the features of the given language [6].

Having the full clear picture of architectural and design patterns, the next path is the result to follow:



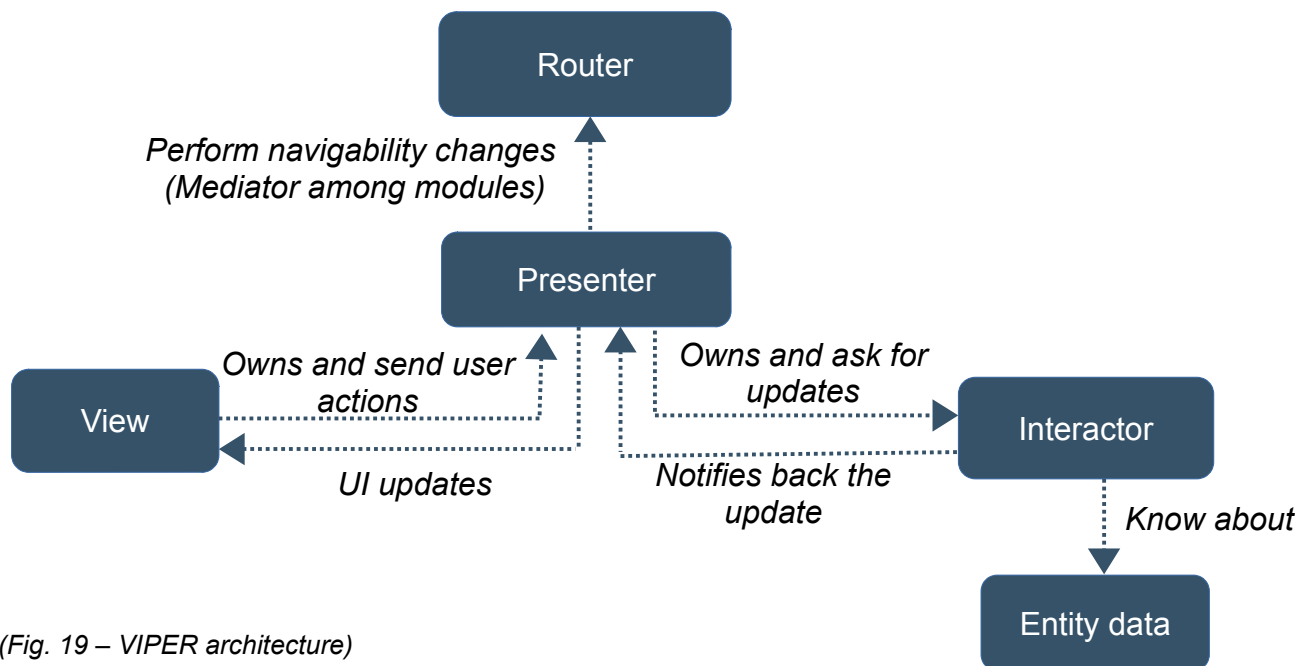
(Fig. 18 – Architectural and design pattern path to VIPER)

3.3 How VIPER works?

VIPER is an architectural interactive pattern technique that minimizes code duplication, prevents high coupling, and standardize a common way of writing code that provides a general solution for the recurring situation when developing mobile applications. It implements “**separation of concern**” paradigm. Mostly like MVP or MVC, it follows a modular approach. One feature, one module. For each module, VIPER has five (sometimes four depending on if that specific module requires the Interactor part, in other words, that module does not need API calls) different classes with distinct roles. No class goes beyond its sole purpose. These classes are the following:

- **View:** A class that has all the code to show the app interface to the user and get their responses. Upon receiving a response view alerts the presenter.

- **Presenter:** The nucleus of a module. It gets user response from the view and works accordingly. The only class to communicate with all the other components. Calls the router for wire-framing, Interactor to fetch data (network calls or local data calls), view to update the UI.
- **Interactor:** Has the business logic of an app. Primarily make API calls to fetch data from a source. Responsible for making data calls but not necessarily from itself.
- **Router:** Does the wire-framing or navigability of the application. Listens from the presenter about which screen to present and executes that.
- **Entity:** Contains plain model classes used by the interactor.



(Fig. 19 – VIPER architecture)

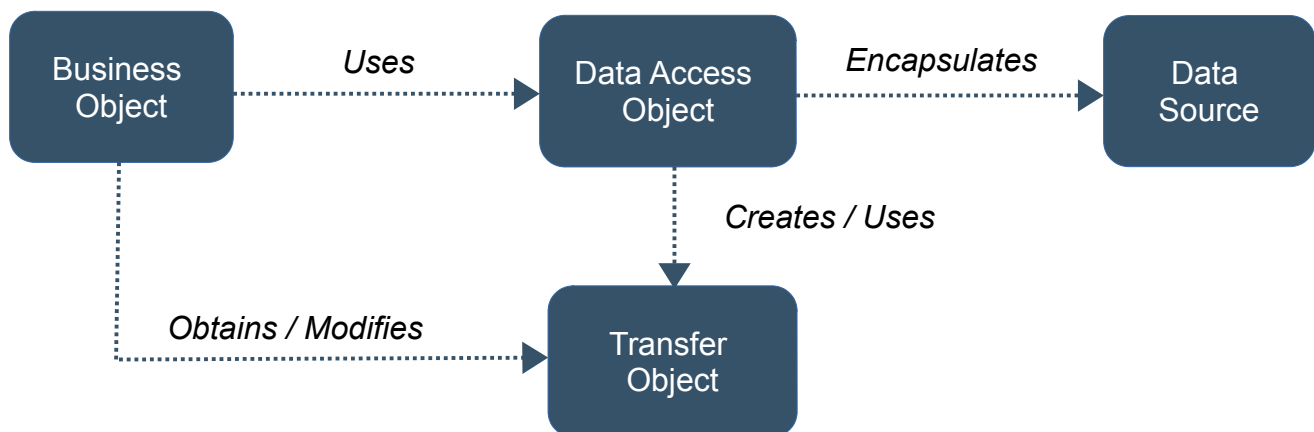
The purpose of this TFM work is focused on both “***the Interactor***” and “***the Entity data***”. On the one hand, the entity data is just the raw representation that the module needs in order to satisfy the user’s actions and his interactions. In mobile applications, the service response will become the entity data. The class definition with their properties and encapsulation methods.

On the other hand, the interactor is going to be the interface with the service method definition using one design pattern called Data Access Object (DAO).

3.4 Data Access Object (DAO)

This design pattern aims to avoid direct dependence situations between the application code and the data access code. The patterns supply an interface of data access to the business and presentation components, providing and independence of data source and APIs of storage persistence used, contribute to the portability of the systems's components and facilitate the migration to different data sources [8].

Then, there is the DAO pattern interface on one side, but the interface should have an implementation side that it will provide the access mechanism required to work with the data source. The original structure of the DAO pattern is the following:



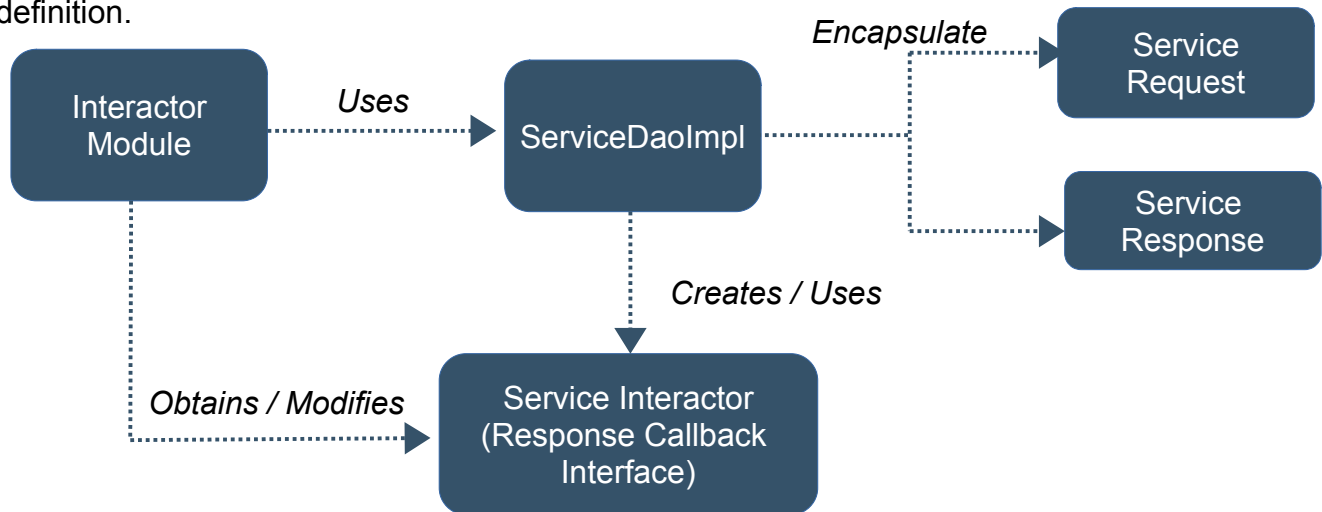
(Fig. 20 – Data Access Object (DAO) structure diagram)

As it is noticed, the data access object encapsulates the data source, enabling transparent access to the data source. In this case, the business object represents the data client. It is the object that requires access to the data source to obtain. On the other hand, the transfer object represents the object that will be used as a data carrier.

Now, let us move to our purpose, applying this pattern inside the Interactor module. The interactor handles the communication between the app (the client) and a server (API calls). Then, the interactor module is identified as the business object that requires access to a certain API call. Besides, the transfer object is the class that will contain the final raw data

model, in our TFM work it is going to be called the interactor interface as a callback to carry the response resultant. The service request is going to be considered as the data source, where the interactor needs information.

As a result, the following structure will be the first five code files needed by each service definition.



(Fig. 21 – Data Access Object (DAO) applied in VIPER perspective)

3.5 Example of a service implementation using VIPER and Data Access Object

Imagine that there is a company that requests us to build a mobile application to search and display hotels in a certain location, as the previous chapter an API where all the web services are listed is needed for that app. If you skipped the second chapter, it seems that you already know about web services and how to represent them in Postman or whatever another context.

Well, keep it simple as the objective is the separation of concerns, so this application will start with a search screen to search hotels once the search finished the data should be displayed inside the map. This is one of the features required. Then VIPER enters the scene this VIPER module will handle that exact functionality. The communication to the server is the call request to search hotels the parameters to search the hotel is the locale code, and the query will refer to the city that the user wants to search.

- **VIPER Interactor Response Callback Interface:** [Appendix B]. The VIPER Interactor interface then will be composed of whether the service call is successful or it fails. As a consequence, if the call is success the data model will be provided if not an error

occurs. Later on, when a user action requires a hotel search this action will be carried until the interactor module that will use the DAO implementation pattern in order to get the service response callback through the interface.

- **DAO Interface:** *[Appendix C]*. The DAO interface will have access methods whereby the request will be launched asking the data source (final API server) for the data needed. So, the request params should be sent in addition to the service interactor callback that will work as a listener whether to succeed or fail.
- **Service Request:** *[Appendix D]*. The request will be a class that will extend all the functionality to make a request. As this request does not require other kinds of parameters more than URL params an empty request class is enough. Detail information regarding the service request (Functionality inside the service request master class) will be explained in the service mobile architecture section.
- **Service Response:** *[Appendix E]*. The response will be a class where all the raw data in the response body definition, that is all the POJO models, and properties, for example, the list of hotels resulted by the API. Commonly, following by status code or an error message. (This is not the case). This API has five properties the important one is the list of suggestions, that will contain the hotel information.
- **DAO Implementation:** *[Appendix F]*. Now, all the little pieces are done, the implementation is needed to allow the interactor module to access the response data model. Due to it is the most complicated to understand, let divide the class into four pieces:
 - ◆ **Class definition and interface implementation:** As Figure 20 shows, the DAO provides the client with an access method to request or use some data, hence this class will implement the DAO interface previously created. With this, the class will be forced to add the implementation of such a method. It is noticed, giving us the opportunity to interact with the params sent by the user and with the interactor callback to acts accordingly. Secondly, the implementation of Retrofit callback this method will provide us with the response data or with an error. More details about it in the following topics and at [12].
 - ◆ **Properties or variables:** Those properties needed in the implementation, the interactor listener to sent back the response status, and the request to be built.
 - ◆ **DAO method implementation:** The DAO interface method to implement, receiving the request parameters as the user action, in addition to the client interface that will return the data (interactor callback). Inside the method will be the code that refers to the API call sending the headers, request parameters, and all that Retrofit needs in the request [12].

- ◆ **Retrofit callback:** Retrofit callback methods that are whether the call results in a successful API communication or it fails. If the response succeeds the object response will be auto-filled with the data structure provided in the response data model. However, the response could result in a successful server communication but it does not result in a correct response format. Then, the code should handle whether if the body response is correct and give back a response to the interactor client listener.

All the pieces are armed, thereupon, the VIPER flow can act, the user action requires to searching for hotels located in Madrid, Spain, those are the URL params. VIPER will pass the parameters from the view module to the presenter, and the presenter will order the Interactor module to make a service request using the DAO Implementation method.

- **VIPER Interactor Client Module:** *[Appendix G]*. Once the presenter ordered the interactor what service call, the interactor should implement the interactor interface that was first declared. Imagine that this module is launched when the splash screen appears. “*The SplashInteractorInput*” interface is just the data access method that the entire module will use as input protocol. In our example, a simple interface with a service call named searchHotels with the parameters that will be carried from the view side. After, the interface method should be implemented (There are many ways of doing it). One simple form is by implementing a thread task where the Dao method will be called sending the parameters and the instance of the interactor callback. Finally, the listeners of the interactor interface should be implemented in order to catch whether the service was successful or failure. Sending back the corresponding signals to the presenter module to take the pertinent actions.

That procedure is repeated by each service, and the services are grouped by VIPER modules. To fulfill the complete understanding of VIPER and the DAO pattern and how a Multi-Agent System could be easily implemented to follow the rules and obtain in an automated, quick and clear way, saving developing time, human effort, and writing errors in the backend side.

3.6 Service mobile architecture

Until now, we have seen the way that the VIPER pattern is used together with the DAO pattern, in order to build the different files that will communicate the user interaction to the server side by each VIPER module. However, in order to reduce the code that is write in each DAO implementation part, a service architecture was built to improve the architectural and design pattern. Five additional files are needed to accomplish this approach.

- **API (Retrofit Interface):** *[Appendix H]*. This class will have all the services exposed through the API as an abstract method which ones are going to be defined, with the request type, headers, and parameters needed whether as request body or field parameters, and the response data model that will return.
- **Service Backend:** *[Appendix I]*. This class will have the URL definition, that are the following properties:

- ◆ **Protocol:** Usually http or https.
- ◆ **Domain:** The namespace used as the URL of the API.
- ◆ **Port:** If explicit a port number such as 443 for https, or 8080 for http.

As a default backend endpoint, only a development (DEV) environment is considered. Due to a mobile application could have many of them, such as production, test or beta environment, this Multi-Agent system application only uses one of them to avoid handling many URL or endpoints. It will be composed of three parts: the variables needed previously mentioned.

- ◆ **Constructor:** Receiving the environment to be used.
- ◆ **Encapsulation method:** That will return the full URL to point the request.
- **Service Error:** *[Appendix J]*. This class will handle all the errors in the application, normally know as a POJO class, with the following properties:
 - ◆ **Code:** Defined by the developer to represent a specific error.

- ◆ **Msg:** Defined by the developer to represent a specific message.

It will be composed by four parts: the variables needed previously mentioned.

- ◆ **Constructor:** Receiving a string that will be the message to be shown as an error.
- ◆ **Encapsulation method:** It will give accessibility to the properties.
- ◆ **To String method:** That will be used to pretty printing the class when debugging.

➤ **Service Configuration:** *[Appendix K]*. This class will handle the different backend endpoints available, in order to provide the developer the way of adding more than the default environment this class is needed to handle which one of the endpoints the service request will use to communicate and obtain some response. It will be composed of four parts:

- ◆ **Singleton instance:** A simple design pattern that handles the initialization of a class, verifying if the object is already instantiated will return that object, as a result, saving memory leaks.
- ◆ **Backend variable:** A variable of the Backend class to initialize the environment to be used.
- ◆ **Setup method:** The method that initializes the variable.
- ◆ **Encapsulation method:** It will give accessibility to the properties.

➤ **Service Request:** *[Appendix L]*. This class will handle the Retrofit platform initialization using the API interface of services, obtaining the service backend to be used to point the request and if needed the initialization of a JSON client that the Retrofit platform will use to pretty-print the request and response models. It will be composed by four parts:

- ◆ **Api variable:** A variable of the API interface class to initialize the API interface.
- ◆ **Constructor:** Call the setup method to initialize all the stuff.
- ◆ **Setup method:** Call the service configuration setup to initialize the backend and the environment to be used, following by the Json client and the Retrofit object, creating

in this way the API interface variable, as a result, each service request extends this master class, and it has access to the API variable to call its respective service.

- ◆ **Encapsulation method:** It will give accessibility to the properties.

Chapter 4: Multi-Agent Systems

The purpose of this chapter is to introduce more details regarding Multi-Agent Systems (MAS), how the JADE platform works (the one used to build the TFM work). Proving some basic definitions from a simple agent, its behaviour in a certain environment where other agents may be or not collaborating, sharing resources in order to achieve their goals. In fact, more than one agent can share the same goal, at different levels such as writing a code file, the task can be shared among agents, so that, they should collaborate to avoid overwriting the file.

Agents are considered one of the most important paradigms that on the one hand may improve on current methods for conceptualizing designing and implementing software systems, and on the other hand, may be the solution to the legacy software integration problem. Hence, take advantage of this AI paradigm in order to improve mobile application development. To develop MAS requires addressing a diverse range of complex challenges such as coordination among agents, learning, and security.

A multi-agent system may be seen as a collection of deliberative, autonomous, collaborative agents, that may communicate and cooperate with each other to solve complex distributed problems. Therefore, problems can be solved quickly, due to the inherent concurrency or parallelism. Agents working at the same time in different parts of the problem.

4.1 Agents, behaviours and architectures

Agents consist of autonomous entities, they collaboratively solve tasks yet they offer more flexibility due to their inherent ability to learn and make autonomous decisions. Agents use their interactions with neighboring agents or with the environment to learn new contexts and actions. Subsequently, agents use their knowledge to decide and perform an action on the environment to solve their allocated task. It is this flexibility that makes MAS suited to solve

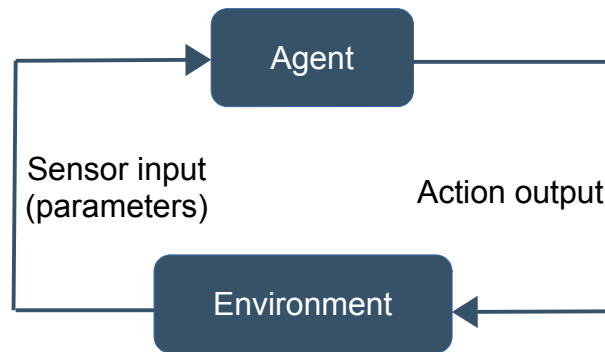
problems in a variety of disciplines including computer science. There are many definitions of what an agent and its behaviour are shown in [13,14, 15], presenting different perspectives resulting from diverse application-specific features of agents. Then, generalizing a concrete definition considering the fundamental abilities and features of agents:

- **Agent:** An agent is an entity that is placed in an environment and senses different parameters that are used to make a decision based on the goal of the entity. The entity performs the necessary action on the environment based on this decision. There are some keywords that can be further elaborated.
 - ◆ **Entity:** Refers to the type of agent. An agent can be software such as daemon security agents, a hardware component such as a thermostat, or a combination of both, a robot.
 - ◆ **Environment:** This refers to the place where the agent is located. The environment can be a network in the case of traffic monitoring agents, software when the agent is monitoring the actions of software components, etc. An agent uses the information sensed from the environment for decision making. The environment has multiple features that affect the complexity of an agent-based system.
 - **Accessibility:** This refers to the accuracy with which agents can sense data from the environment. That means the data is up to date without noisy and / or incomplete data.
 - **Determinism:** Refers to the predictability of the results of an action. That means the results are predictable. Agents precisely know the next state for each action.
 - **Dynamism:** This refers to the changes that occur inside the environment that are independent of the actions taken by the agents. If the changes can only occur as a consequence of the action of the agents is considered to be static, otherwise, it will be dynamic. Agents must detect a change in the state of the environment and update the sensed information.

- **Continuity:** A continuous environment affects the agent's state through a continuous function such as an agent that moves within a physical environment. While a discrete environment constrains the agent to enter into a set of predetermined states.

- ◆ **Parameters:** The different types of data that an agent can sense from the environment. For instance, the web service information will be spread among the agents.

- ◆ **Actions:** Each agent can perform an action that results in some changes in the environment. Those actions can be a set of continuous or discrete actions. That is performing unlimited actions or finite actions correspondingly.



(Fig. 22 – Agent in its environment)

➤ **Behaviours:** Is the way that an agent acts in a given situation or set of situations. The situation is defined by the environmental conditions, its own circumstances, and the knowledge the agent currently has available to it. If the agent has insufficient knowledge for a given situation, then it may choose to search for further knowledge about the situation. The distinction between an action and a particular behaviour is that a behaviour comprises one or more actions performed by an agent in a particular order or manner. Firstly, it allows an agent's behaviour to adapt to the behaviour of the agents with which it interacts. Knowing the plans of other agents, agents can better plan their actions, whether in competitive or cooperative settings. Secondly, it allows users to understand and study agent's behaviour in a multi-agent system of interest, determining its strengths and weaknesses. This can be used to improve the behaviour of the system.



(Fig. 23 – Agent behaviour visualization)

- **Architectures:** Agent architectures are fundamental mechanisms underlying the autonomous components that support effective behaviour in real-world, dynamic and open environments. The purpose is to offer a particular methodology for building an autonomous agent. How the construction of the agent can be decomposed into the construction of a set of component modules and how these modules should be made to interact. There are mainly three kinds of architecture approaches:
- ◆ **Reactive:** Focused on fast reactions/responses to changes detected in the environment. This has at most a very simple internal representation of the world and a tight coupling of perception and action. Intelligence is a product of the interaction between an agent and its environment. The characteristics that distinguish this architecture are the following:
 - **Emergent functionality:** Simple agents with simple interactions. Complex behaviour patterns appear as a result of dynamic interactions. The global behaviour of the system is not specified a priori.
 - **Task decomposition:** Agents composed of autonomous models, each module manages a given task sensor, control, computations, etc. Minimal low-level

communication between modules. There is not any world global model or any planning or controller agent.

- **Raw data:** Basic data without complex symbolic management of data.
- ◆ **Deliberative:** Focused on long term planning of actions, centered on a set of basic goals. Based on the explicit symbolic model of the world. Decisions are made via logical reasoning, based on pattern matching and symbolic manipulation. The sense-plan-act problem-solving paradigm of classical AI planning systems. Hence the characteristics are:
 - **Beliefs:** Agent's view of the environment or world.
 - **Plans:** Sequences of actions that are needed to achieve the intentions, given the agent's beliefs.
 - **Desires:** Follow from the beliefs, can be unrealistic and inconsistent.
 - **Goals:** A subset of the desires. Realistic and consistent. Determining the potential processing.
 - **Intentions:** A subset of the goals. A goal becomes an intention when an agent decides to commit to it.
- ◆ **Hybrid:** Combine both reactive and deliberative approaches. An agent out of at least two subsystems: A deliberative one, containing a symbolic world model, which develops plans and makes decisions in the way proposed by symbolic AI. In addition to a reactive one, which is capable of reacting quickly to events without complex reasoning.

4.2 Multi-Agent systems for distributed problem solving

As it was mentioned, agents work together to solve problems that require collective effort. To do so, the systems require coherence and competence. In other words, the agents want to

work together and need to know how to work together. As our main objective of using MAS to solve the generation of web services, java files needed in the Interactor module.

There are four steps to accomplish a distributed problem:

- **Task decomposition:** Divide a complex problem into a set of tasks into subproblems, decomposing recursively if necessary, large tasks into subtasks that can be tackled by different agents. Tasks can be totally independent or can have some relations. This decomposition may be done by the user, by a central coordinator agent or can be negotiated at run time by all the agents of the system.
- **Task allocation:** Assign the subproblems to different agents, an easy case is that only one agent capable of solving each subproblem with no flexibility. On the other side, the same subproblem can be solved by different kinds of agents. Dynamic mechanisms of run time task allocation. Even, having the possibility of recovering from individual failures.
- **Task accomplishment:** Each agent solves (on its own) its assigned subproblems. This phase can be done, in principle, without communication if the problems are independent. There may be dependencies between subproblems, as a result, potential conflicts may appear such as two sub-solutions are incompatible and / or conflicts in the use of shared resources.
- **Result synthesis:** Put together the results of all agents to find the complete solution, therefore, which agent will be responsible to make it?, and how it is made?. If each subproblem has a unique solution, it is a relatively easy step, otherwise, there may be need of conflict detect, task re-assignment.

4.3 Agent communication mechanisms

Agents require communication, in order to coordinate their actions and facilitate this by allowing individual agents to interact. Cooperation and information sharing is allowed. In

particular, agents interact with each other by using some special communication languages, called agent communication languages.

The basic options used in MAS are the blackboard systems and the direct message passing.

- **Message passing:** In this method agents directly send messages to each other. The agents use point to point or broadcast communication to talk to other agents. The nature of this information can be varied. Speech acts provide one way to describe this variety in [17]. There are many types of speech act:

Inform: Other agents about some data.

Query: Others about their current situation.

Answer: Questions.

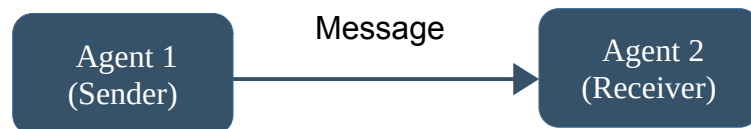
Request: Others to act.

Promise: To do something.

Offer: Deals.

Acknowledge: offers and request

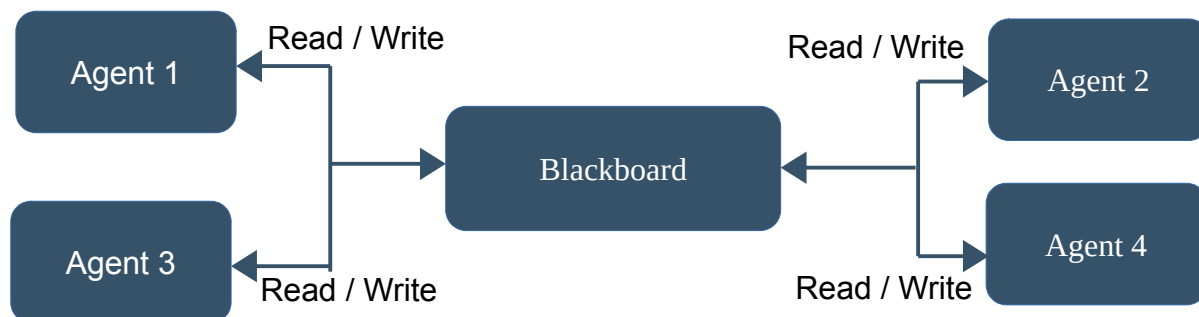
The FIPA-ACL message (Foundation for Intelligent Physical Agents-Agent Communication Language) is a standard that allows the implementation. Later on, this topic is going to be detailed in the following topics.



(Fig. 24 – Direct passing message)

- **Blackboard:** The blackboard is a global shared data structure. Every result an agent produces is stored on the blackboard. Using as a central repository each agent stores its data on the common blackboard that is readable by other agents. Sharing the results

and / or tasks solving subtasks. Providing flexible mechanisms that allow cooperation among agents.



(Fig. 25 – Blackboard indirect communication)

4.4 Voting protocol

This protocol is a mechanism which chooses the outcome of a negotiation based on the inputs (votes) given by all agents to a set of competing options. It is assumed that voters are truthful and agents will vote for the candidate that they consider the best option according to the problem to solve. The purpose of using a voting protocol is to set a competitive environment among agents, in this way, in the end, obtain a rating list of the best performance from the best to the worst agent. The metrics to be used will depend on every system.

There are many kinds of voting protocols, in this case, just the simple protocols will be mentioned. Due to, one of them was used in the implementation of this TFM work. All the protocols are problematic in one sense or another.

- **Plurality:** Each agent can give one vote to one of the alternatives, the alternative with the highest number of votes wins. The most simple voting mechanism efficient from the computation point of view.
- **Anti-plurality:** Each vote gives a negative vote to the alternative the agent considers the worst, the alternative with the fewer votes wins.

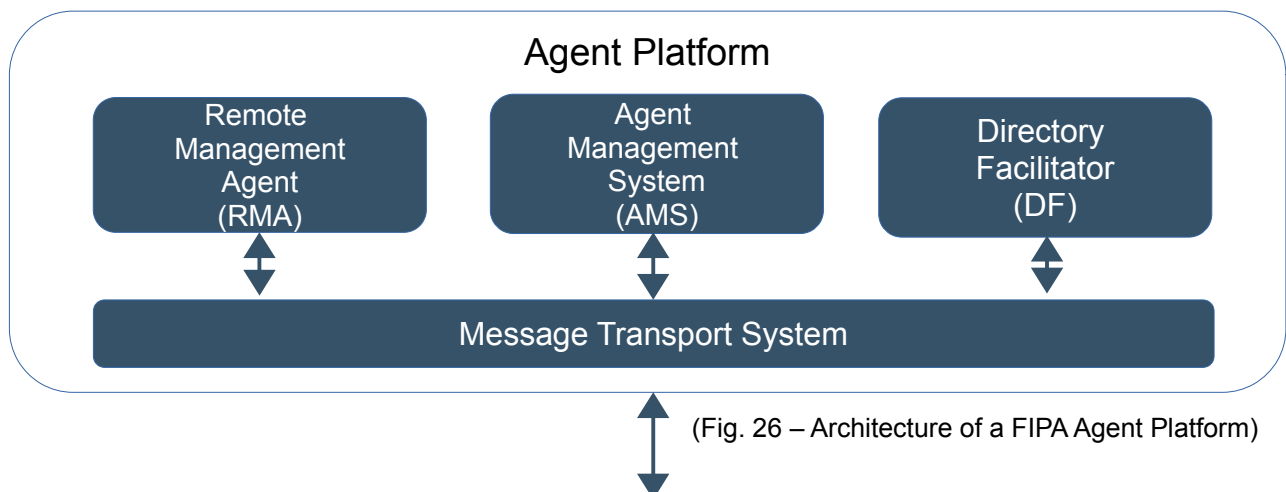
- **Best-worst:** Each agent gives a positive vote to his best alternative and a negative vote to his worst alternative. Hence, each alternative receives $\alpha > 0$ points for each positive vote and $-\beta < 0$ points for each negative vote. The alternative with more points wins. Following the operation $\alpha - \beta$.
- **Approval:** Each voter selects a subset of the candidates, the candidate with most votes wins k-approval voting.

$K = 1$: Plurality

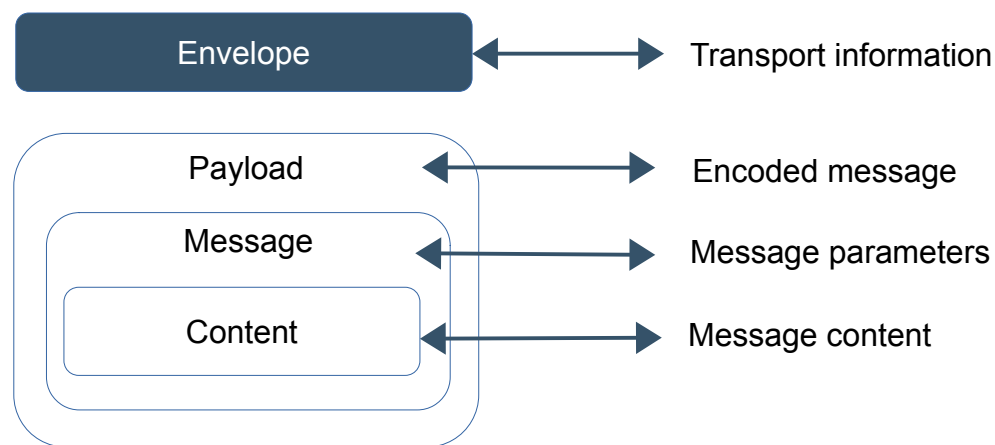
$K = n-1$: Anti-plurality

4.5 JADE platform

The Java Agent Development Framework (JADE) is a software development framework aimed at developing multi-agent systems and applications conforming to the Foundation for Intelligent Physical Agents (FIPA) standards at [19] for intelligent agents. It includes two main products: a FIPA-compliant agent platform and package to develop java agents. JADE has been fully coded in java and agent programmer, to exploit the framework. Therefore, the only software requirements to execute JADE is the Java Run Time Environment (JRE) available at [21]. There are many specifications of the FIPA standards inside the JADE platform, at least two of them are important, on one hand, the **“Agent Management specification”** (FIPA no. 23) and the **“Agent Communication Language”** (FIPA no. 61). The agent platform and its standard model as defined by FIPA, are represented in the following figure.



- **The Agent Management System (AMS):** Is the agent who exerts supervisory control over access to and use of the Agent platform. Only one AMS will exist in a single platform. The AMS provides white-page and life-cycle services, maintaining a directory of agent identifiers (AID) and agent state. Each agent must register with an AMS in order to get a valid AID.
- **The Directory facilitator (DF):** Is the agent who provides the default yellow page service in the platform.
- **The Message Transport System:** Also called Agent Communication Channel (ACC), is the software component controlling all the exchange of messages within the platform, including messages to/from remote platforms.



(Fig. 27 – FIPA message structure)

JADE fully complies with this reference architecture and when a JADE platform is launched the AMS and DF are immediately created and the ACC module is set to allow message communication. The agent platform can be split into several hosts. Only one java application, and therefore only Java Virtual Machine (JVM), is executed on each host. Each JVM is a basic container of agents that provides a complete run time environment for agent execution and allows several agents to concurrently execute on the same host. The main-container, or front-end, is the agent container where the AMS and DF lives and where the RMI registry,

which is used internally by JADE, is created. The other agent containers, instead, connect to the main container and provide a complete run-time environment for the execution of any set of JADE agents.

A FIPA-ACL (Foundation for Intelligent Physical Agent-Agent Communication Language) message contains a set of one or more message parameters. Precisely which parameters are needed for effective agent communication will vary according to the situation; the only parameter that is mandatory in all ACL messages is performative. Although it is expected that most ACL messages will also contain sender, receiver and content parameter. The FIPA-ACL message parameter is shown in the table below, without regard to specific encodings.

Parameter	Description
Performative	Type of the communicative act of the message.
Sender	Identity of the sender of the message.
Receiver	Identity of the intended recipients of the message.
Reply-to	Which agent to direct subsequent message to within a conversation thread.
Content	Content of the message.
Language	Language in which the content parameter is expressed.
Encoding	Specific encoding of the message content.
Ontology	Reference to an ontology to give meaning to symbols in the message content.
Protocol	An interaction protocol is used to structure a conversation.
Conversation-Id	The unique identity of a conversation thread.
Reply-with	An expression to be used by a responding agent to identify the message.
In-reply-to	Reference to an earlier action to which the message is a reply.
Reply-by	A time/day indicating when a reply should be received.

(Tab. 2 – FIPA-ACL message parameters and description)

In order to see more precisely how the exchange of message works among agents. The following figure shows each agent has a message queue (mailbox) where messages can be sent. When a message is placed in the mailbox, the agent is notified. It is the agent's responsibility when to read the message and how to react.



(Fig. 28 – FIPA-ACL workflow among agents)

Those are the main topics that are fundamental to know to understand the next chapter regarding how the JADE platform was used to create agents, exchanging messages to collaborate in the tasks assigned. However, there are more specifications and standards that the FIPA provides for the multi-agent systems. Technical details could be found at [14, 19, 20].

Chapter 5: Multi-Agent System to improve mobile application development

The purpose of this chapter is to fully explain the integration of all the technologies, software tools and different paradigms previously mentioned to achieve one platform which will take the Postman collection file as input, then the platform will split it into pieces, spreading through agent's branches (One branch by one service). Collaborating in their respective branches in order to produce as a result generated java code files to be used in a mobile application for each service defined in the JSON file. [Appendix A-L with G as exception].

In addition, the software tools that were used for the implementation as well as the software requirements. Then, I am going to introduce the architecture that was used to archive the global goal of generating each one of the files, each agent role and its corresponding behaviour, following by how each file was split into micro tasks that could be easily performed. The way agents communicate each other acting in the created hierarchy, creating competitiveness between branches due to each branch will be evaluated taking into account the execution time that each branch expends, additionally running a plurality vote protocol to obtain the faster branch and the most popular. Those branches will be rewarded with additional tasks to create the files at [Appendix H-L] know as the second round.

The starting point of the MAS is the JSON file of defined web services, it is going to be the input of the system, defined at [25]. To do so, a configuration file was defined, thinking regarding in future works or updates, adding more features, such as, selecting the platform (Android with Kotlin, or iOS with swift), the language which the files will be generated. Even, selecting other sorts of architectural patterns, not only VIPER, but also MVI that supports reactive development, MVP or MVVM.

By now, it contains just a little attributes, starting with the API Json file's name, the package's name of the mobile application (this field is required when building mobile applications). It is a string name that works as an identifier, and usually has the following format "com.companyname.appname", this works whether Google Store or Apple Store. Finally, a generic network error that will be used to be returned if something when wrong and Retrofit could not set an error. First of all, lets set up the software requirements that were used in this MAS project.

5.1 Software requirements

- The Multi-Agent system was implemented through Eclipse IDE Version: 2019-09 R (4.13.0) [22].
- Maven project structure with the version 3.6.3 [23].
- Java language 11.0.5 by Oracle Corporation [21].
- JADE platform maven dependency version 4.4.0 [24].

5.2 MAS file configuration

The Multi-Agent System should be capable of being configurable in such a way that allows scalability, in the sense of being able to achieve different goals in future updates. That is the reason for taking advantage of a configuration file based on the XML standard, with simple tags regarding the different settings needed in this first version.

Attribute	Description
<api></api>	The JSON format file name, when exported from Postman.
<package_name> </package_name>	The mobile package name, as mentioned commonly in the format: "com.companyname.appname" .
<response_generic_error></response_generic_error>	Text that will be returned from the service core, when something failures in the server communication.

(Tab. 3 – XML configuration file attributed)

As a visual example of how the XML configuration file looks like:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<Configuration>
  <api>HotelsAPI</api>
  <package_name>com.navi.skeleton</package_name>
  <response_generic_error>No se pudo conectar al servidor</response_generic_error>
</Configuration>
```

(Fig. 29 – XML configuration file for the HotelsAPI)

It is worth to mention, both files the configuration file and the JSON format should be placed into the resources folder of the TFM project. Then the MAS can easily read from the resources the configuration file, extracting the configuration attributes to start working as a Multi-Agent Application. Then let's start defining each agent sort and its role inside this MAS project.

5.3 Agent sorts and roles

The definition of an agent was already mentioned in the previous chapter, this TFM will describe three different sorts of agents by branch, in other words, to achieve the goal of generating java code files for API services, a hierarchy of agents is needed, each one of them handling different micro tasks and roles.

- **Manager Agent:** These agents will be the main handler of a service task, there will be one manager by each service defined in the JSON file. The role of a “Manager”, is to split the JSON service definition extracting the parameters needed such as service name, URL, headers, request parameters, request type, response parameters, etc. Consequently, a resource called “Service Configuration” is created [Section 5.5], together with all the POJO classes needed [Appendix A]. Creating and delegating to five workers that each manager should create, each one of them to work with one file of the

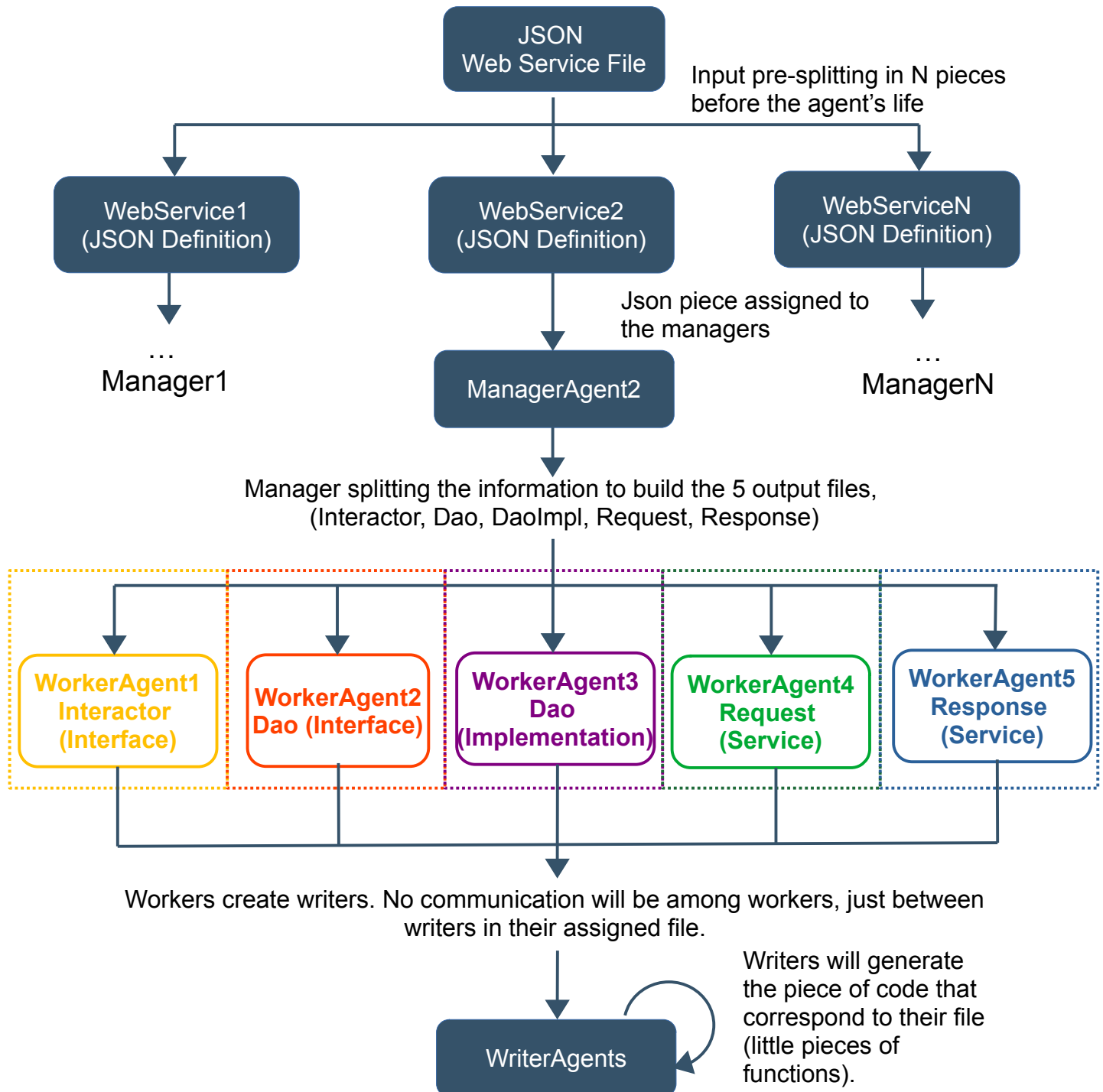
architectural and design patterns mentioned [Appendix B-F]. Finally, waiting for the code files, to physically store them.

- **Worker Agent:** These agents will be the file handler, in other words, each worker should be assigned to generate just one file. It does not matter what kind of file, whether one of the VIPER design patterns, the DAO pattern, or the service architecture. The role of a “worker” is to create a set of writers, generating micro coding tasks for each one of them. Finally, waiting for their response, to gather the little pieces of codes, and then merge them, in a complete “**Generated code file**”. It will be an agent resource that will contain information such as the package name, the imports, the class name, and all the pieces generated by the writers, sending the resource back to its respective manager. [Section 5.5].
- **Writer Agent:** These agents will be the microcode handler and the basic agent, it will take a microcode generation task, in order to generate small pieces of code, sending back to their respective worker. These tasks should be really simple such as variables, constants declaration, methods definition and implementation such as constructors, encapsulation or setups. Details regarding all the possible tasks at [Section 5.6].

Therefore, a hierarchy of agents is set, taking a global goal that is in some way difficult to achieve using just one agent. However, splitting it into small pieces, give us, as a result, the improvement in the execution time that it takes to do the development by a developer.

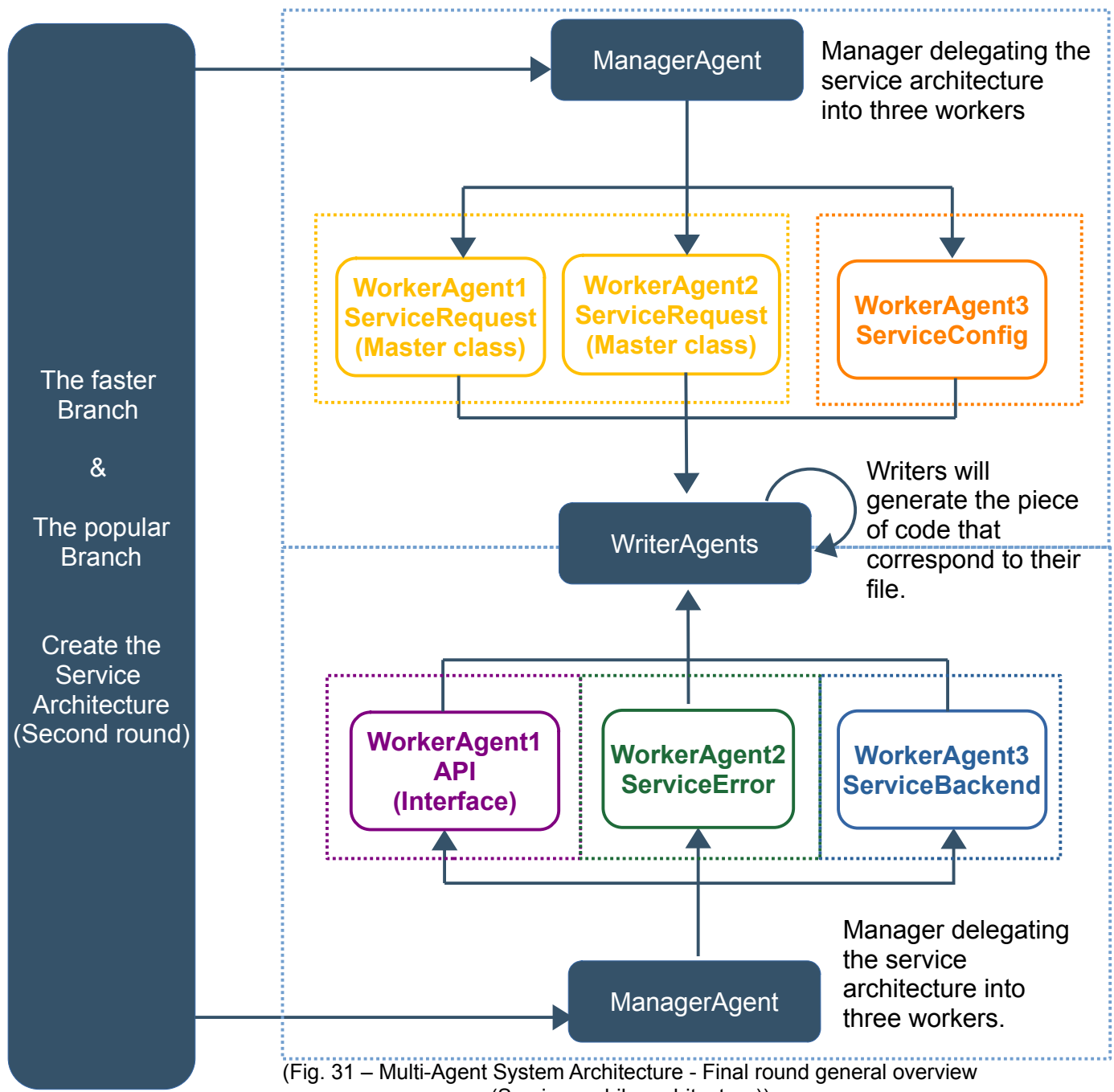
5.4 MAS architecture

Taking into account of using the technique of “*divide and conquer*” the Multi-Agent System architecture extracting one single branch of agents looks as:



(Fig. 30 – Multi-Agent System Architecture – First round general overview (VIPER Interactor & DAO design pattern))

Then, each agent branch will work as a thread were a bunch of agents will communicate with each other sending messages, delegating tasks, creating resources, in order to build five files by branch. This is going to be the first round. Later on, when all branches finished with their respective services, a final round is going to take place, selecting the faster branch (branch with less execution time) and the most popular, chosen through a plurality vote among all the agents. The two selected branches will build the service mobile architecture.



(Fig. 31 – Multi-Agent System Architecture - Final round general overview (Service mobile architecture))

Each worker is identified to work in an individual file, with the exception of the Service Request (Master Class) two workers will work together building the same file. Now, that the different kinds of agents was explained including the architecture that should be followed, let's define the shared resources files that agents will use.

5.5 Shared resource files

There will be some shared resources files that agents will use to fulfill their corresponding tasks. In addition to those already mentioned (the JSON web service, and the XML Configuration).

➤ **Service Configuration:** This resource file will storage the service attributes needed to build and code, all the web services (First round). Those attributes will be extracted from the JSON web service definition by the Manager.

Data type	Name	Description
String	Name	The service name that is used
String	Protocol	Whether http or https
String	Host	The domain name that the api have
String	Path	The additional URL context
String	Port	The port number. If explicit
Boolean	HasUrlQuery	A flag that determines if exists URL params.
String	Mode	Specify the mode in which the parameters are encoded. Such as raw, url encoded, etc.
String	Auth	The authorization required to perform a request, such as Bearer token.
Map	Headers	A collection of values in the format "key": "value" where the key will be the header name and its corresponding worth
String	Request Type	Whether GET, POST, PUT, DELETE or PATCH
Map	Request params	A collection of values in the format "key": "value" where the key will be the param name and its corresponding value
Map	Response params	A collection of values in the format "key": "value" where the key will be the param name and its corresponding value.

(Tab. 4 – Service configuration, resource file were all the data needed by each service will be stored)

- **Generated Code File:** This resource file will storage all the generated code resources by the writers, additionally, list of imports, package name, class name, and the file name that it dealing with. Each worker will generate one of this resources.

Data type	Name	Description
String	Filename	The file name that the worker is dealing with.
String	Package Name	The final mobile application package name, setting in the configuration file.
List	Imports	The list of possible imports that are needed in each file.
String	Class name	The class definition, possible interface implementations or super classes.
List	Generated codes	The list of generated codes resources that each writer generated.
Map	Execution Time	A collection of values in the format "key": "value" where the key will be an agent name, and the value will be the execution time that took that agent in accomplish its task.

(Tab. 5 – Generated Code File, resource file used by Workers to build each file needed)

- **Generated Code:** This resource file will store all the pieces of code generated by the writers, adding the agent's name that produces the code and the order number of the index that each fragment has, for example with order 1, will be variables, constants, or the singleton instance. Followed by the constructor, and then the encapsulation, there are many variants of how the code should be ordained.

Data type	Name	Description
String	Agent name	The agent name of the writer that is generating some kind of code (a certain task).
String	Code gen	The string of code that was generated.
Int	Order index	The index to which the code belongs.
Long	Execution time	The time measured in nano seconds that took to accomplish certain task.

(Tab. 6 – Generated Code, resource file used by Writer to build each piece of code needed in a certain file)

The global goal of the MAS project is to automatize the coding of different files, those needed in the mobile application, through the VIPER interactor. Requesting access to the response data model of each web service. Then, using a single agent to code all of them, it is not worth due to the performance will decrease considerably. Thereupon, lets split each mentioned file into micro-tasks that agents can easily achieve.

5.6 Agent tasks

The advantage of using architectural and design patterns, allows us to follow the guidelines of the way of coding by each class, making easy the way of how the Multi-Agent System can split the different tasks to accomplish the main goal of coding different files. Hence analyzing each individual file and then split it into micro tasks that can be handled by a single writer without work overload.

- **VIPER Interactor Response callback Interface:** *[Appendix B]*. It is a simple interface that will have two method definitions, whether the service is successful or it fails. Then two different tasks to code this file are needed:

WRITER TASK: GENERATE_INTERACTOR_SUCCESS_METHOD

WRITER TASK: GENERATE_INTERACTOR_FAILURE_METHOD

- **DAO Interface:** *[Appendix C]*. A simple interface method definition that will provide access to the data service, passing all the params needed to execute the request. Then just one task to code this file is needed:

WRITER TASK: GENERATE_DAO_METHOD

- **Service Request:** *[Appendix D]*. The service representation that will extend the Retrofit functionality, because of this class is used as a model to send all the parameters to the server, it is formed by the parameters and a constructor. Then two different tasks to code this file are needed:

WRITER TASK: GENERATE_REQUEST_PARAMS

WRITER TASK: GENERATE_REQUEST_CONSTRUCTOR

- **Service Response:** *[Appendix E]*. The POJO model that will storage the response obtained in the JSON web service file, it is formed by the parameters, the constructor and the encapsulation methods. Then, three different tasks to code this file are needed:

WRITER TASK: GENERATE_RESPONSE_PARAMS

WRITER TASK: GENERATE_RESPONSE_CONSTRUCTOR

WRITER TASK: GENERATE_RESPONSE_ENCAPSULATION

- **DAO Implementation:** *[Appendix F]*. The service implementation that will give access to the information to the interactor module. It is formed by the local variables, the implementation of the interface method, and the implementation of two Retrofit methods, whether Retrofit call is successful or it fails. Then four different tasks to code this file are needed:

WRITER TASK: GENERATE_DAOIMPL_PARAMS

WRITER TASK: GENERATE_DAOIMPL_INTERFACE_METHOD

WRITER TASK: GENERATE_DAOIMPL_CALLBACK_RESPONSE_METHOD

WRITER TASK: GENERATE_DAOIMPL_CALLBACK_FAILURE_METHOD

- **API Retrofit Interface:** *[Appendix H]*. Each service interface method that will be used to call the server. It is formed by one method per service. Due to it is a repetitive task, it does not make sense use more than one agent in a loop. Then just one task to code this file is needed:

WRITER TASK: GENERATE_SERVICE_API_METHODS

- **Service Backend:** *[Appendix I]*. Class that will manage the endpoint and the URL data that each request will point to. It is formed by the local variables and the default environment, the constructor and the encapsulation methods. Then three different tasks to code this file are needed:

WRITER TASK: GENERATE_SERVICE_BACKEND_PARAMS

WRITER TASK: GENERATE_SERVICE_BACKEND_CONSTRUCTOR

WRITER TASK: GENERATE_SERVICE_BACKEND_ENCAPSULATION

- **Service Error:** *[Appendix J]*. Class that will manage the errors on the server-side. It is formed by the local variables, the constructor, the encapsulation methods, and with a method to pretty print itself called to string. Then four different tasks to code this file are needed:

WRITER TASK: GENERATE_SERVICE_ERROR_PARAMS

WRITER TASK: GENERATE_SERVICE_ERROR_CONSTRUCTOR

WRITER TASK: GENERATE_SERVICE_ERROR_ENCAPSULATION

WRITER TASK: GENERATE_SERVICE_ERROR_TOSTRING

- **Service Config:** [Appendix K]. Class that will manage which backend environment is set. It is formed by the singleton pattern, the local variables, a setup method and the encapsulation methods. Then four different tasks to code this file are needed:

WRITER TASK: GENERATE_SERVICE_CONFIG_SINGLETON

WRITER TASK: GENERATE_SERVICE_CONFIG_PARAMS

WRITER TASK: GENERATE_SERVICE_CONFIG_SETUP

WRITER TASK: GENERATE_SERVICE_CONFIG_ENCAPSULATION

- **Service Request (Master class):** [Appendix L]. The request master class that will contain Retrofit initialization. It is formed by the local variables, the constructor, a setup method, and the encapsulation methods. Then four different tasks to code this file are needed:

WRITER TASK: GENERATE_SERVICE_REQUEST_PARAMS

WRITER TASK: GENERATE_SERVICE_REQUEST_CONSTRUCTOR

WRITER TASK: GENERATE_SERVICE_REQUEST_SETUP

WRITER TASK: GENERATE_SERVICE_REQUEST_ENCAPSULATION

All the previous ones are tasks designed to be attended only by agents with the role of writer, those are the lower level in the MAS project. Besides, we can clearly see the number writers needed to accomplish each file. As a result, we can obtain how many agents will live by each branch [Section 5.7]. Now, let's level up and define the next hierarchy of tasks that will be handled by workers, which is easy to guess. There are explicit and implicit tasks, the implicit tasks are the following:

- **WORKER TASK:– CREATE_INTERACTOR_FILE:** It refers to the file at [Appendix B].
- **WORKER TASK: – CREATE_DAO_FILE:** It refers to the file at [Appendix C].
- **WORKER TASK:– CREATE_REQUEST_FILE:** It refers to the file at [Appendix D].
- **WORKER TASK: – CREATE_RESPONSE_FILE:** It refers to the file at [Appendix E].
- **WORKER TASK: – CREATE_DAOIMPL_FILE:** It refers to the file at [Appendix F].

The explicit tasks declared into the MAS project are:

- **WORKER TASK: – CREATE_SERVICE_API_FILE:** It refers to the file at [Appendix H]

- **WORKER TASK:- CREATE_SERVICE_BACKEND_FILE:** It refers to the file at [Appendix I].
- **WORKER TASK:- CREATE_SERVICE_ERROR_FILE:** It refers to the file at [Appendix J].
- **WORKER TASK:- CREATE_SERVICE_CONFIG_FILE:** It refers to the file at [Appendix K].
- **WORKER TASK:- CREATE_SERVICE_REQUEST_FILE:** It refers to the file at [Appendix L].

Those tasks are divided into micro subtasks that will be delegated to the corresponding writers. As was mentioned in the writer's tasks. In addition to the own task such as writer creation and the generation code file resource at [Tab 5].

Finally, the tasks that will be performed by agents with the manager role, and the last hierarchy, will be the following:

- **MANAGER TASK:- SPLIT_JSON_WEB_SERVICE:** This is a task that will handle the reading of the JSON web service definition, generating the service configuration resource [Tab 4]. Then continuing with the creation of all the workers.
- **MANAGER TASK:- PHYSICALLY_FILE_STORAGE:** This is a task that will perform the writing of the generated code files (resource list) physically.
- **MANAGER TASK:- CONFIGURE_SERVICE_ARCHITECTURE:** This task refers to the first round, which is the creation of all the web services [Appendix B-F]. One folder will be assigned to one web service.
- **MANAGER TASK:- CONFIGURE_SERVICE_BACKEND:** This task refers to the final round, after obtaining the faster branch and the popular branch. Files at [Appendix H-L].

5.7 MAS communication (workflow)

All the sorts of agents and roles, the MAS architecture, all the agent's resource files, and the different tasks that each role can have, are already defined. The next step is to define how all of those concepts work together, that is the communication among agents, allowing the collaboration of them, in this way creating a powerful team of agents. As was mentioned in the last chapter dedicated to Multi-Agent Systems, there were two important communication channels that were used in this TFM work, the blackboard technique and direct messages. The combination of both of them helps to improve the communication involved. That is,

separating the task's assignation in a decomposition way, as a result, of facing this distributed Multi-Agent System.

On one hand, the blackboard technique is used to control the tasks performed by the agents. In other words, inside the blackboard, tasks can be written using the following structure:

Attribute	Description
Id	A unique consecutive identifier starting by one.
Task	All the different tasks that were defined in the previous section.
State	Define the status the task is in. Providing with the mechanism that will be possible to manage which tasks are already taken, or finished. Using the values: PENDING IN_PROCESS FINISHED.
Role	Define the kind of agent that can take this task. Providing with the mechanism to control the hierarchy tasks already defined.

(Tab. 7 – Blackboard TASK structure)

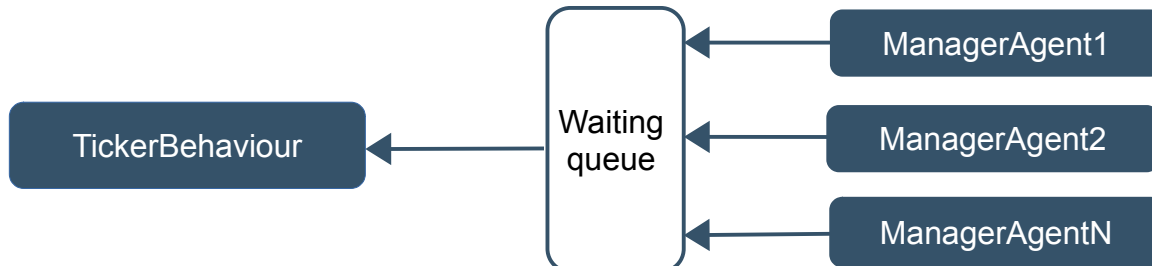
On the other hand, direct messages will allow the MAS project, all the shared resources files that each agent is going to need according to the phase of the task in turn, throughout the workflow. Direct messages work using the agent platform created by JADE, that is, when an agent is created it is registered into the Directory facilitator. That will allow other agents the possibility of communicating to that agent throughout a search by an AID (Agent Identifier) that is a unique value for every existing agent in the platform.



(Fig. 32 – Agent registration into the directory facilitator to handle direct communication)

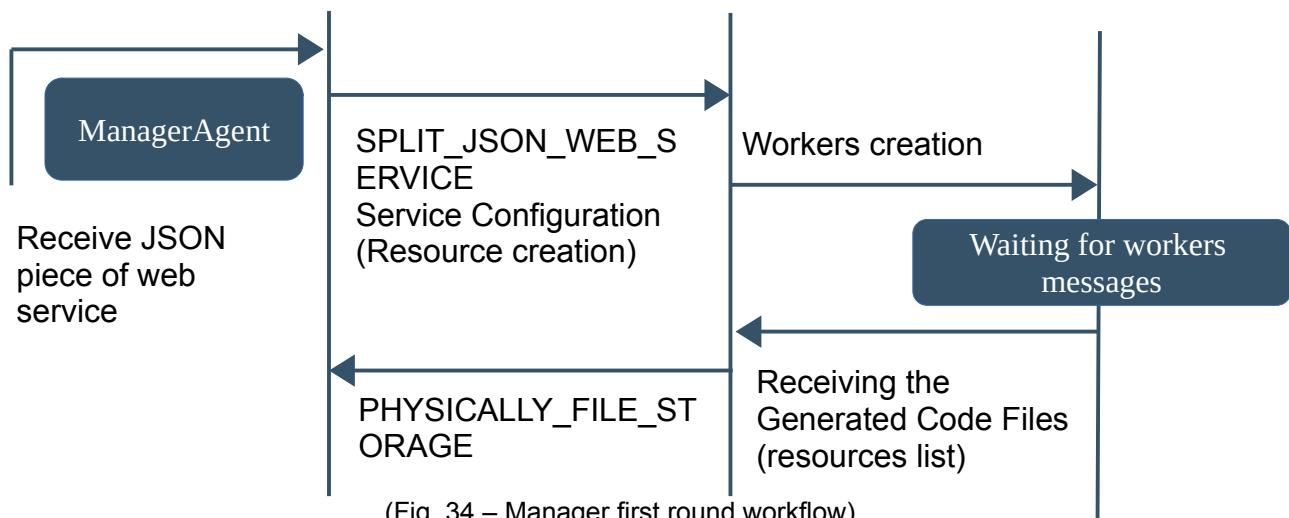
Moreover, each agent starts working according to their tasks assigned or waiting if necessary. It totally will depend on the behaviour that was programmed. That is, each agent

will extend the ***“TickerBehaviour”*** class, this is a predefined class inside the JADE platform. It is an abstract class that implements a behaviour that periodically executes a user-defined piece of code.



(Fig. 33 – Manager Agent behaviour assigned to a waiting queue)

Because of the MAS architecture, that class fits perfect, each manager agent after being registered, will try to achieve its task, then in order to avoid a bottleneck, each agent will be assigned to a waiting queue from one to three seconds, randomly chosen. Then, the response time of every agent will be equilibrated, some of them will wait, more than others, breaking with a possible bottleneck. This behaviour will follow the following task graphs:



(Fig. 34 – Manager first round workflow)

After all, managers finished the first round, an evaluation will take place, obtaining the faster and the popular branch to continue with the final round.

The faster branch will be obtained calculating the sum of the execution time in their corresponding branch:

$$Manager(t) = \sum_{i=0}^5 worker_i(t) + waiting\ time \quad Worker(t) = \sum_{i=0}^n writer_i(t) + waiting\ time$$

Where $n = 2$ in Interactor Interface

Where $n = 1$ in DAO Interface

Where $n = 2$ in Service Request

Where $n = 3$ in Service Response

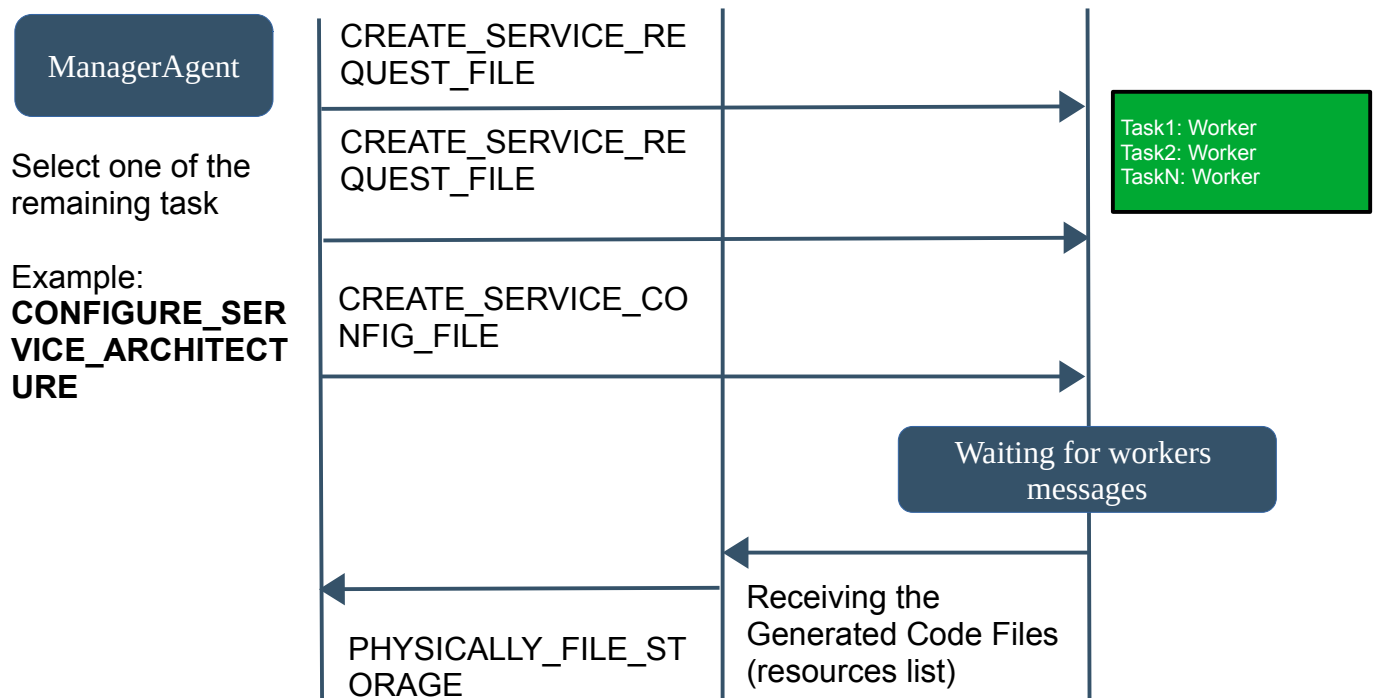
Where $n = 4$ in DAO Implementation

As a result, n determines how many writers are needed by each file (In this first round).

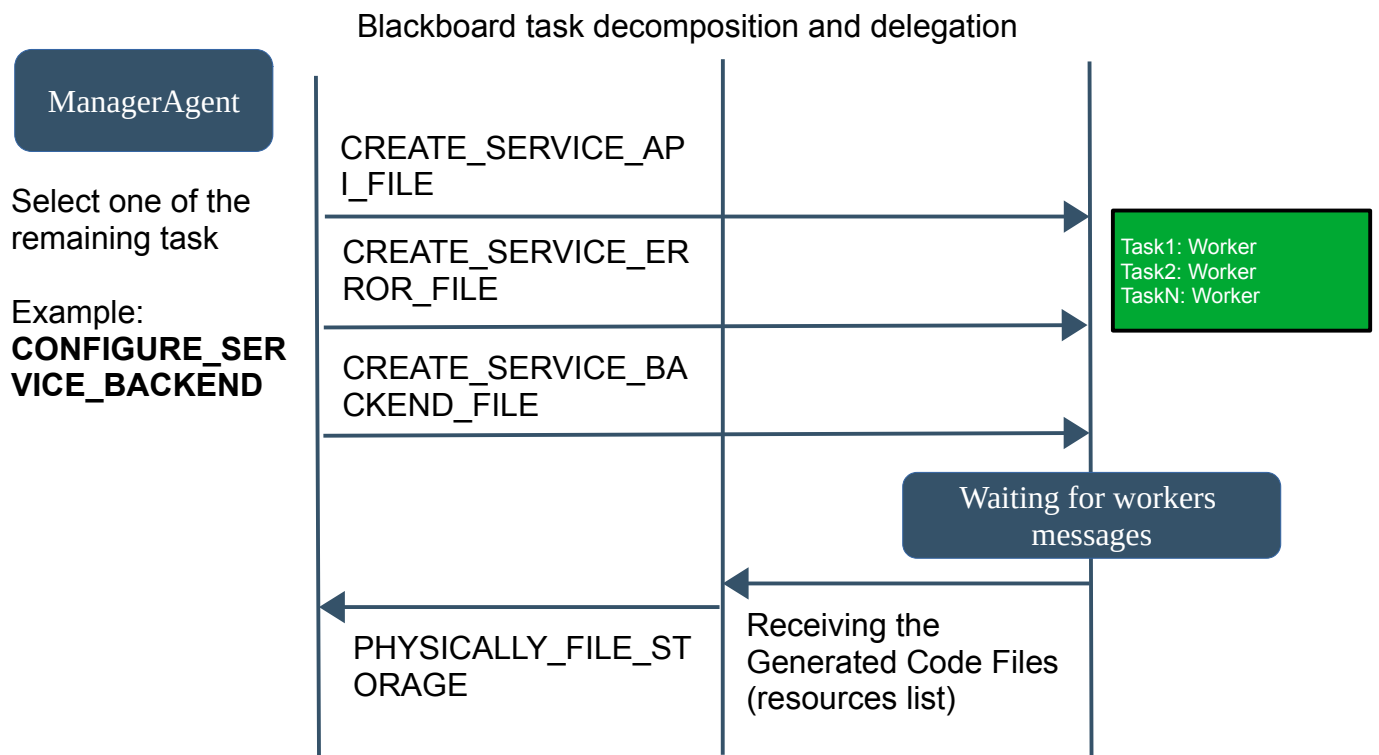
Therefore, each branch will be formed by, 12 writers + 5 workers + 1 manager = 18 agents.

The popular branch is selected by a plurality voting protocol, all agents will randomly select one manager, just workers and writers can vote. The winner branches will receive the proposal of those tasks. The first that arrives, the first that takes one of them.

Blackboard task decomposition and delegation

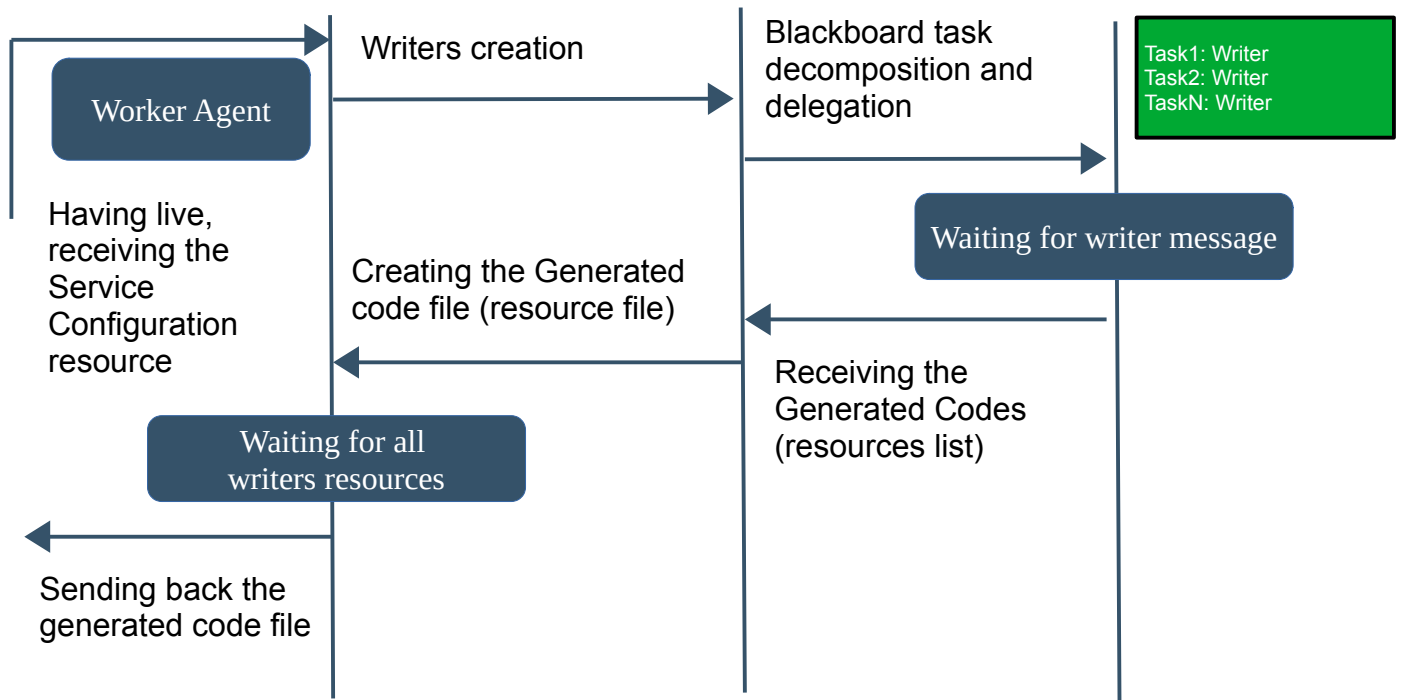


(Fig. 35 – Manager final round workflow (Configure Service Architecture))

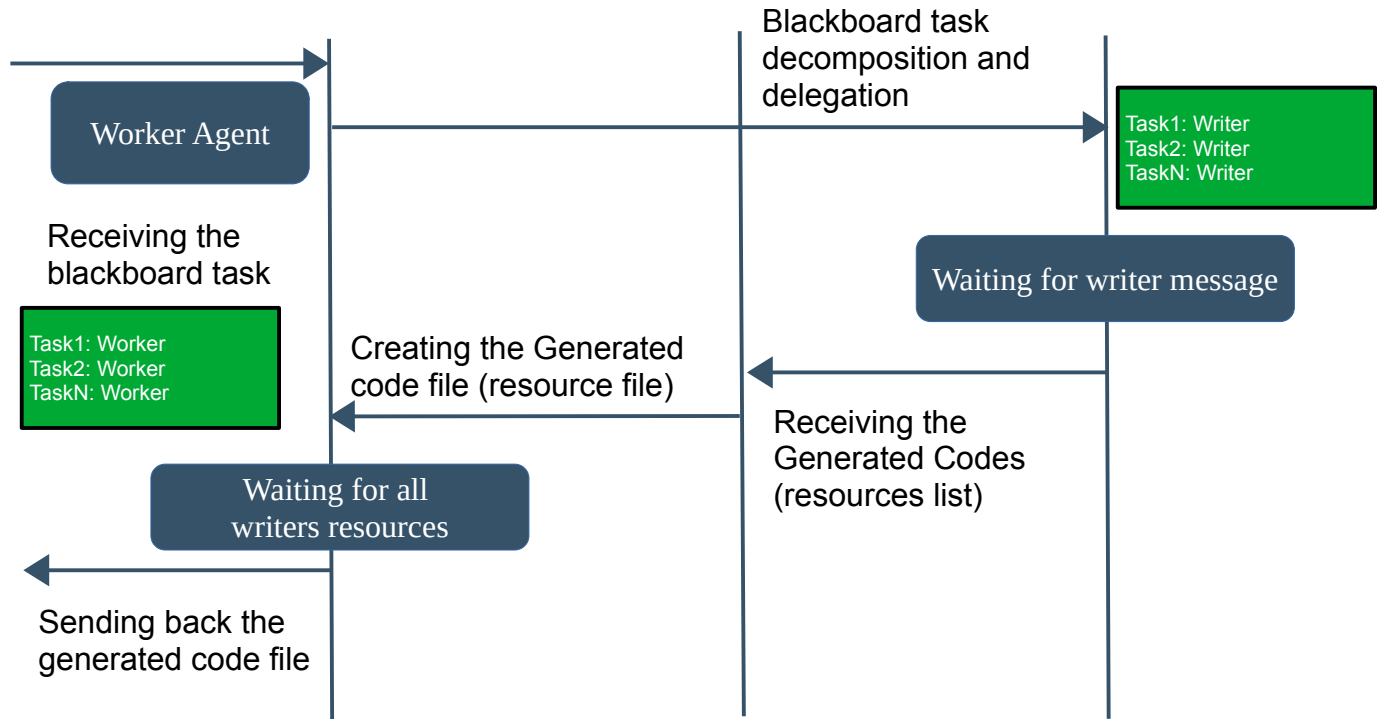


(Fig. 36 – Manager final round workflow (Configure Service Backend))

Now, let's go deeper into the hierarchy of the workflow that corresponds with the workers.

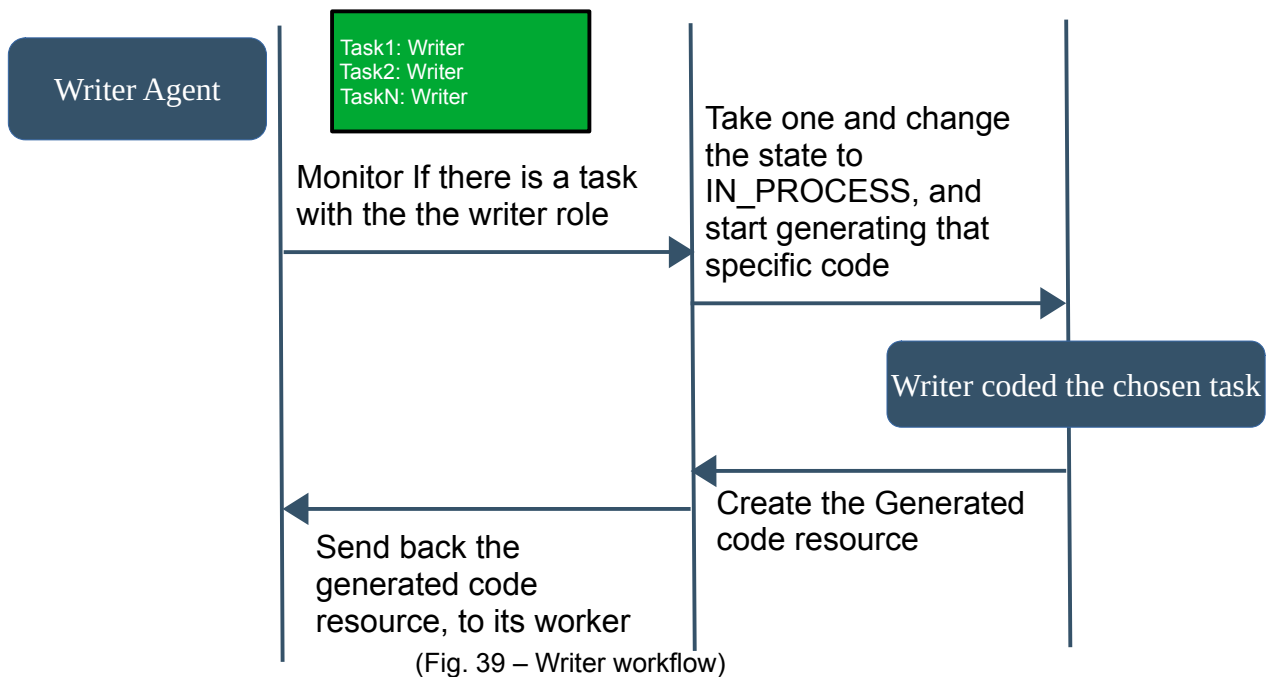


(Fig. 37 – Worker first round workflow)



(Fig. 38 – Worker final round workflow)

Now, let's go deeper in the hierarchy of the workflow that corresponds with the writers.



(Fig. 39 – Writer workflow)

As it can be noticed, the writer workflow does not care about rounds, whether the first round or the final round is executing, writers just have to monitor the blackboard, selecting a task that has the writer role. The state of a task provides with the required control, preventing task coalition. Therefore, tasks with a PENDING state, are the only ones that can be selected by agents. Once an agent chooses one task the first step is to change its state to IN_PROCESS.

In the final round, as is shown in its corresponding architecture diagram, only three workers by each branch, are needed, performing the construction of the mobile service architecture files at [Appendix H-L]. Once the blackboard is empty, and the global goal is achieved and all the files are physically stored inside a folder path. The execution of the MAS project will ends. It is worth to mention, that the MAS project in every step is using the console for logging purpose. And at the end of the first round, a log is also storage.

The log will contain the execution time by each agent in every branch, additionally, the faster agent found will be printed, and a sum of all the agents involved through the process. Finally, a list of all the managers and how many votes they obtained to be chosen the most popular agent branch, at the end the respective winner.

Chapter 6: Conclusions and future related works

The purpose of this chapter is to analyze the output when the MAS project was tested, as well as the performance obtained. In this way, write what are the conclusions that were achieved, in order to see if using a Multi-Agent System really can improve in some way, a mobile development environment. Furthermore, discuss regarding possible future related works or improvements that this MAS project can get, to increase the scope of this research and to increase the use of Multi-Agent Systems in more environments, for example generating other sorts of patterns with different languages such as MVP, MVI that takes a reactive approach. Using SOAP based on XML-Language as starting point to be interpreted by the agents.

Using one MAS architecture by each language to avoid overworking and possible memory leaks at implementation step. In this way, leaving agents to know about specific tasks with specific languages and patterns, keeping it as simple as possible. In other words, one MAS architecture will handle specific input format, patterns, and output language. Another architecture will handle other input format, patterns and output language. Therefore, depending on the input given together with the desire output. Will be the MAS architecture to be implemented.

6.1 Results obtained

The process described in the previous chapter was tested using different public APIs found at [10], all of them were tested and then exported using Postman. In order to obtain more accurate results, a variety of web services were used throughout five intents. It is worth to mention that the name of each web services should be unique, in order to identify it correctly, preventing overwriting. Let's start analyzing an API with few web services, called Airlines.

AirlinesAPI	
# Services	2 (<i>GetAirlines, GetAirlinesDetails</i>)
# Agents	36 Each service composed by 18 agents, then 18 * 2
Execution time	
Iteration 1	40.39906810 secs
Iteration 2	35.43024739 secs
Iteration 3	41.46235415 secs
Iteration 4	37.31227504 secs
Iteration 5	34.37043655 secs
Iteration 6	41.51275181 secs
Iteration 7	39.74625394 secs
Iteration 8	41.10632500 secs
Iteration 9	37.60492270 secs
Iteration 10	35.50808847 secs
Average time	38.44527231 secs

(Tab. 8 – Airlines API execution time through 10 rounds and its average, mapping 2 services)

HotelsAPI	
# Services	5 (<i>SearchHotel, GetHotelDetails, GetMetadata, GetHotelPhotos, GetPropertiesList</i>)
# Agents	90 (<i>Each service composed by 18 agents, then 18 * 5</i>)
Execution time	
Iteration 1	40.32873103 secs
Iteration 2	39.15168995 secs
Iteration 3	40.69914809 secs
Iteration 4	39.10317678 secs
Iteration 5	39.43657669 secs
Iteration 6	38.88301183 secs
Iteration 7	40.56687368 secs
Iteration 8	39.53240958 secs
Iteration 9	39.59447333 secs
Iteration 10	35.64845500 secs
Average time	39.29445459 secs

(Tab. 9 – Hotels API execution time through 10 rounds and its average, mapping 5 services)

RecargasTuCell	
# Services	8 (<i>GetLogin, GetMovements, GetChildren, GetPaymentServices, GetProducts, GetProviderTAE, GetTopups, SetUser</i>)
# Agents	144 (<i>Each service composed by 18 agents, then 18 * 8</i>)
Execution time	
Iteration 1	42.50258758 secs
Iteration 2	38.42525252 secs
Iteration 3	40.68984799 secs
Iteration 4	38.27012665 secs
Iteration 5	40.18736641 secs
Iteration 6	37.03883562 secs
Iteration 7	40.14802296 secs
Iteration 8	38.32014624 secs
Iteration 9	40.22989605 secs
Iteration 10	38.12995724 secs
Average time	39.39420392 secs

(Tab. 10 – RecargasTuCell API execution time through 10 rounds and its average, mapping 8 services)

CocktailAPI	
# Services	12 (<i>GetAlcoholFilters, GetIngredients, GetClasses, GetCategories, GetByIngredient, FilterByCategory, GetByName, GetByNamev2, FilterByAlcoholic, GetIngredientsById, GetRandomIngredient, GetCocktailDetails</i>)
# Agents	216 (<i>Each service composed by 18 agents, then 18 * 12</i>)
Execution time	
Iteration 1	39.51814087 secs
Iteration 2	39.30475045 secs
Iteration 3	38.27966994 secs
Iteration 4	39.50272685 secs
Iteration 5	40.29332448 secs
Iteration 6	40.65932721 secs
Iteration 7	38.58545108 secs
Iteration 8	40.23565613 secs
Iteration 9	40.40350138 secs
Iteration 10	40.38934882 secs
Average time	39.71718972 secs

(Tab. 11 – CockTail API execution time through 10 rounds and its average, mapping 12 services)

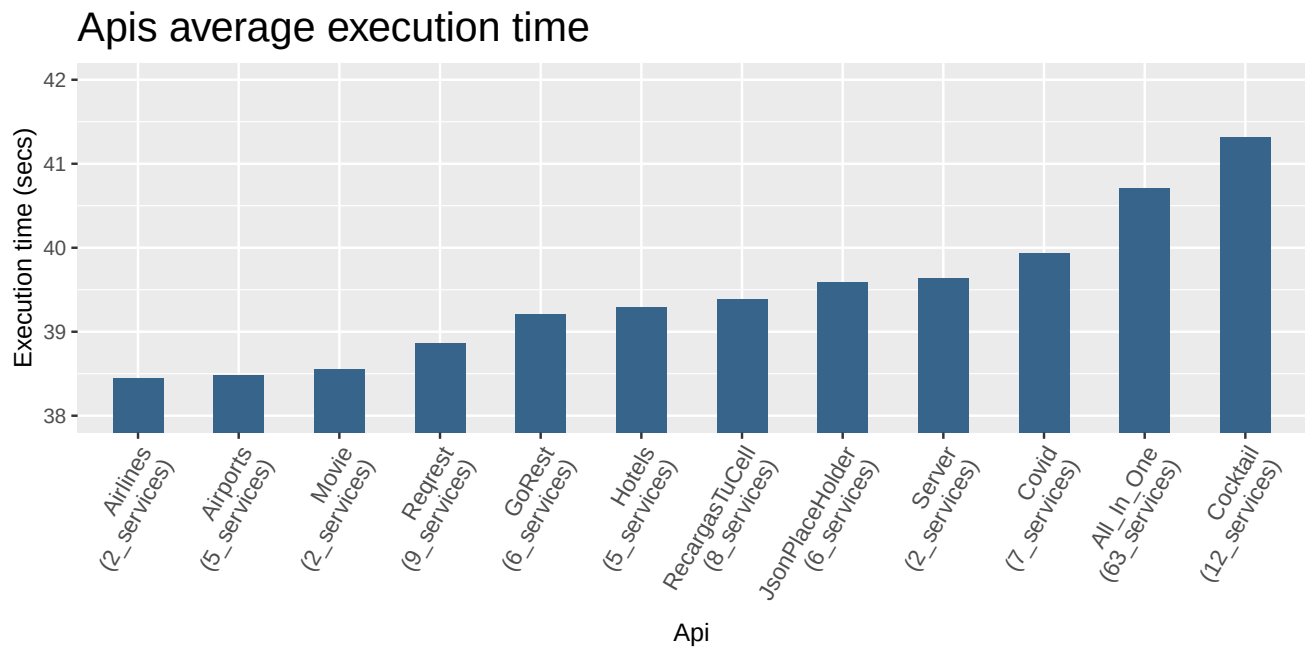
In order to hard test the capabilities of the TFM project, all the API's found were manually merged into one big JSON file, raising the number of services that should be mapped until 63 web services.

ALL_IN_ONE	
# Services	63 (<i>All the previously service mentioned plus others at [26]</i>)
# Agents	1,134 (<i>Each service composed by 18 agents, then 18 * 63</i>)
Execution time	
Iteration 1	40.00655432 secs
Iteration 2	40.77475594 secs
Iteration 3	40.67340656 secs
Iteration 4	39.54129527 secs
Iteration 5	40.69804088 secs
Iteration 6	41.20028389 secs
Iteration 7	42.22022965 secs
Iteration 8	40.46373763 secs
Iteration 9	41.03273731 secs
Iteration 10	40.50795574 secs
Average time	40.71189971 secs

(Tab. 12 – All In One API execution time through 10 rounds and its average, mapping 63 services)

# Services	API	# Agents	Average executed time
2	Server	36	39.63526563 secs
2	Movie	36	38.55619211 secs
2	Airlines	36	38.44527231 secs
5	Airports	90	38.48830112 secs
5	Hotels	90	39.29445459 secs
6	GoRest	108	39.20940600 secs
6	Jsonplaceholder	108	39.59334701 secs
7	Covid	126	39.94117531 secs
8	RecargasTuCell	144	39.39420392 secs
39	Reqrst	162	38.86271463 secs
12	Cocktail	216	41.31742993 secs
63	ALL_IN_ONE	1134	40.71189971 secs

(Tab. 13 – APIs tested in the TFM project)



(Fig. 40 – APIs tested in the TFM project)

As can be seen, no matter how many services were tested by the MAS project architecture, the interval values after 10 iterations (each service) do not exceed 42 seconds.

6.2 Conclusions

The use of agent branches allows the Multi-Agent System architecture to be really faster than having just one or a small team of physical developers, that is, making the architecture more efficient through task decomposition and parallelism. In this way, dividing a complex problem into a set of sub-tabs or subproblems, decomposing large tasks into sub-tasks that can be tackled by different agents inside the hierarchy proposed. That is why communication through agents must be really precise in the different phases of the distributed problem.

The methods proposed using shared files, deal perfectly with excellent performance, accuracy according to the tested results obtained.

Another important point, lies in the use of a waiting queue in the agent behaviour, due to, trying with different sort of behaviours, such as sequential or cyclical, agents were executed really fast producing a bottleneck in task assignation. Sometimes, both or more agents

working in the same task. Therefore, applying a random waiting queue between 1 and 3 seconds allows to take a task, giving them enough time to change the task state, as a result, the delayed agents will not be able to take those tasks.

It is true that 63 web services is not a high number of services, but, the multi-agent system perfectly meets the needs for the development of small, medium and slightly large applications. It is worth to mention that the API called RecargasTuCell is a real application. The web services were taken and reproduced in a local environment. The system can be tested using other real APIs. However, an important aspect is that how many memory the Java and JADE environment in a local host can correctly be managed by each process. Java compiler settings allows the developer to modify this value, but it will depend in the RAM available in the system. Normally it is set in 2048 MB, enough for this test. Other alternatives that can affect the performance of this MAS project is the processor capabilities.

In summary, the MAS architecture is a functional tool that can provide the enough performance to speed up the development of mobile applications.

6.3 Future related works

This Multi-Agent system can be improved, increasing the scope that was originally set for this TFM work. As it is known, this project uses the architectural design pattern of VIPER, together with the DAO pattern through a RESTful API of web services. Nevertheless, there is a huge variety of how a mobile application can be developed and using a different combination of them, for example, instead of using the VIPER pattern, using a MVI pattern that is focused on a reactive architecture. At present, having more popularity by the community of developers.

Another alternative is instead of using RESTful API web services, using the SOAP standard with XML approach. As it was seen by this TFM work, there are many alternatives. To do so, future related works, by using the MAS configuration file, where new attributes can be added, such as:

Attribute	Description
<ach_pattern>	Whether VIPER, MVI, MPV, MVVM, etc.
<service_standard></service_standard>	The web service standard to follow, whether JSON or XML
.	.

(Tab. 14 – Future MAS configuration attributes to increase the scope capabilities)

Imaging a possible perspective, let's say that this MAS project will be the VIPER layer that is going to manage that specific architectural pattern, then adding by each pattern a new MAS layer that can handle the respective tasks to code different patterns. On the other hand, establishing a new input format where both XML and JSON standards can be easily converted readable by each MAS layer, otherwise building each MAS layer to understand both XML and JSON concepts will add more complexity the code consumption.

Appendices

First round process (All the manager branches will generate)

```
1 package com.navi.skeleton.entities;
2
3
4 /**
5  * Created by TFM on 10/06/20.
6  */
7 public class HotelLocation {
8
9     //-----
10    //MARK:- Variables & Complements
11    //-----
12    private String locationName;
13    private Coordinates coordinates;
14    private String resolvedLocation;
15
16    //-----
17    //MARK:- Constructor
18    //-----
19    public HotelLocation () {
20
21    }
22
23    //-----
24    //MARK:- Encapsulation
25    //-----
26    public void setLocationName(String locationName) { this.locationName = locationName; }
27    public String getLocationName() { return this.locationName; }
28
29    public void setCoordinates(Coordinates coordinates) { this.coordinates = coordinates; }
30    public Coordinates getCoordinates() { return this.coordinates; }
31
32    public void setResolvedLocation(String resolvedLocation) { this.resolvedLocation = resolvedLocation; }
33    public String getResolvedLocation() { return this.resolvedLocation; }
34
35 }
```

Auto generated by TFM

(Appendix A – Plain Old Java Object (POJO), a simple class with their own properties and encapsulation methods)

```
1 package com.navi.skeleton.app.network.services.SearchHotel;
2
3 import com.navi.skeleton.app.network.error.Error;
4
5 /**
6  * Created by TFM project on 16-07-2020.
7  */
8 public interface SearchHotelInteractor {
9     void onSearchHotelSuccess(SearchHotelResponse response);
10    void onSearchHotelFailure(Error error);
11 }
12
```

(Appendix B – VIPER Interactor Response Callback Interface, the interface use to return the response model to the VIPER presenter)

```

1 package com.navi.skeleton.app.network.services.SearchHotel;
2
3
4 /**
5  * Created by TFM project on 16-07-2020.
6  */
7 public interface SearchHotelDao {
8     void searchHotel (String query,
9                       String locale,
10                      SearchHotelInteractor listener);
11
12 }
13

```

Auto generated by TFM

(Appendix C – DAO Interface, the interface use to have data access through the request sending the params needed together with the interactor interface, providing the callback to the user interaction)

```

1 package com.navi.skeleton.app.network.services.SearchHotel;
2
3 import com.navi.skeleton.app.network.services.ServiceRequest;
4
5 /**
6  * Created by TFM project on 16-07-2020.
7  */
8 public class SearchHotelRequest extends ServiceRequest {
9
10
11     //-----
12     //MARK:- Constructor
13     //-----
14     public SearchHotelRequest () {}
15
16 }

```

Auto generated by TFM

(Appendix D – Service Request, the service child that will extends all the functionality of the retrofit service request master class [Appendix L])

```

1  package com.navi.skeleton.app.network.services.SearchHotel;
2
3  import java.util.List;
4
5  import com.navi.skeleton.entities.*;
6
7  /**
8   * Created by TFM project on 16-07-2020.
9   */
10 public class SearchHotelResponse {
11
12     //-----
13     //MARK:- Variables & Complements
14     //-----
15     private Long moresuggestions;
16     private Boolean misspellingfallback;
17     private List<Suggestions> suggestions;
18     private String term;
19     private String trackingID;
20
21     //-----
22     //MARK:- Constructor
23     //-----
24     public SearchHotelResponse (Long moresuggestions,
25                                Boolean misspellingfallback,
26                                List<Suggestions> suggestions,
27                                String term,
28                                String trackingID) {
29         super();
30         this.moresuggestions = moresuggestions;
31         this.misspellingfallback = misspellingfallback;
32         this.suggestions = suggestions;
33         this.term = term;
34         this.trackingID = trackingID;
35     }
36
37     //-----
38     //MARK:- Encapsulation
39     //-----
40     public void setMoresuggestions(Long moresuggestions) { this.moresuggestions = moresuggestions; }
41     public Long getMoresuggestions() { return this.moresuggestions; }
42
43     public void setMisspellingfallback(Boolean misspellingfallback) { this.misspellingfallback = misspellingfallback; }
44     public Boolean getMisspellingfallback() { return this.misspellingfallback; }
45
46     public void setSuggestions(List<Suggestions> suggestions) { this.suggestions = suggestions; }
47     public List<Suggestions> getSuggestions() { return this.suggestions; }
48
49     public void setTerm(String term) { this.term = term; }
50     public String getTerm() { return this.term; }
51
52     public void setTrackingID(String trackingID) { this.trackingID = trackingID; }
53     public String getTrackingID() { return this.trackingID; }
54 }

```

(Appendix E – Service Response, the POJO model that will be sent back through the interactor interface [Appendix B])

```

1 package com.navi.skeleton.app.network.services.SearchHotel;
2
3 import androidx.annotation.NonNull;
4 import androidx.annotation.Nullable;
5
6 import com.navi.skeleton.app.network.error.Error;
7 import com.navi.skeleton.app.network.services.ServiceResponse;
8
9 import retrofit2.Call;
10 import retrofit2.Callback;
11 import retrofit2.Response;
12
13 /**
14  * Created by TFM project on 16-07-2020.
15  */
16 public class SearchHotelDaoImpl implements SearchHotelDao,
17     Callback<SearchHotelResponse> {
18
19     //-----
20     //MARK:- Variables & Complements
21     //-----
22     private SearchHotelInteractor listener;
23     private SearchHotelRequest request;
24
25     //-----
26     //MARK:- SearchHotel Implementation
27     //-----
28     @Override
29     public void searchHotel (String query,
30         String locale,
31         SearchHotelInteractor listener) {
32         this.listener = listener;
33         this.request = new SearchHotelRequest();
34
35         try {
36             Call<SearchHotelResponse> call = request.getApi()
37                 .searchHotel("a82156eb53msh83ccd3c830f58f5p10d463jsn6c5c06889085",
38                     "hotels4.p.rapidapi.com",
39                     query,
40                     locale);
41             call.enqueue(this);
42
43         } catch (Exception e) {
44             Error error = new Error(e.getLocalizedMessage());
45             this.listener.onSearchHotelFailure(error);
46         }
47     }

```

Auto generated by TFM

(Appendix F – DAO Implementation, the class that will handle the communication through the interactor module and the server, giving access to the POJO response model [Appendix E], using the VIPER Response Callback Interface [Appendix B] to handle whether the request was success or it failures)

```

1  package com.navi.skeleton.modules.Search.interactor;
2
3  import com.navi.skeleton.app.network.error.Error;
4  import com.navi.skeleton.modules.Search.protocols.SearchInteractorInput;
5  import com.navi.skeleton.modules.Search.protocols.SearchInteractorOutput;
6
7  /**
8   * Created by van on 01/01/20.
9   */
10 public class SearchInteractor implements SearchInteractorInput,
11     SearchHotelInteractor {
12
13     //-----
14     //MARK:- Variables & Complements
15     //-----
16     public SearchInteractorOutput presenter;
17
18     //-----
19     //MARK:- For service SearchHotel
20     //-----
21     @Override
22     public void searchHotel(final String query, final String locale){
23         Thread task = new Thread(new Runnable() {
24             @Override
25             public void run() {
26                 new SearchHotelDaoImpl().searchHotel(query, locale, ,SearchInteractor.this);
27             }
28         });
29         task.start();
30     }
31     //MARK: Listener
32     @Override
33     public void onSearchHotelSuccess(SearchHotelResponse response){
34         // presenter.updateUI(response);
35     }
36     @Override
37     public void onSearchHotelFailure(Error error){
38         // presenter.showError(error.getMsg());
39     }
40 }

```

Generated by the Developer

(Appendix G – VIPER Interactor Module Implementation, the class that will request for access through the DAO Implementation [Appendix F], the presenter module will send the parameters from the user view and the response result will be sent back whether it is success or it failures, how the presenter and the VIPER protocols are not the full scope of this TFM.)

Second round process (Faster & popular branches will generate)

```
1 package com.navi.skeleton.app.network.services;
2
3 import com.navi.skeleton.app.network.services.GetMetadata.GetMetadataResponse;
4 import com.navi.skeleton.app.network.services.GetPropertiesList.GetPropertiesListResponse;
5 import com.navi.skeleton.app.network.services.SearchHotel.SearchHotelResponse;
6 import com.navi.skeleton.app.network.services.GetHotelDetails.GetHotelDetailsResponse;
7 import com.navi.skeleton.app.network.services.GetHotelPhotos.GetHotelPhotosResponse;
8
9 import retrofit2.Call;
10 import retrofit2.http.Body;
11 import retrofit2.http.GET;
12 import retrofit2.http.Header;
13 import retrofit2.http.Query;
14
15 /**
16  * Created by TFM project on 16-07-2020.
17  */
18 public interface Api {
19
20     //-----
21     //MARK:- Service Methods to be consumed by the user
22     //-----
23     @GET("get-meta-data")
24     Call<GetMetadataResponse> getMetadata(@Header("x-rapidapi-key") String x_rapidapi_key,
25     @Header("x-rapidapi-host") String x_rapidapi_host);
26
27     @GET("properties/list")
28     Call<GetPropertiesListResponse> getPropertiesList(@Header("x-rapidapi-key") String x_rapidapi_key,
29     @Header("x-rapidapi-host") String x_rapidapi_host,
30     @Query("pageNumber") String pageNumber,
31     @Query("checkIn") String checkIn,
32     @Query("sortOrder") String sortOrder,
33     @Query("pageSize") String pageSize,
34     @Query("currency") String currency,
35     @Query("locale") String locale,
36     @Query("destinationId") String destinationId,
37     @Query("checkOut") String checkOut,
38     @Query("adults1") String adults1);
39
40     @GET("locations/search")
41     Call<SearchHotelResponse> searchHotel(@Header("x-rapidapi-key") String x_rapidapi_key,
42     @Header("x-rapidapi-host") String x_rapidapi_host,
43     @Query("query") String query,
44     @Query("locale") String locale);
45
46     @GET("properties/get-details")
47     Call<GetHotelDetailsResponse> getHotelDetails(@Header("x-rapidapi-key") String x_rapidapi_key,
48     @Header("x-rapidapi-host") String x_rapidapi_host,
49     @Query("checkIn") String checkIn,
50     @Query("currency") String currency,
51     @Query("id") String id,
52     @Query("locale") String locale,
53     @Query("checkOut") String checkOut,
54     @Query("adults1") String adults1);
55
56     @GET("properties/get-hotel-photos")
57     Call<GetHotelPhotosResponse> getHotelPhotos(@Header("x-rapidapi-key") String x_rapidapi_key,
58     @Header("x-rapidapi-host") String x_rapidapi_host,
59     @Query("id") String id);
60 }
```

Auto generated by TFM

(Appendix H – API (Retrofit Interface), the class that will contain all the service method definitions, the DAO implementation [Appendix F] will give access to these resources)

```

1 package com.navi.skeleton.app.network.backend;
2
3 /**
4  * Created by TFM project on 16-07-2020.
5  */
6 public class ServiceBackend {
7
8     //-----
9     //MARK:- Variables & Complements
10    //-----
11    private String protocol;
12    private String domain;
13    private String port;
14    private String context;
15
16    public enum Environment { DEV };
17
18    //-----
19    //MARK:- Constructor
20    //-----
21    public ServiceBackend(Environment env){
22        switch(env){
23            case DEV:
24                this.protocol = "https";
25                this.domain = "hotels4.p.rapidapi.com";
26                this.port = "";
27                break;
28            default:
29                break;
30        }
31    }
32
33    //-----
34    //MARK:- Encapsulation
35    //-----
36    public String getUrl(){
37        if(this.port == null || this.port.equals("")){
38            return this.protocol + "://" + this.domain;
39        } else {
40            return this.protocol + "://" + this.domain + ":" + this.port + "/";
41        }
42    }
43 }

```

Auto generated by TFM

(Appendix I – Service Backend, the class that will contain the environment and build the server endpoint (URL))

```

1 package com.navi.skeleton.app.network.error;
2
3 import androidx.annotation.NonNull;
4
5 /**
6  * Created by TFM project on 16-07-2020.
7  */
8 public class Error extends Exception {
9
10     //-----
11     //MARK:- Variables & Complements
12     //-----
13     private int code;
14     private String msg;
15
16     //-----
17     //MARK:- Constructor
18     //-----
19     public Error(String msg) {
20         this.code = 1003;
21         this.msg = msg;
22     }
23
24     //-----
25     //MARK:- Encapsulation
26     //-----
27     public int getCode() { return this.code; }
28     public String getMsg() { return this.msg; }
29
30     //-----
31     //MARK:- To String
32     //-----
33     @NonNull
34     @Override
35     public String toString() {
36         return "Code: " + this.code +
37             "Msg: " + this.msg;
38     }
39 }

```

Auto generated by TFM

(Appendix J – Service Error, the class that will handle all the possible error messages when the server connection is refused or something when wrong at execution time)

```

1 package com.navi.skeleton.app.network.services;
2
3 import com.navi.skeleton.app.network.backend.ServiceBackend;
4
5 /**
6  * Created by TFM project on 16-07-2020.
7  */
8 public class ServiceConfig {
9
10     //-----
11     //MARK:- Singleton
12     //-----
13     public static ServiceConfig configInstance;
14
15     public static ServiceConfig sharedInstance() {
16         if(configInstance == null){
17             configInstance = new ServiceConfig();
18         }
19         return configInstance;
20     }
21
22     //-----
23     //MARK:- Variables & Complements
24     //-----
25     private ServiceBackend backend;
26
27     //-----
28     //MARK:- Backend setup with X environment
29     //-----
30     public void setup() {
31         backend = new ServiceBackend(ServiceBackend.Environment.DEV);
32     }
33
34     //-----
35     //MARK:- Encapsulation
36     //-----
37     public ServiceBackend getBackend() { return this.backend; }
38 }

```

Auto generated by TFM

(Appendix K – Service Configuration, the class that will setup the backend environment and give access to the retrofit service request master class [Appendix L])

```

1  package com.navi.skeleton.app.network.services;
2
3  import com.google.gson.Gson;
4  import com.google.gson.GsonBuilder;
5
6  import retrofit2.converter.gson.GsonConverterFactory;
7  import retrofit2.Retrofit;
8
9  /**
10   * Created by TFM project on 16-07-2020.
11   */
12  public class ServiceRequest {
13
14      //-----
15      //MARK:- Variables & Complements
16      //-----
17      private Api api;
18
19      //-----
20      //MARK:- Constructor
21      //-----
22      public ServiceRequest(){
23          this.setup();
24      }
25
26      //-----
27      //MARK:- Service Request Setup
28      //-----
29      private void setup() {
30          ServiceConfig.sharedInstance().setup();
31
32          Gson gson = new GsonBuilder()
33              .setPrettyPrinting()
34              .setLenient()
35              .create();
36
37          Retrofit retrofit = new Retrofit.Builder()
38              .baseUrl(ServiceConfig.sharedInstance().getBackend().getUrl())
39              .addConverterFactory(GsonConverterFactory.create(gson))
40              .build();
41
42          api = retrofit.create(Api.class);
43      }
44
45      //-----
46      //MARK:- Encapsulation
47      //-----
48      public Api getApi() { return this.api; }
49  }

```

Auto generated by TFM

(Appendix L – Service Request, this class will handle the retrofit initialization using the API interface [Appendix H], together with the initialization of the service configuration [Appendix K])

References

- [1] Program development in java, Barbara Liskov & John Guttag, Chapter 15 – Design patterns pag. 369, 2000.
- [2] Reviews of iOS architectural design pattern for testability, Modifiability and performance quality, Fauzi Scholichin, Schahiza Abd Halim, Mohd Adham Isa & M Firdaus Harum, ISSN:1992-8645, E-ISSN:1817-3195, pag. 4024, 2019.
- [3] Planning your mobile application, Segue Technologies Inc. – Ron Novak, Version 2.0. pag. 14, 2016.
- [4] How much does it cost to create an app, Kim Smith, GoodFirms, 2018
- [5] Design patterns: Elements of reusable object-oriented software, Erich Gamma, Richard Helm, Ralph Johnson, John Vissides. pag. 93-153, 1994.
- [6] Pattern-oriented software architecture a system of patterns, Frank Buschamann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal, Volume 1, pag. 25, 221, 383, 1996.
- [7] [Cristopher Alexander Bibliography. Paragraph. 4. 2012.](#)
- [8] Oracle Core J2EE Pattern – Data Access Object, Sun Microsystems. 2020.
- [9] [Postman, Postman official web site documentation. 2020.](#)
- [10] [Rapidapi, find and connect to thousands of APIs, Public and Premium, Iddo Gino CEO, 2020.](#)
- [11] [Postman Collection Scheme, JSON Structure, Postdot Technologies, 2020.](#)
- [12] [Retrofit, Android documentation to implement API calls, Square Inc. 2013.](#)
- [13] An introduction to Multi-Agent Systems, Michael Wooldridge, Second Edition. Chapter 5 pag. 85, Chapter 7 pag. 131. 2009.
- [14] Developing Multi-Agent Systems with JADE, Fabio Bellifemine, Giovanni Caire & Dominic Greenwood. Chapter 3 pag. 29-42, 2007.
- [15] Principles and practice of Multi-Agent Systems, Nirmal Desai, Alan Liu and Michael Winikoff, International conference, 13th edition, Chapter 1 pag. 11, PRIMA 2010.
- [16] JADE programmer's guide, Fabio Bellifemine, Giovanni Caire, Tiziana Trucco (TILAB S.p.A., formerly CSELT), Giovanni Rimassa (University of Parma), JADE. 4.0, Chapter 3 pag 6-22, 2002.
- [17] John Langshaw Austin, How to do things with words, Hermeneus, Revista de traducción e interpretación, pag. 483-508, 2018.
- [18] [Postman application download, Postman Inc. 2020.](#)
- [19] [FIPA official website, IEEE Foundation for Intelligent Physical Agents, 2020.](#)
- [20] [JADE TILAB official website, Telecom Italia SpA. 2020.](#)
- [21] [Oracle Technology Network, Java platform standard edition, Sun Microsystems, 2020.](#)
- [22] [Eclipse IDE official website, Download, Eclipse Foundation Inc. 2020.](#)
- [23] [Apache Maven Project official website, The Apache Software Foundation, 2020.](#)
- [24] [JADE TILAB Maven dependency official website, Telecom Italia SpA. 2020.](#)
- [25] [Multi-Agent Systems in mobile application development, Resources used in this TFM stored in Google Drive for free access, Ivan Alatorre, 2020.](#)