



**University of Camerino**

---

**SCHOOL OF SCIENCES AND TECHNOLOGIES**

Master degree in Computer Science (LM-18)

## **Wine Suggestion System**

KEBI - Project

Group Members

**Filippo Lampa - 124079**

**Francesco Moschella - 122435**

Supervisors

**Prof. Dr. Knut Hinkelmann**

**Prof. Dr. Holger Wache**

---

A.A. 2022/2023



# Contents

<b>1</b>	<b>Introduction</b>	<b>5</b>
1.1	Project Description . . . . .	5
1.2	Project Tasks . . . . .	7
<b>2</b>	<b>System Design</b>	<b>9</b>
2.1	Input Format . . . . .	9
2.2	Output Format . . . . .	10
2.3	After First Review . . . . .	11
2.3.1	Input Format . . . . .	11
2.3.2	Output Format . . . . .	11
<b>3</b>	<b>Knowledge Based Solution</b>	<b>15</b>
3.1	Decision Tables . . . . .	15
3.1.1	After First Review . . . . .	16
3.2	Prolog Software . . . . .	18
3.2.1	After First Review . . . . .	20
3.3	Knowledge Graph . . . . .	22
3.3.1	After first review . . . . .	23
3.4	Ordering Reccomendations . . . . .	25
3.4.1	After first review . . . . .	26
<b>4</b>	<b>Graphical Model</b>	<b>29</b>
4.1	Metamodelling . . . . .	29
4.2	Modelling . . . . .	30
4.3	After First Review . . . . .	33
<b>5</b>	<b>Conclusions</b>	<b>39</b>
5.1	Filippo Lampa Conclusions and Preferences . . . . .	39
5.1.1	DMN Tables . . . . .	40
5.1.2	Prolog . . . . .	40
5.1.3	Knowledge Graphs . . . . .	40
5.1.4	Comparison . . . . .	41
5.1.5	After First Review . . . . .	41
5.2	Francesco Moschella Conclusion and Preferences . . . . .	43
5.2.1	Decision Tables . . . . .	43

5.2.2	Prolog Implementation . . . . .	44
5.2.3	Knowledge Graphs . . . . .	44
5.2.4	Comparison . . . . .	45
5.2.5	<b>After First Review</b> . . . . .	45

# 1. Introduction

This document aims to describe the process followed to complete the Knowledge Engineering and Business Intelligence project. In Section 1.1 the description of the system to be realised will be reported, while the tasks to accomplish will be defined in Section 1.2.

Following, we will explore the System Design (Chapter 2) phase, the implemented Knowledge Based Solutions (Chapter 3) and finally the Graphical Model (Chapter 4). Finally, we draw some Conclusions (Chapter 5) on the project and present our personal points of view on the presented solutions.

All the solutions developed for this project and discussed within the following chapters can be found at <https://github.com/HarlockOfficial/Knowledge-Engineering-and-Business-Intelligence/tree/main/Exam>.

## 1.1 Project Description

With COVID-19 many restaurants have their menus digitised. Guests can scan a QR code and have the menu presented on their smartphones. A disadvantage is that the screen is very small, and it is difficult to get an overview. This is especially true if a restaurant offers a big wine menu.

Usually customers have preferences. Some prefer wine only from a specific country, e.g. Italy, or a specific region<sup>1</sup> like the Lombardy. Others exclude or prefer specific grapes<sup>2</sup>, e.g. some don't like Pinot Noir. Some wines are made out of several grapes. Most people are not wine experts and would like to select their wine by describing the taste (i.e. dry/not-dry, tannin/less-tannin).

But a very prominent decision influencer is the meal<sup>3</sup>. Red wine usually is offered to meat dishes; white wine usually to fish. But there are exceptions<sup>4</sup>, e.g. white wine with chicken.

The objective of the project is to represent the knowledge about wines. Menus and guest preferences are needed to support the selection process (i.e. they are input). Create a system that allows to select those wines that fit the guest preferences and the menu.

---

<sup>1</sup>[https://en.wikipedia.org/wiki/List\\_of\\_wine-producing\\_regions](https://en.wikipedia.org/wiki/List_of_wine-producing_regions)

<sup>2</sup>[https://en.wikipedia.org/wiki/List\\_of\\_grape\\_varieties](https://en.wikipedia.org/wiki/List_of_grape_varieties)

<sup>3</sup><https://winefolly.com/wine-pairing/getting-started-with-food-and-wine-pairing>

<sup>4</sup><https://media.winefolly.com/food-and-wine-poster.jpg>

The knowledge base shall contain information about typical wines (of an international restaurant) with wines from different regions and countries. For the taste, the grapes and the meals we focused on five major representatives. As for the Taste, we decided to allow the user to filter their favourite wine using the following flavour selectors:

- Bold
- Dry
- Fruity
- Savory
- Tannin

For each flavour, the user can decide on an amount from 1 to 5.

As for the Grapes, we allow the user to either include or exclude grapes from the following list:

- Cabernet Franc
- Merlot
- Pinot Noire
- Syrah
- Viura
- Chardonnay

The same goes for Meals, the user can either include or exclude a dish from the following list:

- Crab and Parmesan Stuffed Shrimp
- Turkey and Maitake Mushroom Risotto
- Beef Meatballs with Mushroom Sauce
- Caprese Salad
- Grilled Salmon
- Grilled Steak

Is also possible for the user to either include or exclude countries and regions from which the wine comes.

## **1.2 Project Tasks**

The following are the tasks to tackle for completing the Knowledge Engineering and Business Intelligence projects:

1. Define input and output of the Knowledge-Based system
2. Create different Knowledge-Based solutions based on
  - (a) Decision Tables
  - (b) Prolog
  - (c) Knowledge Graph
3. Design a graphical modelling language, which allows a chef to represent meals and wines in a graphical way, such that it contains all information relevant for the customers to select according to their preferences.



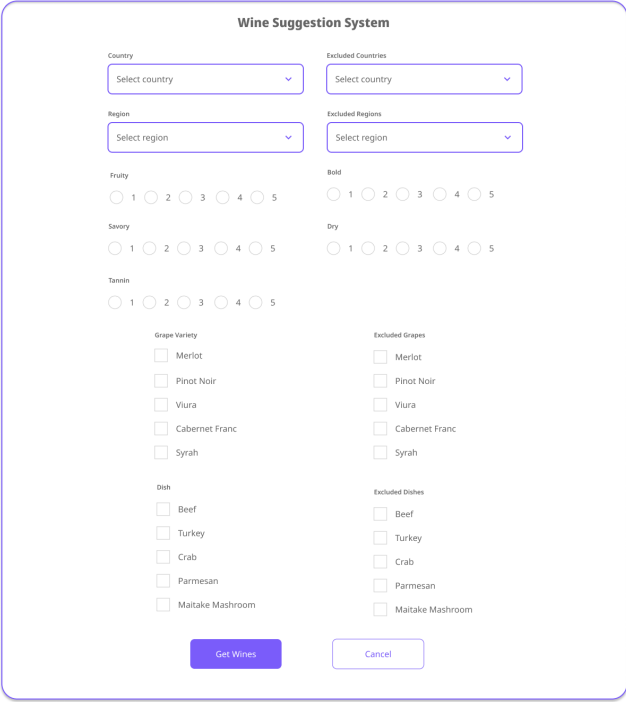


## 2. System Design

In this Chapter, we will explore our idea about the data input and output that the system will analyse

### 2.1 Input Format

Based on the already presented homework and on the exercises completed during the several lectures, we came up with our idea of an input form on which we will base the rest of this project. Following a picture representing our idea of input form is shown (Figure 2.1). The form is supposed to be filled out by the customer in order to help it get the best wine.



The form is titled "Wine Suggestion System" and is organized into several sections. At the top, there are two dropdown menus for "Country" and "Excluded Countries", both with "Select country" as the placeholder text. Below these are two more dropdown menus for "Region" and "Excluded Regions", both with "Select region" as the placeholder text. The next section contains five rows of radio button scales, each with a label and a scale from 1 to 5: "Fruity", "Savory", "Tannin", "Bold", and "Dry". Each scale has five radio buttons labeled 1, 2, 3, 4, and 5. The final section is divided into two columns. The left column is titled "Grape Variety" and "Dish", and the right column is titled "Excluded Grapes" and "Excluded Dishes". Each of these four sub-sections contains a list of items with checkboxes: Merlot, Pinot Noir, Viura, Cabernet Franc, Syrah for grapes; and Beef, Turkey, Crab, Parmesan, Maitake Mushroom for dishes. At the bottom of the form, there are two buttons: "Get Wines" and "Cancel".

Figure 2.1: Example Input form for the customer

In this form, presented in Figure 2.1, we can distinguish four groups of rows, country and region, tastes, grapes variety and dishes. Except for the taste-related group, all the rows are divided into two columns, the left one that allows the user to select elements to take into account and the right one that allows to mark elements to avoid. In the tastes group, we have all the tastes, the user can choose a value between 1 and 5 for

each of them, with 1 meaning the absence (or the wine with as less as possible) of the specified taste, and 5 meaning the wine with the strongest presence of that taste. In each field of each group, is also possible for the user to not provide an answer, in that case, we assume that the specified parameter does not matter and should not be used to filter out wines.

## 2.2 Output Format

The purpose of the project is to obtain the most suitable wine to match a specified meal in a restaurant. We are also aware that the best wine from a mathematical point of view might not be the best choice for the taste of a customer. Therefore we decided to provide as output a list containing the most suitable wines that match the user preferences with a certain tolerance. The list will be ordered and will provide as the first result the most appropriate match, followed by the wines that seem to match most of the user's requirements and preferences up to a certain threshold. Following we present a hypothetical case scenario, with a filled-out form and the related wine list that we ought to get from the solution.

**Wine Suggestion System**

Country:  Excluded Countries:

Region:  Excluded Regions:

Fruity: ☐ 1 ☐ 2 ☐ 3 ☒ 4 ☐ 5

Bold: ☐ 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5

Savory: ☐ 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5

Dry: ☐ 1 ☒ 2 ☐ 3 ☐ 4 ☐ 5

Tannin: ☐ 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5

Grape Variety:

- ☐ Merlot
- ☐ Pinot Noir
- ☐ Viura
- ☐ Cabernet Franc
- ☐ Syrah

Excluded Grapes:

- ☐ Merlot
- ☐ Pinot Noir
- ☐ Viura
- ☒ Cabernet Franc
- ☐ Syrah

Dish:

- ☒ Beef
- ☐ Turkey
- ☐ Crab
- ☒ Parmesan
- ☐ Maitake Mushroom

Excluded Dishes:

- ☐ Beef
- ☐ Turkey
- ☐ Crab
- ☐ Parmesan
- ☐ Maitake Mushroom

Figure 2.2: Sample filled out form

Figure 2.2 presents a possible selection made by a customer. Here we find the preferred wine to be originated from either Italy or France, with no specified region. Also, the wine should be highly Fruity (4) and not Dry (2). The customer does not like the wines from the Champagne region nor wines containing the Cabernet Franc grapes. The customer wants to drink this wine while eating Parmesan and beef.

Figure 2.3 presents the output provided by the parameters provided by the form in Figure 2.2. We can notice the presence of the Syrah wine, listed as the most suitable wine that follows the user's choices followed by other wines that follow the customer's choices but with less strictness.

The image shows a web application window titled "Wine Suggestion System". Inside the window, there is a section labeled "Best Choice" with a single entry: "1) Syrah". Below this is a section labeled "Alternatives" with four entries: "2) Rhone", "3) Supertuscan", and "4) Bordeaux". At the bottom of the window is a blue button labeled "Close".

Figure 2.3: Sample output matching the form in Figure 2.2

## 2.3 After First Review

In the second version of the project, the focus has been moved from single ingredients to dishes. As can be seen in the figures reported below, the input form will now offer a choice of six different dishes the user can choose from. Also, a new grape variety, namely the Chardonnay, has been added to the varieties available.

### 2.3.1 Input Format

Using the knowledge from the previous version of the project, the input format was adapted and slightly modified to include the changed fields. Figure 2.4 shows an empty form that the user can fill out to receive help in deciding the best wine to match his/her meal.

### 2.3.2 Output Format

Figure 2.5 shows an example of the new form filled out by a user. Here, the user would like a wine from France, with a dryness of 4 out of 5. He doesn't specify any particular grape variety, but he knows this wine should be a good match for a Caprese Salad.

The resulting output would be defined as a perfect match and some alternatives which could still suit the meal but not at its best. In this example, the best wine to match the salad turns out to be a Chardonnay, with its crispness. Some other alternatives, taking all the inputs of the user into account, could be Merlot, Northwest or Rose.

Wine Suggestion System

Country

Select country

Excluded Countries

Select country

Region

Select region

Excluded Regions

Select region

Fruity

1

2

3

4

5

Bold

1

2

3

4

5

Savory

1

2

3

4

5

Dry

1

2

3

4

5

Tannin

1

2

3

4

5

Grape Variety

Merlot

Pinot Noir

Viura

Cabernet Franc

Syrah

Super Tuscan

Rose

Bordeaux

Rhone

Northwest

Chardonnay

Excluded Grapes

Merlot

Pinot Noir

Viura

Cabernet Franc

Syrah

Super Tuscan

Rose

Bordeaux

Rhone

Northwest

Chardonnay

Dish

Crab and Parmesan Stuffed Shrimp

Turkey and Maitake Mushroom Risotto

Beef Meatballs with Mushroom Sauce

Caprese Salad

Grilled Salmon

Grilled Steak

Excluded Dishes

Crab and Parmesan Stuffed Shrimp

Turkey and Maitake Mushroom Risotto

Beef Meatballs with Mushroom Sauce

Caprese Salad

Grilled Salmon

Grilled Steak

Get Wines

Cancel

Figure 2.4: Sample Input form for the customer

Wine Suggestion System

Country

France

Excluded Countries

Select country

Region

Select region

Excluded Regions

Select region

Fruity

☐ 1
☐ 2
☐ 3
☐ 4
☐ 5

Bold

☐ 1
☐ 2
☐ 3
☐ 4
☐ 5

Severy

☐ 1
☐ 2
☐ 3
☐ 4
☐ 5

Dry

☐ 1
☐ 2
☐ 3
☒ 4
☐ 5

Tannin

☐ 1
☐ 2
☐ 3
☐ 4
☐ 5

Grape Variety

☐ Merlot
☐ Pinot Noir
☐ Viura
☐ Cabernet Franc
☐ Syrah
☐ Super Tuscan
☐ Rose
☐ Bordeaux
☐ Rhone
☐ Northwest
☐ Chardonnay

Excluded Grapes

☐ Merlot
☐ Pinot Noir
☐ Viura
☐ Cabernet Franc
☐ Syrah
☐ Super Tuscan
☐ Rose
☐ Bordeaux
☐ Rhone
☐ Northwest
☐ Chardonnay

Dish

☐ Crab and Parmesan Stuffed Shrimp
☐ Turkey and Maitake Mushroom Risotto
☐ Beef Meatballs with Mushroom Sauce
☒ Caprese Salad
☐ Grilled Salmon
☐ Grilled Steak

Excluded Dishes

☐ Crab and Parmesan Stuffed Shrimp
☐ Turkey and Maitake Mushroom Risotto
☐ Beef Meatballs with Mushroom Sauce
☐ Caprese Salad
☐ Grilled Salmon
☐ Grilled Steak

Get Wines

Cancel

Figure 2.5: Sample filled out form

Wine Suggestion System

Best Choice

1) Chardonnay

Alternatives

2) Merlot

3) Northwest

4) Rose

Close

Figure 2.6: Sample output matching the form in Figure 2.5



## 3. Knowledge Based Solution

Relying on the input form and format previously defined, we can query our knowledge base in three different ways. Firstly, by using Decision Tables (Section 3.1) to match user preferences and filter out undesired wine or grape qualities. With Prolog (Section 3.2), we can extend the logic provided by the Decision Tables by applying the functionalities provided by a programming language. This allows the users to manipulate data and perform even more actions. Finally, using Knowledge Graphs (Section 3.3), we can store our data in a convenient format and query them as if they were stored in a database. Following, in this chapter, we will discuss each of the three solutions, their main advantages and drawbacks, after which, we will propose a solution regarding the recommended wine ordering (Section 3.4).

### 3.1 Decision Tables

After defining our knowledge base and the user input form. The decision tables were the first step towards our solution. We initially followed a brute force approach by creating a unique table containing all records and combinations. After that, since the result we were about to obtain was a gigantic table that would have been complicated to read, understand and eventually fix, we decided to apply a more scalable solution. Therefore, we split the table into smaller ones, one for each decision and filter we had to take into account and procedurally filter out unwanted wines.

Specifically, in our Decision Table solution, we have five tables:

1. Filter By Dish
2. Filter By Nation
3. Filter By Region
4. Filter By Taste
5. Wine

These tables follow each other in the order they are listed. Each time, some wines are removed from the list. After the last table, a list containing all the wines that can be associated with the meal is retrieved. For each parameter, is possible to express the absence of a preference, usually, this is accomplished by not providing input, in the specific case of the taste, a placeholder parameter is required. Each of these tables is further described below.

**Filter By Dish** The initial table uses the dishes that a user provides as input for both categories, dishes that the user is going to eat and dishes that the user wants to exclude, and produces a list of wines that are related to the specified meal.

**Filter By Nation** Using the previous table output with the nations specified by the user, this table makes ulterior filtering, thus lowering the list of matching wines.

**Filter By Region** Using the output from the Nation table, along with the specified user region preferences, this table filters the input list of wines to produce a subset of the latter and provides its result to the following table.

**Filter By Taste** This table is used to match the taste preferences of a user with the available wines. Requires as input the list of wines filtered from the previous step. Also, a value between 1 and 5 (both included) is required for each of the taste parameters:

- Fruitiness
- Boldness
- Savoriness
- Dryness
- Tannity

If the user does not want to express a preference for the specified parameters, it is possible to use the value “-1”. After this filter, the biggest among the five, we reach the final table.

**Wine** This table uses the list of wines provided as input, along with the grapes that users want and do not want in their wine, to match the perfect list of wines.

This approach sure was simple to implement, simplicity is one of the core points in DMNs, but because of this simplicity, some problems occur. Firstly, the whole work is quite mechanical. Also, eventual changes, updates and edits in the list of wines or the conditions, requires a careful update of all the tables, even a minor error might completely change the outcome. Finally, in this approach, expressing concepts like ranges is not trivial, for example, if a user wants the Tannity to be between 2 and 4, in the current DMN is necessary to search three times, each time changing the Tannity level, and if we want to change the tables to allow for ranges, as said before, the change is far from trivial.

### 3.1.1 After First Review

Thanks to the review provided, we got a deeper overview of the proposed solution. Using the pieces of advice in the review from the professors, we were able to extend the initial version and improve it to obtain a better solution. Using the professor's pieces of advice, we noticed that, the previously proposed solution was based on the concept of pipeline, where data were filtered by subsequent tables by feeding one table with the output from the previous one. That solution was simple to implement and



straightforward, but as a major drawback, we had to retain the information regarding each of the wines that were not filtered out during the previous steps. This solution will have some major problems related to the scaling, especially if the amount of wine types increases. Also, it is not common to retain information about previous steps, and using our first column in the DMN's tables introduced unnecessary data redundancy. Finally, we have not sufficiently tested the solution. In fact, the execution of our software turned out to be more complex than we expected. Using those tips and checking again our work, we came up with a different solution, whose overview is shown in Figure 3.1. This time, we tried to remove the pipeline-like architecture and to reduce the size of

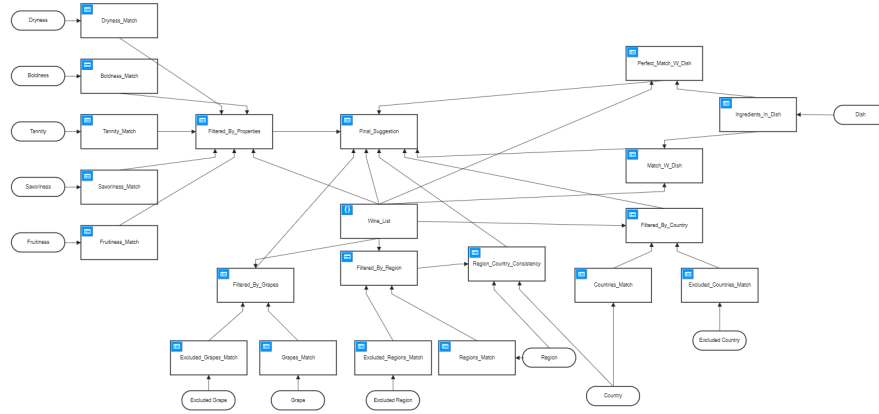


Figure 3.1: New solution based on decision tables

the tables by exploring and exploiting more the functions offered by the FEEL language<sup>1</sup> to operate on lists, which have been neglected in the previous version of the project. We introduced a literal expression containing the full list of possible wines, which could be manipulated and filtered by the several tables according to some input preferences such as the country of the wine, the features, grape, region, and, most important, the dish. Moreover, the list can also be filtered through the negation (exclusion) of the aforesaid preferences, except for the meal which, since the goal is to select wines for a meal from the menu as specified in the project description, we thought was inane to exclude within this kind of solution. The pattern is almost the same for every input included in our solution:

1. The input preference is fed into a first table (\*PREFERENCE\*\_Match) matching it to one or more wines through a collect hit policy
2. The exclude input preference is fed into a different table (Excluded\_\*PREFERENCE\*\_Match) matching it to one or more wines through a collect hit policy
3. The output lists of wines obtained from the two tables are inputted into a single table (Filtered\_By\_\*PREFERENCE\*) where, through FEEL functions, are merged into a single list by getting the intersection of the two.
4. The merged list is passed to the Final\_Suggestion table which collects all the lists filtered by the different preferences and takes care of merging all of them into two

<sup>1</sup>Camunda Feel Documentation Reference: <https://docs.camunda.org/manual/7.19/reference/dmn/feel/>

final output lists, the first one containing the perfect matching wines for the dish, and the second one including just the matching ones.

A first exception to this flow is the Region\_Country\_Consistency table which has the purpose of prevent the user selecting not correlating countries and regions by propagating the "region\_country\_check\_failed" string to the output in that case. A second exception is the filtering on the dish, where the selected dish is first split into ingredients, which are then inputted into the two tables which, by matching ingredients with wines based on the information on <https://winefolly.com/wine-pairing/simple-food-and-wine-pairing/>, are meant to find the match and the perfect match, namely Perfect\_Match\_W\_Dish and Match\_W\_Dish. Moving to the testing, in addition to the standard Camunda DMN Simulator <sup>2</sup> where we tried to check all the possible combinations and boundary cases, we found an official community-driven project<sup>3</sup>, also mentioned in the Camunda DMN documentation <sup>4</sup>. To use this tool, we had to write the configuration files needed to correctly test the DMNs, and then run the Docker image while providing the created config files. Since the whole process could have been tedious and long, and also to avoid introducing errors in the created files, we decided to implement a Python script to extract all the possible values, find all the combinations, predict the expected results given our rules and test the tables. Knowing that others may benefit from our Python script, we tried to develop it as modular and generic as possible, by only constraining the number of output columns to one while letting all the other parameters be as generic as possible. As of now, the script should allow the generation of valid configuration files for several cases. As presented before, the only problems currently known are the extraction of the wrong rule index and the generation of input values that are not context-aware. We are conscious that the script may have some hidden constraints that we could not have thought of, and we consider it as an initial work.

## 3.2 Prolog Software

Having our knowledge base defined and some initial work on the Decision tables, we implemented a software equivalent to the tables in Prolog. Here, having all the means of a programming language, we further expanded the functionalities provided by the DMN diagram and allowed the user to perform a thresholded search in our knowledge base. For simplicity, maintainability and extensibility, we divided the software into two files. One contains all the knowledge base, while the other includes all the horn clauses.

**Knowledge Base** This file has been organised in 9 "blocks" and in triples as much as possible, the reason behind the use of triples is the later simplicity in accessing the data and transforming the Prolog into a Knowledge Graph. Each block contains some pieces of information. Firstly we have the block containing all the countries. Followed by the block where regions are listed and connected to a country. After that is possible to find the block where grapes are defined. Right after grapes, the list of dishes is determined. Finally, the last five blocks are used to respectively match each wine with the grape that

---

<sup>2</sup>DMN Simulator Reference Page: <https://consulting.camunda.com/dmn-simulator/>

<sup>3</sup>Link to the GitHub repository of the project <https://github.com/camunda-community-hub/camunda-dmn-tester>

<sup>4</sup>Link to the documentation <https://camunda.com/blog/2021/02/testing-dmn-tables-automatically/>

contains, the region from which the wine comes or where is produced, the taste of the mapping between the wine and fruitness, boldness, savoriness, dryness and tannicity levels, the matching dish and finally the wine name. This file as is, makes simple the maintenance of the list of wines, countries, regions, dishes, tastes and matches. Also, to add a single wine, only a few lines have to be added to the respective block.

**Horn Clauses** This file contains all the rules used to provide the desired matching wines list to the user. There are only two functions directed to the user. The clause “query(...)” that requires as an input:

- a list of countries or “[A]” if there are no preferences
- a list of regions or “[B]” if there are no preferences
- a list of grapes or “[C]” if there are no preferences
- the value of the fruitness level or “D” if there are no preferences
- the value of the boldness level or “E” if there are no preferences
- the value of the savoriness level or “F” if there are no preferences
- the value of the dryness level or “G” if there are no preferences
- the value of the tannicity level or “H” if there are no preferences
- a list of dishes or “[I]” if there are no preferences
- a list of countries to exclude or “[J]” if there are no preferences
- a list of regions to exclude or “[K]” if there are no preferences
- a list of grapes to exclude or “[L]” if there are no preferences
- a list of dishes to exclude or “[M]” if there are no preferences

The other clause is called “softened\_query(...)” and requires as input all the previously mentioned parameters and adds a threshold parameter. This parameter defines a circular range around each value provided as input for the taste levels. As a side note, in this clause, is mandatory to specify a preference for each parameter regarding the taste.

The other clauses present in the file, although callable by the user, should be almost meaningless by themselves.

Those clauses are mainly used to verify the input data and to find the matching values for the wine pairings, origin and name.

In this task, we faced the problem of testing and learning a new programming language while working with it. It was a nice experience and the previously learned programming languages helped in rapidly understanding this one. We found the absence of an IDE to negatively impact the software developer experience, also the testing was initially slow because we only relied on the provided Prolog website. After some time, we decided to install a Prolog interpreter on our machines and development become much faster and smoother.

### 3.2.1 After First Review

The part of the review related to this section pointed out our relation “wineusedwith” which mapped a specific wine to a group of dishes. We thought about it, and we found an alternative solution which, although was not implemented, could be a transitive relation between a dish and its features/properties first, and from the dish properties to the wine then. Instead, we developed a relation between a dish and its ingredients and defined the matching wines using the dish ingredients. Thanks to the review, we decided to modify the mapping, add dishes composed of ingredients, and convert the relation to a connection between the wine and the food ingredients.

As defined in the other sections, this allows the solution to be more scalable and flexible. We are aware that the map between ingredients and wine to create a perfect pairing may not be exhaustive. Indeed, is important to consider the dish itself along with the organoleptic features of each of its ingredients to perfectly pair a wine. For this project we decided to simplify this aspect and accepted a direct relation between an ingredient and a wine, not considering the impact of the eventual cooking process and the change in the ingredient features.

After changing the knowledge base, we fixed the old horn clauses to obtain valid results. We then continued by defining a new horn clause that would have taken advantage of the new predicate “weakwineusedwith” that is now used to map weak relations between a wine and an ingredient, while the old “wineusedwith” now maps the strong one. Using both of these horn clauses, we created two new clause (namely “query2”, and “query3”) that retrieves the strongly and weakly suggested wines using two slightly different approaches.

The “query2” function takes advantage of the Prolog internal function “setof” to extract a complete set of valid results. Unfortunately, due to the nature of Prolog, which extracts one result for each of the available options, in many cases only changing one parameter and maintaining the other results identical, the result set is still big. Therefore, when running the function, in addition to the time spent waiting for an answer, the same result, where only a value different from the suggested wine column changed, was returned before showing the second suggested wine. While these parameters might be interesting to have more information on the wine, the obtained information seemed to us of no use to the final customer and had to be removed. To overcome this problem, we started by taking advantage of the Prolog lambda functions provided by the “yall” library. Using a lambda function we were able to filter out unwanted columns and have a cleaner output composed only of the two columns representing the strong and weak suggestions. After lowering the column number we noticed that the same output was repeated multiple times, this happened because even if the other columns were hidden, the data were still extracted and considered as meaningful for the row creation. To lower the number of rows we took advantage of the distinct function present in the “solution\_sequences” library. By using the “distinct” function we were able to limit the number of rows by considering the couple “Strongly suggested wine - Weakly suggested wine” as unique. Is important to notice that the use of the “set\_of” function adds a high initial wait time because the results have to be fetched completely, added to a set, and then only one pair at a time will be provided. This may lead to timeout results or extremely long waits for broad searches that do not have a lot of parameters set.

On the other hand, the function “query3” uses “distinct” instead of “set\_of” adopting a more Prolog-like approach and retrieving only a pair Strongly suggested Wine - Weakly suggested Wine at a time. Adopting the “distinct” search proved to be faster

from the very initial results. Indeed there was almost no initial wait time and the first results were returned almost immediately. The main problem was the repetition of data that in “query2” was solved by applying “set\_of”. To overcome this problem we initially wrapped the call in a lambda expression that was also wrapped in a “distinct” call, we later simplified by taking advantage of the Prolog auxiliary variable syntax (eg. “\_EXAMPLE” instead of “EXAMPLE”), this kind of variables are correctly interpreted by the function but not considered as part of the function output. With the “distinct”-related solution, we experienced a higher wait time with respect to the standard “?-query3(...)” call, but were able to extract all unique results (up to a certain degree of freedom).

Lastly we applied the same “distinct” logic to the “query2(...)” call. In both cases, we experienced a higher wait time with respect to the “simple” call and a lower wait time with respect to the lambda-oriented solution that was applied to filter the output data.

As anticipated, we were able to reduce the amount of lines up to a certain degree of freedom. Unfortunately due to the nature of Prolog, it was impossible to consider the two columns as separated and we still experience some repeated output. Specifically, the output set is in the form<sup>5</sup>:

$$\begin{array}{c}
 SW_1 - WW_1 \\
 SW_1 - WW_2 \\
 \dots - \dots \\
 SW_1 - WW_m \\
 SW_2 - WW_1 \\
 SW_2 - WW_2 \\
 \dots - \dots \\
 SW_2 - WW_m \\
 \dots - \dots \\
 SW_n - WW_1 \\
 SW_n - WW_2 \\
 \dots - \dots \\
 SW_n - WW_m
 \end{array}$$

Instead of the expected output of:

$$\begin{array}{c}
 SW_1, SW_2, \dots, SW_n \\
 WW_1, WW_2, \dots, WW_m
 \end{array}$$

The main difference here is the number of results equal to  $n + m$  in the expected behaviour versus a  $n * m$  in our solution.

Even if the resulting number may be acceptable for a small amount of data, is important to consider that each of the results has to be retrieved separately or may

<sup>5</sup>With SW\_x we denote the “Strongly suggested Wine x” and with WW\_y the “Weakly suggested Wine y”

require some fetch time and given this difference in output sizes, the wait time is way longer. We also noticed, during our experiments, that the wait time grows by an exponential factor, therefore the wait time to extract the  $x^{th}$  element is always at least as long as all the time that was waited to extract the previous  $x - 1$  elements. It is also important to consider that the user does not know the “n” value a priori, therefore is impossible to stop on  $SW_n - WW_1$  and is necessary to extract all data until the function fails.

It is also important to consider the actual purpose of this software, which is finding the most suitable wine for a user, therefore, in the average case it will not be necessary to reach the very last combination, and the user will be able to choose a wine before the wait time will be excessively long, therefore we do not consider by the wait times as extremely problematic.

### 3.3 Knowledge Graph

Last but not least, part of this project’s tasks was focused on the knowledge graphs. These graphs map the Prolog knowledge base to a graph where more information on the data may be discovered by using the connections among information. To accomplish this subtask, we used the Protégé software by Stanford<sup>6</sup>. This task’s result is a single RDF file. We can identify four major parts:

- Classes
- Object Properties
- Data Properties
- Individuals

**Classes** The classes can be mapped directly from Object Oriented Programming, indeed, the classes were the types used to initialise objects and provide them functionalities and parameters. In our project, we distinguish five sibling classes.

- Country
- Dish
- Grape
- Region
- Wine

Each of which defines a core component of our model.

**Object Properties** The object properties can be mapped to the functions provided by each object. In our model we defined four object properties.

- <region> belongs to <country>
- <wine> belongs to region <region>

---

<sup>6</sup>The tool website can be found at the following link <https://protege.stanford.edu/>

- <wine> contains grape <grape>
- <wine> is paired with <dish>

In the previous list, we mentioned the classes between angular brackets.

As is possible to notice, these kinds of property effectively relate two classes of objects one to the other.

**Data Properties** The data properties are parameters of an object and define indeed some values directly related to the object itself. In our project, we identified six data properties.

- name
- bold
- dry
- fruity
- savoury
- tannin

Except for the first one, all the properties belong to the wine. The first one, belongs to each class since each class can have a personal name. Also, providing names this way lowers the coupling between the “Individuals” names and their actual names.

**Individuals** The individuals can be seen as the instances of the classes of OOP. we have a discrete amount of individuals, therefore we are not going to list them here. We have a total of three countries, four regions for each country, five dishes and ten wines. After defining all these entities, that as for the DMN was quite mechanical work, we defined a way to query the data and reported a SPARQL Query. This query can be used in two ways, firstly to list all the entries in our dataset, and with a minor change to filter the dataset and obtain wines related to a user’s taste.

A nice feature provided by the Protégé software is the Reasoner, this tool uses the provided knowledge base, creates a graph and infers other information. Sadly not all the inferred data can be used, for example, in our project, as previously presented, we had the parameter name, this parameter was part of each class and therefore part of each object. After starting the reasoner, we found that following its logic, since the wine had a name, and the country had a name, an individual of type wine was also typed as a country. Of course, this behaviour has to be prevented and may lead to unwanted results or problems.

### 3.3.1 After first review

The first change applied to the previous version, as done for the other solutions, is the introduction of strong and weak pairings, and the distinction between ingredient and dish. Moreover, thanks to the review, we checked again our solution, effectively noticing that the differentiation between white and red wine was meaningful. This

separation was not previously present, because we considered all wines equal but, after checking online, we discovered major differences between the taste and the organoleptic features of a wine also related to its colour<sup>7</sup>. For this reason, we decided to reflect this difference in a subclass of wine (Figure 3.2).

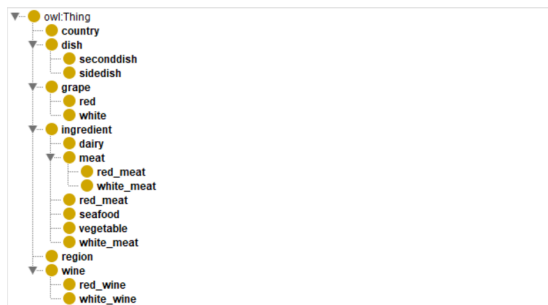


Figure 3.2: Protégé hierarchy with the new subclasses

This division was introduced for dishes, ingredients and grapes as well. In this new version, there is now a distinction between a second course and a side dish. Moreover, an ingredient can be dairy, seafood, vegetable or meet, with the latter having an additional distinction between red meet and white meet. Finally, the grapes are separated into red and white ones. We think that an eventual division for countries and regions is possible, for example by dividing for continents, but meaningless with respect to the objectives of this project. If, in the future, it will be necessary to consider also sparkling wines, like prosecco or champagne, and rosé wines, it will be possible to easily add these subcategories and extend the expressiveness of the project. Specifically, the rosé will be a subcategory of wine while the sparkling/non-sparkling will be subcategories of each of the wine types red, white and rosé. The same goes for eventual first courses or desserts and for more specific varieties of red and white grapes such, for example, moon drop, sultana, or one of the approximately 10'000 varieties<sup>8</sup> currently present in the world. In the first review, we have also been suggested to create SWRL rules and allow the reasoner to infer knowledge using them. Since our solution was not extremely complicated, we had already mapped all the properties to the related instances. Given the suggestion, we tried to revise the solution and ended up adding more properties, to adequately map in a bidirectional way the ones previously defined. This was accomplished by defining the new properties and marking those new properties as the inverse of the already present ones. Other than this, we noticed that a mapping between a country and their produced wines could have been worth being introduced and allowed the solution to scale better with an increasing number of wines. This mapping also allows simpler queries, for example, a user can retrieve the wines that belong to a certain country with a simple query or just by using Protégé, starting the reasoner and examining the inferred properties in the country individuals. To accomplish this last type of mapping, instead of manually matching countries and produced wines, we used an SWRL rule that takes into account the relations between a country and its regions and between a region and its produced wines. Having this knowledge, the rule adds the inference and maps countries to their produced wines. Two other rules have been added in this new version, namely the one to infer the wine matching to a dish without

<sup>7</sup>Article discussing the relation between food and drinks colour and perceived taste: <https://winefolly.com/tips/how-color-affects-your-perception-of-taste-of-wine>

<sup>8</sup>Source of the information on the amount of grape varieties: <https://www.winestyr.com/blog/wine/how-many-different-types-of-wine-grapes-are-there>



having to take care of the ingredients in first person, and the one to infer the perfect match. Some examples of queries can be found within the `query.md` file present in the folder relative to the task.

### 3.4 Ordering Recommendations

In our opinion, an important feature of a wine suggestion system is the possibility to order and rank results by matching criteria. The three kinds of knowledge solutions require the application of different approaches to obtain this result. Following, we will discuss possible approaches to apply when dealing with Decision Tables, Prolog and Knowledge Graphs.

**Decision Tables** In this kind of knowledge-based solution, applying a form of output ordering is not possible. Wines are decided with a Collect policy, therefore the output is treated as a set, and not as an orderable object. All rules are indeed applied at the same time to the whole input, therefore it is not possible to provide ordering of the suggested wines. On the other hand, it is possible to apply a form of weak suggestion. To apply a weak suggestion, we can let the user define a threshold, we can then use the threshold to filter by wine tastes. For this purpose, a new set of rules would need to be introduced. To be precise, a simple threshold of  $\pm 1$  would require, only to weakly suggest the taste, the initial rule containing the target values, 5 more rules in which one single value is changed by  $+1$  or  $-1$ , and finally all the combinations of these five rows, resulting in a huge amount of entries. Other than allowing for a threshold in these values, we could also allow the customer to specify for all fields on which a decision was made (for example the grape variety has to be only Merlot and Syrah) if that decision could be ignored. This opens a Pandora's box, resulting in a lot more rows necessary to either filter out a wine or retain it if the choice was marked as possible to ignore. Moreover, in this case, it is important to underline that strongly suggested wines and weakly suggested wines would end up in the same final unordered set, therefore not being distinguishable, at least if we stick to the single-column output. Trying to move to a double-column output, we would be able to distinguish strongly and weakly suggested wines while opening a higher set of problems. One way to face this situation may be by duplicating most of the tables and creating a pipeline that will only filter suggested wines and the other that will provide the weakly suggested ones. A second approach could be trying to filter both weak and strong recommendations altogether, leading to a huge amount of rows for each table.

**Prolog** Moving to Prolog, a real programming language, matters become simpler, we already provided a form of weak suggestion system indeed. As presented, we introduced the two clauses `query(...)` and `softened_query(...)`, where the latter is actually a softened version of the former. The softening only regards the strictness of the taste values but can be easily extended to also provide support for other parameters, such as the chosen dishes or countries. This change might require some more horn clauses and some testing, but the core idea would be to have a specific parameter in the clause, such as `CAN_SKIP_DISH` expecting a parameter of either 1 if the dish selection can be skipped or 0 if the dish selection has to be followed. After this, inside the horn clause, we should call some other clauses that will require as input the dish list and the parameters, then will either retain or clean up the provided input list. For the ordering,

we can rely on the Prolog interpreter. For the tests done within our local machines and in the online web service, results are already returned with a specific ordering, which is firstly based on the order present in the knowledge base, and then based on the rules that matched before. Indeed, improving our version of the softened query and allowing the threshold to increase during the query execution may allow the engine to return an ordered list of suggested wines that will be ordered following the strength on which the taste and parameters will match the user choices.

**Knowledge Graphs** Actually, when defining a Knowledge Graph we are defining a specific form of database, then the reasoner will apply logic to the database and allow us to extract knowledge. Since the query language used to retrieve information from the data in the Knowledge Graph is similar in a way to standard SQL, and knowing that SQL allows results ordering, we can also correctly assume that the SPARQL language allows to order results and query the data in many ways. Actually, applying strong and weak suggestions to this solution might result in simpler than expected. What we should do is create a specific query that retrieves all the results and adds to each resulting row a “special” identifier, the difference between the user choice and the result. This way the “special” column will hold a value representing the distance between the user choice and the actual result, we can then reverse-order the results following this column and we can also cut out the results with a value higher than a specified threshold.

This will not only allow the user to choose from a higher amount of wines but also can show what changes between its choice and the result, allowing a more aware decision of the wine.

### 3.4.1 After first review

As of now, all of our solutions provide the distinction between strong and weak suggestions. However, there could be some ways to reach an ordering of the output at a more detailed level, from the best match down to the worst one. Starting with Prolog, a possible way to obtain an ordered output exploiting only the operations provided by the language and avoiding the use of external scripts could be to pair every wine with a variable, an integer to be precise, which would be increased by one every time an input property specified by the user is satisfied by that specific wine. For example, with an input in the form of:

```
query ([A],[rioja],[C],3,E,4,G,1,[crab_parmesan_stuffed_shrimp],
[I],[J],[K],[L],OUT).
```

The integer corresponding to the Viura wine, once all the horn clauses have been checked, would have a value of 4, since the region, the fruitiness, the tanniness and the dish are the ones requested by the user. This would be possible thanks to the key-value pairs<sup>9</sup> offered by Prolog, which would come in handy to store the values corresponding to the several wines and to retrieve them in order to build the final sorted output. To execute this sorting, Prolog also offers the *keysort/2*<sup>10</sup> predicate which is specifically designed to work with the aforesaid key-value pairs.

---

<sup>9</sup>Official Pairs Documentation on SWI-Prolog: [https://www.swi-prolog.org/pldoc/doc/\\_SWI\\_/library/pairs.pl](https://www.swi-prolog.org/pldoc/doc/_SWI_/library/pairs.pl)

<sup>10</sup>Official Keysort Documentation on SWI-Prolog: <https://www.swi-prolog.org/pldoc/man?predicate=keysort/2>

The text above, even if is new, with respect to the past submission, was written before the new version of the assignment came out.

With the new requirements, we tried that approach and the result was not satisfying, the values were extracted in a non-Prolog-like way. It was necessary to extract lists of suggested wines and iteratively tag the results to the amount of matching parameters. This solution add a huge amount of delay since the extraction of all the results was computationally complicated. The results obtained were only partial and the function that computed them was excessively long and hard to maintain. Also, the “keysort” function provides as output a new list that cannot be bound, due to Prolog constraints (once a variable is bound to a value, the value cannot be changed), to an already existing list, this would have required an increased number of variables and parameters either in the function or as parameters passed to the function and may make its use non-intuitive.

In the Knowledge Graphs, on the other side, the first way to achieve the ordering could be using some machinery and scripting. By running a query multiple times and varying some or all the provided parameters, it is possible to extract a list containing all the preferred and suggested wines.

In the project we opted to implement the ordering with a SPARQL Query. The query retrieves all results and for each extracted wine associates a likelihood value, finally the results are sorted following this parameter in descending order.

More in general, for each solution, it is possible to order the results by concatenating first the results from the default queries, followed by the results obtained by weakening the query subtracting and adding 1 to each taste value, and subsequently attaching the results obtained with 2, 3 and 4, slowly enlarging the search field. In our wine-choosing system, the taste values range from 1 to 5, therefore, in the average case, 3 is the maximum value that has to be used for creating a circular range and obtaining all the wines present, while 4 is only necessary when edge values (1 and 5) are chosen. After modifying the taste values, by removing part of the input, resetting the taste values and repeating the process until no more parts of the input can be removed, is possible to obtain more results. In our opinion, changes in taste values may be less influential (for the average user) than changes to other input parameters, therefore the removal of input constraints is suggested only after reaching the maximum threshold range. As for the order to follow for removing limitations, we suggest removing, if present and in order, region, dish and/or nation and finally grapes. The reason for this is that a similar wine can be blended or created in the same country but in a different region and the taste may be similar, after this, the choice depends on the user, if the wine taste is more important than the pairing with the dish, we suggest to loosen the dish constraint, otherwise is better to remove the nation constraint. Lastly, it is possible to remove the grapes limitation, this choice has the most impact on the wine taste, therefore is suggested to do so only if the user did not find a valid wine with the procedure described above.

While for the Knowledge Graphs, we provided a query that allows the ordering and does not follow this approach, is possible to modify the query and give personalised weights to each change so that the ordering is more faithful, for example, each of the taste weight can be the absolute value of the difference between the actual value and the expected one.

As we mentioned above and in the dedicated subsection 3.2, the application of this approach using the default language is too complicated in Prolog, and it is suggested

to apply it using a higher-level language, such as Python/Java for desktop applications or JavaScript/PHP for web applications. All these languages support lists and sets and are able to query the knowledge base using the related functions, therefore obtaining a list of results and merging them in a saved list is much simpler than specifying a horn clause that accomplishes this operation.

As a proof of concept, we provide some Python scripts to run queries and try the mechanisms described in this chapter first-hand. Of course, these scripts have to be considered just as an initial work, indeed the full procedure has been implemented, but not fully tested, only in Prolog. Applying the full procedure to the SPARQL query would have required query manipulation. To effectively manipulate the input query we had to use or define a way to interpret it and this process might have required some time. Also, since providing an implemented version of the ordering was not mandatory, we thought that an initial and proof of concept work would have been a valid enough solution.

## 4. Graphical Model

Differently from the previous tasks, the graphical modelling of the system aims to be as human-readable as possible. The goal of a conceptual model is to provide a visual representation of a system or concept by depicting entities, their relationships, and the flow of information, to create a clear and intuitive representation that aids understanding and communication. The process of modelling can be split into two main parts: metamodelling, i.e. the definition of domain-specific concepts, which will be taken into account in Section 4.1 and the modelling phase where the previously defined concepts get instantiated, described in Section 4.2.

### 4.1 Metamodelling

This phase focuses on the construction of the general knowledge about the domain, which will subsequently allow the actual modelling of the system. This can be obtained through the definition of concepts, attributes and relationships, by exploiting the functionalities of the ADOxx Development Toolkit <sup>1</sup>. The first step was therefore to create those classes and relation classes along with their attributes within a new ADOxx library, namely the WineLib library, associated to a new user. Through the specification of a hierarchy among the classes, ADOxx provides inheritance mechanisms which make the definition of entities and attributes more straightforward, by using an OOP style. Out of fairness, no sub-classes have been created during the development of our project, since it seemed to be a stretch, actually not needed. The only super-class extended is the construct, which acts as the root of the class hierarchy. Following the structure used to construct the knowledge graph in Section 3.3, the classes that we can find within our library are the following:

- Wine
- Grape
- Region
- Country
- Dish

Additionally, to allow clear inter-model referencing, three more classes have been added:

---

<sup>1</sup>The official documentation of the tool can be found at <https://www.adoxx.org/live/introduction-to-adoxx>

- RegionWineRef
- WineGrapeRef
- WineDishRef

Their goal is to redirect the user to a different model when clicked, defining a flow between the models and allowing to better address the user through the exploration of the knowledge base, towards the choice of the right wine. We will get back to the underlying concept later in Section 4.2, when we will specifically talk about the actual models. For now, what needs to be underlined is that these links can be easily implemented in ADOxx by adding an INTERREF attribute to the class we want to make clickable, further defining the reference type (Model reference or Instance reference) and the target model/class. Within our implementation we used only Instance references, to draw a more precise path between the entities. An additional step that needs to be carried on within the development toolkit consists of the editing of the GraphReps of the several classes and relationships. GraphReps are specific textual attributes which, by being interpreted as a script by the GRAPHREP interpreter, allow to define the style and appearance of the classes within the actual model in the ADOxx Modelling Toolkit, aiming to enhance the expressiveness of the model. This field allows not only to define standard geometric shapes such as rectangles to represent an entity, but also to edit text, labels, and to dynamically apply icons based on the instance's attributes by exploiting conditional scripting constructs such as the IF-ELSIF one. Custom icons and graphics can be applied also to the relation classes to personalise, for instance, the appearance of the arrows. All the classes and the relation classes defined in the hierarchy, will not be available within the modelling toolkit until we specify them in the add-ons section of the library attributes. Each model is specified through the MODELTYPE keyword, and the classes to be included in the model are set by using the INCL keyword. Always based on our protege work, we can find four models in our implementation:

**RegionBelongsToCountry** draws the relations of belonging between regions and countries

**WineBelongsToRegion** defines the relations of belonging between each wine and its region of production

**WineContainsGrape** explains the grapes contained inside each wine

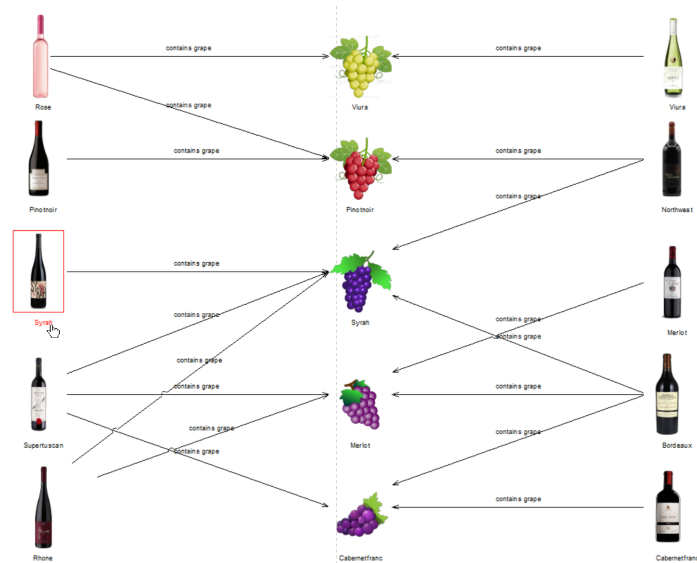
**WinePairedWith** matches each wine with the foods which better suit its flavour

## 4.2 Modelling

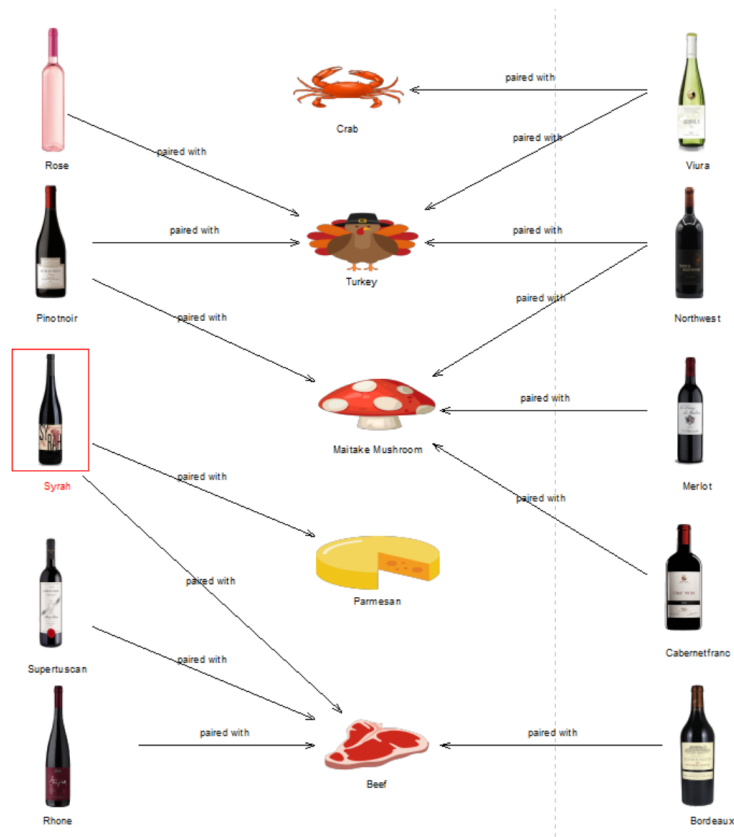
Within the ADOxx Modeling Toolkit is where the actual instantiation of the meta-model takes place. Here it is possible to use the concepts defined in the development toolkit, specifically the ones declared within the add-ons section of the library attributes, to draw the models and the relationships among the classes in a visual way. It starts by laying down the entities, giving them names of actual instances. It is also possible, as we did for example for the wine instances, to give values to the attributes of the class, for instance, the values from 1 to 5 for Fruitiness, Boldness, Savoriness, Dryness and Tannity. After having our instances placed in each corresponding model,



Choose a wine based on the preferred region and click on its name to be redirected to the WineContainsGrape model in which all the grapes used to make that specific wine are described

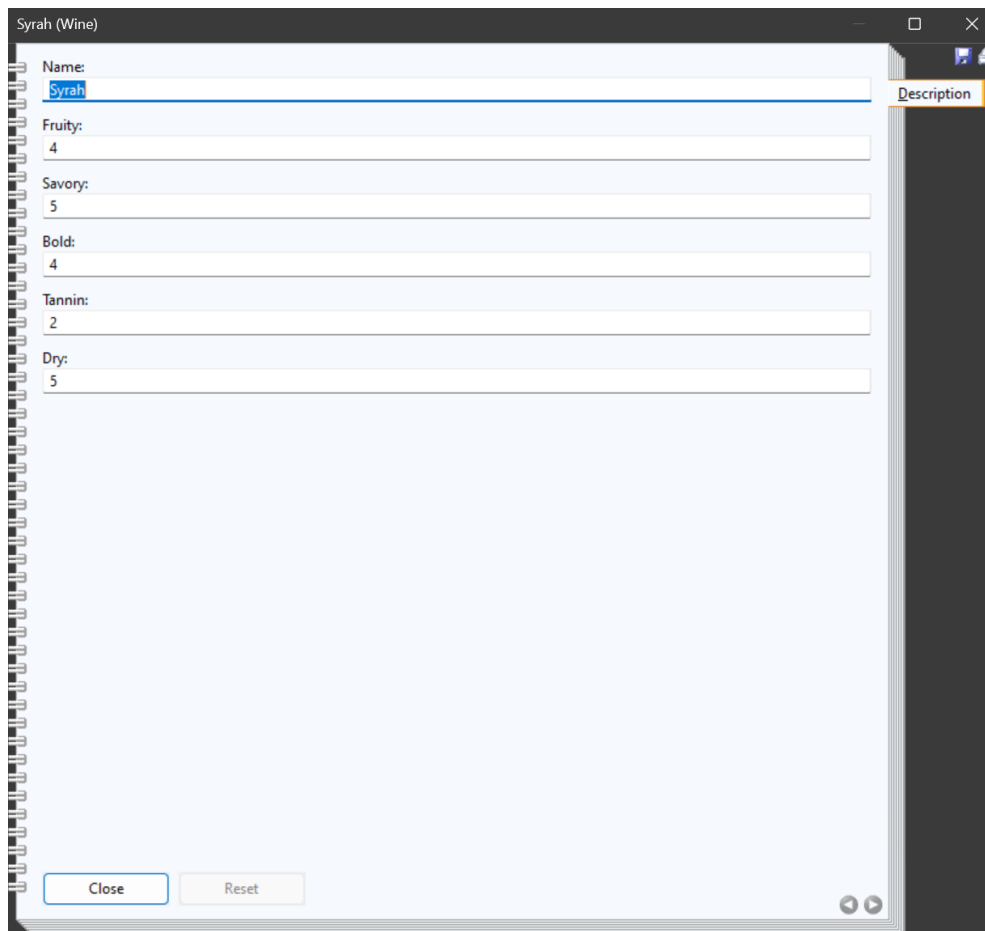


Choose a wine based on the preferred grapes and click on its name to be redirected to the WinePairedWith model in which all the dishes matching the wine are drawn, allowing the user to make its final choice based on it.





On top of that, within this model the user has also the possibility to open the description of a wine through a double click to see the values associated to its taste



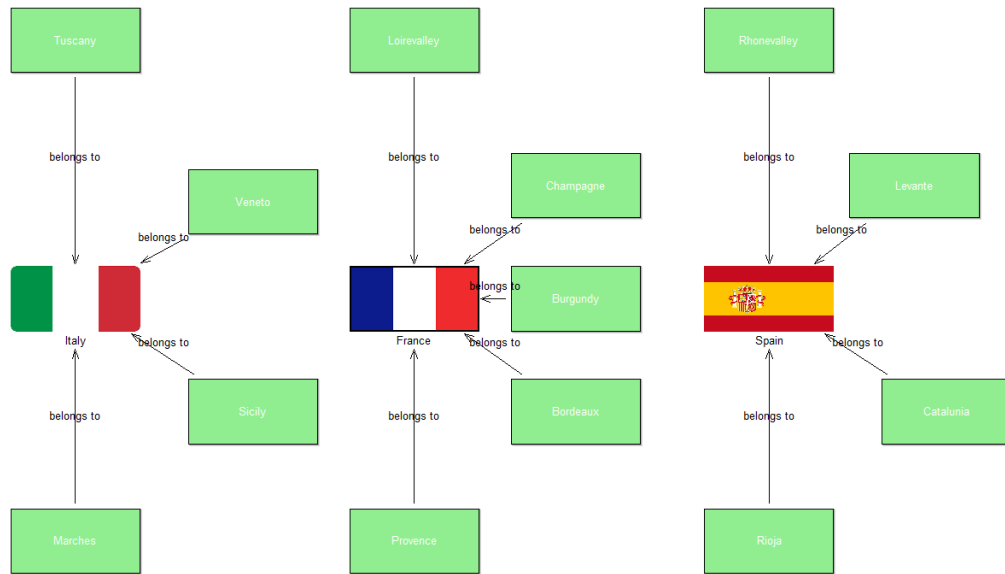
The screenshot shows a software window titled "Syrah (Wine)". On the left, there are several input fields with labels: "Name:" (containing "Syrah"), "Fruity:" (containing "4"), "Savory:" (containing "5"), "Bold:" (containing "4"), "Tannin:" (containing "2"), and "Dry:" (containing "5"). To the right of these fields is a vertical panel labeled "Description". At the bottom of the window are two buttons: "Close" and "Reset". The window has a standard operating system title bar with minimize, maximize, and close buttons.

After following this flow, the user will either be aware of the wine that better suits their preferences, or start the choice process from the beginning, changing their ideas and following different paths.

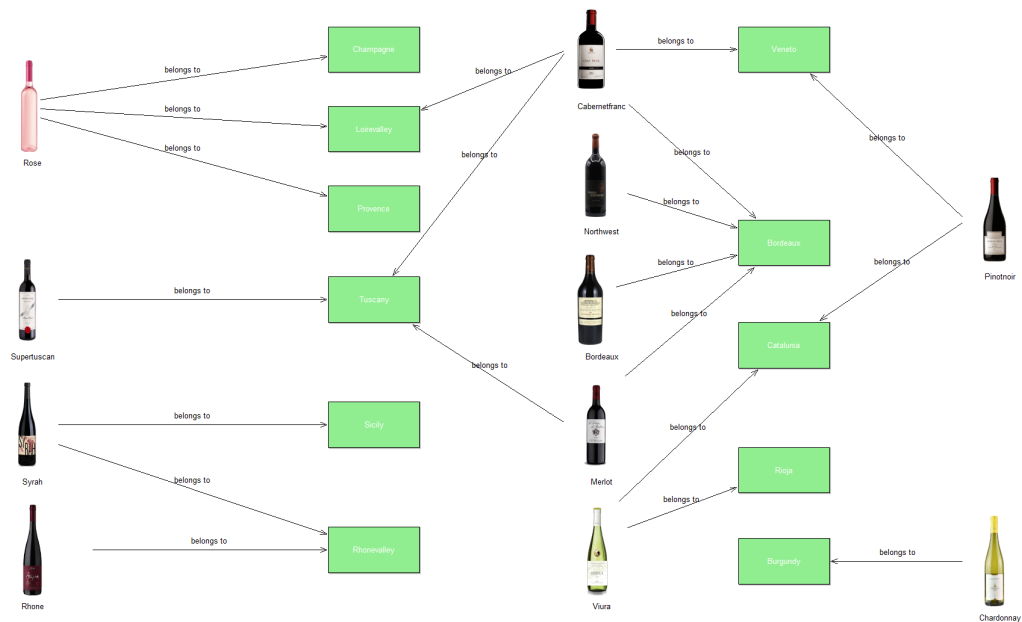
### 4.3 After First Review

For this second submission, we adapted our graphical model to introduce the ingredients related to the meals and to distinguish a match from a perfect match, without neglecting the new elements of our knowledge base introduced in this second version of the project. We will now go through the new information flow, to see one more time how the users can graphically find the wine which best suits their preferences:

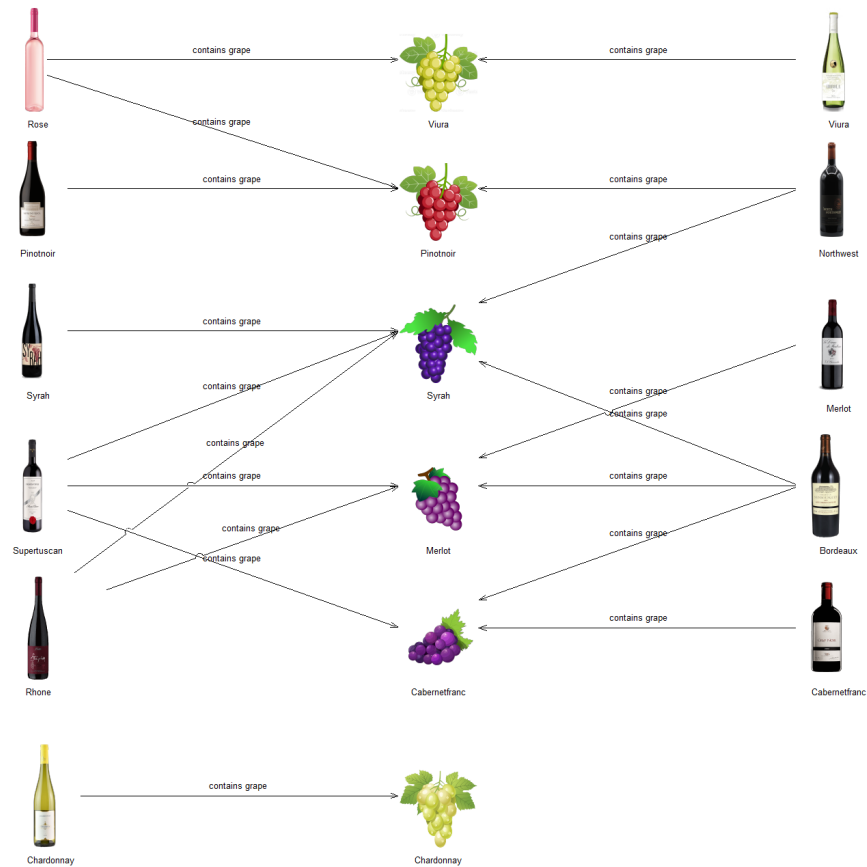
Again, we start from the RegionBelongsToCountry model, where the Burgundy region has been added.



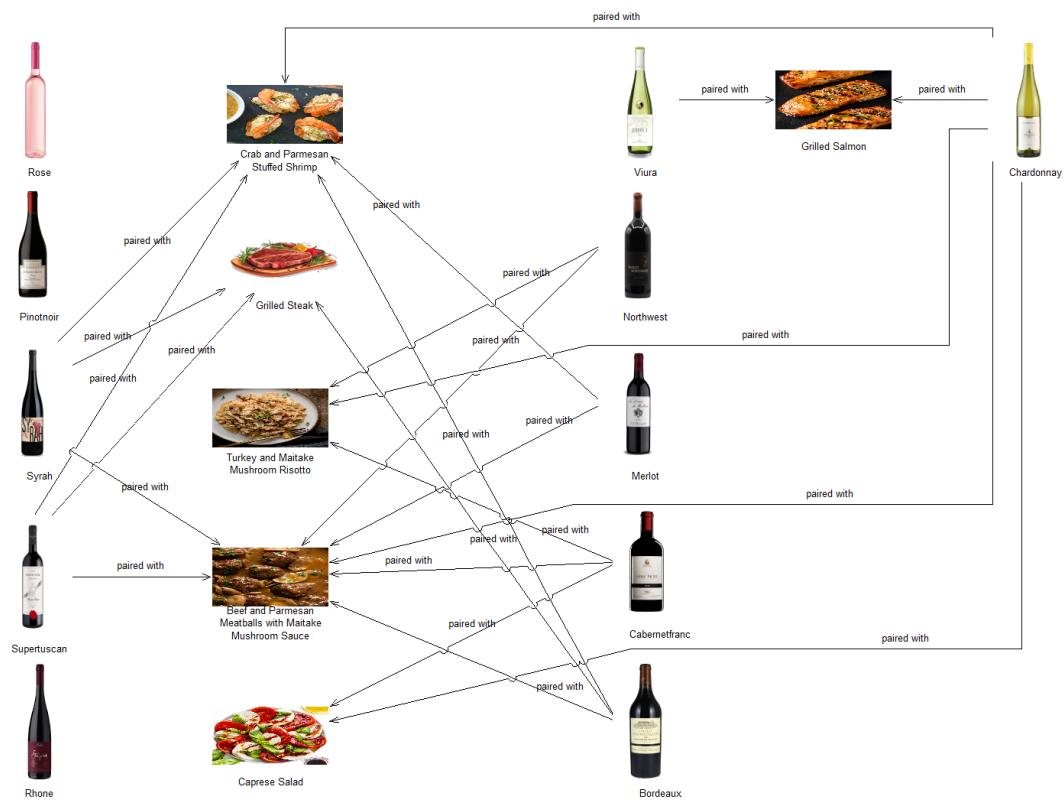
Here we can explore the available wines in a region by clicking the region's name, to be redirected to the WineBelongsToRegion model where, in addition to the wines present from the previous version of the model, we find the Chardonnay wine.



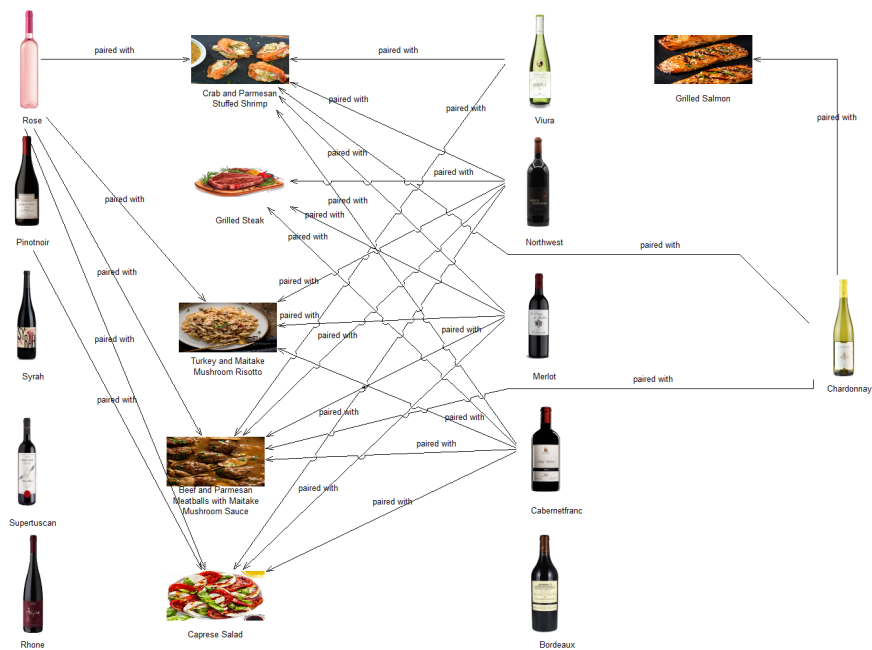
Choose a wine based on the preferred region and click on its name to be redirected to the WineContainsGrape model in which all the grapes used to make that specific wine are described.



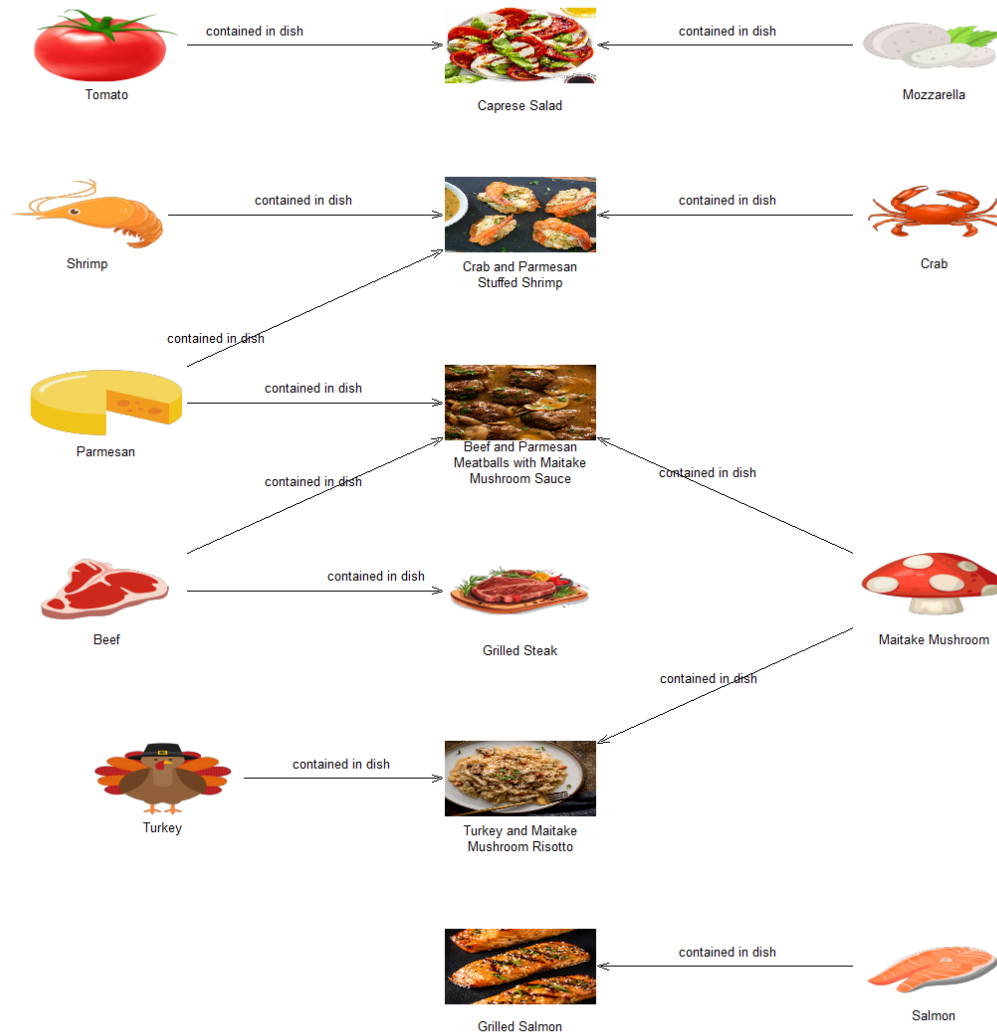
Choose a wine based on the preferred grapes and click on its name to be redirected to the WineBestPairedWith model in which all the dishes best matching the wine are drawn, allowing the user to make its final choice based on it.



In case a base match is preferred, the WinePairedWith model can be opened to find the base matches and explore some other wines that could still suit the meal.



Clicking on the name of a meal, we find ourselves inside the IngredientInDish model, useful for understanding the various ingredients composing each meal, in order to make a more conscious choice of the wine.



Once inside this model, it is possible to go back to the WineBestPairedWith model by clicking again on the name of one of the meals.



## 5. Conclusions

This report described the process followed to complete the Knowledge Engineering and Business Intelligence project. Starting from the project description, the goal was to develop a wine suggestion system that allows restaurants to provide a simple, yet detailed and user-friendly interface to customers that wish to receive help in choosing the perfect wine for their meal. Moving to the project tasks development, the first step was to define and describe an input format. The input format consisted of two columns and four rows, the columns were to decide elements to either include or exclude. The rows were mono-thematic, with the first one allowing the customer to decide on country and region to include or exclude, the second row related to the user's wine tastes, the third about the grapes present in the wine, and the last one related to the dish. After this form and a sample case of its use, we moved to the actual implementation, starting with Decision Tables, a strict, yet simple method to represent knowledge and to query a dataset to retrieve answers. The subsequent step consisted of the Prolog implementation. This time, a programming language allowed for a more flexible implementation, that as we have shown can quickly scale in both the number of wines and the rules that can be applied to query the knowledge base. As for the last part of the chapter, we presented a solution using Knowledge Graphs, a form of database, that uses reasoning software and allows extracting knowledge from a set of data. After the Knowledge Graphs, we discussed the ordering recommendation possibilities and some plausible ways to implement this feature. We have shown that in some instances providing this kind of feature to the user is far from trivial and implies using cumbersome and time-consuming solutions. We also have shown that in other cases, such as Prolog and Knowledge Graphs, the implementation of a weak recommendation system in the former was partially implemented or simple to implement, and non-necessary in the latter, since we could take advantage of the query language to filter and sort data as preferred. The following chapter was destined for the last task of the project. We presented the graphical model, with a small digression in metamodeling and modelling and how those were applied using the ADOxx software.

The project followed a bottom-up approach, with each step being the base for the next one, and we are both satisfied with the learning outcomes.

Following in this chapter we will present our point of view on the project, talking about the most liked parts and the ones that for us proved to be the most boring.

### 5.1 Filippo Lampa Conclusions and Preferences

The development of the project described in this report allowed both me and my colleague to get hands-on several tools and techniques. All of them proved to have strengths and weaknesses, therefore in this section, I will compare and discuss the ones

that, in my own opinion, are the pros and cons of every knowledge-based solution used within this project.

### **5.1.1 DMN Tables**

The first feature the eye falls on is the ease of understanding and visualising data: DMN tables provide a representation that is intuitive to both write and comprehend, making it easier to get the key relationships between the data within them. On top of that, even if we did not take advantage of that in this specific project, DMN tables can be quickly integrated with business process modelling tools, enhancing business automation capabilities. This makes them a powerful tool if used in the right context and with awareness, providing the possibility of both quick implementation and fast queries. On the other hand, if used when should not, this kind of solution rapidly shows its drawbacks. DMN tables may not be the best choice to represent highly complex and intricate knowledge or reasoning, since their ease of usage does not come without some limitations given by the tabular representation, which may restrict the flexibility required for capturing complex relationships and constraints present in some domains, where there could even show some non-deterministic behaviours impossible to capture with this kind of structure. Moreover, as decision models grow in size and complexity, managing DMN tables becomes more and more challenging, leading to increased maintenance efforts and tables exponentially bigger, especially if designed without attention.

### **5.1.2 Prolog**

Being based on a powerful logical reasoning syntax and on predicates with high expressiveness power, it proves to be efficient even where sophisticated inference capabilities are needed. It allows for easy manipulation and extension of knowledge bases, making it adaptable to the most disparate problem domains and making it rather scalable, allowing it to handle complex bases just as easily as simple ones if the right design choices are made. The first drawback of this language is its syntax, not too complex but still rather counter-intuitive. Most of the operators and constructs do not follow the standard mechanisms found in most of the other programming languages, making the learning curve steeper and making it sometimes necessary to check the documentation when trying to represent slightly more complex scenarios. The second problem is, Prolog may suffer from performance issues when dealing with large-scale data processing or computationally intensive tasks. It is not rare to see the engine working for high amounts of time after querying the knowledge base, sometimes even not replying at all due to the bad management of exceptions and errors.

### **5.1.3 Knowledge Graphs**

This structure provides a flexible and extensible way to model complex knowledge domains and capture relationships between entities effectively. Especially in Protege, the ontology editor used during the development of this project, powerful reasoning capabilities are offered, allowing for advanced inferencing and semantic querying. This flexibility and reasoning capabilities, unfortunately, do not come for free. To exploit the reasoner and the inferences at their best, significant efforts in terms of ontology design, data modelling, and ontology population are required to build and maintain knowledge graphs. Adding more classes, relationships, properties and individuals may



result in a cumbersome and time-consuming operation, resulting even in a threat to scalability, not to mention the querying and reasoning operations which can become computationally expensive as knowledge graphs grow in size and complexity.

#### 5.1.4 Comparison

To sum up, DMN tables are user-friendly and excel at representing decision logic but may lack expressiveness for complex knowledge domains. We can find more expressiveness in Prolog, which provides powerful logical reasoning capabilities but has a slightly steeper learning curve and may occasionally show performance issues, given sometimes also the massive need for recursion enforced by its design. Knowledge Graphs with Protege offer representational flexibility, and advanced semantic reasoning but require significant knowledge engineering efforts and high efforts to maintain and update the knowledge base. Ultimately, it is quite clear that the choice of the knowledge-based solution depends on the specific requirements of the problem domain, the complexity of knowledge, the reasoning involved and the available knowledge base, without forgetting to look ahead and take into account the needed scalability.

#### 5.1.5 After First Review

##### DMN Tables

The first feature the eye falls on is the ease of understanding and visualising data: DMN tables provide a representation that is intuitive to both write and comprehend, making it easier to get the key relationships between the data within them. This resulted quite useful since, when working asynchronously with my colleague, the idea behind the changes applied by the other was easily understandable by having a look at the structure and the content of the tables without needing many explanations. This makes them a powerful tool if used in the right context and with awareness, providing the possibility of both quick implementation and fast queries. On the other hand, if used when should not, this kind of solution rapidly shows its drawbacks. DMN tables may not be the best choice to represent highly complex and intricate knowledge or reasoning, since their ease of usage does not come without some limitations given by the tabular representation, which may restrict the flexibility required for capturing complex relationships and constraints present in some domains, where there could even show some non-deterministic behaviours impossible to capture with this kind of structure. For instance, if we think about starting to take into account, as mentioned in Chapter 3.2, the relationship between a wine and the properties of a dish, to be able to pair wines to dishes in a more precise way by also considering organoleptic properties, the tables would not be the easiest choice to represent and query the knowledge base, since the relationships between these factors are not always straightforward and some wines might have unexpected flavours that pair well with certain foods. Moreover, as decision models grow in size and complexity, managing DMN tables becomes more and more challenging, leading to increased maintenance efforts and tables exponentially bigger, especially if designed without attention. For instance, our first implementation was designed in a pipeline way, which may not be the best solution to exploit the declarativeness DMN tables could potentially offer. This first solution, as well as being the result of some bad design decisions, presented some errors when running the simulation. However, by exploiting more the FEEL functions offered as we did in the second approach we used described in Chapter 3.1.1, the size of the tables has been signifi-

cantly reduced, lowering the cumbersomeness and making the solution more scalable. In addition, the debugging in this kind of solution is not trivial when using the official Camunda DMN simulator. However, as mentioned in Section 3.1.1, some open-source tools are able to support this process and ease the testing phase.

## **Prolog**

Being based on a powerful logical reasoning syntax and on predicates with high expressiveness power, it proves to be efficient even where sophisticated inference capabilities are needed. It allows for easy manipulation and extension of knowledge bases, making Prolog adaptable to the most disparate problem domains and making it rather scalable, allowing it to handle complex bases just as easily as simple ones if the right design choices are made. Introducing new wines, dishes and relationships between them, for example, would not imply the change of previous ones, since everything needed would be to add them to the knowledge base and take them into account within the horn clauses, in addition to the ones already present. The first drawback of this language is its syntax, not too complex but still rather counter-intuitive. Most of the operators and constructs do not follow the standard mechanisms found in most of the other programming languages resulting, from my point of view, in a steeper learning curve, and making it sometimes necessary to check the documentation when trying to represent slightly more complex scenarios. Moreover, we could not always easily find this documentation, since the dialects of the language are several and the changes introduced through the versions are not easy to follow, making it possible to stumble upon deprecated constructs and functions. The second problem is, from what I could experience, the errors were not enough clear and detailed enough to allow easy debugging. It is not rare to see the engine working for high amounts of time after querying the knowledge base, sometimes even not replying at all due to what looks like bad management of exceptions and a lack of output information.

## **Knowledge Graphs**

This structure provides a flexible and extensible way to model complex knowledge domains and capture relationships between entities effectively. Especially in Protégé, the ontology editor used during the development of this project, powerful reasoning capabilities are offered, allowing for advanced inferencing and semantic querying. Moreover, the querying part, with the exception of some standards related to naming and URIs, was something I could quickly get confident with since the syntax is almost the same as the one provided by other query languages, making the learning curve less steep. The flexibility and reasoning capabilities, unfortunately, do not come for free. To exploit the reasoner and the inferences at their best, significant efforts in terms of ontology design, data modelling, and ontology population are required to build and maintain knowledge graphs. Adding more classes, relationships, properties and individuals may result in a cumbersome and time-consuming operation, resulting even in a threat to scalability, not to mention the querying and reasoning operations which can become computationally expensive as knowledge graphs grow in size and complexity. Manually inputting data, may it be new wines, regions or dishes, can be time-consuming and error-prone. Moreover, mismatched or inconsistent data can negatively impact the quality of recommendations, and make it hard for the reasoner to infer correct relationships and behaviours. Anyway, as a silver lining, the management of the hierarchy is something that remains easy to accomplish even when the size of the project grows.

For instance, the addition of the subclasses that we introduced in our second version of the knowledge graph, didn't require a severe distortion of the knowledge base previously implemented and turned out to be quite straightforward. In addition, since after the first review we introduced some more levels in the hierarchy and, in general, the knowledge base is prone to getting bigger and more complex, we had the opportunity to exploit the SWRL language to try lightening the work of the reasoner through the specification of some rules like, for example, the relation describing the production of a given wine in a certain country or the match of a dish to a wine without directly dealing with ingredients. This was, from my point of view, the most enjoyable part of the implementation of the knowledge graph, since it made us deal with a completely new syntax and mechanisms we had never seen before this course. Anyway, despite the syntax being new to us, getting in tune with it was quite straightforward given its intuitiveness and given some parts shared with the syntax of SPARQL queries, for instance, the declaration of variables.

## Comparison

To sum up, DMN tables are user-friendly and excel at representing decision logic but may lack expressiveness for complex knowledge domains. We can find more expressiveness in Prolog, which provides powerful logical reasoning capabilities but has a slightly steeper learning curve, not made any easier by the expressiveness of the errors which may arise during the development. Knowledge Graphs with Protégé offer representational flexibility, and advanced semantic reasoning but require significant knowledge engineering efforts and high efforts to maintain and update the knowledge base. Ultimately, it is quite clear that the choice of the knowledge-based solution depends on the specific requirements of the problem domain, the complexity of knowledge, the reasoning involved and the available knowledge base, without forgetting to look ahead and take into account the needed scalability.

## 5.2 Francesco Moschella Conclusion and Preferences

Throughout the course of this project, my colleague and I had the opportunity to work on various aspects of knowledge engineering. All the knowledge-based solutions present pros and cons. In this section, I am going to present my own impressions about the provided solutions, also taking into account what, from my point of view, were the positive and negative aspects.

### 5.2.1 Decision Tables

These tables are quite straightforward. They receive inputs and provide outputs by following some rules, specified as rows in the table while the output is collected following a so-called "Hit Policy". There are many policies to group the results, and each of them comes with personal features, advantages and disadvantages.

In this project, we used a "Collect" policy that allows all the rules to be checked concurrently, thus exploiting the full potential of the DMNs.

Using this policy, the input-related part of the tables follow a simple format, the columns are bonded by an "and" relation, and all rows can be considered as using an "or" relationship with each other, while the output of the matching conditions is just collected and add to a set for each output column.

This way of functioning is indeed simple to understand and easy to apply.

But, if we try to use these tables in highly complicated contexts, where there are many variables to take into account and the conditions are not so simple, the application of this method may become impossible or extremely complex.

As an example, in the project development chapter, we outlined the complexity related to maintaining or enhancing the behaviour provided by the current tables.

Also, this method of manipulating data is quite cumbersome and repetitive, even in our project, at a certain point, we thought of creating a Python script to simplify our work. The creation of all entries was quite monotonous and at a certain point tedious. Also, the debugging was not simple, indeed we only had access to an online tool that did not completely support our solution.

### 5.2.2 Prolog Implementation

The Prolog development was the part that I enjoyed the most. I had the possibility to learn a new programming language and experiment with it. This language does not present a steep learning curve, albeit finding sources and material to study is not simple. It enforces a high use of recursion and does not provide other loop-related operators. Also, classic mathematical operations are not straightforward. Definitely, this language has to be used in its own context, where it seems to perform really well. In fact, managing the data part of our knowledge base and retrieving information from those data is quite simple and intuitive. Although learning and using this language was quite interesting and fun, I also have to recognise that Prolog presents some major problems.

First of all, and as mentioned, there is no clear documentation online. Each document that I found, refers to its own dialect of Prolog, therefore it is never possible for the developer to be certain that a provided solution or function will work in its dialect, is important to notice that this enlarges the development time since each function has to be tested singularly and with the others, to check whether the presented behaviour matches the expected one. Again related to the presence of many dialects and the testing problems, we can find the lack of standards. Since anyone can define their dialect of Prolog, the function signature may change for each version, further complicating this language. Finally, even if Prolog is highly performing and optimised to work with identifiers, working with a big knowledge base, may slow down the interpreter, resulting in computation that requires lots of time to provide a result.

### 5.2.3 Knowledge Graphs

I saw this solution as a midway between DMN and Object Oriented programming languages. Here we are allowed to define objects, with their relations and fields and extract knowledge from the instances of these objects. It was intriguing to model knowledge this way, also the querying method was close to standard SQL, which made the use of it simple. Unfortunately, in knowledge graphs, I found some negative sides that resemble the ones present in Decision Tables. In my opinion, this directly derives from the Protégé software that was used to model the solution. Indeed, the creation of the model structure and of the instances was quite repetitive and uninteresting. There was no way to automate this part, and we had to repeat the same operations several times. After developing the project, we tried to use the reasoner to infer information from our data. As presented in the Knowledge Graphs related section, we discovered

that the provided reasoner was not context-aware. Among the extracted information, there were invalid or useless pieces of information. The presence of this non-necessary information could have impacted in a negative way the querying process used to retrieve knowledge from the data.

#### 5.2.4 Comparison

In conclusion, while there were parts of the project that I thoroughly enjoyed, such as Prolog Software development and the SPARQL Query creation, there were also areas that I found less appealing, such as Decision Tables definition, and Knowledge Graphs creation. Nevertheless, this project provided me with valuable experiences, growth opportunities, and a deeper understanding of the subject matter. After completing this project, I think that each of the presented and used technologies has its personal field of use, and is important to not misuse them.

#### 5.2.5 After First Review

##### Decision Tables

The DMN tables are quite straightforward. They receive inputs and provide outputs by following some rules, specified as rows in the table while the output is collected following a so-called “Hit Policy”. There are many policies to group the results, and each of them comes with personal features, advantages and disadvantages.

In this revision of the project, we moved away from the previous approach that relied on a pipeline pattern, we continued to have some tables with the “Collect” hit policy, retaining all the related advantages, but we had to integrate some FEEL expressions to filter data in a simpler way. I am aware that we use a programming-like approach, but implementing the solution only using tables would have been extremely repetitive and tedious, instead, the FEEL scripts allowed us to speed up the development and create a more robust and scalable solution.

Using the “Collect” hit policy, the input-related part of the tables follow a simple format, the columns are bonded by an “and” relation, and all rows can be considered as using an “or” relationship with each other, while the output of the matching conditions is just collected and add to a set for each output column.

This way of functioning is indeed simple to understand and easy to apply.

While the FEEL expressions use input sets to manipulate data and provide output sets.

If we try to use only the tables, without FEEL expressions, in highly complicated contexts, where there are many variables to take into account and the conditions are not so simple, the application of this method may become extremely complex in most cases and impossible in some other.

While with the FEEL expressions, some proper documentation and use of patterns, as shown in the project development chapter, the maintainability and robustness of the software are increased.

After the development, we used the DMN Tester, a community-driven tool, also mentioned in the official Camunda documentation. This tool allowed us to speed up the testing phase and avoid most of the manual testing. The tool was quite straightforward, and, after the initial setup, we could proceed to test and fix our tables. To correctly work, this tool relies on some configuration files that contain the input and the expected

output. To create these files, we had to write some Python scripts that computed many possible inputs and generated the configuration file.

During the enhancement of the DMN we had some other negative sides, for example the non-dynamicity of the Camunda platform was really visible and we had to agree on a specific version of the installed software to avoid problems such as feature unavailability or the impossibility of opening a tab.

## **Prolog Implementation**

The Prolog development was the part that I enjoyed the most. I had the possibility to learn a new programming language and experiment with it. This language does not present an extremely steep learning curve, albeit finding sources and material to study is not simple. It enforces a high use of recursion and loop-related operations are highly discouraged, also because they require a lot of computational time. Also, classic mathematical operations are not straightforward. Definitely, this language has to be used in its own context, where it seems to perform really well. In fact, managing the data part of our knowledge base and retrieving information from those data is quite simple and intuitive. Although learning and using this language was quite interesting and fun, I also have to recognise that Prolog presents some major problems.

First of all, and as mentioned, there is no clear documentation online. And the presented documentation always refers to a specific dialect of Prolog, and we had no clue if that version matched ours, all we could do was test that solution and verify the outcome.

Due to a lack of standards, it was complicated to track back the small amount of online documentation to the right Prolog version. This caused us to try deprecated and invalid functions and slowed down our development.

The last problem we found was related to the Prolog running time and the absence of information or errors regarding infinite loops and slow operations. We had to include in the code some print functions that are usually only considered for debug purposes because they slow down the computation, but this addition was necessary to have some insight into what Prolog was doing under the hood.

We were in the end able to create a function to define strong and weakly suggested wines. However, we failed to create a function to retrieve the ordered list of wines.

To solve the problem we had to rely on a high-order language, Python, which we used to create a script to order the results retrieved from the Prolog and noticed that given some inputs, the Prolog query kept running even for minutes. While testing this feature we had no certainty about the operation outcome, we only had to keep waiting for some time and hope for an output. Sometimes, we just did not want to wait anymore and killed the process deciding that the query filters were not strict enough. To double-check the truthfulness of the received data we also used the online Prolog interpreter provided by SWI-Prolog and installed a version of it in our local machine. We actually are not sure if those filters were valid or not since we got the same results when launching the query directly in the Prolog interpreters. In my opinion, having more information or errors from the running interpreter may help the development of more complicated solutions.

## **Knowledge Graphs**

I saw this solution midway between DMN and Object Oriented programming languages. Here we are allowed to define objects, with their relations and fields and extract knowledge from the instances of these objects. It was intriguing to model knowledge this way, also the querying method was close to standard SQL, which made the use of it simple. Unfortunately, I found some negative sides in knowledge graphs that resemble those present in Decision Tables. In my opinion, this directly derives from the Protégé software that was used to model the solution. Indeed, the creation of the model structure and of the instances was quite repetitive and uninteresting. There was no way to automate this part, and we had to repeat the same operations several times. After developing the project, we tried to use the reasoner to infer information from our data. Among the extracted information, there were invalid or useless pieces of information. The presence of this unnecessary information could have impacted in a negative way the querying process used to retrieve knowledge from the data. Because of this, we had to modify our solution by removing some fields that interfered with the reasoner type inference. After this we added some new fields, to correctly match bidirectional operations and some SWRL query to create the relation between the country and the produced wine. This latter part was quite interesting and made me see some of the potential of this kind of solution in representing complicated class relations. We finally had to write queries again, this time to extract all the data with the new hierarchy. Thanks to the similarities to SQL, we were also able to create a query that represents an initial ordering technique.

## **Comparison**

In conclusion, while there were parts of the project that I thoroughly enjoyed, such as Prolog Software development and the SPARQL Query creation, there were also areas that I found less appealing, such as DMN definition, and Knowledge Graphs creation. Nevertheless, this project gave me valuable experiences, growth opportunities, and a deeper understanding of the subject matter. After completing this project, I think that each of the presented and used technologies has a personal field of use, and it is important to not misuse them. Thanks to the Python scripts I could also see more clearly where these techniques can be used, for example, websites, and the possibility related to applying them in most various contexts. Using Knowledge Graphs instead of heavy SQL-like databases, given their relation with non-structured databases like NoSQL and GrapDB, might have made the development of some of my previous projects faster and less complicated.