**University of Camerino**
School of Advanced Studies
Master of Science in
Computer Science & Mathematics


# KEBI: Project Report

***Supervisor***
Prof. Knut Hinkelmann
Prof. Emanuele Laurenzi

***Students***
Francesco Finucci
Fabio Michele De Vitis

Academic Year 2023-2024

# Contents

# Part I

# PROLOGUE

# 1

# *Introduction*

In today's digital era, many restaurants have transitioned to digitized menus accessible through QR codes. While this technological advancement offers convenience, it also presents challenges, especially for guests with specific dietary preferences or restrictions. The small screen size of smartphones can make it difficult to get a comprehensive and full view of a menu, and guest often have to shift through numerous options that may not align with their dietary needs.

To address these issues, this project aims to develop a knowledge-based system that tailors menu recommendations based on individual guest preferences. By leveraging knowledge engineering techniques, we can create a system that filters and presents only those meals that match the guest's dietary profile. This enhances the dining experience by providing a more personalized and manageable menu.

Knowledge engineering plays a crucial role in this project, as it involves the creation, representation and utilization of knowledge to solve complex problems. The solutions created in this project include decision tables, Prolog, and knowledge graphs/ontologies, each offering unique advantages for representing and querying knowledge.

In developing our solution, **we utilized the menu from the Italian restaurant Nero Balsamico as a case study** (https://www.nerobalsamico.it/). This real-world example provided a rich dataset of typical Italian meals, including pizzas, pastas, and main dishes, along with detailed information about their ingredients and nutritional content. It surely added complexity rather than mocking a simple and common Menu but we gladly went through with the challenge.

The report is structured as follows:

1. **Introduction**
    - Overview of the project, its objectives, and the importance of knowledge engineering.

2. **Decision Tables and DRD**
    - Explanation of decision tables and Decision Requirements Diagrams (DRD).
    - Creation and implementation of decision tables for menu recommendations.
3. **Prolog Implementation**
    - Introduction to Prolog and its use in knowledge representation.
    - Development of Prolog facts and rules for guest-specific meal recommendations.
4. **Knowledge Graphs and Ontologies**
    - Explanation of knowledge graphs and ontologies.
    - Use of SWRL, SPARQL, and SHACL for meal recommendation queries and rules.
5. **AOAME**
    - Introduction to the AOAME tool.
    - Modeling of our ontology leveraging the power of AOAME.
6. **Conclusions and Personal Thoughts**
    - De Vitis Fabio Michele's Conclusions
    - Francesco Finucci's Conclusions

By the end of this report, we will have demonstrated how knowledge-based systems can significantly enhance the dining experience by providing personalized and relevant menu recommendations. This project not only showcases the practical application of knowledge engineering but also highlights the potential for such systems to be integrated into modern restaurant operations. You can view all the project implementation and try it out yourself by checking the following **Github Repository**: (https://github.com/Meguazy/project_KEBI).

# 2

# *General Logic*

In this chapter, we will provide a comprehensive analysis and explanation of the key logical concepts that recur throughout our paper. These concepts form the backbone of our approach, enabling us to develop solutions that are both efficient and adaptable. By understanding these foundational principles, the reader will gain deeper insights into how our system operates and how it manages complex decision-making processes, particularly in the context of dietary restrictions and meal recommendations.

## 2.1 SYSTEM INTERFACE

To design an intuitive and user-friendly interaction with our Knowledge System, we envisioned how a user, referred to as the Client, would engage with the system to obtain a personalized menu based on their preferences. For demonstration purposes, we created a scenario that illustrates this interaction, which could be integrated through a pipeline with one of the solutions proposed in this project.

Following the project guidelines, the Client would ideally begin by scanning a QR code, like the one shown below (which is actually scannable):

FIGURE 2.1: Example QR Code

After scanning the QR code, the Client would be directed to a simple form where they can enter their preferences. These preferences would then be processed by our Knowledge System to filter the initial menu, presenting the Client with a personalized selection of meals that match their requirements. An example of the interface where the Client can input their preferences is shown in Figure 2.2:

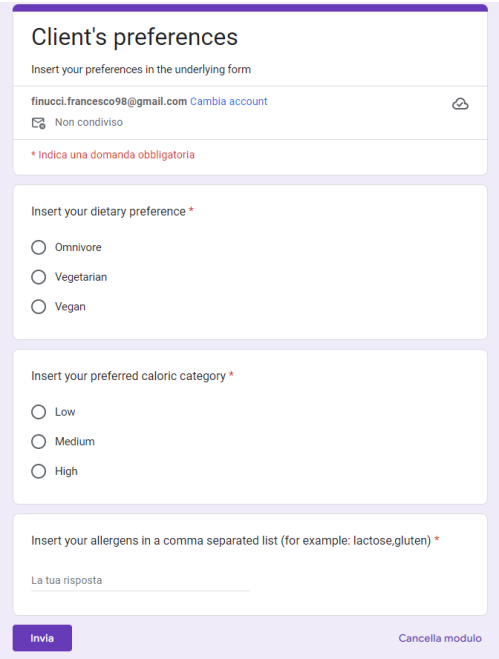


FIGURE 2.2: Client preferences form

## 2.2 HIERARCHICAL LOGIC IN DIETARY RESTRICTIONS

In this project, we implemented a hierarchical logic system to effectively manage dietary restrictions across all the solutions. This hierarchy plays a crucial role in streamlining the decision-making processes, particularly when determining the suitability of ingredients for various diets. The hierarchy is structured from the most general to the most specific dietary categories: **omnivore**, **vegetarian**, and **vegan**.

### 2.2.1 *Structure of the Hierarchy*

The logic behind this hierarchical structure is simple yet powerful. It categorizes ingredients based on their compatibility with different dietary restrictions, enabling a more efficient and flexible filtering process.

- **Omnivore**: This is the most general category. Ingredients labeled as omnivore include both animal-based and plant-based foods. These ingredients are only suitable for individuals who follow an omnivorous diet, meaning they do not avoid any food groups. By labeling an ingredient as omnivore, we indicate that it is not suitable for vegetarians or vegans.

- **Vegetarian**: Moving one step more specific, vegetarian ingredients exclude meat and fish but may include dairy and eggs. These ingredients are suitable for both vegetarians and omnivores. Vegetarian ingredients are a middle ground, offering more specificity than omnivore ingredients while still being less restrictive than vegan ingredients.
- **Vegan**: At the highest level of specificity, vegan ingredients exclude all animal products, including meat, dairy, eggs, and any other animal-derived substances. These ingredients are the most restrictive but also the most inclusive in terms of dietary compatibility. Vegan ingredients can be consumed by everyone—vegans, vegetarians, and omnivores alike.

#### 2.2.2 *Application of the Hierarchy*

This hierarchical system is applied across all the solutions in the project to simplify the logic and make the decision-making process more efficient. By categorizing ingredients according to this hierarchy, the system can quickly and accurately filter out unsuitable options based on the client's dietary needs.

**Example:**

- If a client is vegan, only vegan ingredients are considered, ensuring that the meal recommendations are strictly aligned with their dietary preferences.
- For vegetarians, both vegetarian and vegan ingredients are available, but omnivore ingredients are excluded from the options.
- Omnivores have the most flexibility, with all ingredients being considered unless specific allergens are noted.

#### 2.2.3 *Benefits of Hierarchical Logic*

The use of hierarchical logic in this project brings several key benefits:

- **Efficiency**: The hierarchy allows the system to quickly filter and identify suitable ingredients, reducing the complexity of the logic needed to make these decisions.
- **Scalability**: As new ingredients are introduced, they can be easily integrated into the existing hierarchy without requiring major changes to the system's logic.
- **Flexibility**: This approach ensures that the system can cater to a wide range of dietary needs, from the most general to the most specific, without sacrificing accuracy or personalization.

In summary, the hierarchical logic system is a fundamental component of the project's overall structure. It ensures that the solutions are both adaptable and robust, capable of delivering accurate and personalized recommendations across a diverse set of dietary requirements.

## 2.3 CALORIC CATEGORIES

To accommodate the caloric preferences of our guests, we implemented a classification system based on **Caloric Categories**. This system plays a **dual role**: it is used to classify each dish on our menu and to tailor recommendations according to the specific caloric preferences of our guests. However, it's important to note that there is a significant distinction between how **Caloric Categories** are applied to dishes and how they are used to reflect guest preferences.

Each dish on our menu is assigned to a specific **Caloric Category** based on its total caloric content, which is calculated by summing the calories of all its individual ingredients. This classification helps us ensure that the caloric content of the meals aligns with the guests' dietary goals.

1. **Low Caloric Category:** This category includes dishes with a total caloric content of up to 400 calories. These meals are suitable for guests who prefer lighter options or are on a strict diet.
2. **Medium Caloric Category:** Dishes in this category have a caloric content that falls between 401 and 900 calories.
3. **High Caloric Category:** Dishes in this category have a caloric content exceeding 900 calories, with no upper limit.

On the other hand, when a guest selects a caloric preference—such as "Medium"—the system will recommend dishes according to an **upper bound** principle, it's basically the maximum caloric category he prefers, for example:

- If a guest selects the **Low** caloric preference, they will only receive recommendations for dishes that fall within the Low Caloric Category.
- If a guest selects the **Medium** caloric preference, they will receive recommendations for dishes within both the Low and Medium Caloric Categories.
- If a guest selects the **High** caloric preference, they will receive recommendations for dishes from all three categories: Low, Medium, and High.

By establishing these caloric categories for both dishes and guest preferences, we created a flexible and dynamic recommendation system that adapts to the diverse caloric needs of our guests. This dual approach not only enhances the guest experience by aligning with their dietary goals but also simplifies the decision-making process, ensuring that each guest receives meal recommendations that are both appropriate and satisfying.

# Part II

# IMPLEMENTATION & TECHNOLOGIES

# 3

# *Decision Tables and DRD*

*"A decision table is the simplest form of representing the decision logic in a tabular manner, which helps in organizing and ensuring completeness."*
—

## 3.1 DRD DESCRIPTION

Decision tables were the first step towards our solutions; they are a powerful tool used in decision-making processes. They provide a structured way to represent decision logic by mapping different conditions to corresponding actions through a tabular representation. We leveraged the power of decision tables to model our system with rules that allowed us to filter the initial ingredients list to ensure we would consider the Guest's dietary needs and allergens. During our solutions, we made extensive use of the FEEL language, which enabled us to create a more dynamic system. We followed a course on Camunda about Decision Tables to deeply understand their advantages and managed to elaborate every single table in our DRD with powerful FEEL expressions.

To have a deep understanding of the solution we provided through the decision tables, we shall start by describing our DRD, a Decision Requirements Diagram, which illustrates the decision-making process for recommending meals based on client-specific dietary information.

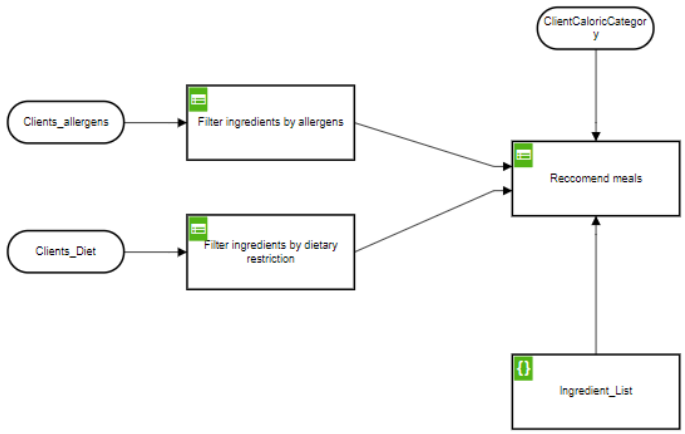


FIGURE 3.1: Decision Requirements Diagram (DRD) for Recommending Meals Based on Dietary Preferences

As we can see from Figure 3.1, this DRD is composed of several elements. We will analyze each element individually and provide an accurate description. The diagram is made up of the following components:

- **Filter Ingredients By Allergens:** This is a decision table that filters out ingredients containing allergens specified by the clients.
  - These allergens are provided as input data by the component **Clients_allergens.**
- **Filter By Dietary Restriction:** This decision table is responsible for filtering out ingredients based on the client's dietary restrictions, such as Vegan, Vegetarian, and Omnivore.
  - The specific diet of the client is given by an input data shape called **Clients_Diet.**
- **Ingredient_List:** This, unlike the others, is a literal expression and represents the comprehensive list of all possible ingredients available for meal preparation, mapped to their kcal values.
- **Recommend Meals:** This central decision node integrates filtered results based on allergens, dietary restrictions, and caloric categories to recommend appropriate meals based on the client's needs. It represents the heart of our DRD and is responsible for collecting and combining information from other decision tables. As we can see from the inputs, it is linked with almost every element. There is also a new element called **ClientsCaloricCategory**, which we will discuss further in the project report.

First and foremost, it is important to note that we initially used a sequential filtering approach. We later concluded that a sequential approach might have some potential problems in the future regarding extensibility and maintainability, as it was very difficult to retrieve information from previous tables that weren't directly linked. The new DRD's modular design allows each decision component (e.g., filtering by allergens, dietary restrictions, and caloric categories) to be developed, tested, and maintained independently. We have a single responsibility where each table has one logical responsibility. This modularity makes it easier to add new decision criteria or modify existing ones without affecting the entire system.

Another significantly improved aspect is the **Scalability**. As new types of dietary requirements or allergens might emerge, additional decision nodes can be incorporated into the DRD without overhauling the entire decision-making process. This ensures the system remains scalable and adaptable to evolving needs. In fact, unlike sequential filtering, where each step depends on the previous one, the DRD allows for parallel processing of decision rules, leading to more efficient data processing as multiple conditions can be evaluated simultaneously.

The DRD also enhances **Maintainability** by making it straightforward to update specific decision rules or add new ones. This flexibility is crucial for adapting to new dietary trends and regulations, ensuring that the system remains relevant and accurate over time.

In summary, the use of DRD in our project significantly improved the system's extensibility, scalability, and maintainability, providing a robust framework for dynamic and personalized meal recommendations.

## 3.2 DECISION TABLES ANALYSIS

In this section we are going to analyze the single decision tables, explaining their implementation and semantics.

### 3.2.1 *Filter Ingredients By Allergens Decision Table*

The **Filter Ingredients by Allergens** decision table plays a crucial role in ensuring that the meal recommendations are safe for clients with specific allergen sensitivities. This decision table utilizes the FEEL (Friendly Enough Expression Language) to dynamically filter out ingredients that contain allergens specified by the client. Below, we describe the table in detail and explain the FEEL expressions used.

Filter ingredients by allergens | Hit policy: Collect

| | When | Then | Annotations |
|---|---|---|---|
| | Clients_allergens (string) | Filtered_Ingredients_By_Allergens (string) | |
| 1 | - | "lemon_sorbet" | |
| 2 | - | "fresh_fruit_salad" | |
| 3 | - | "caramelized_orange" | |
| 4 | not(contains(lower case(?), "gluten")) | "pink_peppercorn_tuille" | |
| 5 | not(contains(lower case(?), "lactose")) | "anise_panna_cotta" | |
| 6 | - | "fresh_raspberries" | |
| 7 | not(contains(lower case(?), "lactose")) | "zabaglione_chantilly_cream" | |
| 8 | not(contains(lower case(?), "gluten")) | "crisp_wafer" | |
| 9 | not(contains(lower case(?), "gluten") or contains(lower case(?), "lactose") or contains(lower case(?), "nuts")) | "chocolate_macaron" | |
| 10 | - | "custard" | |
| 11 | - | "alchermes" | |
| 12 | not(contains(lower case(?), "gluten") or contains(lower case(?), "lactose")) | "sponge_cake" | |
| 13 | - | "coffee_coral" | |
| 14 | not(contains(lower case(?), "lactose")) | "tiramisu_parfait" | |
| 15 | not(contains(lower case(?), "lactose")) | "mascarpone_cream" | |
| 16 | not(contains(lower case(?), "lactose")) | "barozzi_cake" | |

FIGURE 3.2: Decision Table: Filter Ingredients by Allergens

### 3.2.2 *Understanding the Decision Table*

The decision table is structured into the following main columns:

- **When (Conditions)**: This column checks the presence of specific allergens in the client's allergen list.

- **Then (Actions)**: If the condition is met, the action specifies which ingredient should be included or excluded from the filtered list based on the allergen content.

The decision table operates under the **Hit policy: Collect**, which means that all applicable rules are executed, and their results are collected. This policy is particularly useful in this case since we need to filter **multiple** ingredients based on the client's allergen profile.

**FEEL Expressions in Detail**

The FEEL expressions used in the **When** column are as follows:

- **Expression 1:** `not(contains(lower case(?), "gluten"))`
  This expression checks if the client's allergen list does not contain the term "gluten". The function `lower case(?)` converts the allergen list to lowercase to ensure case-insensitive matching, and `contains` checks for the presence of the string "gluten". If "gluten" is not found, the corresponding ingredient (in the **Then** column) is considered safe and added to the filtered list.
- **Expression 2:** `not(contains(lower case(?), "lactose"))`
  Similar to the first expression, this checks for the absence of "lactose" in the allergen list. If lactose is not present, the ingredient is deemed safe for inclusion.
- **Expression 3:** `not(contains(lower case(?), "egg"))`
  This checks the absence of egg in the client's allergen list.
- **Expression 4:** `not(contains(lowercase(?),"nuts"`
  This last expression checks for the absence of nuts in the client allergens list.
- **Expression 5:** `-`
  A hyphen (`-`) in the **When** column indicates a default rule that always applies if no specific condition is matched. In this table, it typically allows non-allergenic ingredients to pass through the filtering process.

The previous FEEL expressions take the client's data as an input and suggest ingredients that do not contain something that could harm the client based on his allergens. In our example we considered just gluten, lactose, egg and nuts. Since they are the most common ones, but in future implementations we could consider inserting more allergens to make the system more inclusive. It is important to note that in our table there are **combinations of previous FEEL expressions**, in fact by using a logic OR, we contemplated the cases where an ingredient contains more than just one allergen.

**The output** of the *Filter Ingredients By Allergens Table* will be a list of Ingredients that are suitable for the client allergens.The use of FEEL

expressions allows for a flexible and dynamic approach to filtering ingredients, making the system both powerful and adaptable to various client needs. This list will be used later on on our main table.

### 3.2.3 *Filter Ingredients By Dietary Restriction Decision Table*

The **Filter Ingredients by Dietary Restriction** decision table is essential for tailoring meal recommendations according to the client's dietary preferences, such as omnivore, vegetarian, and vegan diets. This table, similar to the allergen filtering table, uses FEEL (Friendly Enough Expression Language) to dynamically filter ingredients based on the client's diet type. Below, we describe the table in detail and explain the FEEL expressions used.

Filter ingredients by dietary restriction | **Hit policy:** Collect

| | When<br>Clients_Diet<br>string | Then<br>Filtered_Ingredients_By_Diet<br>string | Annotations |
|---|---|---|---|
| 1 | contains(lower case(?), "omnivore") | "prosciutto_crudo" | |
| 2 | contains(lower case(?), "omnivore") | "coppa" | |
| 3 | contains(lower case(?), "omnivore") | "salame" | |
| 4 | contains(lower case(?), "omnivore") | "mortadella" | |
| 5 | contains(lower case(?), "omnivore") | "beef" | |
| 6 | contains(lower case(?), "omnivore") | "mullet_roe" | |
| 7 | contains(lower case(?), "omnivore") | "octopus" | |
| 8 | contains(lower case(?), "omnivore") | "meat_broth" | |
| 9 | contains(lower case(?), "omnivore") | "beef_ragu" | |
| 10 | contains(lower case(?), "omnivore") | "red_shrimp_tartare" | |
| 11 | contains(lower case(?), "omnivore") | "beef_filet" | |
| 12 | contains(lower case(?), "omnivore") | "tuna" | |

FIGURE 3.3: Decision Table: Filter Ingredients by Dietary Restriction

### 3.2.4 *Understanding the Decision Table*

The decision table is structured into the following main columns:

- **When (Conditions)**: This column checks the specific dietary restriction of the client.
- **Then (Actions)**: If the condition is met, the action specifies which ingredient should be included or excluded from the filtered list based on the client's dietary restriction.

The decision table, like the previous one, operates under the **Hit policy: Collect**, which means that all applicable rules are executed, and their results are collected.

**FEEL Expressions in Detail**

The FEEL expressions used in the **When** column are as follows:

- **Expression 1:** `contains(lower case(?), "omnivore")`
  This expression checks if the client's diet is omnivore. The function `lower case(?)` converts the diet input to lowercase to ensure case-insensitive matching. If the client's diet is omnivore, the corresponding ingredient (in the **Then** column) is considered suitable and added to the filtered list.
- **Expression 2:** `contains(lower case(?), "vegetarian")`
  This expression checks if the client's diet is vegetarian. It filters out ingredients that are not compatible with a vegetarian diet.
- **Expression 3:** `contains(lower case(?), "vegan")`
  This expression checks if the client's diet is vegan. It filters out ingredients that are not suitable for a vegan diet, such as any animal products.
- **Expression 4:** `-`
  A hyphen (`-`) in the **When** column indicates a default rule that always applies if no specific condition is matched. This typically allows ingredients that are universally acceptable (i.e., suitable for all diet types) to pass through the filtering process.

**Note:** The FEEL expressions are **case-sensitive**, meaning that the input data must match the expected case format. This is why the function `lower case($)` is used to normalize the input, ensuring that the condition check is not affected by case variations in the input data.

**Example Rules and Outcomes**

- **Rule 1**: `contains(lower case(?), "omnivore")` ⇒ `"prosciutto_crudo"`
  If the client's diet is omnivore, "prosciutto_crudo" is considered suitable and added to the filtered ingredients list.
- **Rule 2**: `contains(lower case(?), "omnivore")` ⇒ `"coppa"`
  If the client's diet is omnivore, "coppa" is included in the filtered list.
- **Rule 3**: `contains(lower case(?), "omnivore")` ⇒ `"salame"`
  If the client's diet is omnivore, "salame" is considered appropriate and added to the list.

### 3.2.5 *Conclusion*

The *Filter Ingredients By Dietary Restriction* decision table dynamically adjusts the list of suitable ingredients based on the client's dietary profile. The use of FEEL expressions allows for a flexible and robust filtering mechanism, ensuring that the system can adapt to various dietary needs while maintaining accuracy and relevance in meal recommendations.

This filtered list will then be utilized in subsequent decision-making steps to recommend meals that align with the client's dietary restrictions.

### 3.2.6 *Recommend Meals Decision Table*

The **Recommend Meals** decision table integrates the filtered results from both allergens and dietary restrictions, along with the client's desired caloric intake, to recommend appropriate meals. This table uses FEEL expressions to dynamically assess the suitability of a meal based on the combined criteria. Below, we describe the table in detail, including how the ingredients are mapped to their caloric values using the **Ingredient_List**.

Recommend Meals | Hit policy: Collect

| | When | And | And | And | Then | And | Annotations |
|---|---|---|---|---|---|---|---|
| | Filtered_Ingredients_By_Allergens | Filtered_Ingredients_By_Diet | Ingredient_List | ClientCaloricCategory | Reccomended_Meals | Menu_Category | |
| | list | list | list | "low","medium","high" | string | string | |
| 7 | list contains(Filtered_Ingredients_By_Allergens, "spinach") and list contains(Filtered_Ingredients_By_Allergens, "parmesan_cheese") and list contains(Filtered_Ingredients_By_Allergens, "mullet_roe") | list contains(Filtered_Ingredients_By_Diet, "spinach") and list contains(Filtered_Ingredients_By_Diet, "parmesan_cheese") and list contains(Filtered_Ingredients_By_Diet, "mullet_roe") | (Ingredient_List[name="spinach"].calories[1] + Ingredient_List[name="parmesan_cheese"].calories[1]) + Ingredient_List[name="mullet_roe"].calories[1]) <=400 | "low" | "Spinach flan on Parmigiano Reggiano cream with mullet roe" | "Appetizers" | |
| 8 | list contains(Filtered_Ingredients_By_Allergens, "spinach") and list contains(Filtered_Ingredients_By_Allergens, "parmesan_cheese") and list contains(Filtered_Ingredients_By_Allergens, "mullet_roe") | list contains(Filtered_Ingredients_By_Diet, "spinach") and list contains(Filtered_Ingredients_By_Diet, "parmesan_cheese") and list contains(Filtered_Ingredients_By_Diet, "mullet_roe") | (Ingredient_List[name="spinach"].calories[1] + Ingredient_List[name="parmesan_cheese"].calories[1]) + Ingredient_List[name="mullet_roe"].calories[1]) <=900 | "medium" | "Spinach flan on Parmigiano Reggiano cream with mullet roe" | "Appetizers" | |
| 9 | list contains(Filtered_Ingredients_By_Allergens, "spinach") and list contains(Filtered_Ingredients_By_Allergens, "parmesan_cheese") and list contains(Filtered_Ingredients_By_Allergens, "mullet_roe") | list contains(Filtered_Ingredients_By_Diet, "spinach") and list contains(Filtered_Ingredients_By_Diet, "parmesan_cheese") and list contains(Filtered_Ingredients_By_Diet, "mullet_roe") | | "high" | "Spinach flan on Parmigiano Reggiano cream with mullet roe" | "Appetizers" | |

FIGURE 3.4: Decision Table: Recommend Meals

### 3.2.7 *Understanding the Decision Table*

The decision table is structured into several key columns:

- **Filtered_Ingredients_By_Allergens**: This column contains the list of ingredients that have passed the allergen filtering step.
- **Filtered_Ingredients_By_Diet**: This column contains the list of ingredients that have passed the dietary restriction filtering step.
- **Ingredient_List**: This column uses the provided **Ingredient_List** to calculate the total caloric value of the selected ingredients.
- **ClientCaloricCategory**: This column represents the caloric category selected by the client (e.g., "low", "medium", "high").
- **Recommended_Meals**: If all conditions are satisfied, this column specifies the meal to be recommended.
- **Menu_Category**: This column identifies the category of the recommended meal (e.g., "Appetizers", "Main Course").

The decision table operates under the **Hit policy: Collect**, meaning that all applicable rules are executed and their results are collected. This is crucial for ensuring that the system can recommend multiple meals that fit the client's preferences and dietary needs.

### Ingredient List and Caloric Mapping

The **Ingredient_List** (shown in Figure 3.5) maps each ingredient to its corresponding caloric value. This list is used in the **Recommend Meals** table to calculate the total caloric content of the ingredients in a meal. The FEEL expressions used for this purpose include the following:

- Expression: `Ingredient_List[name="spinach"].calories[1] + Ingredient_List[name="parmesan_cheese"].calories[1] + Ingredient_List[name="mullet_roe"].calories[1]`
  This expression sums the caloric values of "spinach", "parmesan_cheese", and "mullet_roe" to determine the total calories of a potential meal. The `calories[1]` syntax indicates that we are accessing the first (and typically only) entry of the caloric value for each ingredient in the list.

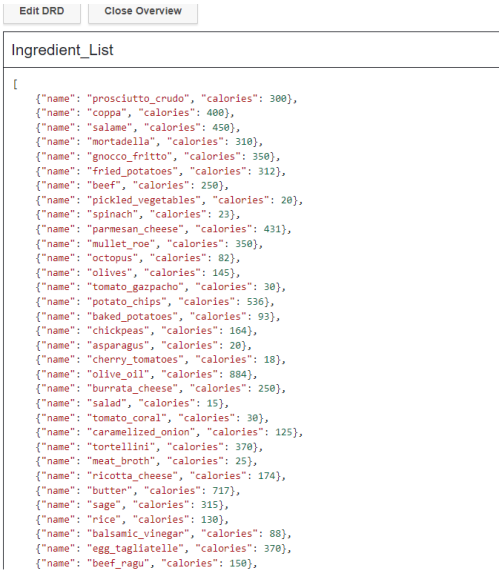


FIGURE 3.5: Ingredient List with Caloric Mapping

The *Recommend Meals* decision table effectively integrates dietary restrictions, allergen considerations, and caloric preferences to provide personalized meal recommendations. By leveraging the **Ingredient_List** for caloric calculations, the system ensures that clients receive meals that are not only safe and suitable but also align with their nutritional goals.

## 3.3 SIMULATION EXAMPLES

This section explains the results generated by the Camunda[1] simulator based on various inputs. We will explore how the inputs lead to the specific outputs and how the decision tables are executed to provide meal recommendations.

[1]Camunda is an open-source platform for workflow and process automation that enables organizations to design, execute, and monitor business processes. It is based on the Decision Requirements Diagrams (DRD) standard, which allows users to create visual models of business processes.

### 3.3.1 *Simulator Inputs and Outputs Overview*

The first image (3.6) shows the interface where inputs are provided and outputs are generated. The inputs include the decision table selection,

allergens, dietary restrictions, and caloric category. The outputs display the filtered ingredients and the recommended meals.

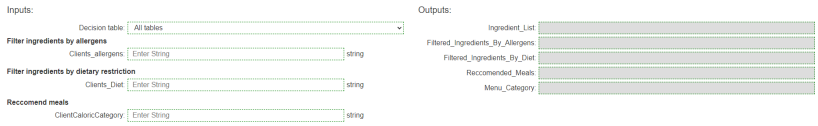


FIGURE 3.6: Camunda Simulator Inputs and Outputs Interface

**Inputs:**

- **Decision Table**: The user can select "All tables" to apply the full decision logic across all available decision tables.
- **Filter Ingredients by Allergens**:
  - **Clients_allergens**: Specifies any allergens the client may have, such as "lactose" and "gluten".
- **Filter Ingredients by Dietary Restriction**:
  - **Clients_Diet**: Specifies the dietary preferences of the client, such as "Vegetarian", "Vegan", or "Omnivore".
- **Recommend Meals**:
  - **ClientCaloricCategory**: Specifies the desired caloric intake category: "low", "medium", or "high".

**Outputs:**

- **Ingredient_List**: The complete list of ingredients with their respective caloric values.
- **Filtered_Ingredients_By_Allergens**: Ingredients that have passed the allergen filtering step.
- **Filtered_Ingredients_By_Diet**: Ingredients that match the client's dietary restrictions.
- **Recommended_Meals**: The meals that meet all the client's criteria, including allergens, diet, and caloric category.
- **Menu_Category**: The category under which the recommended meal falls, such as "Appetizers" or "Main Dishes".

### 3.3.2 *Scenario 1: Vegetarian Client with Lactose and Gluten Allergies*

In the second scenario (3.7), the inputs are as follows:

- **Clients_allergens**: "lactose, gluten"
- **Clients_Diet**: "Vegetarian"
- **ClientCaloricCategory**: "medium"

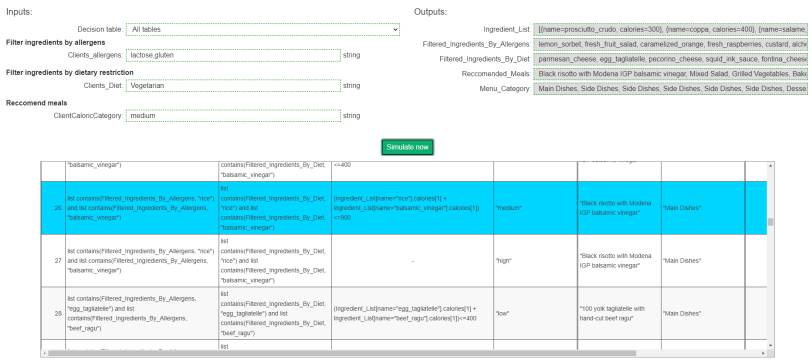


FIGURE 3.7: Example Scenario: Vegetarian Client with Lactose and Gluten Allergies

**Outputs:**

- **Filtered_Ingredients_By_Allergens**: The list includes ingredients like "lemon sorbet", "fresh fruit salad", and "custard", which are free from lactose and gluten.
- **Filtered_Ingredients_By_Diet**: The list includes vegetarian-friendly ingredients such as "parmesan_cheese", "egg_tagliatelle", and "pecorino_cheese".
- **Recommended_Meals**: The simulator recommends "Black risotto with Modena IGP balsamic vinegar", which fits the medium-calorie requirement and is suitable for the client's dietary needs, and all the others meals that suit the requirements.
- **Menu_Category**: The recommended meal is categorized under "Main Dishes".

### 3.3.3 *Scenario 2: Omnivore Client with High-Calorie Preference*

In the third scenario (3.8), the inputs are:

- **Clients_allergens**: None (input left empty)
- **Clients_Diet**: "Omnivore"
- **ClientCaloricCategory**: "high"

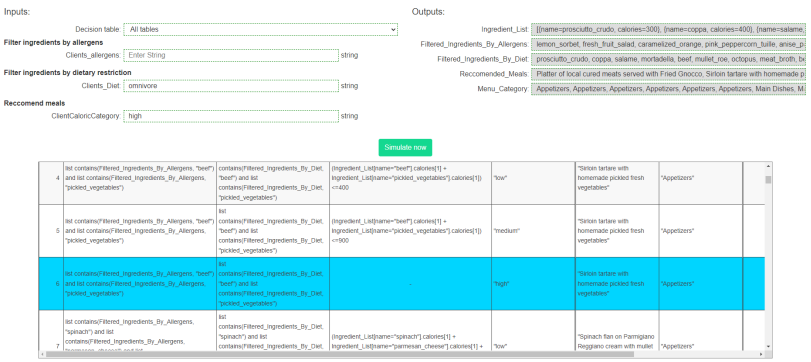


FIGURE 3.8: Example Scenario: Omnivore Client with High-Calorie Preference

**Outputs:**

- **Filtered_Ingredients_By_Allergens**: Since no allergens were specified, the output includes all ingredients.
- **Filtered_Ingredients_By_Diet**: As the client is omnivorous, the list includes all types of ingredients, including meat.
- **Recommended_Meals**: The simulator recommends every meal in the menu since no ingredients where filtered out in the previous steps. An example is "Sirloin tartare with homemade pickled fresh vegetables" as a high-calorie option.
- **Menu_Category**: This meal is categorized under "Appetizers".

These scenarios demonstrate how the Camunda simulator processes the inputs and provides personalized meal recommendations based on the client's dietary needs and caloric preferences, you can download the DMN file called DecisionTablesFinalVersion.dmn on https://github.com/Meguazy/project_KEBI/tree/main/Part%20One/Decision%20tables, go on the camunda web simulator and test it with different allergens and dietary restrictions.

# 4

# *Prolog*

## 4.1 PROLOG CODE EXPLANATION

Following our exploration of decision tables, we turned our attention to Prolog to enhance our meal recommendation system. While decision tables offer a robust framework for straightforward decision-making, they can become cumbersome when dealing with more complex scenarios involving multiple variables. Prolog, on the other hand, is particularly well-suited for managing such intricate and dynamic systems, making it an ideal choice for our needs.

This section explores the Prolog code written for the Nero Balsamico restaurant in Italy's meal recommendation system. In order to recommend the best dishes, the system is built to take into account a number of client-specific factors, such as dietary restrictions, allergies, and preferred calorie intake. Through the utilization of Prolog's logical inference powers, the system is able to efficiently sort and suggest meals that correspond with the individual preferences of the client.

### 4.1.1 *Defining Ingredients and Their Attributes*

The initial step in the code involves defining a comprehensive list of ingredients available at Nero Balsamico, each accompanied by relevant attributes such as caloric content, allergens, and dietary restrictions. This baseline data is essential for the later filtering of ingredients according to customer preferences.

```
% Define the facts with ingredient, calories, allergens,  and dietary
    restrictions
ingredient_info(prosciutto_crudo, 250, [], [omnivore]).
ingredient_info(coppa, 300, [], [omnivore]).
ingredient_info(salame, 400, [], [omnivore]).
ingredient_info(mortadella, 350, [lactose], [omnivore]).
ingredient_info(gnocco_fritto, 300, [gluten], [vegan]).
ingredient_info(fried_potatoes, 300, [], [vegan]).
```

Listing 4.1: Defining Ingredients and Their Attributes

Every ingredient has three essential characteristics that are carefully recorded. First, the amount of calories in each serving is indicated by the caloric content. The second section is called allergens, and it lists any possible allergies, like lactose or gluten, that may be present in

the ingredient. Last but not least, the dietary restrictions specify which diets—vegetarian, vegan, or omnivore—the ingredient is appropriate for.

For example, `mortadella` is described as having 350 calories, containing `lactose`, and being suitable for those following an `omnivore` diet.

### 4.1.2 *Dish Definition and Meal Assembly*

The definition of dishes is the subject of the following section of the code. Dishes are prepared here by identifying ingredients and grouping them into meal categories, like appetizers, main dishes, sides, and desserts.

```
% Appetizers dish facts
dish('Platter of local cured meats served with Fried Gnocco', appetizer, [salame, gnocco_fritto]).
dish('Sirloin tartare with homemade pickled fresh vegetables', appetizer, [beef, pickled_vegetables]).
dish('Spinach flan on Parmigiano Reggiano cream with mullet roe', appetizer, [spinach, parmesan_cheese, mullet_roe]).
```

Listing 4.2: Defining Dishes

Each dish is characterized by its name, the type of course it belongs to, and the list of ingredients it contains. This structure ensures that each dish is accurately described and its components are clearly identified.

For instance, the dish `'Spinach flan on Parmigiano Reggiano cream with mullet roe'` is classified as an `appetizer` and comprises `spinach`, `parmesan_cheese`, and `mullet_roe`.

### 4.1.3 *Client Preferences and Meal Recommendation Logic*

Client preferences form the cornerstone of the meal recommendation process. These preferences are encapsulated in Prolog as facts, detailing the client's desired caloric intake level, allergens they need to avoid, and their dietary restrictions.

```
% Client preferences fact
client_preferences(client1, low, [lactose], vegetarian).
client_preferences(client2, high, [gluten], vegetarian).
client_preferences(client3, medium, [], vegan).
client_preferences(client4, medium, [egg], omnivore).
```

Listing 4.3: Client Preferences and Meal Recommendation Logic

The preferences include a client's preferred caloric intake, which might be low, medium, or high, alongside any allergens they need to avoid. Dietary restrictions are also recorded, identifying whether the client follows an omnivore, vegetarian, or vegan diet.

The system then utilizes these preferences to identify dishes that are appropriate for the client by cross-referencing the dish ingredients with the client's dietary needs.

```
% Find suitable dish for the client based on client preferences
find_suitable_dish(ClientID, DishName, CourseType, [TotalCalories, AllAllergens, HighestDietaryRestriction], [ClientCaloriePref, ClientAllergens, ClientDietaryRestriction]) :-
    client_preferences(ClientID, ClientCaloriePref, ClientAllergens, ClientDietaryRestriction),
    assemble_dish(DishName, CourseType, [TotalCalories, AllAllergens, HighestDietaryRestriction], [CalorieCategory]),
    matches_calorie_preference(ClientCaloriePref, CalorieCategory),
    % Ensure the dish does not contain any allergens the client is allergic to
```

```
7     forall(member(Allergen, ClientAllergens), \+ member(Allergen,
          AllAllergens)),
8     % Ensure the dish matches the client's dietary restrictions
9     matches_dietary_restriction(ClientDietaryRestriction,
          HighestDietaryRestriction).
```

Listing 4.4: Finding Suitable Dishes

The predicate in the code works by first assembling the dish, which involves calculating the total calories, determining the caloric category, aggregating any allergens present, and identifying the most restrictive dietary requirement among the ingredients. Then, it ensures that the dish's characteristics match the client's preferences, filtering out any options that don't align.

### 4.1.4 *Advantages of the Prolog Approach*

Prolog is especially well suited for this task because it provides a number of unique advantages. Prolog's logical inference capability is one of its main advantages since it enables dynamic and flexible filtering of ingredients and meals according to the client's changing preferences. This adaptability is essential to guaranteeing that the suggestions are precise and unique.

Prolog's scalability is also impressive. New ingredients, allergies, and dietary restrictions can be added to the knowledge base with ease without requiring major adjustments to the fundamental reasoning. Because of its versatility, Prolog is an effective tool for managing intricate systems, such as the Nero Balsamico meal recommendation engine.

Lastly, Prolog's declarative syntax contributes to the clarity and maintainability of the code. The rules are straightforward and easy to understand, which facilitates both development and ongoing maintenance.

By leveraging Prolog's strengths, we have developed a meal recommendation system that is both sophisticated and user-friendly, capable of meeting the unique needs of each client at Nero Balsamico.

## 4.2 USE CASE RESULTS ANALYSIS

In this section, we analyze the results generated by the Prolog-based meal recommendation system for four different clients, each with distinct dietary preferences and restrictions. The outputs illustrate how the system tailors its recommendations to meet the specific needs of each client.

**It does not** contain every single suitable item since the list would be too long to show on a screenshot.

### 4.2.1 *Client 1: Low-Calorie Vegetarian with Lactose Allergy*

For Client 1, who has a low-calorie preference and follows a vegetarian diet with a lactose allergy, the system recommended two dishes:

- **Main Dish:** *Black risotto with Modena IGP balsamic vinegar*
  This dish falls under the main dish category and is suitable for vegans. It contains 218 kcal, making it an appropriate low-calorie option for the client.
- **Side Dish:** *Mixed Salad*
  The system also recommended a mixed salad as a side dish. This dish is also vegan and contains only 15 kcal, aligning well with the client's dietary restrictions and low-calorie preference.


FIGURE 4.1: Prolog System Output for Client 1: Low-Calorie Vegetarian with Lactose Allergy

### 4.2.2 *Client 2: High-Calorie Vegetarian with Gluten Allergy*

Client 2 prefers high-calorie meals, follows a vegetarian diet, and has a gluten allergy. The system provided the following recommendations:

- **Appetizer:** *Chickpea hummus with sautéed asparagus, confit cherry tomatoes, and basil oil*
  This dish is vegan and contains 1086 kcal, making it suitable for the client's high-calorie requirement.
- **Appetizer:** *Apulian burrata on a bed of fresh salad with tomato coral and caramelized onion*
  Although this dish contains lactose, it is still considered a vegetarian option with 375 kcal, making it a suitable recommendation for the client's dietary preferences.

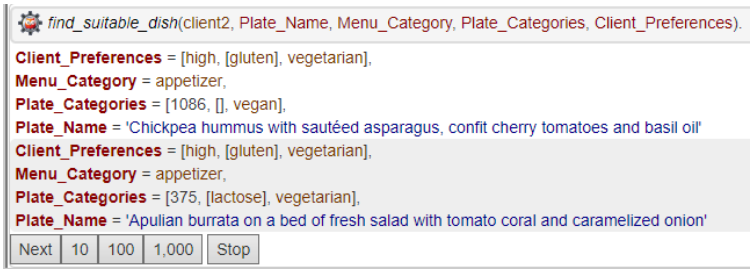


FIGURE 4.2: Prolog System Output for Client 2: High-Calorie Vegetarian with Gluten Allergy

### 4.2.3 *Client 3: Medium-Calorie Vegan*

Client 3 follows a vegan diet and prefers meals with a medium caloric intake. The system recommended the following dishes:

- **Main Dish:** *Black risotto with Modena IGP balsamic vinegar*
  This vegan dish contains 218 kcal and is classified as a main dish, making it a suitable medium-calorie option for the client.
- **Side Dish:** *Mixed Salad*
  Another recommendation is a mixed salad, a vegan side dish with 15 kcal, fitting well with the client's caloric preference.
- **Side Dish:** *Grilled Vegetables*
  Lastly, the system recommended grilled vegetables, a vegan side dish containing 20 kcal, which also aligns with the client's preferences.

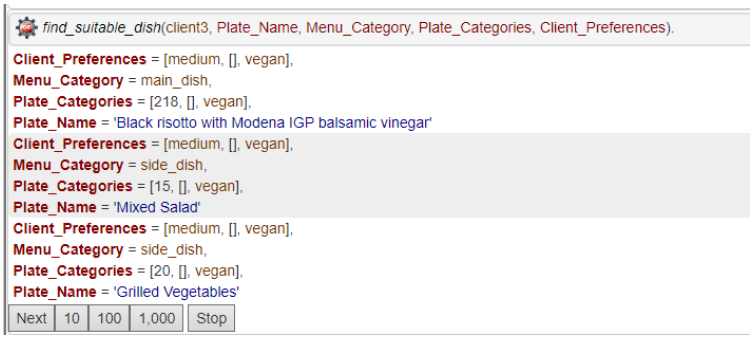


FIGURE 4.3: Prolog System Output for Client 3: Medium-Calorie Vegan

### 4.2.4 *Client 4: Medium-Calorie Omnivore with Egg Allergy*

Client 4 is an omnivore with an egg allergy and a preference for medium-calorie meals. The system suggested the following dishes:

- **Appetizer:** *Platter of local cured meats served with Fried Gnocco*
  This dish contains 700 kcal and is suitable for omnivores, though it contains gluten.
- **Appetizer:** *Sirloin tartare with homemade pickled fresh vegetables*
  This dish is another appetizer suitable for omnivores with 280 kcal and no allergens that the client is concerned about.

- **Appetizer:** *Spinach flan on Parmigiano Reggiano cream with mullet roe*
  This dish contains 704 kcal and includes lactose, but it is still appropriate for an omnivorous diet.

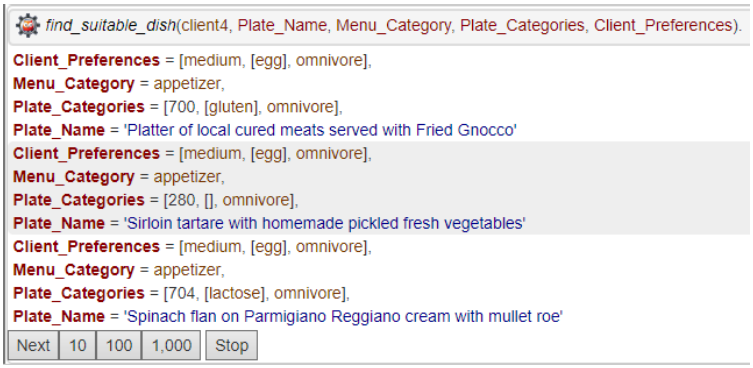


FIGURE 4.4: Prolog System Output for Client 4: Medium-Calorie Omnivore with Egg Allergy

In this chapter we demonstrated how our Prolog system successfully customizes meal suggestions to each client's specific dietary needs and caloric preferences, guaranteeing a suitable and personalized dining experience. You can find the prolog file, called menu_suggestion.pl containing the script at the following link: https://github.com/Meguazy/project_KEBI/tree/main/Part%20One/prolog. You can try out with different combinations of dietary restrictions and allergens to find out which meals are suggested.

# 5

# *Ontology & Knowledge Graphs*

In this chapter we will explore our third knowledge based solution, in this case we leveraged on the use of Knowledge Graphs and the tool Protege to model our ontology. We will explore the details of our implementation starting from the structure and description of our classes, object properties and data properties and explain the logic we followed on doing so. Later on we will introduce our SHACL for the validation of our graph's shape, SWRL to define rules for the reasoning, and the SPARQL query we used to retrieve the list of meals suitable to the user.

## 5.1 STRUCTURE

In this section we are gonna explore the structure of our ontology, explaining the different hierarchies firstly from a general standpoint then slowly getting more and more specific about all the relations.

### 5.1.1 *Class Hierarchy*

First of all, we have the class hierarchy, which is the foundational structure of our ontology. The image displayed shows the class hierarchy as modeled in Protégé. Let's break down this hierarchy to understand how each class and subclass contributes to our overall knowledge graph. It is important to underline however, that we modeled our classes following the schema.org module, this way we defined classes and their respective subclasses in the most extensible way.

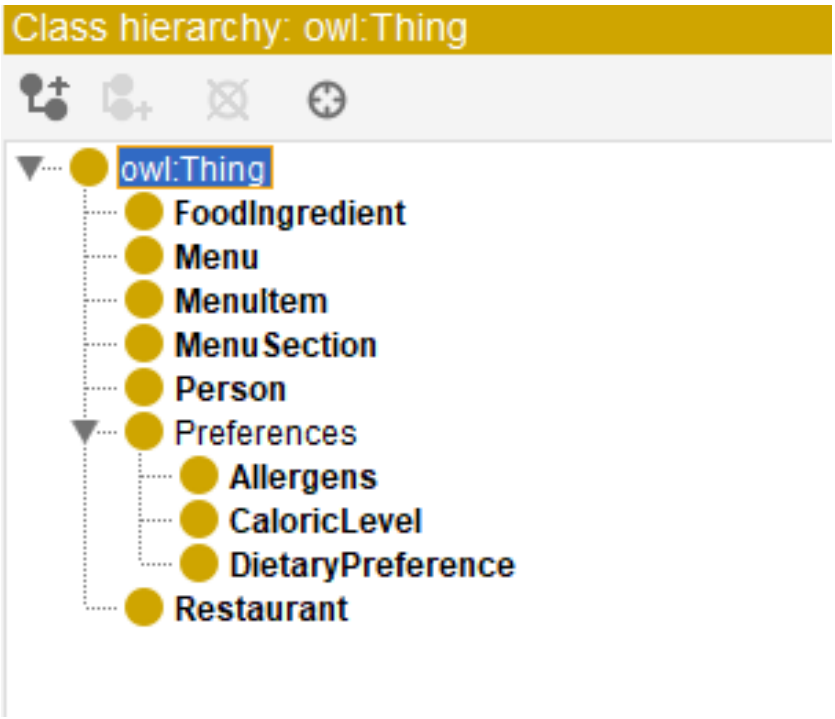


FIGURE 5.1: Protege Class Hierarchy

As you can see from Figure 5.1, at the top of our hierarchy, we have **owl:Thing**, which is the most general class, representing all entities in the ontology. All other classes are subclasses of owl:Thing.

Next we are gonna provide an analysis of all the rest of the classes:

- **FoodIngredient:** This class defines all the food ingredients that constitute the base of our ontology and compose the dishes. Each ingredient is later gonna be defined throught properties like caloric content, allergens and dietary categories.
- **MenuItem:** The class Menu Item represents individual items on the menu, such as specific dishes. These are the elements that atre going to be recommended to clients based on their needs.
- **Menu:** This class encapsulates the overall menu structure, which includes all the dishes available at the Restaurant.
- **MenuSection:** It categorizes the menu into different sections like appetizers, main dishes, side dishes and desserts. Providing an organized structure to the Menu.
- **Person:** This class defines an individual who interacts with the system, such as costumers.
- **Preferences:** The preferences class is strictly linked to the costumer, and it defines the preferences of the latter. It is further broken down into specific subclasses such as:
    1. **Allergens:** Representing the allergens that the user needs to avoid.

2. **Caloric Level:** Defines the caloric intake preference based on three categories: high, medium or low.
3. **DietaryPreference:** Captures the dietary choices of the user.

- **Restaurant:** This class defines the Restaurant class itself.

This hierarchy allows us to model complex relationships between different entities in the domain of meal recommendations. By building the ontology in this manner, we ensure that every aspect of the meal recommendation process—from the ingredients to the menu items to the user preferences—is effectively captured.

For example, we can finely filter and tailor meal recommendations based on highly specific user needs by defining Preferences as a class and then subclassing it into Allergens, CaloricLevel, and DietaryPreference. To find appropriate recommendations, each preference type can be evaluated independently against the properties of MenuItem and FoodIngredient.

This structure improves the scalability of the ontology in addition to supporting the data's logical organization. It is simple to incorporate newly introduced menu items or dietary trends into this hierarchy without upsetting the current order.

### 5.1.2 *Object Properties*

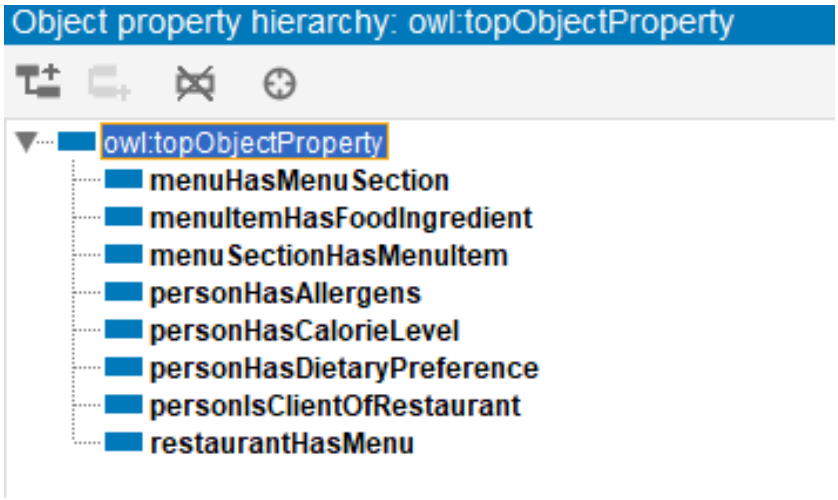


FIGURE 5.2: Protege Object Property Hierarchy

Object properties in our ontology define the relationships between instances of classes, effectively linking different entities in the knowledge graph. Each object property has a domain and range, which specify the classes it connects. Let's walk through the key object properties depicted in Figure 5.2 and explain how they link various classes in our ontology.

- **menuHasMenuSection:**
  - **Domain:** Menu

  - **Range:** MenuSection
  - This property links a **Menu** to its constituent **MenuSections**. For instance, a menu at a restaurant might be divided into sections such as appetizers, main courses, and desserts. This property ensures that each menu is properly organized into these sections.

- **menuItemHasFoodIngredient:**
  - **Domain:** MenuItem
  - **Range:** FoodIngredient
  - This property associates a **MenuItem** (such as a dish) with the **FoodIngredient**s that compose it. For example, a pizza (**MenuItem**) would be linked to ingredients like dough, tomato sauce, and cheese (**FoodIngredients**).

- **menuSectionHasMenuItem:**
  - **Domain:** MenuSection
  - **Range:** MenuItem
  - This property connects a **MenuSection** to the **MenuItem**s it contains. For example, an appetizer section (**MenuSection**) might include items like bruschetta and caprese salad (**MenuItems**).

- **personHasAllergens:**
  - **Domain:** Person
  - **Range:** Allergens
  - This property links a **Person** to their specific **Allergens**. It ensures that the system is aware of any ingredients that must be avoided when making meal recommendations for that person.

- **personHasCalorieLevel:**
  - **Domain:** Person
  - **Range:** CaloricLevel
  - This property ties a **Person** to their preferred **CaloricLevel** (low, medium, high). It allows the recommendation system to tailor meal suggestions based on the caloric intake preferences of the individual.

- **personHasDietaryPreference:**
  - **Domain:** Person
  - **Range:** DietaryPreference
  - This property associates a **Person** with their **DietaryPreference** (e.g., vegetarian, vegan, omnivore). This is crucial for filtering out dishes that do not align with the person's dietary restrictions.

- **personIsClientOfRestaurant:**
  - **Domain:** Person
  - **Range:** Restaurant
  - This property establishes the relationship between a **Person** and the **Restaurant** they are visiting or ordering from. It signifies that the person is a client of a particular restaurant, potentially allowing for personalized services or recommendations.
- **restaurantHasMenu:**
  - **Domain:** Restaurant
  - **Range:** Menu
  - This property connects a **Restaurant** to its **Menu**. It indicates that a particular menu belongs to a specific restaurant, encapsulating all the dishes and options available at that establishment.

By defining these object properties, we are able to model complex relationships between different entities in the ontology. For instance, by linking a **Person** to their **Allergens**, **CaloricLevel**, and **DietaryPreference**, the system can make informed and precise meal recommendations from the **Menu** that take into account the specific **MenuItems** and their **FoodIngredients**.

This structure not only provides a clear and logical organization of data but also ensures that the ontology can be easily extended or modified. New object properties can be added to accommodate additional relationships as needed, without disrupting the existing model.

### 5.1.3 *Data Properties*

Finally we defined data properties, the name are pretty self explanatory and are structured in a way that it is easily understandable the domain of the data property, as you can see in Fig 5.3. In this case we are basically defining the attributes of the different objects, every **has name** data property has string as a range. We also attributed to each dish the **hasTotalCalories** attribute that was calculated by summing the calories of each ingredient composing a meal.

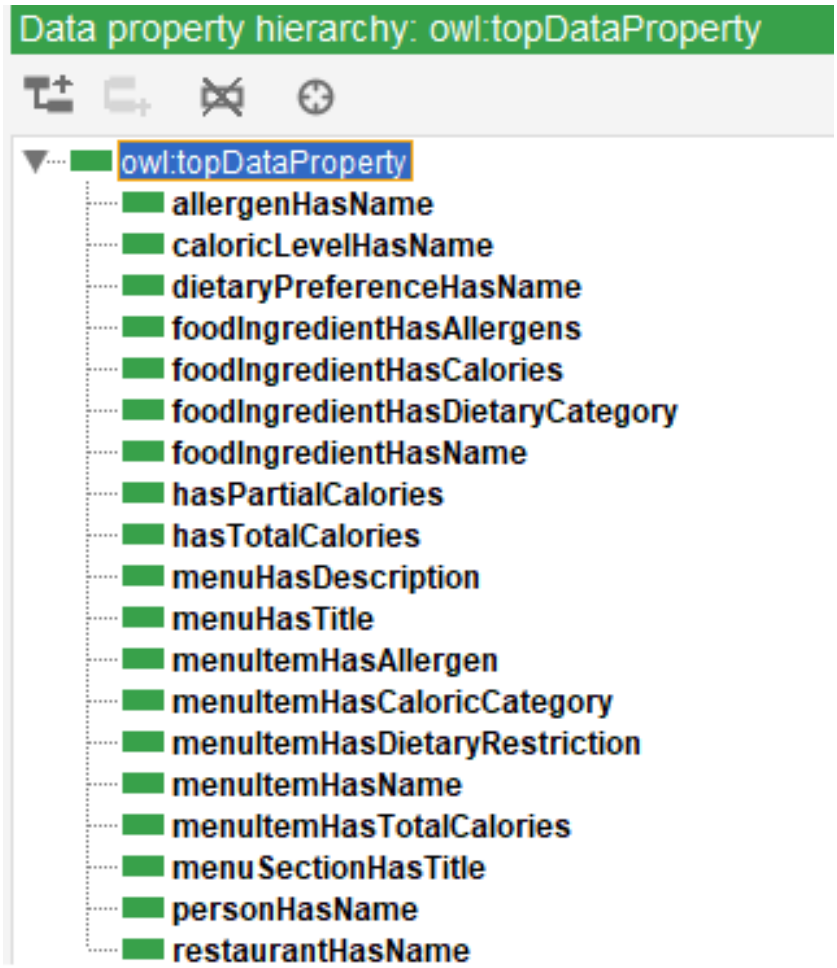


FIGURE 5.3: Data Property Hierarchy

## 5.2 SHACL

SHACL (Shapes Constraint Language) is a W3C standard used to define constraints and validate data in RDF (Resource Description Framework) graphs. It allows you to specify shapes, which are sets of conditions or rules that RDF data must satisfy. SHACL is commonly used to ensure data consistency and integrity in semantic web applications by checking whether the data conforms to a predefined structure, such as requiring certain properties to be present, of specific types, or within certain value ranges. SHACL shapes are written in RDF themselves, making them highly interoperable within semantic web frameworks.

### 5.2.1 *Shapes description*

The following is the SHACL document used to test the shape of the ontology:

```
prefix : <http://www.semanticweb.org/byjus/ontologies/2024/6/
    KebiProject/> .
prefix sh: <http://www.w3.org/ns/shacl#> .
prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

```

```
### PersonShape
:PersonShape
    a sh:NodeShape ;
    sh:targetClass :Person ;
    sh:property [
        sh:path :personHasName ;
        sh:datatype xsd:string ;
        sh:minCount 1 ;
    ] ;
    sh:property [
        sh:path :personHasAllergens ;
        sh:class :Allergens ;
        sh:minCount 0 ;
    ] ;
    sh:property [
        sh:path :personHasCalorieLevel ;
        sh:class :CaloricLevel ;
        sh:minCount 1 ;
    ] ;
    sh:property [
        sh:path :personHasDietaryPreference ;
        sh:class :DietaryPreference ;
        sh:minCount 1 ;
    ] .

### FoodIngredientShape
:FoodIngredientShape
    a sh:NodeShape ;
    sh:targetClass :FoodIngredient ;
    sh:property [
        sh:path :foodIngredientHasName ;
        sh:datatype xsd:string ;
        sh:minCount 1 ;
    ] ;
    sh:property [
        sh:path :foodIngredientHasCalories ;
        sh:datatype xsd:nonNegativeInteger ;
        sh:minCount 1 ;
    ] ;
    sh:property [
        sh:path :foodIngredientHasAllergens ;
        sh:datatype xsd:string ;
        sh:minCount 0 ;
    ] ;
    sh:property [
        sh:path :foodIngredientHasDietaryCategory ;
        sh:datatype xsd:string ;
        sh:minCount 1 ;
    ] .

### MenuItemShape
:MenuItemShape
    a sh:NodeShape ;
    sh:targetClass :MenuItem ;
    sh:property [
        sh:path :menuItemHasName ;
        sh:datatype xsd:string ;
        sh:minCount 1 ;
    ] ;
    sh:property [
        sh:path :menuItemHasFoodIngredient ;
        sh:class :FoodIngredient ;
        sh:minCount 1 ;
    ] .

### MenuShape
:MenuShape
    a sh:NodeShape ;
    sh:targetClass :Menu ;
    sh:property [
        sh:path :menuHasTitle ;
        sh:datatype xsd:string ;
        sh:minCount 1 ;
    ] ;
    sh:property [
```

```
        sh:path :menuHasDescription ;
        sh:datatype xsd:string ;
        sh:minCount 0 ;
    ] ;
    sh:property [
        sh:path :menuHasMenuSection ;
        sh:class :MenuSection ;
        sh:minCount 1 ;
    ] .

### MenuSectionShape
:MenuSectionShape
    a sh:NodeShape ;
    sh:targetClass :MenuSection ;
    sh:property [
        sh:path :menuSectionHasTitle ;
        sh:datatype xsd:string ;
        sh:minCount 1 ;
    ] ;
    sh:property [
        sh:path :menuSectionHasMenuItem ;
        sh:class :MenuItem ;
        sh:minCount 1 ;
    ] .

### RestaurantShape
:RestaurantShape
    a sh:NodeShape ;
    sh:targetClass :Restaurant ;
    sh:property [
        sh:path :restaurantHasName ;
        sh:datatype xsd:string ;
        sh:minCount 1 ;
    ] ;
    sh:property [
        sh:path :restaurantHasMenu ;
        sh:class :Menu ;
        sh:minCount 1 ;
    ] .

### CaloricLevelShape
:CaloricLevelShape
    a sh:NodeShape ;
    sh:targetClass :CaloricLevel ;
    sh:property [
        sh:path :caloricLevelHasName ;
        sh:datatype xsd:string ;
        sh:minCount 1 ;
    ] .

### DietaryPreferenceShape
:DietaryPreferenceShape
    a sh:NodeShape ;
    sh:targetClass :DietaryPreference ;
    sh:property [
        sh:path :dietaryPreferenceHasName ;
        sh:datatype xsd:string ;
        sh:minCount 1 ;
    ] .

### AllergensShape
:AllergensShape
    a sh:NodeShape ;
    sh:targetClass :Allergens ;
    sh:property [
        sh:path :allergenHasName ;
        sh:datatype xsd:string ;
        sh:minCount 1 ;
    ] .
```

Listing 5.1: SHACL constraints

The SHACL document defines a series of "shapes" that correspond to various classes within the ontology. Each shape specifies the expected

properties for instances of these classes, including the data types of those properties, the minimum number of occurrences required, and relationships to other classes. These shapes function as templates or blueprints that data must follow, ensuring that the data conforms to the intended structure and meaning.

## 5.3 SWRL

SWRL (Semantic Web Rule Language) is a language used for expressing rules in the context of the Semantic Web. It extends OWL (Web Ontology Language) by allowing users to write rules that can infer new knowledge from existing data. SWRL combines OWL's ontology modeling capabilities with the expressive power of rule-based logic. A SWRL rule consists of an antecedent (a set of conditions) and a consequent (a set of conclusions), where if the conditions in the antecedent are met, the conclusions in the consequent are inferred. SWRL is often used for tasks such as automated reasoning, data validation, and enhancing the expressiveness of ontologies.

### 5.3.1 *Rules description*

Let's discuss the rules one by one.

**menuItemHasAllergen**

```
MenuItem(?m) ^ menuItemHasFoodIngredient(?m, ?i) ^
    foodIngredientHasAllergens(?i, ?allergen) ^ swrlb:notEqual(?
    allergen, "") -> menuItemHasAllergen(?m, ?allergen)
```

Listing 5.2: menuItemHasAllergen rule

This rule identifies allergens present in a menu item. It checks if a menu item (?m) has any food ingredients (?i) that contain allergens (?allergen). If an allergen is present and is not an empty value, the rule infers that the menu item has the identified allergen, associating it with the menu item through the property menuItemHasAllergen.

**menuItemHasDietaryRestriction**

```
MenuItem(?m) ^ menuItemHasFoodIngredient(?m, ?i) ^
    foodIngredientHasAllergens(?i, ?allergen) ^ swrlb:notEqual(?
    allergen, "") -> menuItemHasAllergen(?m, ?allergen)
```

Listing 5.3: menuItemHasDietaryRestriction rule

This rule determines if a menu item is categorized as suitable for omnivores. It checks if any ingredient (?i) of a menu item (?m) belongs to the "Omnivore" dietary category. If so, the menu item is labeled with the dietary restriction "Omnivore" using the menuItemHasDietaryRestriction property.

**High Caloric Category Rule**

```
MenuItem(?menuItem) ^ menuItemHasTotalCalories(?menuItem, ?
    calories) ^ swrlb:greaterThan(?calories, 900) ->
    menuItemHasCaloricCategory(?menuItem, "High")
```

Listing 5.4: High Caloric Category Rule

This rule classifies a menu item as "High" in calories. It checks if the total caloric content of the menu item exceeds 900 calories. If true, the rule assigns the "High" caloric category to the menu item using the menuItemHasCaloricCategory property.

#### Low Caloric Category Rule

```
MenuItem(?menuItem) ^ menuItemHasTotalCalories(?menuItem, ?
    calories) ^ swrlb:lessThanOrEqual(?calories, 400) ->
    menuItemHasCaloricCategory(?menuItem, "Low")
```

Listing 5.5: Low Caloric Category Rule

This rule classifies a menu item as "Low" in calories. It checks if the total caloric content of the menu item is 400 calories or less. If this condition is met, the menu item is assigned the "Low" caloric category.

#### Medium Caloric Category Rule

```
MenuItem(?menuItem) ^ menuItemHasTotalCalories(?menuItem, ?
    calories) ^ swrlb:greaterThan(?calories, 400) ^ swrlb:
    lessThanOrEqual(?calories, 900) ->
    menuItemHasCaloricCategory(?menuItem, "Medium")
```

Listing 5.6: Medium Caloric Category Rule

This rule classifies a menu item as "Medium" in calories. It checks if the total caloric content of the menu item is greater than 400 calories but not more than 900 calories. If this range is satisfied, the menu item is categorized as having a "Medium" caloric level.

### 5.3.2 *Putting the rules in the ontology*

In order to make those rules accessible in the ontology even without a reasoner, we've used Protege[1] by converting the rules into OWL language. This was done by using the operations in Figure 5.4.

[1]Protégé is a free, open-source ontology editor and knowledge management tool developed by Stanford University. It is widely used in the field of semantic web and artificial intelligence to create, edit, and manage ontologies, which are formal representations of knowledge within a specific domain.

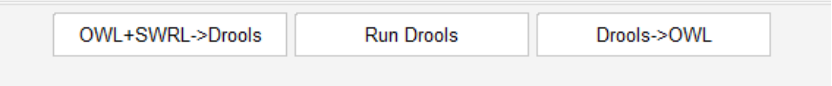


FIGURE 5.4: Protege drills for using SWRL rules

## 5.4 SPARQL QUERY

SPARQL (SPARQL Protocol and RDF Query Language) is a powerful query language used to retrieve and manipulate data stored in RDF (Resource Description Framework) format. It allows users to query datasets by specifying patterns in the data, similar to how SQL is used for querying

relational databases. SPARQL can be used to extract information, filter data, perform complex joins, and even modify RDF graphs. It supports queries across multiple datasets, making it particularly useful in the context of the Semantic Web, where data from diverse sources can be integrated and queried. SPARQL is essential for working with linked data and ontologies.

### 5.4.1 *Filtering the menu using SPARQL*

The following is the query that was used to filter the menu based on client's preferences

```
PREFIX : <http://www.semanticweb.org/byjus/ontologies/2024/6/KebiProject/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

SELECT ?itemName ?menuCaloricCategory ?personCaloricCategoryName ?allergen ?personAllergen ?dietaryRestriction ?personDietaryPreference
WHERE {
  # Retrieve Fabio's caloric preference
  ?person rdf:type :Person .
  ?person :personHasName "Fabio" . # Specify the individual's name
  ?person :personHasCalorieLevel ?personCaloricCategory .
  ?personCaloricCategory :caloricLevelHasName ?personCaloricCategoryName .

  # Retrieve Fabio's dietary preference
  ?person :personHasDietaryPreference ?personDietaryPreference .
  ?personDietaryPreference :dietaryPreferenceHasName ?personDietaryPreferenceName .

  # Retrieve Fabio's allergens
  OPTIONAL {
    ?person :personHasAllergens ?personAllergen .
    ?personAllergen :allergenHasName ?personAllergenName .
  }

  # Find menu items with the same caloric preference
  ?menuItem rdf:type :MenuItem .
  ?menuItem :menuItemHasCaloricCategory ?menuCaloricCategory .
  ?menuItem :menuItemHasName ?itemName .

  OPTIONAL { ?menuItem :menuItemHasAllergen ?allergen . }
  OPTIONAL { ?menuItem :menuItemHasDietaryRestriction ?dietaryRestriction . }

  # Caloric category filter
  FILTER(
    (?personCaloricCategoryName = "High" && (?menuCaloricCategory = "Low" || ?menuCaloricCategory = "Medium" || ?menuCaloricCategory = "High")) ||
    (?personCaloricCategoryName = "Medium" && (?menuCaloricCategory = "Low" || ?menuCaloricCategory = "Medium")) ||
    ?menuCaloricCategory = ?personCaloricCategoryName
  )

  # Dietary preference filter
  FILTER(
    (?personDietaryPreferenceName = "Omnivore" && (?dietaryRestriction = "Omnivore" || !bound(?dietaryRestriction))) ||
    ((?personDietaryPreferenceName = "Vegetarian" || ?personDietaryPreferenceName = "Vegan") && (!bound(?dietaryRestriction)))
  )

  # Allergen filter
  FILTER NOT EXISTS {
```

```
    ?menuItem :menuItemHasAllergen ?menuAllergenName .
    ?person :personHasAllergens ?personAllergen .
    ?personAllergen :allergenHasName ?menuAllergenName .
  }
}
}
ORDER BY ?itemName
```

Listing 5.7: SPARQL query to filter the menu

This SPARQL query is designed to identify menu items that align with the dietary needs and preferences of an individual named Fabio. The query works by first retrieving Fabio's caloric category, dietary preference, and any allergens he may have. It then searches for menu items that match these criteria.

Specifically, the query looks for menu items with a caloric category that fits Fabio's preferences, ensuring that the item falls within an acceptable range—whether that be high, medium, or low calories—depending on Fabio's personal caloric level. It also checks the dietary restrictions associated with each menu item, comparing them with Fabio's dietary preference, such as being omnivorous, vegetarian, or vegan.

Additionally, the query considers allergens, filtering out any menu items containing allergens that Fabio needs to avoid. The goal is to ensure that the recommended menu items not only meet his caloric and dietary needs but are also safe for him to consume.

Finally, the results are ordered by the menu item name, providing a list of food options that are suitable for Fabio based on all these considerations. The properties of the menu items, such as caloric categories and dietary restrictions, are derived from inferences made using SWRL rules, which ensure the data aligns with Fabio's specific requirements.

# 6

# *AOAME Implementation*

**AOAME** is a tool that allows to extend the BPMN from the ontology. In our case, **AOAME** was paired with the **Jena Fuseki** tool that is used to query the BPMN 2.0 and access its attribute so that they can be used inside the SPARQL queries.

## 6.1 BPMN DESCRIPTION

The image depicts a Business Process Model and Notation (BPMN) diagram designed to represent a process flow for a meal selection system. Below is a detailed description of the BPMN diagram for use in an academic report:

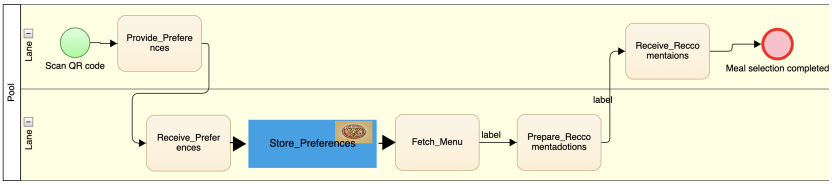


FIGURE 6.1: Full BPMN

The diagram is divided into two main horizontal sections, referred to as *lanes*, which together form a *pool*. Each lane represents a different participant or system involved in the process. The diagram captures the end-to-end process, from scanning a QR code to the completion of a meal selection based on personalized recommendations.

## 6.2 COMPONENTS

1. **Pool and Lanes**: The diagram consists of a single pool divided into two lanes. The top lane represents the customer interacting with the system. The bottom lane represents the backend system or service handling user preferences and recommendations.

2. **Start Event**. Scan QR Code (Green Circle): The process begins with the user scanning a QR code, which triggers the start of the workflow. This is represented by a green circle, indicating a start event.

3. **Provide Preferences (Top Lane)**: After scanning the QR code, the user is prompted to provide their meal preferences. This task is represented by a rounded rectangle.

4. **Receive Preferences (Bottom Lane)**: Once the user submits their preferences, the system receives this information, as depicted by a task in the bottom lane.

5. **Store Preferences (Bottom Lane)**: The system then stores the user's preferences. This task is highlighted with a blue background and includes an image of a pizza, suggesting the process is extended using AOAME.

6. **Fetch Menu (Bottom Lane)**: The system fetches the entire menu, in order to filter it later. This task is directly connected to the previous task.

7. **Prepare Recommendations (Bottom Lane)**: The system prepares meal recommendations using the fetched menu and stored preferences. This is the final processing step before results are returned to the user.

8. **Receive Recommendations (Top Lane)**: The prepared recommendations are then sent back to the user, completing the main user interaction process.

9. **End Event**. Meal Selection Completed (Red Circle): The process concludes when the user receives the recommendations, indicated by a red circle, representing the end event.

## 6.3 FLOW AND SEQUENCE

The process flow is indicated by arrows connecting each task and event in sequence. The user initiates the process by scanning the QR code, which leads to providing preferences. The system then processes these preferences through several steps before sending the recommendations back to the user.

## 6.4 EXTENDING THE BPMN USING AOAME

In this section we will talk about how we extended the **BPMN** and defined the **BPMN 2.0**. The task that was extended is the **store preferences** task, that is used to store the client's preferences in order for them to be accessed later into the Jena Fuseki tool. First thing first, we created a new canvas for in task so that it is recognizable. The canvas is the

one in Figure 6.2, it has a blue background and a pizza on the upper right corner. The pizza was used to put emphasis on the fact that we are talking about food.

FIGURE 6.2: New canvas design

The task was extended using the **extend task** button that can be reached by right clicking on the task option. This can be seen in Figure 6.3.

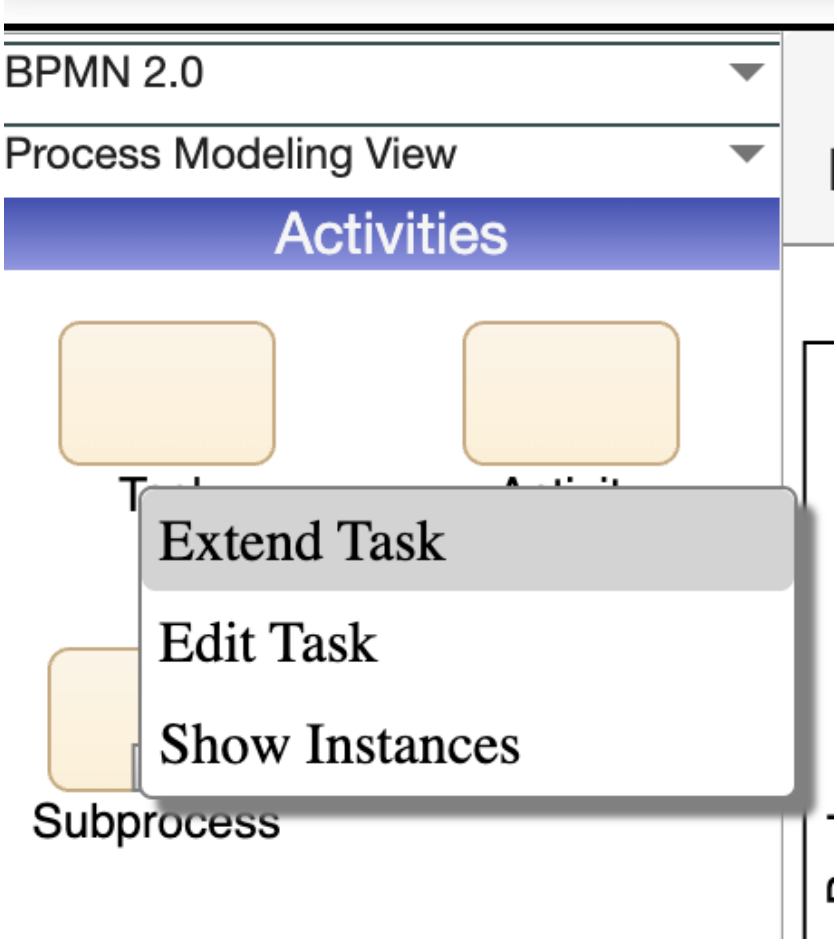


FIGURE 6.3: Extending task

Once inside the extend task menu, we were able to add the client's preferences as a *data property* inside the newly created *Store_Preferences* task. We added three new properties:

- **clientHasCaloricCategory**. This is the property that can contain either *low*, *medium* or *high*, depending on the client's preference;
- **clientHasDietaryCategory**. Same as before, this also can contain only the following three values: *vegan*, *vegetarian* and *omnivore*. This tells the system how the client should be categorized in terms of his diet;
- **clientHasAllergen**. This can contain various allergens, like for example *lactose*, *gluten*, *nuts* or *eggs*. A client can have multiple allergies, so we can just add the variable multiple times with all the allergens.

In Figure 6.4 we can see the 3 data properties we've just discussed.

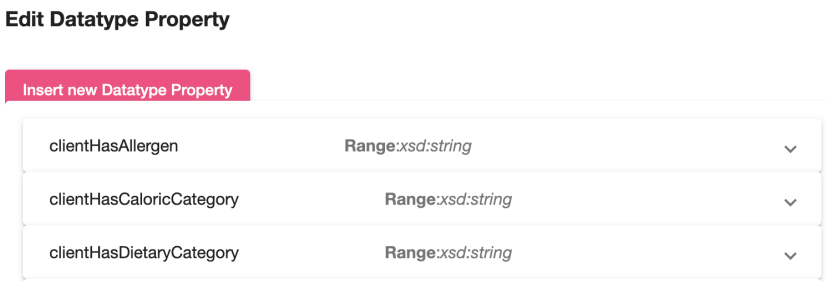


FIGURE 6.4: New data properties

As we can see they all are of type string. An example that we made is a client with a lactose allergy, an omnivore diet and a low caloric intake that can be seen in Figure 6.5.

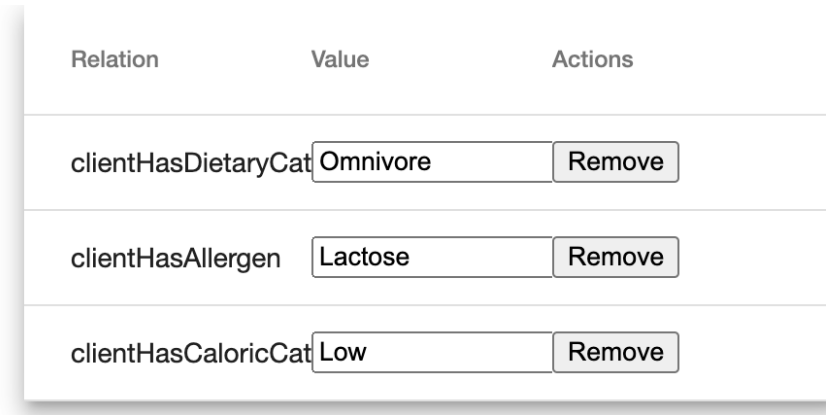


FIGURE 6.5: Client example

In order to use those properties in the *Jena Fuseki* tool we need the ID of the extended task. To get it, the following query was used

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX bpaas: <http://ikm-group.ch/archimeo/bpaas#>
PREFIX mod: <http://fhnw.ch/modelingEnvironment/ModelOntology#>
SELECT *
WHERE {
    ?subject ?relation ?object .
    ?subject rdfs:label "Store_Preferences".
}
```

Listing 6.1: Getting the ID

Using this, we extracted the following ID: **mod:prepareReccomendations_827b32b6-de81-48a3-91e8-2d2f009fa22c**.

We then used this id to get the client's preferences by just modifying the query previously mentioned in Section 5.4 and taking the preferences from the BPMN instead of the ontology. For example, the following code extracts the *clientHasDietaryCategory* property:

```
PREFIX : <http://www.semanticweb.org/byjus/ontologies/2024/6/
    KebiProject/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX bpaas: <http://ikm-group.ch/archimeo/bpaas#>
PREFIX bpmn: <http://ikm-group.ch/archiMEO/BPMN#>
PREFIX mod: <http://fhnw.ch/modelingEnvironment/ModelOntology#>
PREFIX lo: <http://fhnw.ch/modelingEnvironment/LanguageOntology#
    >
PREFIX md: <http://www.w3.org/ns/md#>

SELECT ?personCaloricCategoryName
WHERE {
    # Retrieve Fabio's caloric preference
    ?person rdf:type :Person .
    ?person :personHasName "Fabio" . # Specify the individual's
        name
    mod:prepareReccomendations_827b32b6-de81-48a3-91e8-2
        d2f009fa22c lo:clientHasCaloricCategory ?
        personCaloricCategoryName.
    }
```

Listing 6.2: Getting the data property

Once we were able to extract all of the properties, we queried our ontology like we already did before.

For reference, the following is the entire SPARQL query:

```
PREFIX : <http://www.semanticweb.org/byjus/ontologies/2024/6/
    KebiProject/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX bpaas: <http://ikm-group.ch/archimeo/bpaas#>
PREFIX bpmn: <http://ikm-group.ch/archiMEO/BPMN#>
PREFIX mod: <http://fhnw.ch/modelingEnvironment/ModelOntology#>
PREFIX lo: <http://fhnw.ch/modelingEnvironment/LanguageOntology#
    >
PREFIX md: <http://www.w3.org/ns/md#>

SELECT ?mealName ?menuCaloricCategory ?personCaloricCategoryName
     ?allergen ?personAllergenName ?dietaryRestriction ?
    personDietaryPreferenceName
WHERE {
  # Retrieve Fabio's caloric preference
  ?person rdf:type :Person .
  ?person :personHasName "Fabio" . # Specify the individual's
      name
  mod:prepareReccomendations_827b32b6-de81-48a3-91e8-2
      d2f009fa22c lo:clientHasCaloricCategory ?
      personCaloricCategoryName.

```

```
  # Retrieve Fabio's dietary preference
  mod:prepareReccomendations_827b32b6-de81-48a3-91e8-2d2f009fa22c lo:clientHasDietaryCategory ?personDietaryPreferenceName.

  # Retrieve Fabio's allergens
  OPTIONAL {
    mod:prepareReccomendations_827b32b6-de81-48a3-91e8-2d2f009fa22c lo:clientHasAllergen ?personAllergenName.
  }

  # Find menu items with the same caloric preference
  ?menuItem rdf:type :MenuItem .
  ?menuItem :menuItemHasCaloricCategory ?menuCaloricCategory .
  ?menuItem :menuItemHasName ?mealName .

  OPTIONAL { ?menuItem :menuItemHasAllergen ?allergen . }
  OPTIONAL { ?menuItem :menuItemHasDietaryRestriction ?dietaryRestriction . }

  # Caloric category filter
  FILTER(
    (?personCaloricCategoryName = "High" && (?menuCaloricCategory = "Low" || ?menuCaloricCategory = "Medium" || ?menuCaloricCategory = "High")) ||
    (?personCaloricCategoryName = "Medium" && (?menuCaloricCategory = "Low" || ?menuCaloricCategory = "Medium")) ||
    ?menuCaloricCategory = ?personCaloricCategoryName
  )

  # Dietary preference filter
  FILTER(
    (?personDietaryPreferenceName = "Omnivore" && (?dietaryRestriction = "Omnivore" || !bound(?dietaryRestriction))) ||
    ((?personDietaryPreferenceName = "Vegetarian" || ?personDietaryPreferenceName = "Vegan") && (!bound(?dietaryRestriction)))
  )

  # Allergen filter
  FILTER NOT EXISTS {
    ?menuItem :menuItemHasAllergen ?menuAllergenName .
    ?person :personHasAllergens ?personAllergen .
    ?personAllergen :allergenHasName ?menuAllergenName .
  }
}
ORDER BY ?mealName
```

Listing 6.3: Full SPARQL query

# Part III

# EPILOGUE

# 7

# *De Vitis Fabio Michele's Conclusions*

This chapter presents my reflections on the overall project, with a focus on evaluating the various business knowledge solutions we implemented. I will provide a detailed analysis of each phase of the project, highlighting the strengths and weaknesses of the different tools and approaches we employed. Finally, I will share my personal insights, offering a deeper examination of how these solutions performed and what I learned from the experience.

## 7.1 DECISION TABLES

The development and implementation of **decision tables** was one of the most challenging tasks in our project. We utilized **Decision Requirements Diagrams (DRD)** and **FEEL expressions** to create a dynamic and flexible system capable of handling complex client-specific logic.

Initially, our approach relied on a **sequential filtering method**, which quickly revealed its limitations in terms of **maintainability and extensibility**. The sequential approach made it challenging to retrieve information from previously executed tables that were not directly linked, and it was only possible to have the output data as input in the connected table. This posed a significant challenge when dealing with complex scenarios as it led to data replication. To address these issues, we transitioned to a more **modular design** in our DRD, inspired by the **Single Responsibility Principle (SRP)** from **Object-Oriented Design**. This approach allowed each decision component, such as filtering by allergens, dietary restrictions, and caloric categories, to be developed, tested, and maintained independently, vastly improving the system's **scalability and maintainability** compared to our initial solution.

Another key element in our decision table implementation was the use of **Hit Policies**. We predominantly used the "Collect" hit policy, which allowed us to aggregate outputs from multiple matching rows. This policy worked well when combined with FEEL expressions, enabling a more efficient and less repetitive development process.

Despite the improvements with the modular DRD design and the integration of FEEL expressions, we encountered challenges with the **Camunda Platform's** inherent static nature when **FEEL** was not fully utilized. Without dynamic capabilities, the decision logic was limited to basic conditions and outcomes, making it difficult to handle complex, real-world scenarios and leading to a more rigid system. The lack of detailed documentation, especially for advanced use cases, also required us to spend additional time on experimentation and testing. Since we did not find any detailed documentation regarding the FEEL language, as they only provided basic examples, we followed an online free course on the Camunda platform, which I strongly suggest. Here we learned how to map key-values using FEEL and how to perform particular operations on lists, enabling us to create the most dynamic environment we could think of.

After this, I encountered another limitation of DMN tables: their **data-specific nature**. Adapting them to different restaurant menus or datasets would require significant adjustments, reducing the portability of the decision tables and making them less versatile for various applications.

## 7.2 PROLOG

The next phase of our project involved implementing **Prolog**, a logic programming language that allowed us to address some of the limitations we encountered with decision tables and Decision Requirements Diagrams (DRD). Prolog's unique **logical inference capabilities** provided a higher degree of flexibility and adaptability, making it an excellent choice for handling client-specific meal recommendations.

Prolog distinguishes itself by its **dynamic nature**, which is particularly advantageous for scenarios where client preferences are highly individualized and subject to frequent changes. Unlike decision tables, which can become unmanageable when the decision logic grows in complexity, Prolog allows the system to evolve with minimal adjustments to the underlying logic. Its **extensible design** enables new dietary restrictions or preferences to be incorporated without requiring a complete remodeling of the existing code. Additionally, Prolog's **declarative syntax** plays a crucial role in making the codebase more transparent and maintainable, as it allows for a concise representation of complex rules and relationships.

However, working with Prolog is not without its challenges. One of the primary difficulties lies in the **unconventional programming paradigm** that Prolog employs. Unlike traditional imperative or object-oriented languages, Prolog is based on a form of logic programming that focuses on specifying what the solution should look like rather than detailing the steps to achieve it. This paradigm shift can be difficult for developers who are more used to procedural or object-oriented approaches. Matter of fact understanding concepts such as unification,

backtracking, and recursion in the context of Prolog required a different mindset, which can be intimidating initially.

Another significant challenge is **performance degradation** when handling large datasets or complex decision logic. Prolog's strength lies in its ability to infer solutions through pattern matching and logical reasoning, but this process can become computationally expensive as the volume of data increases. When managing extensive restaurant menus or complex dietary rules, the performance can suffer, resulting in slower response times. Optimizing Prolog code, therefore, becomes essential to ensure that the system remains responsive and capable of delivering recommendations in real-time.

Additionally, debugging in Prolog presents its own set of difficulties. Unlike more conventional programming environments, Prolog lacks the extensive suite of debugging tools that are available for other languages. The declarative nature of Prolog means that tracking down the source of an error often requires tracing through logical inferences, which can be both time-consuming and complex. This can make it harder to identify and correct issues, especially in larger projects with intricate logic.

Finally, Prolog's **dependency on the structure and organization of its knowledge base** poses another challenge.Prolog is capable of handling a large range of datasets, but the logic and rules need to be specifically customized to meet the needs of each application. Reconfiguring the knowledge base significantly is often necessary when adapting Prolog to new domains or drastically different contexts, like switching from one type of restaurant menu to another. Because of this dependency, Prolog can be very flexible within its defined domain, but when applied to other scenarios, it needs to be carefully planned for and modified to maintain its effectiveness.

## 7.3 KNOWLEDGE GRAPHS

The integration of **Protege** and related semantic web technologies with an **ontology-based methodology** added structure and logic to our meal recommendation system. We used **Semantic Web Rule Language (SWRL)** to infer **dietary restrictions** for each dish from its ingredients, but we faced constraints that blocked the complete classification of dishes into **omnivore, vegetarian, and vegan** categories.

SWRL allowed us to classify dishes as omnivore if they contained any animal products, but it could not further distinguish between vegetarian and vegan dishes due to its limitations, such as the inability to handle **negation as failure** and **disjunctions**. This led to a binary classification system instead of the desired three-category system. These limitations highlight the challenges of using rule-based systems like SWRL for complex dietary classification.

Despite these challenges, **SPARQL** proved effective for querying the data, and its similarity to SQL made it easy to learn. The ontology-based approach provided structured data management and dynamic relationship inference, but SWRL's limitations underscored the need for careful tool selection based on the task's requirements. However, the repetitive nature of modeling in Protege, including the manual creation of classes and instances, made the development process less efficient. Additionally, when using the reasoner to infer new knowledge, we encountered issues with unnecessary or incorrect inferences, which could negatively impact the querying process. To address this, we had to carefully modify our ontology and SWRL rules to ensure accurate and relevant results.

## 7.4 AOAME

The integration of **AOAME** (Agile and Ontology-Aided Modeling Environment) marked a significant advancement in our project by enabling the flawless incorporation of semantic rules into our meal recommendation system. AOAME is a web-based modeling tool, that allowed us to extend the BPMN framework, providing the capability to semantically enrich our process models. This enrichment was essential for ensuring that our BPMN workflows accurately reflected the complex dietary needs, preferences, and restrictions of our clients.

AOAME's key strength lies in its ability to extend traditional modeling languages with ontology-based semantics. This feature was fundamental in our project, as it enabled us to maintain a high level of data accuracy and consistency across various stages of the meal recommendation process.

The deployment of AOAME, however, was not without its challenges. The initial installation process was hindered by errors in the installation script and an unresponsive user interface. The last issue was eventually resolved by switching to a different web browser, which allowed us to bypass the interface problems and continue with the integration. Despite these initial setbacks, AOAME proved to be a powerful tool that significantly enhanced the robustness of our BPMN workflows.

Additionally, AOAME's integration with **Jena Fuseki**, allowed us to execute **SPARQL queries** directly within our BPMN models. This capability was crucial for dynamically retrieving and applying client-specific data during the recommendation process. The use of SPARQL within the BPMN framework enabled a level of flexibility that was otherwise unattainable, allowing us to query and manipulate data in ways that were both efficient and contextually relevant.By supporting the integration of ontologies and SPARQL queries, AOAME allows the system to adapt dynamically, meaning we could potentially change or add components such as Dietary Restrictions or the Client's preferences.This dynamic adaptability means that as the complexity or diversity of inputs increases, the system can scale to handle these changes without needing signifi-

cant reconfiguration, making it extremely scalable and overcoming the limitations we have encountered so far.

## 7.5 PERSONAL THOUGHTS & COMPARISONS

Looking back, this project was quite the learning experience for me. Sure, there were a lot of challenges along the way, but I really enjoyed the hands-on approach we took. It gave me a chance to dive deep into different tools and really understand not just what they can do, but also their limitations.

I initially believed Decision Tables to be very simple. They indeed are, since they consist just of a table with rules that, when certain conditions are met, provide you with particular outputs. But as our use cases became more intricate, requiring us to balance numerous components and a great deal of logic, Decision Tables quickly started to cause us headaches. Although initially perceived as a benefit, their simplicity and static nature proved to be a significant drawback. Trying to create a dynamic environment that could suggest meals based on user inputs was tough. We had to lean heavily on **FEEL expressions**, and honestly, we hit a lot of errors during testing. The lack of detailed documentation didn't help either since Camunda's docs were pretty bare-bones, especially when it came to more complex examples and how to work with FEEL. We ended up turning to an online course from Camunda, which was extremely instructive, especially for handling lists and more advanced operations.

Keeping a logical structure with Decision Tables was another challenge. The system only lets you pull from output columns, so avoiding data duplication was tricky and added another layer of complexity. In my opinion, Decision Tables are great for simple scenarios with small datasets, but they're not the best choice when you need something scalable and maintainable. For more complex situations, there are definitely better options out there.

One of those better options was **Prolog**. This logic programming language was a game-changer. It offered way more flexibility and was easier to maintain compared to Decision Tables. We could lay out the rules and facts about our project in a way that was clear and made sense. We managed to implement all the logic we had in mind, and it worked well—Prolog helped us come up with meal suggestions that were spot-on for what the client needed, and it even organized them by menu sections.

What made working with Prolog even more exciting was its continued relevance in the field of artificial intelligence.Its ability to handle complex logical inferences and pattern matching makes it an ideal choice for applications that require deep reasoning, such as diagnostic systems or intelligent agents. Knowing that I was working with a technology that

has played a critical role in AI made the experience even more rewarding.

With Prolog we were able to implement every single aspect we thought about on our project, we ended up with having a list of recommended meals that were also **ordered by Section.** But Prolog wasn't without its challenges either, in fact optimizing the performances was a little bit tricky.

Then there was our work with **Protege** for modeling **knowledge graphs** and using **SPARQL** to query the data. This approach felt to me like a middle ground between Decision Tables and Prolog. Knowledge graphs gave us a structured way to handle the relationships between data, which was really useful. But when we started using **SWRL** to infer dietary categories, we hit some limitations. SWRL's monotonic nature and its lack of support for negation and disjunctions made it hard to fully classify dishes into all the dietary categories we wanted, like omnivore, vegetarian, and vegan. On the bright side, SPARQL turned out to be a powerful tool for querying the data, and it was pretty easy to pick up since it's a lot like SQL.

Finally, we brought everything together with **AOAME** and **Jena Fuseki**. AOAME allowed us to integrate our ontology directly into the BPMN tasks, which was crucial for making sure all the client preferences, dietary restrictions, and allergens were accurately tracked and used in the recommendations. But getting AOAME up and running wasn't exactly smooth. But once we got it working, it was pretty powerful, especially when we paired it with Jena Fuseki. We could run SPARQL queries against the BPMN models, which let us dynamically pull client preferences from the extended BPMN tasks and use them to generate meal recommendations.

In the end, each tool we worked with had its own strengths and weaknesses. **Decision Tables** were simple but got unwieldy with complexity. **Prolog** was super flexible but came with has some optimization concerns. **Knowledge graphs** offered great structure but had limitations in reasoning due to SWRL. And **AOAME** and **Jena Fuseki** brought everything together with some limitations due to the lack of proper documentation. If I had to choose for the best option I would say Prolog, but the combination between Ontologies and AOAME has the potential to surpass it. Despite the challenges, this project gave me a chance to really dig into these tools and see what they're capable of, using a real dataset from an Italian Restaurant made it more challenging and interesting at the same time. I feel like I've come away with a much deeper understanding of knowledge-based systems, and I'm sure the lessons I've learned will help me choose the right tools and approaches for future projects.

# 8

# *Francesco Finucci's Conclusions*

This project aimed to develop an intelligent system capable of filtering a restaurant menu based on the client's dietary preferences, allergen sensitivities, and desired caloric intake. The system leveraged three key technologies: Prolog, decision tables, and ontologies, to achieve a dynamic and responsive filtering mechanism. We were also able to use the new BPMN 2.0 technology that allowed us to perform agile modelling by extending the BPMN starting from the ontology. This conclusion will discuss the results of the project, evaluate the effectiveness of the chosen technologies, and provide an analysis of their pros and cons.

## 8.1 SUMMARY OF ACHIEVEMENTS

The project successfully implemented a system that takes as input the client's dietary preferences, allergen sensitivities, and caloric requirements, and outputs a tailored menu. This system addressed the growing demand for personalized dietary options in the food industry by enabling restaurants to offer menus that meet the specific needs of their clients.

For the first part of the project, three technologies were used:

- **Decision tables** were utilized alongside the FEEL expression language to structure and evaluate filtering rules systematically. FEEL enhanced the expressiveness and readability of the rules, ensuring a comprehensive and understandable way to encode dietary preferences, allergens, and caloric intake;

- **Prolog**, a logic programming language, was used for the reasoning process. Prolog's strengths lie in its pattern matching and backward chaining capabilities, which were essential for handling complex logical queries about dietary requirements and food item properties;

- **Ontologies** were a central component of this project, providing a structured, semantic framework for modeling the relationships between food items, dietary categories, allergens, and nutritional information. By employing ontologies, the system could go beyond simple keyword matching and apply a deeper, more nuanced understanding of how different concepts relate to each other. This

semantic depth was further enhanced by the integration of SHACL, SWRL, and SPARQL, each contributing to the system's reasoning and validation capabilities.

For the second part, we used **AOAME** in combination with the **Jena Fuseki** querying tool. AOAME is used to compose the BPMN and to extend it by adding custom properties, while Jena Fuseki allows us to access those properties so that they can be used inside SPARQL queries.

## 8.2 EVALUATION OF TECHNOLOGIES

### 8.2.1 *Decision Tables*

Decision tables were our first approach to a knowledge-based system that allowed us to filter the menu using the client's preferences. Decision tables are a very organized way of handling knowledge and decisions, since you have to explicitly specify every single rule in order for it to work. This allows an easy understanding of the tables and makes them easily modifiable for newly added rules or if we want to change the logic behind them. Also, the addition of FEEL significantly enhanced the expressiveness of the rules, allowing for more detailed and nuanced conditions to be articulated.

Of course, there are some cons in using decision tables. **Complexity management** is one of those. Decision tables are excellent for organizing rules, but as the number of variables and conditions grows, the tables can become increasingly complex and difficult to manage. This complexity can lead to a situation where the decision tables, initially intended to clarify and simplify rule management, become cumbersome and challenging to maintain. The static nature of decision tables also means that any change in the rules or the introduction of new variables requires manual updates, which can be time-consuming and prone to human error. There are also some **Potential Performance Overhead**. As the tables grow and the menu gets larger and more complex the FEEL expression language isn't able to keep up as FEEL isn't thought out to be able to handle overly complex tasks.

### 8.2.2 *Prolog*

The Prolog-based solution provided a powerful tool for logical reasoning. Prolog's strength lies in its ability to handle complex logical queries and pattern matching, making it a strong candidate for filtering menus based on a wide range of dietary preferences and restrictions. Its declarative nature allowed for concise expression of rules, making the logic behind the system transparent and relatively easy to follow. However, the project also highlighted some challenges associated with using Prolog. Its syntax and logical paradigm presented a steep learning curve, particularly for those unfamiliar with logic programming. Additionally, as the complexity of the queries increased, performance issues became apparent, particularly when handling larger datasets. These challenges

underscored the need for careful optimization when using Prolog for real-time applications.

### 8.2.3 *Knowledge graphs and ontologies*

The third solution leveraged the semantic depth of ontologies, combined with SHACL for data validation, SWRL for complex rule expression, and SPARQL for querying. This approach was particularly effective in modeling and understanding the intricate relationships between food items, dietary preferences, allergens, and caloric values. Ontologies provided a rich, semantic framework that allowed the system to move beyond simple keyword filtering to a more nuanced understanding of how different dietary factors interact. SHACL added a crucial layer of validation, ensuring that the data used in the filtering process was consistent and adhered to defined constraints. SWRL extended the reasoning capabilities by enabling the expression of complex logical rules, which could accommodate specific dietary restrictions or preferences. SPARQL was instrumental in querying the ontology, allowing for precise retrieval of relevant data based on the client's input.

While this approach offered significant advantages in terms of semantic reasoning and data validation, it also introduced several challenges. The complexity of managing and maintaining the ontology, especially when combined with SHACL, SWRL, and SPARQL, required a high level of expertise. The performance overhead associated with semantic reasoning and complex querying posed additional challenges, particularly in scenarios requiring real-time responsiveness.

### 8.2.4 *AOAME and Jena Fuseki*

One of the standout advantages of using AOAME in the menu filtering process is its capability to dynamically extend and update ontologies. In a domain as fluid as dietary preferences and allergens, where new trends and requirements frequently emerge, AOAME provides the flexibility to easily incorporate new concepts into the existing ontology. For example, if a new allergen becomes widely recognized or a new dietary trend emerges, AOAME allows for these to be seamlessly integrated into the system without disrupting the overall structure. This dynamic adaptability ensures that the system can stay up-to-date with minimal effort, making it highly effective for real-world applications where change is constant. AOAME's ability to maintain consistency between human-interpretable modeling constructs (such as visual diagrams or UI elements) and machine-interpretable ontological data is another significant advantage.

The biggest cons is that AOAME is a relatively new tool, one of the major drawbacks is the scarcity of documentation and community resources. Unlike more established tools, where developers can easily find tutorials, examples, and forums to troubleshoot issues, working with AOAME can be challenging due to the lack of readily available support. This can slow down the development process, especially when trying to implement more advanced features or when encountering

unexpected behavior in the system. The limited documentation means that developers might have to spend additional time experimenting and problem-solving, which can be a significant drawback in time-sensitive projects.

## 8.3 FINAL CONSIDERATIONS

Working on this project allowed me to understand knowledge based systems better. I understood how to collect, organize and exploit knowledge in order to make informed decision and allow digital systems to be able to work with knowledge. It is crucial to give knowledge a machine-interpretable format, otherwise it can't be used by computers. Knowledge starts from implicit one like employees of a company knowing something and it needs to be processed until it can be programmed. The process of transforming basic knowledge about italian dishes into multiple complex knowledge-based systems was a challenging but very useful and informative process that will allow me to make better use of knowledge in my future projects.

As far as how i liked the various tools, i've found the decision table to have a quite steep learning curve and even tho they've allowed us to develop a simple but effective solution, i don't think they are suited for tasks more complex than a simple menu filtering. I've also found Camunda and FEEL to be not very documented in general, we did have an hard time figuring out how to use those tools effectively.

Prolog, on the other hand, is a programming-like language that was easier to get a grasp on and that even tho the concept behind it is very simple it allowed us to do very complex reasoning with less effort than decision tables.

Knowledge graphs and ontologies allowed us to describe the menu with a precise graph-based system. The ontology-based solution, enriched with SHACL, SWRL, and SPARQL, provided the most semantically sophisticated approach, capable of handling intricate relationships and complex rules. The experience of using ontologies was also enriched by the use of AOAME and its capabilities of extending the BPMN in order to add new data properties and use them in following SPARQL queries. This allowed an agile-based modelling and a more dynamic way of developing knowledge graphs, making the BPMN part of the ontology.