University of Camerino

School of Sciences and Technologies

Master degree in Computer Science (Class LM-18)

Project "Personalized Menu"

Knowledge Engineering and Business Intelligence - Project

Group members

Francesco Caldarelli - 132380
Edoardo Papa - 131313

Supervisors

Prof. Dr. Knut Hinklemann
Prof. Dr. Emanuele Laurenzi

A.A. 2024/2025

Contents

Abstract

The “Personalized Menu” project addresses the challenge of digitizing menus in restaurants by proposing innovative solutions to customize the display of dishes according to guests’ specific needs and preferences. Considering the limited visibility on smartphone screens, the project aims to create an intelligent system that displays only the appropriate dishes for each guest, taking into account dietary restrictions such as allergies (lactose, gluten) and dietary preferences (vegetarians, carnivores, calorie-conscious).

The solution involves the development of detailed knowledge bases on typical Italian restaurant dishes, including ingredients and nutritional values, and the profile of different types of customers. Three different knowledge representation techniques have been adopted: decision tables, logic programming in Prolog, and knowledge graphs/ontologies with SWRL rules, SPARQL queries, and SHACL constraints.

In addition, the project explores agile and meta-modeling ontology-based integration with the BPMN 2.0 language to support restaurants in dynamically managing customized menus through an intuitive interface, facilitating interaction and enhancing the guest experience.

1. Introduction

1.1 Project Description

In recent years, many restaurants have digitized their menus, allowing guests to access them by simply scanning a QR code with their smartphones. While this solution is convenient and environmentally friendly, it also has some limitations—especially when the menu is large. Viewing many items on a small screen can make it difficult to get a clear overview and find suitable options quickly.

Additionally, guests often have different dietary needs and preferences. Some may be vegetarian, others may suffer from food allergies or intolerances, and some may be particularly mindful of their calorie intake. Showing the full list of meals to every guest, regardless of their needs, can make the user experience overwhelming and inefficient.

The goal of this project is to improve the digital menu experience by making it more personalized and user-friendly. The idea is to develop a system that intelligently filters the available meals, displaying only those that match the guest's dietary preferences and restrictions.

To achieve this, a knowledge base will be created to represent the ingredients used in various meals, along with relevant attributes such as ingredient type (e.g., meat, vegetables, dairy) and nutritional information like calorie content. Guest profiles will also be defined, including categories such as carnivores, vegetarians, calorie-conscious individuals, and people with specific allergies or intolerances (e.g., lactose or gluten).

The resulting system will enhance the dining experience by offering guests a tailored view of the menu, simplifying their choices and aligning with their personal preferences.

1.2 Project Objective

The primary objective of this project is to develop and implement a robust, knowledge-based digital menu system capable of providing personalized meal recommendations. Specifically, the project aims to:

- Create diverse knowledge-based solutions using various representation languages, including:
  1. decision tables with Decision Requirements Diagrams (DRD)
  2. logical programming with Prolog
  3. ontology-based knowledge graphs enhanced with SWRL rules, SPARQL queries, and SHACL shapes.
- Adapt the Business Process Model and Notation (BPMN 2.0) framework through agile and ontology-based meta-modelling, facilitating meal suggestions tailored

specifically to customer profiles. This includes extending BPMN elements and developing intuitive graphical notations easily comprehensible by restaurant managers. Leverage advanced query interfaces (Jena Fuseki triple store) to efficiently manage and retrieve relevant information from the knowledge base.

2. Technologies

2.1 Trisotech

Trisotech is a high-end visual modeling tool for process model development and editing based on standard notations such as Business Process Model and Notation (BPMN), Case Management Model and Notation (CMMN), and Decision Model and Notation (DMN).

It is part of a cloud-based platform that is particularly geared towards business process automation and management, offering advanced tools to facilitate enterprise integration and collaboration.

In this project, Trisotech was chosen because it has the ability to execute directly within the cloud environment, allowing end users to build, share, and edit models in real-time and remotely, thus supporting distributed development.

A few of the most notable features of Trisotech are:

- **BPMN Modeling:** Supports the assembly of complex workflows by defining user tasks, service tasks, scripts, and other key elements required to orchestrate processes.

- **DMN Modeling:** Supports explicit and intuitive definition of decision points, optimizing business decisions using systematic decision models.

- **CMMN Modeling:** Provides facilities for managing complex cases, facilitating the creation of flexible and dynamic models tailored to evolving business situations.

Trisotech provides an intuitive interface by means of which people can easily create and modify process models. It provides collaborative as well as embedded features to concurrently work on models and enable communication among team members, along with integration with the systems already being used. It also allows multiple scenarios to be simulated and analyzed for model performance to enable better decision-making at a strategic level.

In general, Trisotech is an intuitive and all-encompassing business process model and design tool through which organizations can create, modify, and deploy efficient process models, thus achieving objectives in terms of process automation and optimization.

Throughout this project, Trisotech was selected because it was directly usable in the cloud environment so that users could build, share, and modify models in real time and remotely, thus providing a virtual development environment.

2.2 Prolog

Prolog is a declarative programming language that uses formal logic to represent information and perform deductive reasoning. Unlike traditional imperative languages, Prolog expresses code through logical statements that describe facts and rules, rather than sequences of instructions executed procedurally.

Applications in Prolog are primarily composed of three distinct elements:

- **Facts**: These represent basic knowledge about the domain, specifying relationships between objects or their attributes. They are formulated using predicates that connect various arguments.

- **Rules**: These define logical conditions for deducing new information from already established facts. Rules are expressed as implications, using the implication operator (‘:-‘).

- **Queries**: These allow interaction with the program by verifying certain conditions or searching for answers. When executing queries, Prolog uses logical inference techniques, such as backtracking, to systematically explore all possible solutions.

The Prolog inference engine also relies on the concept of **unification** to match queries and rules with available facts, allowing advanced pattern matching. Additionally, Prolog allows the combination of multiple logical conditions through operators like conjunction (‘,‘) and disjunction (‘;‘).

Thanks to its declarative nature and effective handling of pattern matching, Prolog is particularly well suited for applications that involve symbolic reasoning and knowledge-based systems. Its structure allows efficient modeling and manipulation of complex knowledge, making it a valuable tool in many fields where logical deduction is essential.

2.3 Protégé

Protégé is a widely adopted open-source ontology editor and knowledge-based framework, developed by the Stanford Center for Biomedical Informatics Research. Provides a powerful,et user-friendly environment for building, editing, and managing ontologies using the Web Ontology Language (OWL).

Ontologies are formal and structured representations of knowledge that define concepts, relationships, and properties within a specific domain. Protégé facilitates the creation and maintenance of ontologies by offering a broad range of modeling capabilities, such as:

- **Class modeling**: Defining hierarchical class structures to represent domain concepts (e.g., `Dish`, `Dessert`, `Second`).

- **Object properties**: Specifying relationships between individuals, such as `has_ingredient`, `has_allergen`, or `has_category`.

- **Data properties**: Associating individuals with literal data values, including numerical or textual information (e.g., `has_dish_calories`, `has_customer_rate`).

- **Individual instances**: Creating concrete examples or entities within the ontology (e.g., `grilled_seitan` as an instance of `Second`).
- **Logical axioms**: Defining constraints and conditions that support consistency checking and automated reasoning.

Protégé also supports the use of the **Semantic Web Rule Language (SWRL)**, which allows the specification of logical rules for automatic inference. For example, a rule can be defined that classifies dishes according to their calorie content. These rules, combined with integrated reasoners such as **HermiT** or **Pellet**, allow users to:

- Detect inconsistencies in the ontology.
- Infer implicit class memberships and property assertions.
- Apply rule-based reasoning for advanced classification.

Another essential feature is support for **SPARQL**, the query language for RDF data. Protégé includes a built-in SPARQL editor that allows users to retrieve and explore ontology data through structured queries. For example, the following query selects all individuals of type `Dish`:

```
SELECT ?dish WHERE { ?dish a :Dish . }
```

2.4 ADOxx

ADOxx is an platform for creating individual modeling tools. The platform is based on a meta-modeled architecture, in which not only models, but also behavior, rules, and interaction of modeling constructs can be specified. This renders ADOxx particularly suitable for rapid prototyping of individual modeling languages for academic as well as industrial research.

Some highlights are:

Support for meta-model definition: enables creation and management of concepts, attributes, relations and semantic rules.

Custom graphical environment: enables the creation of model-specific user interfaces.

Extensibility: offers a script language (ADOscript) and connectors for integration with other systems (e.g., web services, databases).

Simulation and validation of models: enables checking of process dynamics, logical constraints, and other user-definable properties.

In the context of this project, ADOxx was utilized for (insert purpose here: e.g., "modeling user decision making," "designing a knowledge-based solution," etc.), allowing for structured visual and semantic capture of key concepts.

3. Knowledge Base

A Knowledge Base is an organized collection of information, facts, rules, and concepts, properly organized and stored in order to facilitate access and retrieval. It is a repository of information for an individual, organization, or system.

A Knowledge Base contains a collection of domain-related information, where the main objective is to capture and organize real-world knowledge in such a form so that it can be utilized for decision-making, problem-solving, etc. It can or cannot include explicit knowledge, which is documentable and codifiable with ease, and tacit knowledge, which is experience-related and context-oriented.

3.1 Inputs

In our example, the inputs to the Knowledge Base are the **guest preferences** and the **restaurant menu**. They are needed for the selection process.

The guest preferences are divided into *diet type, allergies, and calories.*

3.1.1 Resturant Menu

The restaurant menu is a comprehensive list of dishes, each described by attributes such as:

- **name**: a unique identifier used to distinctly recognize each dish.
- **course**: specifies the type of course, such as main course, second course, or dessert.
- **category**: indicates the compatibility of the dish with specific dietary needs or restrictions (like carnivore, vegetarian, vegan, pescatarian).
- **totalCalories**: represents the total calorie content of the dish, crucial for managing dietary intake (Calories Conscious).
- **rate**: provides a qualitative rating of the dish, evaluated on a scale from 1 (lowest) to 5 (highest), reflecting its overall appeal or quality.
- **allergens**: enumerates all allergens present in the dish, essential for allergy management.
- **ingredient**: details the list of ingredients included in the dish, ensuring transparency and aiding informed choice.

First Courses

Lasagna with Meat Sauce (850 kcal)
Classic lasagna featuring fresh egg pasta, rich meat sauce, creamy béchamel, and parmesan cheese. *Perfect for lovers of traditional Italian cuisine.*
Allergens: gluten, eggs, milk
Diet type: Carnivore
Rating: 4.8

Pumpkin Risotto (650 kcal)
A creamy, delicate vegetarian risotto with sweet pumpkin and parmesan, finished with butter. *Ideal for a light yet flavorful meal.*
Allergens: milk
Diet type: Vegetarian
Rating: 4.3

Legume Soup (430 kcal)
A healthy vegan soup with mixed legumes, crunchy celery, and fresh carrots, finished with extra virgin olive oil. *Excellent choice for those seeking a light yet tasty dish.*
Allergens: celery
Diet type: Vegan
Rating: 3.0

Spaghetti with Clams (610 kcal)
A maritime classic with al dente spaghetti and fresh clams, seasoned with olive oil and parsley. *Perfect for seafood lovers.*
Allergens: gluten, mollusks
Diet type: Pescatarian
Rating: 2.7

Pasta with Walnut Pesto (650 kcal)
Flavorful vegetarian pasta with a unique walnut and olive oil pesto. *A pleasant alternative to classic pesto.*
Allergens: gluten, tree nuts
Diet type: Vegetarian
Rating: 2.7

Vegetable Couscous (550 kcal)
Light and nutritious vegan couscous served with grilled mixed vegetables. *Excellent balance between health and taste.*
Allergens: gluten
Diet type: Vegan
Rating: 3.3

Second Courses

Grilled Steak (550 kcal)
Juicy beef steak perfectly grilled with extra virgin olive oil. *A must-have for meat*

lovers.
Allergens: none
Diet type: Carnivore
Rating: 2.9

Zucchini Omelet (330 kcal)
Delicate vegetarian omelet with fresh zucchini and parmesan cheese, fluffy and flavorful. *Perfect for a light meal.*
Allergens: eggs, milk
Diet type: Vegetarian
Rating: 2.6

Grilled Seitan (400 kcal)
Vegan seitan accompanied by grilled vegetables, a tasty and protein-rich plant-based alternative. *Ideal for vegan diets.*
Allergens: gluten, soy
Diet type: Vegan
Rating: 2.6

Salmon Fillet (460 kcal)
Grilled salmon fillet, lightly seasoned with lemon and olive oil. *Great for a balanced diet.*
Allergens: fish
Diet type: Pescatarian
Rating: 4.7

Stir-Fried Tofu with Vegetables (370 kcal)
Flavorful stir-fried tofu with fresh vegetables and soy sauce. *Perfect balance between lightness and taste.*
Allergens: soy
Diet type: Vegan
Rating: 4.4

Lupin Burger (500 kcal)
Vegan burger made with lupins, served with crispy bread and fresh vegetables. *A nutritious alternative to traditional meat burgers.*
Allergens: gluten, lupins
Diet type: Vegan
Rating: 4.4

Desserts

Hazelnut Cake (750 kcal)
Rich and delicious vegetarian cake with roasted hazelnuts, butter, and eggs. *Ideal for those who appreciate homemade pastry flavors.*
Allergens: gluten, eggs, milk, tree nuts
Diet type: Vegetarian
Rating: 3.7

Fresh Fruit Salad (150 kcal)
Fresh and light vegan fruit salad, the perfect end to any meal.
Allergens: none
Diet type: Vegan
Rating: 3.0

3.1.2 Guest Preferences

Diet Type

The diet type to be selected may be one among the following types:

- carnivore
- vegetarian
- vegan
- pescatarian

Obviously, only a single option could be selected, and hence, a carnivore cannot select also vegetarian, etc.

Allergies

The following list represents the most common food allergens that need to be considered when preparing meals and designing customized menus. These ingredients can cause allergic reactions, even severe ones, in sensitive individuals. It is essential to clearly mark them on menus to ensure the safety and well-being of all guests.

The Allergies to be selected may be one among the following types:

- Sulfur dioxide and sulfites
- Tree nuts
- Gluten
- Milk
- Lupins
- Mollusks
- Fish
- Celery
- Sesame seeds
- Soy
- Eggs

Calories

The Calories Conscious to be selected may be one among the following types:

- < 400
- ≥ 400 e < 600
- ≥ 600

3.2 Output

The output generated by our knowledge base consists of a set of recommended dishes, selected based on the user's expressed preferences and the restaurant's available menu. Specifically, the system can produce two types of output:

- A list of personalized dishes, filtered according to the user's preferences (allergies, diet, calories, etc.), with a recommended proposal for each course (first course, second course, dessert, etc.), selected from the compatible and top-rated dishes by customers;
- An informational message, in case there are no dishes compatible with the preferences and constraints specified by the user.

4. Decision Table

This chapter will describe the development of the knowledge base in DMN and the creation of the decision tables using Trisotech (Section 2.1 of the delivery). It is important to note that two distinct versions were implemented. The first, which is more procedural in nature, involves extensive use of FEEL programming; while in the second, it was instead reorganized with the goal of being as declarative as possible, minimizing the use of FEEL expressions.

4.1 Decision Model and Notation

Decision Model and Notation (DMN) is an Object Management Group (OMG) standard, an international group engaged in specifying standard process management languages. DMN is being widely adopted in many industries to capture business decisions and drive the automation of their execution. Its formal and shared structure enables collaboration between the business community and IT and facilitates the specification of repeatable, easily maintained decision rules. With DMN, there is real payoff: operational efficiency is increased, the possibility of human errors is minimized, and decision models are simply shared and reused throughout the organization. DMN is one of three significant standard business modeling languages, along with BPMN (Business Process Model and Notation) and CMMN (Case Management Model and Notation). Although each can be employed individually, they are designed to be complementary. In fact, in most business settings in which they are used, they are employed together:

- BPMN for formal process flow definition;
- CMMN for less formal, case-based activities;
- DMN for governing complex, multi-criteria rule-based choices;

Because of the above facts, BPMN, CMMN and DMN are most often the "triple crown" standards for business process optimization.

4.2 DMN Diagram

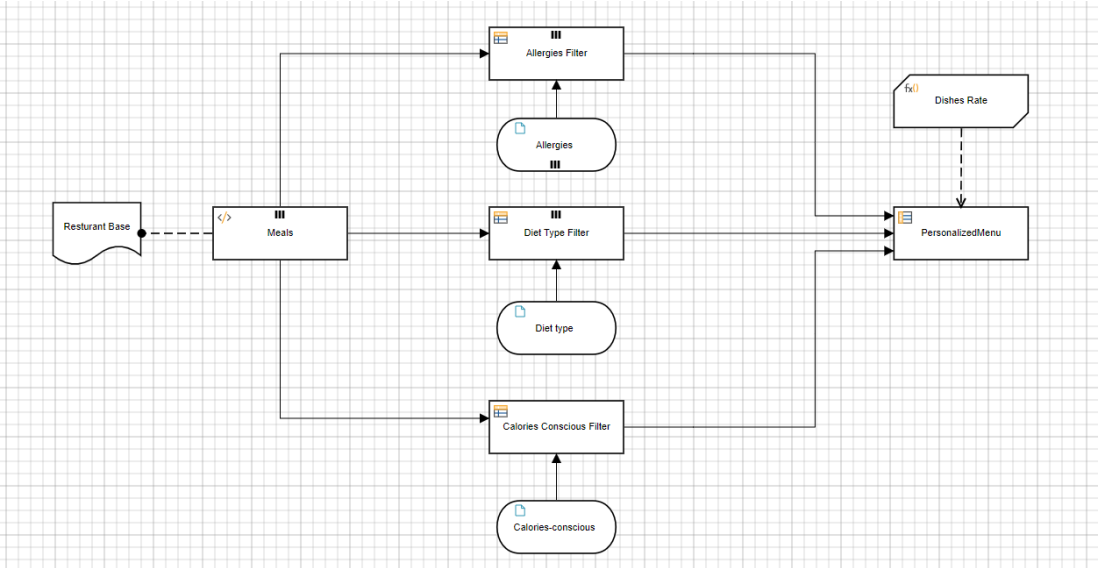
4.2.1 First Version

In this project we focus on the DMN diagram and decision tables. For the knowledge base, we tried to create in this first version a DMN that could be implemented in a realistic or hypothetical scenario of an application through which customers of a restaurant can view a sublist of the menu based on the preferences he/she has chosen

from: diet type, allergies, and calories. The DMN diagram in our project consists of 4 decision tables:

- "Allergies Filter";
- "Diet Type Filter";
- "Calories Conscious Filter";
- "PersonalizedMenu";

We then also deliberately added an additional filter that allows us to suggest to the user one dish per course that has the highest customer rating, but still respecting the user's choices (this filter call "Dishes Rate").



In the context of business process modeling and decision management, FEEL (Friendly Enough Expression Language) is a language specifically designed for expressing decision rules and logic in a human-readable and understandable format. It is used in conjunction with the Decision Model and Notation (DMN) standard. FEEL is a highly expressive language that aims to strike a balance between being business-friendly and having the necessary computational power for decision automation. It allows for the representation of complex expressions and decision logic in a straightforward and intuitive manner.

Allergies Filter

The "Allergies Filter" decision table represents a key component of the DMN model developed to customize menus based on a user's self-reported allergies. The table consists of: inputs and outputs, and is structured to adopt a "Unique" type of evaluation policy, i.e., it selects only one row from the possible rows (although, in this specific case, the row condition is always true, so there would be no ambiguity or need for conflict). The input condition was written following the FEEL syntax and is:

if Allergies instance of list then Allergies else [Allergies]

This ensures that the input is always treated as a list, even if the user has entered only one value.
The output was also written following the FEEL syntax and is:

Meals[not(some i in item.allergens satisfies i in Allergies)]

This FEEL expression returns only those dishes (Meals) for which none of the allergens in the dish matches those declared in the user's Allergies list. In other words, it excludes any dish that contains at least one of the stated allergens.

This table plays an essential role in ensuring users' food safety by dynamically filtering the proposed dishes according to the declared allergies. The use of a compact logical structure and the FEEL language allows an expressive and readable definition of the rules, minimizing complexity while maintaining a high level of customization of the offer.

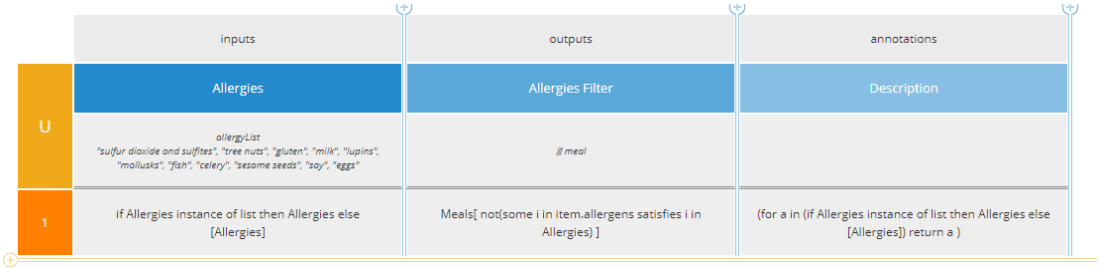


Figure 4.1: Filtro allergie nel DMN.

Diet Type Filter

The "Diet Type Filter" decision table is designed to filter the dishes offered in the menu according to the type of diet selected by the user. This allows only meals compatible with the user's dietary choices, such as a vegetarian, vegan, or pescetarian diet, to be returned to the user. The "First" evaluation policy was adopted, which implies that the output of the first row whose input condition is met is returned. In this case it is appropriate because each row is mutually exclusive of the others: each represents a distinct type of diet. In input we thus have 5 possible data items:

1. carnivore
2. vegetarian
3. vegan
4. pescatarian
5. -

The output returned is a filtered list of dishes (Meals) in which the category of the dish (item.category) corresponds to the type of diet selected:

Meals[item.category = Diet type]

In the last row, where the input Diet type is missing (-), the entire Meals list is simply returned, i.e., no filter is applied. This table is essential to tailor the food

offerings to users' dietary needs, improving the personalized experience. The use of the "First" policy avoids ambiguity, ensuring that each choice is handled directly and unambiguously.

F	inputs	outputs	annotations
	Diet type	Diet Type Filter	Description
	dietTypeList *"carnivore", "vegetarian", "vegan", "pescatarian"*	*meal*	
1	"carnivore"	Meals[item.category = Diet type]	
2	"vegetarian"	Meals[item.category = Diet type]	
3	"vegan"	Meals[item.category = Diet type]	
4	"pescatarian"	Meals[item.category = Diet type]	
5	-	Meals	

Calories Conscious Filter

The "Calories Conscious Filter" decision table allows users to filter the dishes available on the menu based on the total amount of calories each dish contains. This feature allows users to choose dishes consistent with their dietary goals or calorie restrictions.

The table adopts the "First" evaluation policy, which returns the result of the first row whose condition is true. This choice is justified by the structure of the table itself, in which the conditions are mutually exclusive: a dish can only fall within one of the specified calorie ranges.

In input we thus have 4 possible data items:

1. "< 400" - dishes with less than 400 kcal
2. ">= 400 < 600" - dishes between 400 and 600 kcal included
3. ">= 600" - dishes with at least 600 kcal
4. "-" - no preference

The output field returns the dishes (Meals) filtered based on the sum of calories contained in the individual ingredients of the dish, as shown below:

- Row 1 (< 400):

$$\textit{Meals[sum(item.ingredients.calories) < 400]}$$

- Row 2 (>= 400 < 600):

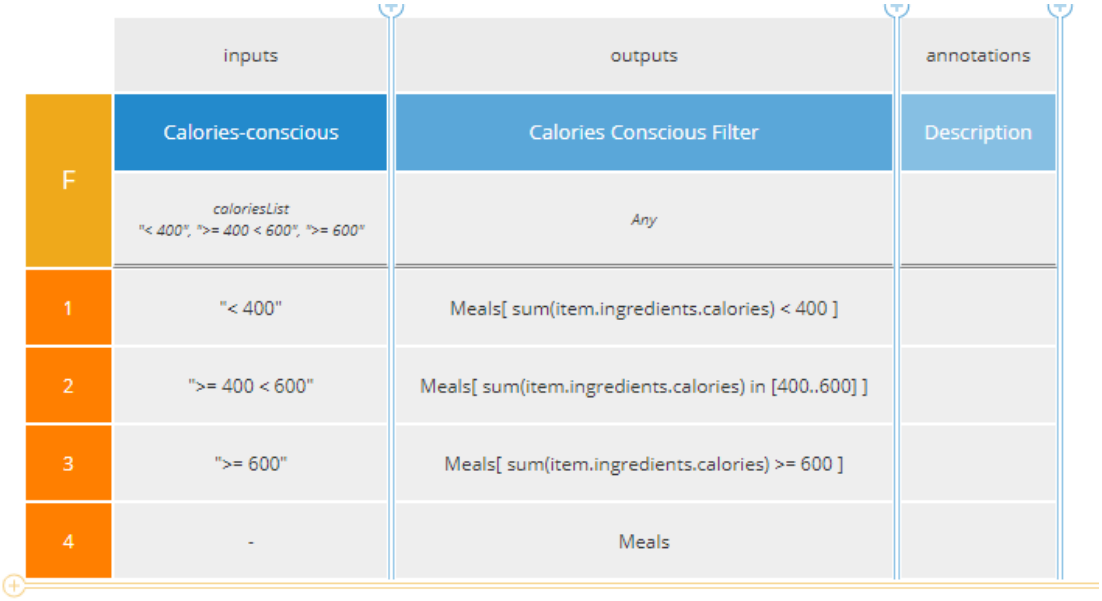
Meals[sum(item.ingredients.calories) in [400..600]]

- Row 3 (>= 600):

Meals[sum(item.ingredients.calories) >= 600]

- Row 4 (- missing input): returns the entire Meals list, i.e., no calorie filter applied.

This table greatly enhances the user experience, making the system capable of meeting specific dietary needs. The use of the sum of calories per ingredient allows a more accurate assessment than using a preaggregated value. The "First" policy avoids ambiguity and ensures that each input falls into only one category.



F	inputs	outputs	annotations
	Calories-conscious	Calories Conscious Filter	Description
	caloriesList *"< 400", ">= 400 < 600", ">= 600"*	*Any*	
1	"< 400"	Meals[sum(item.ingredients.calories) < 400]	
2	">= 400 < 600"	Meals[sum(item.ingredients.calories) in [400..600]]	
3	">= 600"	Meals[sum(item.ingredients.calories) >= 600]	
4	-	Meals	

PersonalizedMenu

The PersonalizedMenu table represents the final level of decision making in our DMN model. Its task is to combine the results of the previous filters (Allergies, Diet Type, and Calories Conscious) to return a personalized menu to the user, along with a descriptive message. This decision-making layer aims to take as input the entire set of available dishes and filters them based on the intersection between the previous three criteria, returning a customized menu and a message that varies depending on the availability of results.

It consists of:

- A column with the name of the rule: PersonalizedMenu (with type Any, because it can return a list or a message).
- A main rule row (row 1), which contains:
  - Main filter named FilteredMenu
  - Final output named Result, containing a conditional block

FilteredBlock (FilteredMenu)

Allergies Filter[list contains(Diet Type Filter, item) and list contains(Calories Conscious Filter, item)]

This filter takes as its basis the list of dishes already filtered by allergies and selects only those items that are also contained in the results of the filters by diet type and by calories. In other words, it constructs an intersection between the three lists of filtered dishes.

Output (Result) block

This block uses an if-then-else construct to handle the presence or absence of dishes in the FilteredMenu list.

Case 1 - Available dishes:

if count(FilteredMenu) > 0 then FinalMenu: FilteredMenu, DishesRate: "Based on our guests' preferences and the filters you selected, we recommend the following dishes: " + string(Rate Dishes(FilteredMenu))

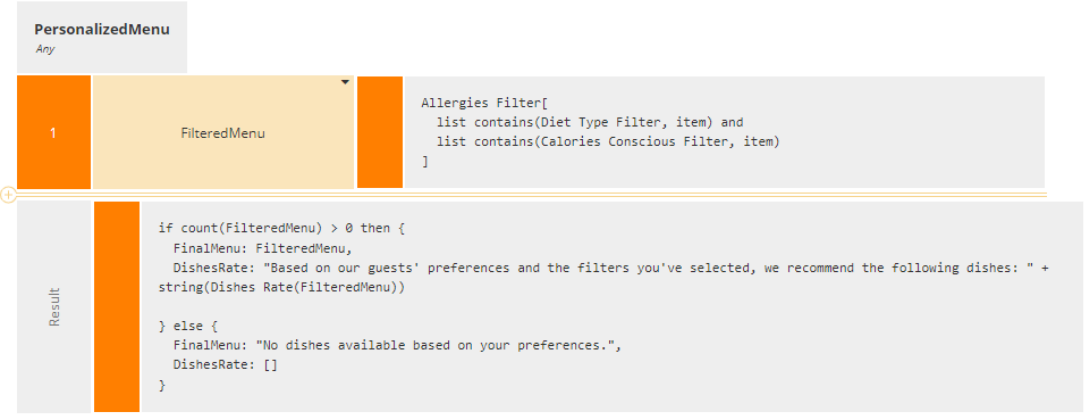
In this case, *FinalMenu* (the list of filtered dishes) and *DishesRate* (which we will see later) are returned

Case 2 – No dishes available:

else FinalMenu: "No dishes available based on your preferences.", DishesRate: []

In case of an empty result, an explicit message is shown that there are no dishes compatible with the preferences entered.

This table acts as a logical composer of the previous filters and is responsible for providing user-friendly and easily readable output. The use of the combined filter and conditional 'if' block allows the behavior of the system to be dynamically adapted, ensuring total customization of the gastronomic offer and communicative clarity towards the user.



Dishes Rate

The "Dishes Rate" table is responsible for generating a summary of the most highly rated dishes (by score) for each course (e.g., first course, second course, dessert, etc.), filtering the customized menu according to the user's preferences. Its main purpose is to identify and return the highest scoring dishes within each course category. It is a supporting logic that is used by the final PersonalizedMenu table to generate the final description of the recommended offering.

The expression is written in FEEL language and follows this structure:

for c in distinct values(MenuFilteredByUserPreferences.course) return (if count(MenuFilteredByUserPreferences[course = c]) > 0 then sort(MenuFilteredByUserPreferences[course = c], function(a, b) a.rate > b.rate)[1].name else [])

The first 'for' allows us to have a loop on flow rates:

for c in distinct values(MenuFilteredByUserPreferences.course)

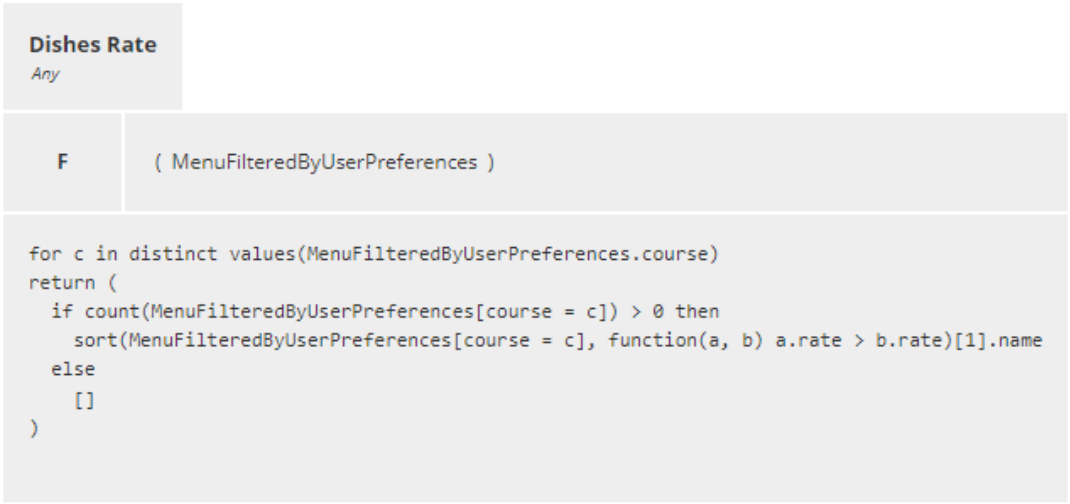
where we have an iteration on each course present in the user-filtered dishes.
Then a check is performed to see if any dishes are available:

if count(MenuFilteredByUserPreferences[course = c]) > 0 then

and then sorting by score takes place:

sort(MenuFilteredByUserPreferences[course = c], function(a, b) a.rate > b.rate)[1].name)

The Dishes Rate table provides a concise and useful output to describe the best of the personalized offering and ensures that at least one dish per course is present, if available. It also makes the user experience clearer by emphasizing the quality of the selected results.



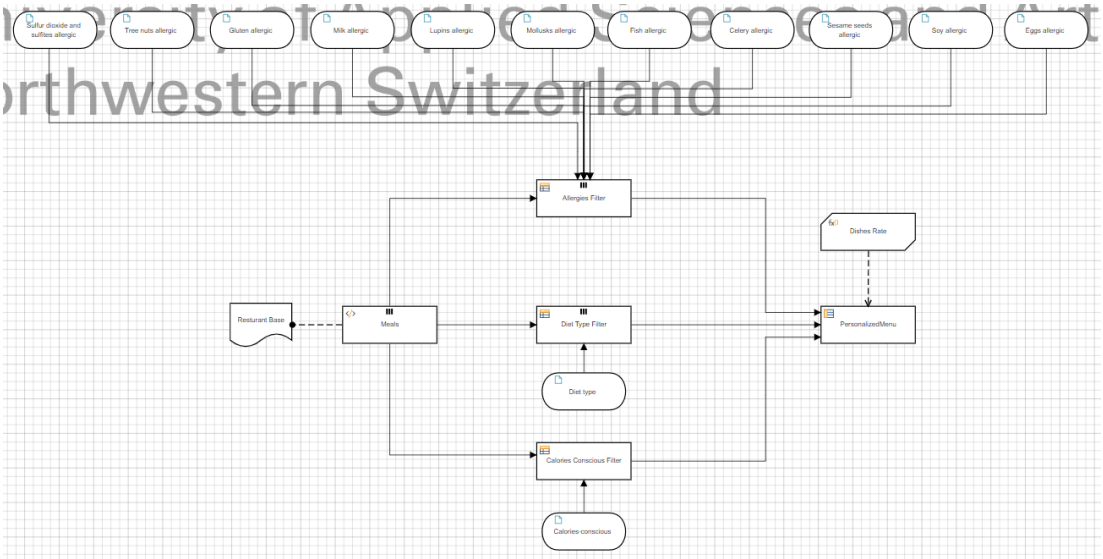
4.2.2 Second version

In the second version of the knowledge base, an attempt was made to make a DMN model as declarative as possible. However, this choice resulted in a limitation: it was

not possible to include all possible combinations of filters. For example, for the allergy filter, complete coverage of all combinations would have required the management of $2^n - 1$ (where n is the number of allergens), generating too large a number of rules for efficient modeling. For this reason, we chose to include only the most frequent and representative combinations in the model. Even in this version, the goal remains to allow restaurant customers to view a customized subset of the menu, filtered according to indicated preferences: diet type, allergies, and caloric threshold. The resulting DMN diagram consists of four main decision tables:

- "Allergies Filter";
- "Diet Type Filter";
- "Calories Conscious Filter";
- "PersonalizedMenu."

To these is added, intentionally, an additional filter designed to offer an intelligent suggestion to the user: for each course, the dish with the highest rating is selected, always respecting the specified preferences.



In this second version, the points that have already been covered and remained unchanged will not be taken up; the focus will be on the most relevant changes only in the points below.

Allergies Filter

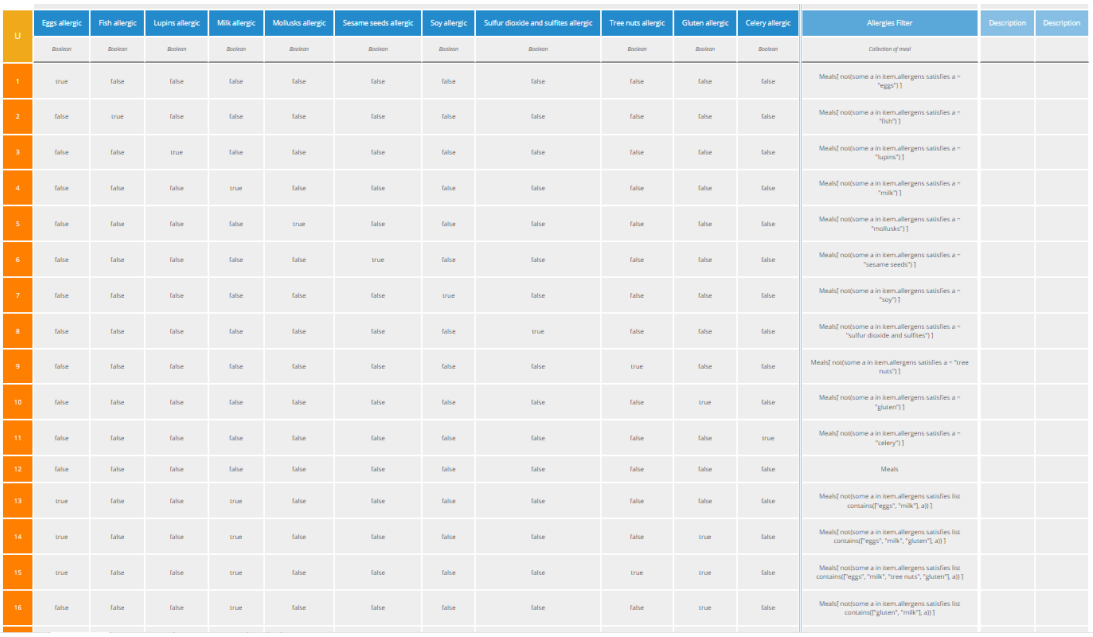
The "Allergies Filter" table is designed to filter the list of dishes based on the allergies declared by a user. The input data is represented by a set of Booleans, each of which corresponds to a specific food allergy (e.g., egg, fish, milk, gluten, etc.). If the user is allergic to a specific ingredient, the corresponding field will be set to true. The filter works as follows: for each allergy set to true, any dish that contains that substance among its allergens is excluded from the menu. This behavior is implemented via the ***Meals(not(some a in item.allergens satisfies a = "name_allergen"))*** function, or with contains() when multiple allergens need to be checked at the same time.

The table can receive two types of input:

- Single boolean value: represents a single allergen.
- Boolean list: represents a set of allergies. In this case, the filter condition is applied for each allergy with value true.

Each row of the table represents a different combination of allergies. The first eleven rows handle cases where only one allergy is active, while the other rows consider multiple combinations of allergies simultaneously, showing the use of the contains() function on a list of allergens.

We decide to adopt a "Unique" type of evaluation policy, i.e., it selects only one row from the possible rows (although, in this specific case, the row condition is always true, so there would be no ambiguity or need for conflict).



U	Eggs allergic	Fish allergic	Lupins allergic	Milk allergic	Mollusks allergic	Sesame seeds allergic	Soy allergic	Sulfur dioxide and sulfites allergic	Tree nuts allergic	Gluten allergic	Celery allergic	Allergies Filter	Description	Description
	Boolean	Boolean	Boolean	Boolean	Boolean	Boolean	Boolean	Boolean	Boolean	Boolean	Boolean	Collection of meal		
1	true	false	false	false	false	false	false	false	false	false	false	Meals[not(some a in item.allergens satisfies a = "eggs")]		
2	false	true	false	false	false	false	false	false	false	false	false	Meals[not(some a in item.allergens satisfies a = "fish")]		
3	false	false	true	false	false	false	false	false	false	false	false	Meals[not(some a in item.allergens satisfies a = "lupins")]		
4	false	false	false	true	false	false	false	false	false	false	false	Meals[not(some a in item.allergens satisfies a = "milk")]		
5	false	false	false	false	true	false	false	false	false	false	false	Meals[not(some a in item.allergens satisfies a = "molluscs")]		
6	false	false	false	false	false	true	false	false	false	false	false	Meals[not(some a in item.allergens satisfies a = "sesame seeds")]		
7	false	false	false	false	false	false	true	false	false	false	false	Meals[not(some a in item.allergens satisfies a = "soy")]		
8	false	false	false	false	false	false	false	true	false	false	false	Meals[not(some a in item.allergens satisfies a = "sulfur dioxide and sulfites")]		
9	false	false	false	false	false	false	false	false	true	false	false	Meals[not(some a in item.allergens satisfies a = "tree nuts")]		
10	false	false	false	false	false	false	false	false	false	true	false	Meals[not(some a in item.allergens satisfies a = "gluten")]		
11	false	false	false	false	false	false	false	false	false	false	true	Meals[not(some a in item.allergens satisfies a = "celery")]		
12	false	false	false	false	false	false	false	false	false	false	false	Meals		
13	true	false	false	true	false	false	false	false	false	false	false	Meals[not(some a in item.allergens satisfies list contains(["eggs", "milk"], a))]		
14	true	false	false	true	false	false	false	false	false	true	false	Meals[not(some a in item.allergens satisfies list contains(["eggs", "milk", "gluten"], a))]		
15	true	false	false	true	false	false	false	false	true	true	false	Meals[not(some a in item.allergens satisfies list contains(["eggs", "milk", "tree nuts", "gluten"], a))]		
16	false	false	false	true	false	false	false	false	false	true	false	Meals[not(some a in item.allergens satisfies list contains(["gluten", "milk"], a))]		

4.3 DMN-Simulation

For this project, the Trisotech DMN Simulator was utilized to verify the behavior of the decision model and the logical correctness of the decision tables implemented.. The simulation interface offers a means for the user to manually enter the input values of the different criteria used in the model: i.e., dietary preferences, allergy constraints, caloric objectives, or any other relevant attributes defined within the decision logic.

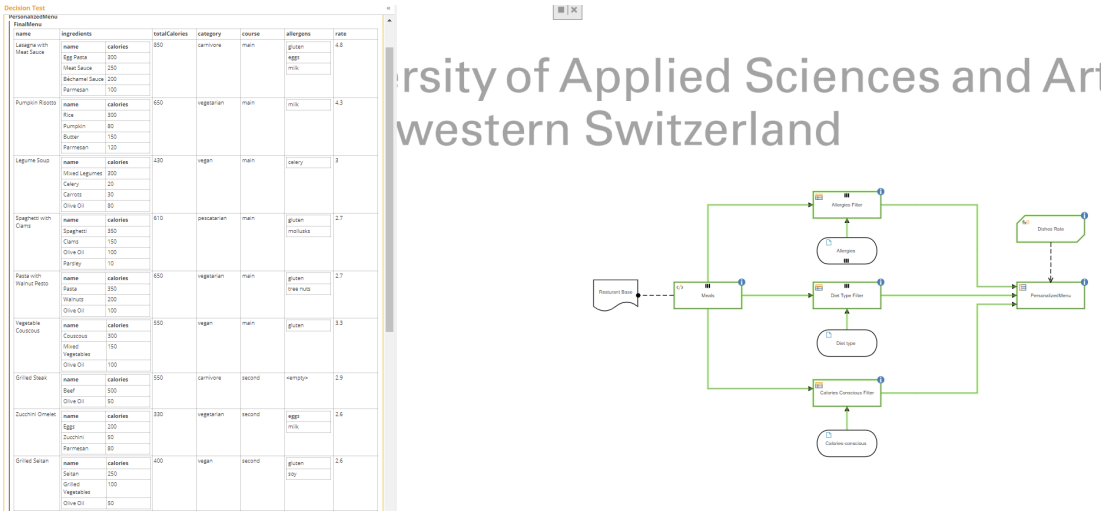
In our sample, the simulation fields represent the structured input data that's used across the various decision tables. Inputs can be directly entered into the form fields or selected from dropdowns where there are specific predefined values. Notably, Trisotech also provides support for sophisticated input structures such as lists, nested objects, and boolean flags, making it well-suited to testing decision logic in the real world.

Once the inputs are defined, the user can click on the "Submit" button to execute the simulation. Trisotech will then compute and display the result of every decision table

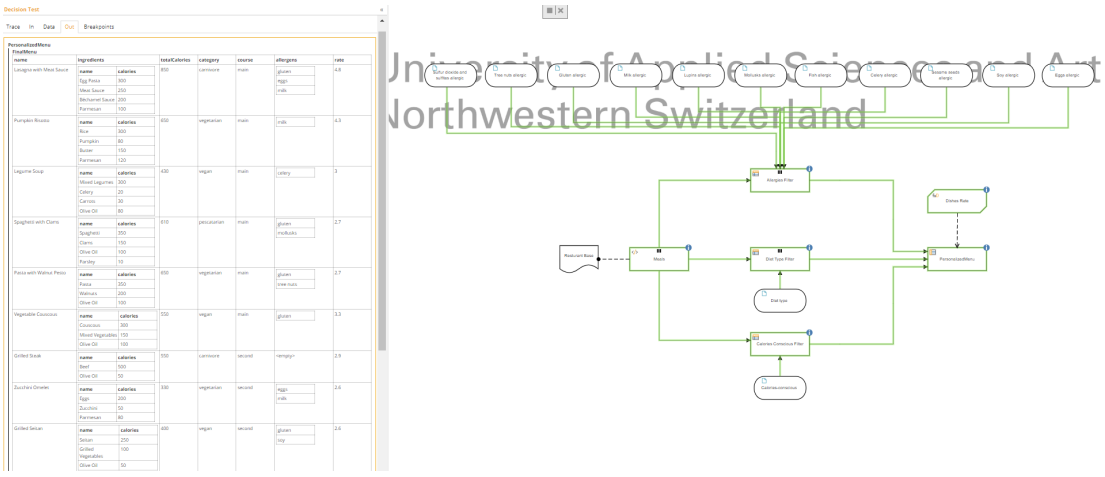
separately. Each output is shown along with its corresponding decision node, making it easy to trace the output back to the logic that generated it.

In our simulation, the system delivers intermediate outcomes—i.e., filtered lists by allergy, diet type, and calorie level—along with the final personalized menu derived from the intersection of all the filters imposed. This last outcome integrates the logic of all the previous decisions and offers a global and consistent recommendation based on the entire input profile.

Simulation V1:



Simulation V2:



5. Prolog

Prolog (short for **PROgramming in LOGic**) is a declarative programming language commonly used in artificial intelligence, logic programming, and symbolic reasoning.

Unlike imperative programming languages (such as C or Java), where developers explicitly define *how* tasks should be performed, Prolog focuses on defining *what* is true through facts and rules. The system itself then logically infers the solutions.

A typical Prolog program consists of **facts** and **rules**. For example:

```
% Facts
parent(john, mary).
parent(mary, susan).

% Rule
grandparent(X, Y) :- parent(X, Z), parent(Z, Y).
```

In this example, Prolog logically infers relationships (e.g., grandparents) based on the provided facts and rules.

Main Characteristics of Prolog

- **Declarative**: Defines relationships and conditions rather than procedural steps.
- **Automatic inference**: The system automatically deduces solutions by logically connecting rules and facts.
- **Natural recursion**: Facilitates intuitive handling of recursive problems.
- **Backtracking**: Automatically explores different potential solutions to reach an answer or multiple answers.

Typical Applications of Prolog

- Expert systems and automated reasoning
- Natural language processing
- Logic puzzles and games
- Symbolic AI and knowledge representation
- Education, particularly for teaching logic and AI principles

5.1 Code Structure

The file is divided into several sections:

- Defining facts about dishes, ingredients, allergens, diet type, course type, calories, and ratings.
- Predicate for filtering dishes.
- Predicates for classifying dishes by course and rating.
- Main predicate `personalized_menu/4`.
- Predicates for profile.

5.2 Definition of Facts

5.2.1 Dishes

Available dishes are defined with the predicate `meal/1`.

```
meal(lasagna_with_meat_sauce).
meal(pumpkin_risotto).
meal(legume_soup).
meal(spaghetti_with_clams).
meal(pasta_with_walnut_pesto).
meal(vegetable_couscous).
meal(grilled_steak).
meal(zucchini_omelet).
meal(grilled_seitan).
meal(salmon_fillet).
meal(stir_fried_tofu_with_vegetables).
meal(lupin_burger).
meal(hazelnut_cake).
meal(fruit_salad).
meal(lemon_sorbet).
meal(chocolate_soy_pudding).
meal(cream_ice_cream).
meal(sesame_cake).
```

Listing 5.1: Prolog meals

5.2.2 Ingredients

Each dish has its ingredients associated with it via the predicate `recipe_ingredient/2`. It is a many-to-many relationship: the same dish appears in multiple facts, and the same ingredient can refer to multiple dishes (not present here but possible).

```
recipe_ingredient(lasagna_with_meat_sauce, egg_pasta).
recipe_ingredient(lasagna_with_meat_sauce, meat_sauce).
recipe_ingredient(lasagna_with_meat_sauce, bechamel_sauce).
recipe_ingredient(lasagna_with_meat_sauce, parmesan).

recipe_ingredient(pumpkin_risotto, rice).
recipe_ingredient(pumpkin_risotto, pumpkin).
recipe_ingredient(pumpkin_risotto, butter).
recipe_ingredient(pumpkin_risotto, parmesan).
```

```
10
11 recipe_ingredient(legume_soup, mixed_legumes).
12 recipe_ingredient(legume_soup, celery).
13 recipe_ingredient(legume_soup, carrots).
14 recipe_ingredient(legume_soup, olive_oil).
15
16 recipe_ingredient(spaghetti_with_clams, spaghetti).
17 recipe_ingredient(spaghetti_with_clams, clams).
18 recipe_ingredient(spaghetti_with_clams, olive_oil).
19 recipe_ingredient(spaghetti_with_clams, parsley).
20
21 recipe_ingredient(pasta_with_walnut_pesto, pasta).
22 recipe_ingredient(pasta_with_walnut_pesto, walnuts).
23 recipe_ingredient(pasta_with_walnut_pesto, olive_oil).
24
25 recipe_ingredient(vegetable_couscous, couscous).
26 recipe_ingredient(vegetable_couscous, mixed_vegetables).
27 recipe_ingredient(vegetable_couscous, olive_oil).
28
29 recipe_ingredient(grilled_steak, beef).
30 recipe_ingredient(grilled_steak, olive_oil).
31
32 % For "grilled steak" we do not specify allergens because there are none.
33
34 recipe_ingredient(zucchini_omelet, eggs).
35 recipe_ingredient(zucchini_omelet, zucchini).
36 recipe_ingredient(zucchini_omelet, parmesan_cheese).
37
38 recipe_ingredient(grilled_seitan, seitan).
39 recipe_ingredient(grilled_seitan, grilled_vegetables).
40 recipe_ingredient(grilled_seitan, olive_oil).
41
42 recipe_ingredient(salmon_fillet, salmon).
43 recipe_ingredient(salmon_fillet, lemon).
44 recipe_ingredient(salmon_fillet, olive_oil).
45
46 recipe_ingredient(stir_fried_tofu_with_vegetables, tofu).
47 recipe_ingredient(stir_fried_tofu_with_vegetables, mixed_vegetables).
48 recipe_ingredient(stir_fried_tofu_with_vegetables, soy_sauce).
49
50 recipe_ingredient(lupin_burger, lupins).
51 recipe_ingredient(lupin_burger, bread).
52 recipe_ingredient(lupin_burger, fresh_vegetables).
53
54 recipe_ingredient(hazelnut_cake, flour).
55 recipe_ingredient(hazelnut_cake, hazelnuts).
56 recipe_ingredient(hazelnut_cake, butter).
57 recipe_ingredient(hazelnut_cake, eggs).
58
59 recipe_ingredient(fruit_salad, fresh_fruit).
60
61 recipe_ingredient(lemon_sorbet, lemon).
62 recipe_ingredient(lemon_sorbet, sugar).
63 recipe_ingredient(lemon_sorbet, sulfur_dioxide).
64
65 recipe_ingredient(chocolate_soy_pudding, soy_milk).
66 recipe_ingredient(chocolate_soy_pudding, cocoa).
67 recipe_ingredient(chocolate_soy_pudding, sugar).
68
69 recipe_ingredient(cream_ice_cream, milk).
70 recipe_ingredient(cream_ice_cream, eggs).
```

```
recipe_ingredient(cream_ice_cream, sugar).

recipe_ingredient(sesame_cake, flour).
recipe_ingredient(sesame_cake, sesame_seeds).
recipe_ingredient(sesame_cake, sugar).
```

Listing 5.2: Prolog ingredients

5.2.3 Allergens

Dish allergens are declared using `allergen/2`.

```
allergen(lasagna_with_meat_sauce, gluten).
allergen(lasagna_with_meat_sauce, eggs).
allergen(lasagna_with_meat_sauce, milk).

allergen(pumpkin_risotto, milk).

allergen(legume_soup, celery).

allergen(spaghetti_with_clams, gluten).
allergen(spaghetti_with_clams, mollusks).

allergen(pasta_with_walnut_pesto, gluten).
allergen(pasta_with_walnut_pesto, tree_nuts).

allergen(vegetable_couscous, gluten).

allergen(zucchini_omelet, eggs).
allergen(zucchini_omelet, milk).

allergen(grilled_seitan, gluten).
allergen(grilled_seitan, soy).

allergen(salmon_fillet, fish).

allergen(stir_fried_tofu_with_vegetables, soy).

allergen(lupin_burger, gluten).
allergen(lupin_burger, lupin).

allergen(hazelnut_cake, gluten).
allergen(hazelnut_cake, eggs).
allergen(hazelnut_cake, milk).
allergen(hazelnut_cake, tree_nuts).

allergen(lemon_sorbet, sulphur_dioxide_and_sulphites).

allergen(chocolate_soy_pudding, soy).

allergen(cream_ice_cream, eggs).
allergen(cream_ice_cream, milk).

allergen(sesame_cake, gluten).
allergen(sesame_cake, sesame_seeds).
```

Listing 5.3: Prolog allergens

5.2.4 Type of Diet

Each dish is classified by diet type with `diet/2`.

```
diet(lasagna_with_meat_sauce, carnivore).
diet(pumpkin_risotto, vegetarian).
diet(legume_soup, vegan).
diet(spaghetti_with_clams, pescatarian).
diet(pasta_with_walnut_pesto, vegetarian).
diet(vegetable_couscous, vegan).
diet(grilled_steak, carnivore).
diet(zucchini_omelet, vegetarian).
diet(grilled_seitan, vegan).
diet(salmon_fillet, pescatarian).
diet(stir_fried_tofu_with_vegetables, vegan).
diet(lupin_burger, vegan).
diet(hazelnut_cake, vegetarian).
diet(fruit_salad, vegan).
diet(lemon_sorbet, vegan).
diet(chocolate_soy_pudding, vegan).
diet(cream_ice_cream, vegetarian).
diet(sesame_cake, vegan).
```

Listing 5.4: Prolog diet

5.2.5 Type of Course

The course (first, second, dessert) is indicated by the predicate `course/2`.

```
course(lasagna_with_meat_sauce, main).
course(pumpkin_risotto, main).
course(legume_soup, main).
course(spaghetti_with_clams, main).
course(pasta_with_walnut_pesto, main).
course(vegetable_couscous, main).
course(grilled_steak, second).
course(zucchini_omelet, second).
course(grilled_seitan, second).
course(salmon_fillet, second).
course(stir_fried_tofu_with_vegetables, second).
course(lupin_burger, second).
course(hazelnut_cake, dessert).
course(fruit_salad, dessert).
course(lemon_sorbet, dessert).
course(chocolate_soy_pudding, dessert).
course(cream_ice_cream, dessert).
course(sesame_cake, dessert).
```

Listing 5.5: Prolog courses

5.2.6 Calories

The calories of each dish are recorded via `calories/2`.

```
calories(lasagna_with_meat_sauce, 850).
calories(pumpkin_risotto, 650).
calories(legume_soup, 430).
calories(spaghetti_with_clams, 610).
calories(pasta_with_walnut_pesto, 650).
calories(vegetable_couscous, 550).
```

```
calories(grilled_steak, 550).
calories(zucchini_omelet, 330).
calories(grilled_seitan, 400).
calories(salmon_fillet, 460).
calories(stir_fried_tofu_with_vegetables, 370).
calories(lupin_burger, 500).
calories(hazelnut_cake, 750).
calories(fruit_salad, 150).
calories(lemon_sorbet, 200).
calories(chocolate_soy_pudding, 300).
calories(cream_ice_cream, 450).
calories(sesame_cake, 550).
```

Listing 5.6: Prolog calories

5.2.7 Rating

The liking of the dishes is represented through the predicate `rating/2`.

```
rating(lasagna_with_meat_sauce, 4.8).
rating(pumpkin_risotto, 4.3).
rating(legume_soup, 3.0).
rating(spaghetti_with_clams, 2.7).
rating(pasta_with_walnut_pesto, 2.7).
rating(vegetable_couscous, 3.3).
rating(grilled_steak, 2.9).
rating(zucchini_omelet, 2.6).
rating(grilled_seitan, 2.6).
rating(salmon_fillet, 4.7).
rating(stir_fried_tofu_with_vegetables, 4.4).
rating(lupin_burger, 4.4).
rating(hazelnut_cake, 3.7).
rating(fruit_salad, 3.0).
rating(lemon_sorbet, 3.7).
rating(chocolate_soy_pudding, 4.5).
rating(cream_ice_cream, 2.7).
rating(sesame_cake, 4.5).
```

Listing 5.7: Prolog ratings

5.2.8 Filter Predicates

5.2.9 Allergen Exclusion

`allergens_excluded(Meal, Allergies)`: checks that a dish does not contain allergens in the list provided.

```
allergens_excluded(Meal, Allergens) :-
    \+ ( allergen(Meal, Ingredient), member(Ingredient, Allergens) ).
```

Listing 5.8: allergens_excluded

- If there is no such Ingredient that is an allergen of the dish and belongs to the Allergens list, the goal is successful.
- The sub-query in round brackets after `\+` is a conjunction (allergen/2, member/2). If at least one solution is found, `\+` fails → the dish is discarded.

5.2.10 Dietary Compatibility

`type_diet_compatible(Meal, Dieta)`: check that the dish meets the required diet.

```
type_diet_compatible(Meal, Dieta) :-
    diet(Meal, Dieta).
```

Listing 5.9: Prolog diet

Useful for keeping data logic and application logic separate, in case I want to handle "more complex" compatibility in the future (e.g., a vegetarian who accepts "lacto-ovo" but not "pescatarian").

5.2.11 Caloric Compatibility

`calories_compatible/2` distinguishes between low, medium and high-calorie dishes.

```
calories_compatible(Meal, low) :-
    calories(Meal, Cal),
    Cal < 400.

calories_compatible(Meal, medium) :-
    calories(Meal, Cal),
    Cal >= 400,
    Cal < 600.

calories_compatible(Meal, high) :-
    calories(Meal, Cal),
    Cal >= 600.
```

Listing 5.10: Prolog calories

5.3 Classification predicates

This mini-pipeline creates, for each course present after the filters, the best dish according to guest evaluation.

5.3.1 Drawing of Distinguished Dishes

`distinct_courses/2`: obtains the list of scope types present.

```
distinct_courses(FilteredDishes, DistinctCourses) :-
    findall(Course,
        ( member(Dish, FilteredDishes),
          course(Dish, Course)
        ),
        Courses),
    sort(Courses, DistinctCourses).
```

Listing 5.11: Prolog distinct courses

The purpose of the distinct_courses/2 predicate is to identify all the course types (e.g., first course, second course, dessert) in a filtered list of dishes. This is useful for further classifying dishes according to their course and for subsequent operations such as selecting the best dish for each course.

- findall/3 collects all Course of the filtered dishes.

- sort/2 removes duplicates because it sorts and deduplicates.

5.3.2 Filter by Course

`filter_by_course/3`: filters dishes belonging to a specific course.

```
%%% If the dish list is empty, then the filtered list will also be empty
filter_by_course([], _, []).

%%% Takes the first dish (Dish) and the rest of the list (Rest)
%%% Checks: course(Dish, Course) -> if this dish belongs to the desired
    course
%%% If true: - includes it in the filtered list ([Dish|Filtered])
%%%          - continues filtering the rest (Rest) with the same rule
filter_by_course([Dish|Rest], Course, [Dish|Filtered]) :-
    course(Dish, Course),
    filter_by_course(Rest, Course, Filtered).

%%% Takes the first dish (Dish) and the rest of the list (Rest)
%%% Checks: course(Dish, Course) -> if this dish belongs to the desired
    course
%%% Verifies: that Dish has a course different from the requested one (Other
     \= Course)
%%% If true: - does not include it in the filtered list
%%%          - continues processing Rest normally
filter_by_course([Dish|Rest], Course, Filtered) :-
    course(Dish, Other),
    Other \= Course, % deve essere diverso
    filter_by_course(Rest, Course, Filtered).
```

Listing 5.12: Prolog filter by course

- List recursion with two mutually exclusive clauses.
- The first clause includes the dish if `course(Dish, Course)` is true; the second clause excludes it.

5.3.3 Sorting by Rating

`sort_by_rating_desc/2`: sorts dishes by descending rating.

```
%%% map_list_to_pairs: Creates pairs (dish, rating) by applying the
    dish_rating/2 predicate to each element of the Dishes list.
%%% keysort: Sorts the pairs in ascending order based on rating.
%%% reverse and pairs_values: Reverses the order to obtain a list of dishes
    sorted by descending rating, extracting only the dishes from the sorted
    pairs.
sort_by_rating_desc(Dishes, Sorted) :-
    map_list_to_pairs(dish_rating, Dishes, Pairs),
    keysort(Pairs, SortedAsc),
    reverse(SortedAsc, SortedDesc),
    pairs_values(SortedDesc, Sorted).

dish_rating(Dish, Rating) :-
    rating(Dish, Rating).
```

Listing 5.13: Prolog rate

- `map_list_to_pairs/3` → creates pairs (Rating-Key, Dish-Value).

- `keysort/2` sorts in ascending order.
- `reverse/2` reverses the list.
- `pairs_values/2` extracts only the dishes.

5.3.4 Best Dish by Course

`best_dish_by_course/3`: selects the best dish for each course type.

```
best_dish_by_course(FilteredDishes, Course, BestDish) :-
    filter_by_course(FilteredDishes, Course, DishesByCourse),
    (   DishesByCourse = []
    ->  BestDish = null
    ;   sort_by_rating_desc(DishesByCourse, [BestDish|_]) % we only need the
     top-rated dish, so we take the head of the sorted list
    ).
```

Listing 5.14: Prolog Best dish by course

- Uses the if–then–else construct: *(Condition -¿ Then ; Else)*

Note: the anonymous variable _ discards the rest of the already sorted list.

5.3.5 Course-Dish Association

`dishes_rate_by_course/2`: for each course, provides the best dish.

```
dishes_rate_by_course(FilteredDishes, CourseDishPairs) :-
    distinct_courses(FilteredDishes, DistinctCourses),
    findall([Course, Dish],
        (
            member(Course, DistinctCourses),
            best_dish_by_course(FilteredDishes, Course, Dish)
        ),
        CourseDishPairs).
```

Listing 5.15: Prolog Course-Dish Association

5.4 Main Predicate

5.4.1 `personalized_menu/4`

```

personalized_menu(Allergens, Diet, CaloriesRange, Result) :-
    findall(Meal,
        (
            meal(Meal),
            allergens_excluded(Meal, Allergens),
            type_diet_compatible(Meal, Diet),
            calories_compatible(Meal, CaloriesRange)
        ),
        FilteredMeals),
    generate_answer(FilteredMeals, Result).

generate_answer([], "No dishes available based on your preferences.").
generate_answer(Meals, Message) :-
```

```
    findall(Rate, (member(Meal, Meals), rating(Meal, Rate)), RateList),
    dishes_rate_by_course(Meals, CourseDishPairs),
    format(string(Message),
        "Based on our guests' preferences and the filters you selected, we
    recommend: ~w with ratings: ~w. We also suggest: ~w",
        [Meals, RateList, CourseDishPairs]).

```

Listing 5.16: Prolog personalized menu

- `findall/3` constructs `FilteredMeals` by applying all filters in conjunction.
- The list is passed to `generate_answer/2`, which produces a formatted string containing:
  - the list of dishes (w),
  - the ratings (`RateList`) calculated using a second `findall/3`,
  - course-to-best-dish suggestions (`CourseDishPairs`).

5.4.2 `generate_answer/2`

Filters dishes based on:

- absence of allergens,
- dietary type,
- requested calorie range.

After filtering:

- if no dish is available, it generates a warning message;
- otherwise, it builds a message suggesting dishes and showing ratings and the best dish per course.

5.5 Profiles

5.5.1 Predefined Standard Profiles

```
/* Profile: Vegetarian host without allergies, average calories  */
profile_vegetarian_medium(Result) :-
    personalized_menu([], vegetarian, medium, Result).

/* Profile: Vegan guest with no allergies, low calories */
profile_vegan_low(Result) :-
    personalized_menu([], vegan, low, Result).

/* Profile: Gluten-allergic pescetarian host */
profile_pescatarian_no_gluten(Result) :-
    personalized_menu([gluten], pescatarian, medium, Result).

/* Profile: Carnivore host without allergies, high calories */
profile_carnivore_high(Result) :-
    personalized_menu([], carnivore, high, Result).
```

```
16
17 /* Profile: Vegetarian host allergic to milk and eggs */
18 profile_vegetarian_no_milk_eggs(Result) :-
19     personalized_menu([milk, eggs], vegetarian, medium, Result).
20
21 /* Profile: Guest vegan allergic to soy, average calories */
22 profile_vegan_no_soy_medium(Result) :-
23     personalized_menu([soy], vegan, medium, Result).
```

Listing 5.17: Prolog Predefined Standard Profiles

This Prolog code defines a series of **standardized dietary profiles**, each of which invokes the main predicate `personalized_menu/4`, which has the following structure:

`personalized_menu(Allergens, DietType, CalorieLevel, Result)`

- **Allergens**: list of allergens to avoid
- **DietType**: type of diet followed (e.g., `vegetarian`, `vegan`, `carnivore`, `pescetarian`)
- **CalorieLevel**: desired calorie level (`low`, `medium`, `high`)
- **Result**: output returned by the system (list of suitable dishes)

These predefined profiles allow the system to quickly model common use cases (e.g., a vegan guest with no allergies, a vegetarian with milk and egg intolerance, etc.), making it easily testable and reusable.

5.5.2 `Profile`

```
1
2
3 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
4 %% Standard Profiles Per Person                                        %%
5 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
6 profile(Person, Result) :-
7     person_diet(Person, Diet),
8     findall(Meal,
9         ( meal(Meal),
10          diet(Meal, Diet),
11          \+ ( allergen(Meal, Allergen), is_allergic(Person, Allergen) )
12        ),
13    MealsRes),
14    generate_answer(MealsRes, Result).
```

Listing 5.18: Prolog Profiles

Details:

- `person_diet/2`: retrieves the diet followed by the person.
- `findall/3`: builds the list `MealsRes` containing all `Meal` entries that are compatible with the diet (`diet/2`) and do not contain allergens to which the person is allergic (a combination of `allergen/2` and `is_allergic/2`).
- The filter `\+ (...)` excludes dishes that may be dangerous for the user.

- `generate_answer/2`: processes and formats the result (`Result`), for example by selecting a representative subset of suitable dishes.

This predicate allows the system to handle **personalized profiles for each individual**, based on knowledge stored in the system regarding their diet and allergies.

5.6 Prolog Operators Used

Symbol	Name and arity	Practical meaning	Code example
`\+`	*negation as failure* (prefix, arity 1)	"Not provable that..."	`\+ (allergen(Meal, Ing), member(Ing, Allergens))`
`,`	*and* conjunction (infix, arity 2)	All goals must succeed	`course(Dish, Course), rating(Dish, R)`
`;`	*or* disjunction (infix, arity 2)	At least one goal must succeed	Implicit in *if–then–else*
`->`	*if–then* (infix, arity 2)	Part of If -> Then `;` Else	`DishesByCourse = [] -> BestDish = null ; ...`
`=`	Unification	Structural match, not assignment	`Cal =< 400`
`\=`	Structural inequality	"Cannot be unified with"	`Other \= Course`

Table 5.1: Main Prolog operators used in the Personalized Menu project

5.6.1 Prolog Query Example

```
personalized_menu([gluten], vegetarian, _, Result).
```

Result = "Based on our guests' preferences and the filters you selected, we recommend: [pumpkin_risotto,zucchini_omelet,cream_ice_cream] with ratings: [4.3,2.6,2.7]. We also suggest: [[dessert,cream_ice_cream],[main,pumpkin_risotto],[second,zucchini_omelet]]"

```
profile(lucrezia, Result).
```

Result = "Based on our guests' preferences and the filters you selected, we recommend: [grilled_steak] with ratings: [2.9]. We also suggest: [[second,grilled_steak]]"

```
personalized_menu([gluten, eggs], vegan, _, Result).
```

Result = "Based on our guests' preferences and the filters you selected, we recommend: [legume_soup,stir_fried_tofu_with_vegetables,fruit_salad,lemon_sorbet,chocolate_soy_pudding] with ratings: [3.0,4.4,3.0,3.7,4.5]. We also suggest: [[dessert,chocolate_soy_pudding],[main,legume_soup],[second,stir_fried_tofu_with_vegetables]]"

```
profile_pescatarian_no_gluten(Result).
```

Result = "Based on our guests' preferences and the filters you selected, we recommend: [salmon_fillet] with ratings: [4.7]. We also suggest: [[second,salmon_fillet]]"

6. Knowledge Graph

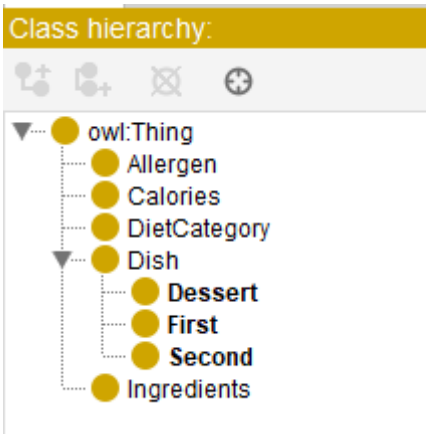
Into this chapter will explain the Ontology Structure using Protégé software editor and the creation of a Knowledge Graph. In Protégé, the hierarchy of classes is a tree structure that organizes the concepts of an ontology by inheritance relationships. Being a structure, it is a taxonomy and is useful in classifying the concepts of a specific domain in a systematic way.

In the class hierarchy, classes are arranged according to parent-child relationship: a subclass (child) inherits properties and characteristics from its superclass (parent). Through this mechanism, you can model generalization and specialization relationships between ideas to specify more narrow categories from broader ones.

The use of hierarchy in Protégé is critical in structuring domain knowledge in a purposeful and reusable manner, and also in enabling queries and logical deductions to be performed on the basis of the indicated relationships. Through this, not only can concepts and relations be expressed but reasoned upon as well, in a formal and efficient way.

6.1 Class Hierarchy

The class hierarchy of this project shows that "Thing" is the top-level supercalss, and it has direct subclasses "Allergen", "Calories", "DietCategory", "Ingredients" and "Dish". "Dish" has direct subclasses "Dessert", "First", "Second". All of them contain a set of individuals created to represent the real scenario cases. For example the class "Allergen" contains all the allergens of an international resturant have. In fact, Individuals in Protégé are typed as members of one or more classes organized and displayed as hierarchy.



6.2 Object Property Hierarchy

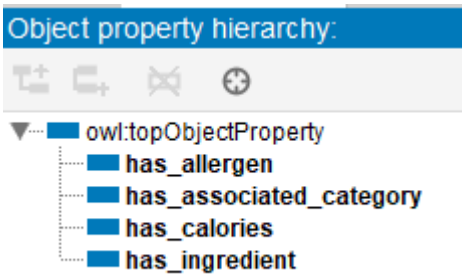
The object property hierarchy defines how object properties are organized in a structured, hierarchical manner. Object properties describe the relationships between individuals within an ontology, specifying how instances of one class relate to instances of another.

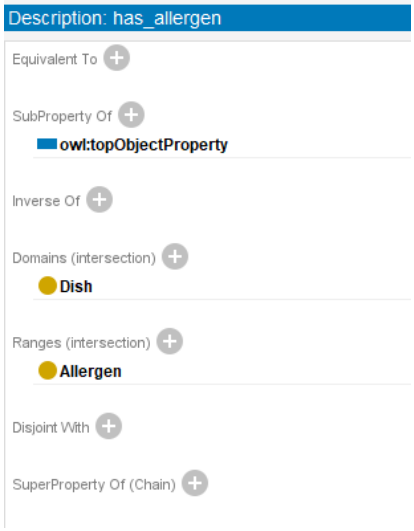
In Protégé, this hierarchy is built through the use of subproperties and superproperties. A subproperty represents a more specific relationship and inherits the characteristics of its corresponding superproperty, while adding further detail or constraint.

Organizing object properties hierarchically allows for a clearer and more systematic modeling of relationships. This structure also supports reasoning processes, enabling inference engines to derive new knowledge and make more precise deductions based on the defined connections.

In particular our case we have the Object Properties:

- has_allergen
- has_associated_category
- has_calories
- has_ingredients





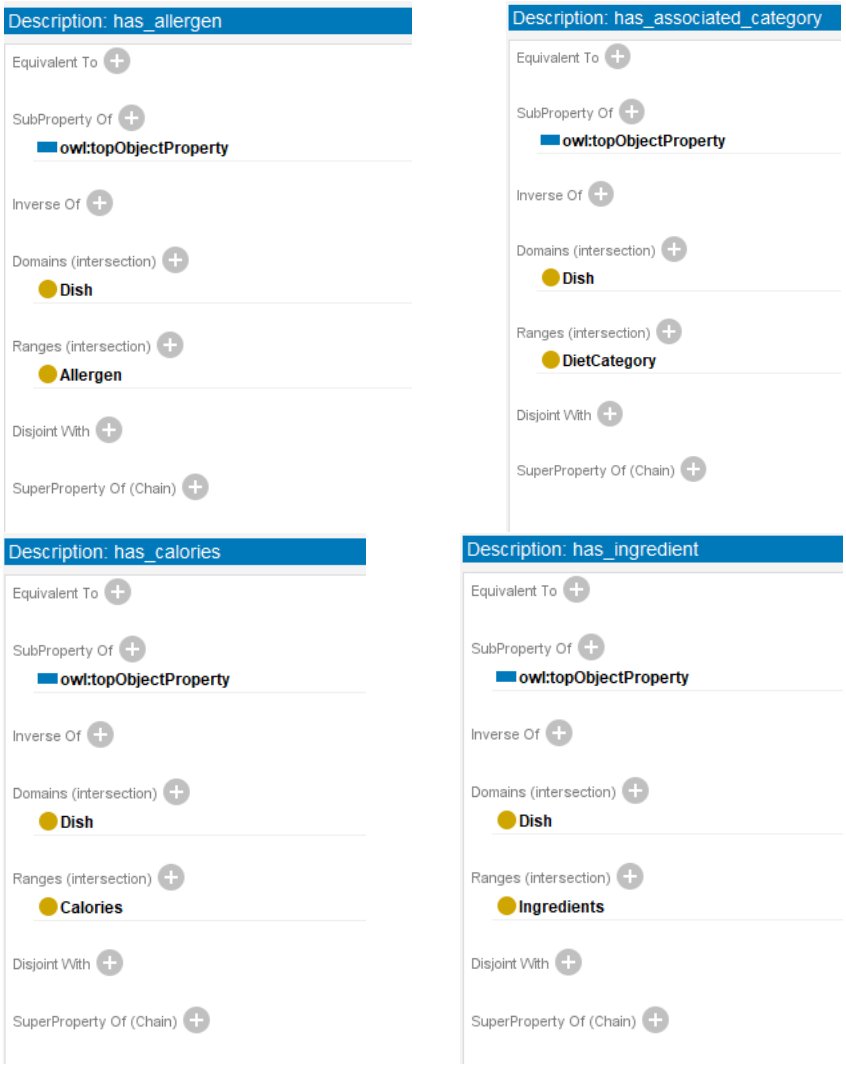
The first property, has_associated_category, links a dish to its corresponding diet category (e.g., vegetarian, vegan, pescetarian). The Domain of this property is Dish, while the Range is DietCategory. This relation is useful for classifying dishes based on dietary needs.

The second property, has_calories, relates each dish to its associated caloric value, represented by an individual of class Calories. Again, the Domain is Dish and the Range is Calories. This enables reasoning over the energy content of the menu.

The third property, has_ingredient, expresses the composition of a dish by linking it to the ingredients it contains. The Domain is Dish and the Range is Ingredients. This relation supports ingredient-based filtering or recommendation.

Finally, the has_allergen property connects each dish to one or more allergens it may contain. This is fundamental for users with dietary restrictions or allergies. Its Domain is Dish and the Range is Allergen.

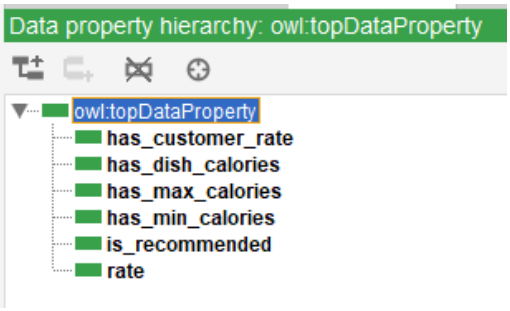
Together, these properties form the core of the semantic structure of our ontology, enabling meaningful reasoning and flexible querying across the dataset.



6.3 Data Property Hierarchy

The data property hierarchy refers to the hierarchical relationships established between data properties. Data properties represent attributes or characteristics of individuals in an ontology. They describe the data values associated with instances of a class. Also here, there are subproperties. In the data properties the predicate connects a single subject with some form of attribute data. Data properties have defined datatypes including string, integer, date, datetime, or boolean.

In our case, we defined a set of Data Properties under the hierarchy of owl:topDataProperty, each of which contributes to enriching the semantics of the Dish and related individuals with literal values such as numbers or booleans.



The property has_customer_rate links each dish to a customer-assigned rating. It has Dish as its domain and xsd:float as range, enabling numerical evaluations.

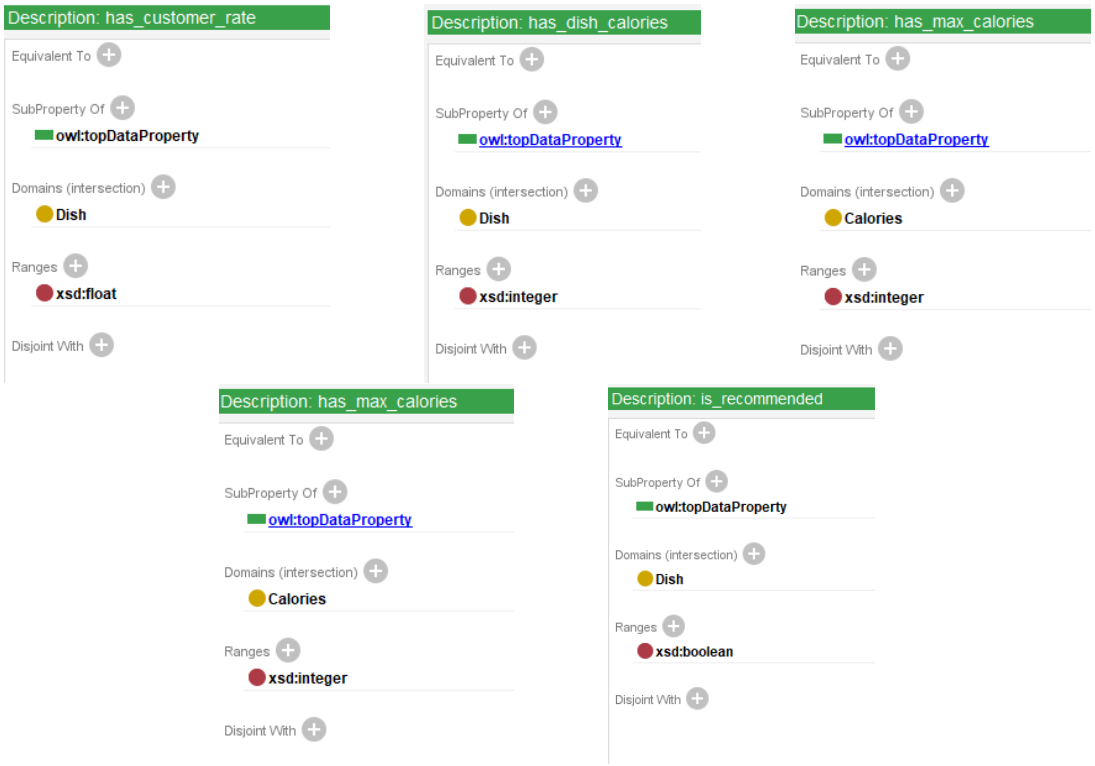
The has_dish_calories property assigns a specific calorie value to a dish. The domain is again Dish, while the range is xsd:integer, allowing precise control over dish energy content.

has_max_calories and has_min_calories are associated with the Calories concept and define thresholds (upper and lower bounds) for allowable caloric values. Both properties use xsd:integer as range. These properties are especially useful for reasoning or filtering operations in diet recommendation systems.

The is_recommended property is a boolean flag (xsd:boolean) that indicates whether a particular dish is recommended. The domain is Dish.

Lastly, the rate property is a generic numeric evaluation property, though in this case, no domain or range is explicitly defined. It may serve as an abstract base for more specific properties like has_customer_rate.

These data properties, together, support fine-grained, literal-based querying and filtering within the ontology, improving its utility in personalized recommendation scenarios and semantic reasoning engines.



6.4 Knowledge Graph

A knowledge graph is a structured representation of knowledge in the form of a graph, where information is organized through interconnected nodes and edges. In Protégé, this graph is built by leveraging ontology modeling features.

Within a knowledge graph, nodes typically correspond to concepts or individuals defined in the ontology, while edges represent the relationships between them, such as object properties or data properties. This structure visually maps out how different elements relate to each other.

Using Protégé, knowledge graphs offer a clear and intuitive way to visualize complex interconnections and dependencies within a domain. They support users in exploring associations between entities, making it easier to analyze, reason about, and gain insights from the modeled knowledge.

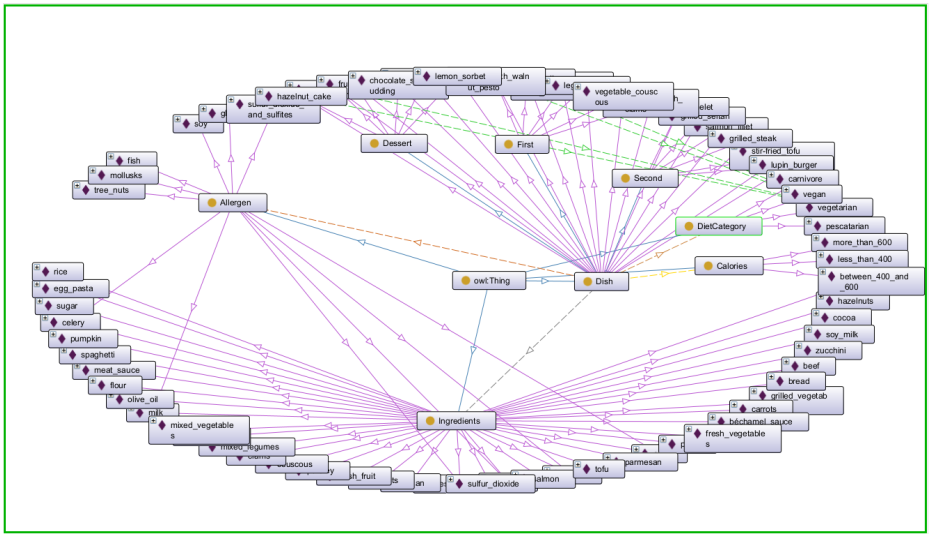
The resulting Knowledge Graph of this project's ontology is centered around the class Dish, which acts as the core concept within the domain. As shown in the figure, Dish is connected to several other classes through a network of Object Properties, providing a structured and semantic representation of meal-related knowledge.

Each dish is linked to one or more Ingredients via the has_ingredient property, allowing a breakdown of its composition. Similarly, the has_allergen property connects a dish to potential allergens (e.g., fish, soy, tree_nuts), which is essential for allergy-aware recommendations.

The has_associated_category property relates dishes to a specific DietCategory, such as vegan, vegetarian, or pescatarian, supporting dietary filtering and classification. Moreover, the has_calories property connects each dish to an individual of the class Calories, which is further detailed by data properties like has_max_calories and has_min_calories.

Dishes are also semantically grouped by type (First, Second, Dessert) through subclassing mechanisms, providing an intuitive categorization of meals. Finally, additional data properties like has_dish_calories, has_customer_rate, and is_recommended enrich the description of each dish with numerical and boolean values, supporting user feedback and recommendation logic.

This ontology-driven graph offers a comprehensive and interconnected model for reasoning about meals, enabling enhanced data querying, personalized filtering, and intelligent menu generation.



6.5 SPARQL

The SPARQL Plugin for Protégé is an extension that enables the execution of SPARQL queries directly within the Protégé environment. SPARQL (SPARQL Protocol and RDF Query Language) is the W3C-recommended language for querying and manipulating RDF-based data.

Thanks to this plugin, users can write and run queries on ontologies or knowledge graphs modeled in RDF, without needing to export the data to external tools. The plugin offers a dedicated interface inside Protégé where SPARQL queries can be composed and executed, allowing for the extraction of specific information, the exploration of relationships between entities, and the testing of reasoning rules over the ontology. In our case, we defined a set of SPARQL queries to retrieve filtered views of the available dishes in the menu.

6.5.1 Queries

Note that each query requires some prefixes, the following are necessary to run our queries.

```
PREFIX rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX owl:  <http://www.w3.org/2002/07/owl#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX xsd:  <http://www.w3.org/2001/XMLSchema#>
PREFIX :     <http://www.semanticweb.org/kebi/PapaAndCaldarelli/personalizedmenu#>
```

1. Subclass Hierarchy Retrieval

This query retrieves all subclass relationships defined in the ontology. It helps to explore the taxonomic structure of concepts.

```
SELECT ?subject ?object
WHERE {
  ?subject rdfs:subClassOf ?object
}
```

2. Dishes with Less Than 400 Calories

Selects all dishes classified as having fewer than 400 calories according to the ontology's calorie categorization.

```
SELECT ?dish
WHERE {
  ?dish a :Dish ;
        :has_calories :less_than_400 .
}
```

3. Vegan Dishes

Retrieves all dishes categorized under the vegan diet.

```
SELECT ?dish
WHERE {
  ?dish a :Dish ;
```

```
        :has_associated_category :vegan .
}
```

4. Dishes Containing Gluten

Returns all dishes that list gluten among their allergens.

```
SELECT ?dish
WHERE {
  ?dish a :Dish ;
        :has_allergen :gluten .
}
```

5. High-Rated Desserts

Retrieves desserts with a customer rating greater than 4.0.

```
SELECT ?dish ?rating
WHERE {
  ?dish a :Dessert ;
        :has_customer_rate ?rating .
  FILTER(?rating > 4.0)
}
```

6. Dishes with 400–600 Calories

Selects all dishes that belong to the "between 400 and 600 calories" category.

```
SELECT ?dish
WHERE {
  ?dish a :Dish ;
        :has_calories :between_400_and_600 .
}
```

7. Vegetarian Dishes Containing Eggs

Finds dishes classified as vegetarian that also list eggs as an ingredient.

```
SELECT ?dish
WHERE {
  ?dish a :Dish ;
        :has_associated_category :vegetarian ;
        :has_ingredient :eggs .
}
```

8. Dishes with More Than 600 Calories

Retrieves all dishes classified in the "more than 600 calories" category.

```
SELECT ?dish
WHERE {
  ?dish a :Dish ;
        :has_calories :more_than_600 .
}
```

9. Ingredients in Vegan Dishes

Returns all distinct ingredients used in dishes categorized as vegan.

```
SELECT DISTINCT ?ingredient
WHERE {
  ?dish a :Dish ;
        :has_associated_category :vegan ;
        :has_ingredient ?ingredient .
}
```

10. First-Course Dishes Under 600 Calories

Selects dishes of type "First" with fewer than 600 calories.

```
SELECT ?dish ?calories
WHERE {
  ?dish a :First ;
        :has_dish_calories ?calories .
  FILTER(?calories < 600)
}
```

11. Dishes Without Allergens

Finds all dishes that are not associated with any allergens.

```
SELECT ?dish
WHERE {
  ?dish a :Dish .
  FILTER NOT EXISTS {
    ?dish :has_allergen ?allergen .
  }
}
```

12. View All Dishes with Their Characteristics

Retrieves all dishes along with their calories, rating, category, ingredients, and allergens, using optional properties and grouping.

```
SELECT
  ?dish
  ?calories
  ?rating
  ?category
  (GROUP_CONCAT(DISTINCT ?ingredient; separator=", ") AS ?ingredients)
  (GROUP_CONCAT(DISTINCT ?allergen; separator=", ") AS ?allergens)
WHERE {
  ?dish rdf:type :Dish .

  OPTIONAL { ?dish :has_dish_calories ?cal . BIND(STR(?cal) AS ?calories) }
  OPTIONAL { ?dish :has_customer_rate ?rate . BIND(STR(?rate) AS ?rating) }
  OPTIONAL { ?dish :has_associated_category ?cat . BIND(STR(?cat) AS ?
    category) }
  OPTIONAL { ?dish :has_ingredient ?ing . BIND(STR(?ing) AS ?ingredient) }
  OPTIONAL { ?dish :has_allergen ?all . BIND(STR(?all) AS ?allergen) }
}
GROUP BY ?dish ?calories ?rating ?category
ORDER BY ?dish
```

13. Filtered Dishes by Category, Calories and Allergen Exclusion

Returns the dishes that match a specific calorie range and diet category, excluding dishes that contain selected allergens (e.g., milk).

```
SELECT
  ?dish
  (GROUP_CONCAT(DISTINCT ?ingredient; separator=", ") AS ?ingredients)
  (GROUP_CONCAT(DISTINCT ?allergen; separator=", ") AS ?allergens)
  ?category
  ?calories
  ?rating
WHERE {
  ?dish a :Dish ;
        :has_calories :between_400_and_600 ;
        :has_associated_category :vegan .

  OPTIONAL { ?dish :has_ingredient ?ingredient . }
  OPTIONAL { ?dish :has_allergen ?allergen . }
  OPTIONAL { ?dish :has_customer_rate ?rating . }
  OPTIONAL { ?dish :has_dish_calories ?calories . }
  OPTIONAL { ?dish :has_associated_category ?category . }

  FILTER NOT EXISTS {
    ?dish :has_allergen ?excludedAllergen .
    FILTER(?excludedAllergen IN (:milk))
  }
}
GROUP BY ?dish ?category ?calories ?rating
ORDER BY ?dish
```

6.6 SWRL

SWRL (Semantic Web Rule Language) is a rule-based language used to express logical rules on top of OWL ontologies. It enables the combination of OWL axioms with Horn-like rules of the form: ***antecedent*** $\rightarrow$ ***consequent*** where both the antecedent (body) and the consequent (head) consist of a conjunction of atoms (such as class membership, property relations, or built-in operations).

In the context of this project, SWRL rules have been employed to enrich the reasoning capabilities of the ontology, allowing for the classification of dishes based on their nutritional values and customer preferences. By using SWRL, it is possible to infer new knowledge from existing data, thus supporting semantic queries and decision-making.

Implemented Rules

The following rules have been implemented using the SWRLTab in Protégé:

1. Caloric Classification: Between 400 and 600

```
Dish(?d) ^ has_dish_calories(?d, ?cal) ^ swrlb:greaterThanOrEqual(?cal, 400)
     ^ swrlb:lessThanOrEqual(?cal, 600) -> has_calories(?d,
    between_400_and_600)
```

This rule classifies a dish into the custom class between_400_and_600 if its caloric value is within the inclusive range of 400 to 600. This supports caloric filtering through categorical tags rather than numeric filters in SPARQL.

2. Caloric Classification: Less than 400

```
Dish(?d) ^ has_dish_calories(?d, ?cal) ^ swrlb:lessThan(?cal, 400) ->
    has_calories(?d, less_than_400)
```

This rule assigns the class less_than_400 to all dishes whose calorie count is strictly below 400, enabling queries focused on low-calorie options.

3. Caloric Classification: Over 600

```
Dish(?d) ^ has_dish_calories(?d, ?cal) ^ swrlb:greaterThan(?cal, 600) ->
    has_calories(?d, more_than_600)
```

Dishes with a calorie value exceeding 600 are categorized as more_than_600. This rule allows the ontology to group high-calorie meals explicitly.

4. Recommended Dishes Based on Rating

```
Dish(?d) ^ has_customer_rate(?d, ?r) ^ swrlb:greaterThanOrEqual(?r, 4.2) ->
    is_recommended(?d, true)
```

This rule marks a dish as recommended if its average customer rating is greater than or equal to 4.2. The inferred boolean property is_recommended can then be used for suggestions or quality-based filtering in queries.

Purpose and Benefits

The integration of these SWRL rules provides multiple advantages:

- Semantic categorization: Numeric values (e.g., calories) are translated into semantic categories that are easier to use in SPARQL queries and reasoning.
- Recommendation logic: The system can automatically determine which dishes are likely to be preferred, based on aggregated user ratings.
- Interoperability: These rules can be processed by reasoners such as Pellet, allowing for dynamic inference without manual updates to the ontology.

6.7 SHACL

SHACL (Shapes Constraint Language) is a W3C standard used to describe and validate RDF data. It allows the specification of constraints, or "shapes", that RDF resources must conform to. These shapes ensure that instances in an ontology respect a predefined structure or set of rules—improving consistency, data quality, and interoperability.

In this project, SHACL was used to validate instances of the Dish class according to specific structural and semantic conditions. This helps verify whether the ontology complies with expected formats before deploying it in reasoning or querying tasks.

Defined Shapes and Constraints

Several validation rules were created and applied using SHACL. Each one enforces constraints on data associated with Dish instances:

1. Dish Must Have a Caloric Value

This shape ensures that every Dish instance must include the has_dish_calories property with a valid xsd:integer value.

2. Dish Must Have a Category

Each dish must be associated with at least one dietary category (:has_associated_category) to support classification and filtering.

3. Dish Must Have at Least One Ingredient

This constraint guarantees that a dish is not left without its basic compositional elements.

4. Rating Must Be a Float Between 0 and 5

For customer ratings, this shape ensures that the value of has_customer_rate is a float between 0 and 5, aligning with standard rating systems.

5. Label Validation

This SHACL shape is used to enforce rules on the rdfs:label property of individuals. It ensures that every label is a valid string, with a specific length and formatting constraint.

```
:DishShape
  a sh:NodeShape ;
  sh:targetClass :Dish ;

  # At least 1 ingredient
  sh:property [
    sh:path :has_ingredient ;
    sh:minCount 1 ;
  ] ;

  # Exactly 1 integer value for calories
  sh:property [
    sh:path :has_dish_calories ;
    sh:datatype xsd:integer ;
    sh:minCount 1 ; sh:maxCount 1 ;
  ] ;

  # Rating of customer (0  5  )
  sh:property [
    sh:path :has_customer_rate ;
    sh:datatype xsd:float ;
    sh:minInclusive 0 ; sh:maxInclusive 5 ;
    sh:maxCount 1 ;
  ] ;
```

```
25
26   sh:property [
27     sh:path :has_associated_category ;
28     sh:in ( :vegetarian :carnivore :pescatarian :vegan ) ;
29     sh:minCount 1 ;  # obbligatoria
30   ] .
31
32 :LabelValidationShape
33   a sh:NodeShape ;
34   sh:targetSubjectsOf rdfs:label ;  # Applica la validazione a chi ha rdfs:
       label
35   sh:property [
36     sh:path rdfs:label ;
37     sh:datatype xsd:string ;
38     sh:minLength 3 ;
39     sh:maxLength 100 ;
40     sh:pattern "^[a-z].*" ;  # Deve iniziare con lettera maiuscola
41     sh:message "L'etichetta deve iniziare con una lettera maiuscola e avere
       una lunghezza tra 3 e 100 caratteri." ;
42   ] .
```

Purpose and Benefits

The SHACL validation rules serve as a quality assurance mechanism in the ontology lifecycle. Specifically, they:

- Detect structural inconsistencies (e.g., missing data or wrong types).
- Enforce domain logic (e.g., mandatory associations between a dish and its components).
- Prevent errors during reasoning or SPARQL querying due to incomplete individuals.
- Facilitate integration with external systems that expect well-structured data.

By integrating SHACL into the development process, the ontology becomes robust, validated, and ready for reasoning, querying, or deployment in real-world applications.

7. Agile and Ontology-based Meta-modelling

This chapter explores an innovative approach to metamodeling that combines the principles of agile development with the use of ontologies. This methodology aims to make modeling languages more adaptable and relevant in different application contexts. Integrating the flexibility of agile with the conceptual framework offered by ontologies provides a dynamic and effective solution for the design and evolution of complex models.

7.1 AOAME

To address the second phase of the project, focused on ontology-based agile metamodeling, we used the AOAME tool. This tool is designed to support the creation and management of Knowledge Graph schemas, enabling the structuring of domain-specific knowledge to facilitate its analysis, reasoning, and integration across heterogeneous data sources.

AOAME enables dynamic modification and management of modeling constructs through metamodeling operators, which generate SPARQL queries and update the triplestore, ensuring alignment between model and data.

Built using Jena Fuseki's Java libraries, AOAME proves particularly well suited to address real-world needs, offering flexible support for metamodeling. In our case, it played a key role in the customization of the BPMN 2.0 language, allowing us to extend its existing classes and properties according to specific project requirements.

7.2 BPMN 2.0

BPMN (Business Process Model and Notation) 2.0 is a widely used standard for modeling business processes through graphical representation. It offers a rich yet intuitive notation designed to be easily understood and implemented, thus facilitating communication both within and across different organizations. This approach helps improve performance management, collaborations, and business transactions.

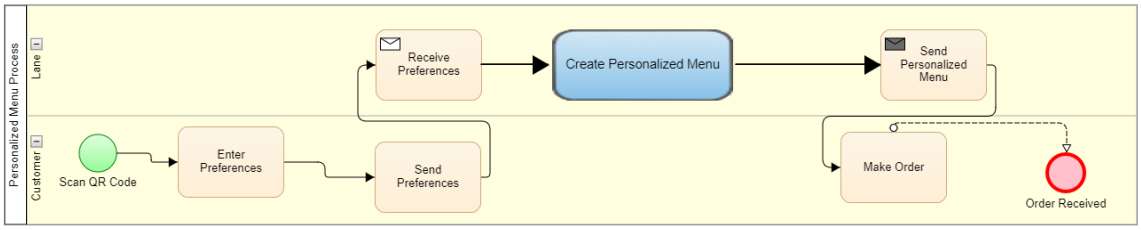
The language includes elements such as activities, events, gateways, and flows, which enable the representation of operational flow, decision points, and interactions between process components. The main goal of BPMN is to bridge the gap between the conceptual design of processes and their operational execution by providing a standardized method for documenting and analyzing business processes.

7.3 Our BPMN model

The Figure below shows the BPMN 2.0 model created using AOAME, which represents the process of generating a customized menu for a specific client. The model is divided into two separate pools: one dedicated to the Customer and the other to the Management System, clearly highlighting the interactions between the two parties.

The flow begins when the customer scans a QR code to start the process. He then enters his dietary preferences, such as dietary restrictions and allergies ecc., and sends them to the system. Once the information is received, the system forwards it to an extended class, ***Suggest Task***, specifically designed to select the dishes that best suit the customer's needs.

After the personalized menu is generated, the system returns it to the customer, thus completing the process with the delivery of the tailored menu.



7.4 Jena Fuseki

Jena Fuseki is a SPARQL server and is one of the main components of the Apache Jena framework. It is designed to host RDF (Resource Description Framework) data and provides a platform for querying and managing such data through the SPARQL language.

Fuseki supports both SPARQL query operations (for reading data) and SPARQL update operations (for modifying data). It can be run as a stand-alone server, offering HTTP endpoints for accessing RDF data, or integrated directly within applications, making it a flexible and powerful tool for developing semantic Web-based applications and knowledge graph systems.

7.4.1 Query in Jena Fuseki

A query was defined on the Jena Fuseki server (see Figure below) to retrieve the custom menu from the ontology developed in Chapter 6, in accordance with the model defined through AOAME. The query takes advantage of the extended class and its properties, specifically designed to support the extraction of dishes in line with customer preferences.

The query logic is similar to that described in Section 6.5, but differs in that additional prefixes are used and preferences are not defined directly in the query: instead, they are retrieved automatically from the extended class.

```
1
2  PREFIX mod:  <http://fhnw.ch/modelingEnvironment/ModelOntology#>
3  PREFIX lo:   <http://fhnw.ch/modelingEnvironment/LanguageOntology#>
4  PREFIX kebi: <http://example.org/restaurant#>
5  PREFIX rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
6
7  PREFIX :    <http://www.semanticweb.org/kebi/PapaAndCaldarelli/
       personalizedmenu#>
8  PREFIX rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
9
10 SELECT DISTINCT
11   ?dishName
12   ?category
13   (GROUP_CONCAT(DISTINCT ?ing;   SEPARATOR=", ") AS ?ingredients)
14   (GROUP_CONCAT(DISTINCT ?all;   SEPARATOR=", ") AS ?allergens)
15   ?caloriesValue
16   ?rating
17 WHERE {
18   ######################################################################
19   # 0)  user parameters
20   ######################################################################
21   mod:SuggestTask_18b92b15-80b1-4c00-8307-57eb15a61823
22         lo:CaloriesConscious ?kcalLit ;
23         lo:DietType          ?catLit ;
24         lo:Allergies         ?allLit .
25
26   ######################################################################
27   # 1) base dish
28   ######################################################################
29   ?dish  rdf:type           :Dish ;
30          :has_dish_calories ?caloriesValue ;
31          :has_customer_rate ?rating .
32
33   ######################################################################
34   # 2) create the IRIs
35   ######################################################################
36   BIND( IRI(CONCAT(STR(:), ?kcalLit)) AS ?kcalIRI )
37   BIND( IRI(CONCAT(STR(:), ?catLit )) AS ?catIRI  )
38   BIND( IRI(CONCAT(STR(:), ?allLit )) AS ?allIRI  )
39
40   ######################################################################
41   # 3) has_calories constrait
42   ######################################################################
43   FILTER (
44          ?kcalLit = "" || ?kcalLit = "none"
45       || EXISTS { ?dish :has_calories ?kcalIRI }
46   )
47
48   ######################################################################
```

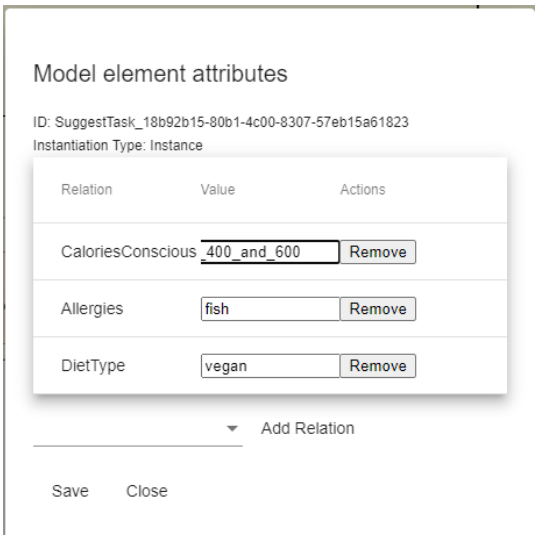
```
  # 4) has_associated_category constrait
  ######################################################################
  FILTER (
         ?catLit = "" || ?catLit = "none"
      || EXISTS { ?dish :has_associated_category ?catIRI }
  )

  ######################################################################
  # 5) has_allergen constrait
  ######################################################################
  FILTER (
         ?allLit = "" || ?allLit = "none"
      || NOT EXISTS { ?dish :has_allergen ?allIRI }
  )

  ######################################################################
  # 6) ingredients and allerges
  ######################################################################
  OPTIONAL {
    ?dish :has_ingredient ?ing .
    BIND( STRAFTER(STR(?ing),  "#") AS ?ingName )
  }
  OPTIONAL {
    ?dish :has_allergen   ?all .
    BIND( STRAFTER(STR(?all), "#") AS ?allName )
  }

  ######################################################################
  # 7) dish name (short)
  ######################################################################
  BIND( STRAFTER(STR(?dish), "#") AS ?dishName )
}
GROUP BY ?dishName ?category ?caloriesValue ?rating
ORDER BY ?dishName
```

Below we'll provide some example of result based on different properties selected by a customer.



Output of the query:

	dishName	category	ingredients	allergens	caloriesValue	rating
1	grilled_seitan		http://www.semanticweb.org/kebi/PapaAndCaldarelli/personalizedmenu#grilled_vegetables, http://www.semanticweb.org/kebi/...	http://www.semanticweb.org/kebi/PapaAndCaldarelli/personalized...	"400"^^<http://www.w3.org/2...	"2.6"^^<http://www.w...
2	legume_soup		http://www.semanticweb.org/kebi/PapaAndCaldarelli/personalizedmenu#carrots, http://www.semanticweb.org/kebi/PapaAndCa...	http://www.semanticweb.org/kebi/PapaAndCaldarelli/personalized...	"430"^^<http://www.w3.org/2...	"3.0"^^<http://www.w...
3	lupin_burger		http://www.semanticweb.org/kebi/PapaAndCaldarelli/personalizedmenu#bread, http://www.semanticweb.org/kebi/PapaAndCal...	http://www.semanticweb.org/kebi/PapaAndCaldarelli/personalized...	"500"^^<http://www.w3.org/2...	"4.4"^^<http://www.w...
4	sesame_cake		http://www.semanticweb.org/kebi/PapaAndCaldarelli/personalizedmenu#sugar, http://www.semanticweb.org/kebi/PapaAndCal...	http://www.semanticweb.org/kebi/PapaAndCaldarelli/personalized...	"550"^^<http://www.w3.org/2...	"4.5"^^<http://www.w...
5	vegetable_co...		http://www.semanticweb.org/kebi/PapaAndCaldarelli/personalizedmenu#couscous, http://www.semanticweb.org/kebi/PapaAnd...	http://www.semanticweb.org/kebi/PapaAndCaldarelli/personalized...	"550"^^<http://www.w3.org/2...	"3.3"^^<http://www.w...

8. Conclusions

8.1 Edoardo Papa

In this project, a customised menu system was designed and implemented, capable of proposing dishes based on the user's dietary preferences and restrictions. The system considers parameters such as lactose intolerance, gluten-free requirements, vegetarian options and different calorie levels, automatically generating customised menus.

DMN decision tables were used for process modelling, while logical reasoning was entrusted to Prolog. Ontology construction and management were carried out in Protégé, using SHACL Shapes and SWRL rules to impose constraints and ensure data consistency. Queries were optimised using SPARQL.

It was possible to use Jena Fuseki and AOAME for agile and flexible ontology metamodelling, which facilitates the creation of complex and adaptable knowledge models. Below, I share my perspective on these methods.

8.1.1 Decision Tables

Decision tables offer a structured method to map inputs and outputs according to a "Hit Policy". In our case, the First and Unique policy were chosen, which avoids ambiguity and ensures that each input falls into only one category. This approach guarantees clarity and simplicity. This approach was most useful in situations where conditions were mutually exclusive, like filtering dishes according to dietary requirements (vegetarian vs. pescetarian) or the ingredients' presence (gluten-free, lactose-free). The system is able to quickly pick the best rule using the First and Unique hit policies, not testing redundant or conflicting alternatives.

Besides, such an organization made it quite simple to follow the initial validation and debugging procedures. Each rule could be tested in isolation, removing the risk of duplicate conditions and ensuring the logic would be understandable regardless of the rules' quantity. The decision tables also eased the process for stakeholders who were non-technical to verify and comment on the rule-defining process because they could be presented in tabulated and readable form.

Briefly, the application of decision tables brought the following advantages:

- A graphical and declarative representation of rules.
- Improved maintainability, especially when it came to adding or changing conditions.
- Improved traceability since each decision path could be explained and documented.

This module eventually served as a good foundation for consistency across the recommendation engine and was an important contributor to consistency between the ontological reasoning and procedural rules used elsewhere in the system.

8.1.2 Prolog

Among the various techniques I experimented with during this project, Prolog quickly became the one I enjoyed—and leveraged—the most. Its declarative nature seemed tailor-made for expressing my personalized-menu system's sophisticated diet rules. Rather than juggling control flow with loops and counters, I could simply assert facts and rules and let the inference engine handle the rest. This shift in attitude was stimulating, but not without struggle. Prolog was a blessing. It permitted me to express complex constraints—such as mutually incompatible allergens, calorie restrictions, and diet types—in a way still accessible to domain experts. Even better, its logical transparency matched the Decision Tables and ontology validations in the rest of the workflow beautifully, giving the entire system one, cohesive reasoning paradigm.

8.1.3 Protégé

Perhaps the most intriguing aspect of the project was applying ontology-based modeling, which yielded a hybrid paradigm combining the firmness of DMN-style structuring and the versatility of object-oriented cognition. With tools like Protégé, I was able to set up entities (e.g., Dish, Allergen, DietType), associate them with attributes, and explain complex relationships with RDF triples.

One aspect that I particularly liked was the query interface, which—due to the SPARQL query language—was straightforward and comfortable, akin to directly playing with SQL. This lowered the learning curve substantially and was beneficial in being able to rapidly iterate over testing ontology-based logic on live data.

Equally useful was the SHACL validator, which allowed me to define shape constraints—such as required fields or allowable value ranges—for various entities. I often drew the parallel with Java assertion-style testing since it aided in picking up inconsistencies or omitted fields at the modeling stage.

Assembling reasoning rules using SWRL, however, did cause some problems. While excellent at making implicit knowledge (e.g., deducing that a dish is "safe" for a user based on the user's allergies and the characteristics of the dish), it required special attention at deployment time.

Although these were limitations, this ontology-based methodology provided a solid semantic basis for the system. It encouraged rules consistency, facilitated modular knowledge management, and supplemented both the Prolog engine and decision table modules.

8.1.4 AOAME

Working with AOAME provided another layer of elegance to the project by merging meta-modeling with usual BPMN flows. By providing tailored additions—like the Suggest Task class, which was a derived form of the native Task element—we could create domain-specific BPMN diagrams that used restaurant parlance. Every extended task had semantically endowed attributes—Allergies, CaloriesConscious, DietType—which

were tied to the underlying ontology effortlessly, allowing for more granular meal suggestions right from the process map.

Tight integration with Apache Jena Fuseki was a blessing. Once enriched and started with a BPMN instance, the engine injected context data into Fuseki, which then served up live menu suggestions through SPARQL endpoints. This pipe assured that all updates—a new allergy found by a user or calorie limit adjustment—would be directly sent to the generated menu without any form of human intervention.

8.1.5 Final Remarks

Overall, this project was a rich and varied tour of the terrain of knowledge-based systems, highlighting the advantages and compromises of each approach. Decision Tables were a simple and structured way of encoding rules, highly readable and useful for quick prototyping, but that soon showed their limitations when faced with intersecting or extremely contextual conditions. Prolog did turn out to be an incredibly effective tool of expression for reasoning, ideal for characterizing constraints and inference structure—but not without its limits in scalability and learning materials.

Employment of Protégé provided formal structuring of concepts and relationships and supported correct semantic modeling. Inference rules, however, needed to be regulated through SWRL and provided reasoning consistency between tools with non-trivial amounts of coordination. Employment of AOAME integrated process modeling within the semantic space, giving visual liveliness to ontology-driven workflows, albeit unstable in its online variant and able to benefit from further polish.

From a development standpoint, while the solutions I developed were functional and aligned with the project goals, there is certainly area for optimization—specifically performance and maintainability for enterprise-scale deployments. However, the experience has been very impactful.

In general, this project was a worth-while experience in experimenting with hands-on logic-based reasoning, semantic modeling, and decision automation, reaffirming my interest in the field of knowledge engineering. Of all the tools experimented with, Prolog is the one I became most impressed with, because of its readability, expressiveness, and intellectual pleasure that comes from crafting elegant rule-based solutions.

"Knowing what your guest cannot *eat is the first step towards offering exactly what they* will *love."*

8.2 Francesco Caldarelli

8.2.1 Project Summary

The *Personalized Menu* project was developed to meet a concrete and increasingly relevant challenge: creating digital restaurant menus that are not only interactive, but also context-aware and responsive to individual dietary needs. In an era where menus are often accessed via QR codes and screens, the ability to intelligently filter and personalize content is not just a technological enhancement—it is a necessity for inclusivity, safety, and user satisfaction.

The central goal was to design a system that, based on the specific profile of each guest, could recommend the most suitable dishes while excluding those that pose health

risks or conflict with personal preferences. The guest's profile may include allergies, dietary regimes such as vegan or pescatarian, or even preferences regarding calorie intake.

To explore this idea comprehensively, the project adopted a multi-paradigm approach to knowledge representation, integrating four distinct methodologies:

- **Decision Tables**, using the DMN standard to model and visualize rule-based logic;
- **Prolog**, for a declarative and logic-based representation of relationships between dishes and user needs;
- **Ontologies**, developed in Protégé with OWL, SWRL, SPARQL, and SHACL to enable semantic reasoning and structured querying;
- **AOAME**, an ontology-based extension of BPMN 2.0, enabling semantic knowledge to be embedded in business processes.

Each of these approaches was implemented and evaluated on its own terms, and then considered in light of its contribution to the broader design of an intelligent, modular, and human-centered recommendation system.

8.2.2 Decision Tables

Decision tables provided a clear and business-friendly structure for modeling rule-based logic. Implemented in Trisotech using DMN, the tables captured filters for allergens, diet types, and calorie thresholds, eventually composing a final decision table named `PersonalizedMenu` that consolidated the outputs.

This method stood out for its transparency and ease of simulation. Even without a technical background, stakeholders could follow the logic, input different scenarios, and observe how the recommendation changed accordingly. While certain complexity limits were encountered (particularly in the declarative modeling of numerous allergen combinations), decision tables ultimately proved to be a solid foundation for modeling structured, explainable logic in constrained domains.

8.2.3 Prolog

Prolog offered a radically different perspective, allowing the modeling of knowledge through facts and rules. The declarative nature of Prolog made it possible to define rich relationships and apply recursive reasoning across multiple criteria, from allergens to calories to user profiles.

The predicate `personalized_menu/4` served as the system's reasoning core, capable of filtering and selecting the most suitable dishes while generating natural-language output. Prolog also allowed for modular testing using predefined user profiles and enabled symbolic inference that would be difficult to express in other paradigms. While less immediately accessible for non-programmers, its power and clarity at the reasoning level made it a valuable component in the overall architecture.

8.2.4 Ontology-Based Modeling

The Protégé-based ontology brought semantic depth to the project. By modeling domain entities—such as `Dish`, `Ingredient`, and `Allergen`—and their relationships, we were able to structure knowledge in a way that supports formal reasoning and rich querying.

SWRL rules were used to infer classifications, such as labeling a dish as "high calorie" based on its nutritional values. SPARQL queries allowed for precise and contextual retrieval of data, while SHACL constraints validated the structural consistency of the model. This approach brought rigor and flexibility, but also required familiarity with semantic technologies and ontology design practices.

8.2.5 Ontology-Based Agile Meta-Modelling (AOAME)

AOAME represented the project's most process-oriented and integrative layer. By extending BPMN 2.0 with semantically aware tasks, we enabled business processes to make decisions based on ontological reasoning. A new task type—graphically adapted and semantically enriched—was created to execute SPARQL queries directly within a process model, powered by the Jena Fuseki triple store.

This allowed, for instance, a restaurant workflow to respond to guest input by querying the ontology and adapting the process in real time—showing dishes, escalating to staff, or offering fallback options. The process itself became "knowledge-aware," reacting not to static rules, but to structured, evolving knowledge.

While more complex to implement, this integration demonstrated how semantic reasoning can become a native part of process automation, enabling systems that are adaptive, explainable, and closely aligned with real-world workflows.

8.2.6 Final Remark

This project has shown that adopting a multi-paradigm approach to knowledge representation enables the design of robust, flexible, and explainable systems for real-world decision-making scenarios. Each method explored—decision tables, logic programming, ontologies, and process integration—has offered distinct advantages and revealed specific use cases in which it excels.

Decision tables have proven particularly suitable for transparent, business-oriented rules and decision logic. **Prolog** allowed for concise expression of complex logical relations and inference chains. **Ontologies** provided a semantic backbone, enabling interoperability, validation, and structured reasoning. Finally, the integration of ontological knowledge into **BPMN 2.0 through AOAME** demonstrated the feasibility of embedding semantic logic directly within business workflows.

This layered strategy not only made the system more modular and reusable, but also showed how different representations can complement one another when orchestrated toward a shared goal. The overall architecture enabled both high-level abstraction and fine-grained control, while ensuring that system behavior remains understandable and verifiable.

By applying and comparing these methodologies in a unified application scenario personalized food recommendation—the project offered a concrete example of how knowledge-based systems can support decision processes in dynamic and constraint-sensitive environments.