AN ONTOLOGY-BASED RESTAURANT SYSTEM

A REPORT SUBMITTED TO THE UNIVERSITY OF MANCHESTER
FOR THE DEGREE OF BACHELOR OF SCIENCE
IN THE FACULTY OF SCIENCE AND ENGINEERING

2023

Student id: 10452849

Department of Computer Science

Contents

Word Count: 1626

List of Tables

List of Figures

Abstract

AN ONTOLOGY-BASED RESTAURANT SYSTEM
Thomas Hayter
A report submitted to The University of Manchester
for the degree of Bachelor of Science, 2023

The aim of the thesis is to ...

Declaration

No portion of the work referred to in this report has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

Copyright

Acknowledgements

First of all, I would like to thank my supervisor who has guided me through the process of this third year project. They have constantly kept me on track, reassuring me when I needed it while also emphasizing the urgency of the project. I would also like to thank all of the participants of my user studies, who picked out so many problems that I missed and whose feedback drastically improved the system in many ways.

Chapter 1

Introduction

1.1 Overview

My project explores how ontologies can be used to store information about a restaurant's menu. I will also explore what extent it is possible to store information, for example whether you can store calorific information about ingredients, calculate dishes suitable for certain diets and calculate new information about a dish with certain ingredients removed or extra ingredients added.

The final product will consist of two user interfaces. The first will allow a customer to view and filter the menu. The second will allow the owner of a restaurant to create and edit the menu.

1.2 Project Plan

1.3 This Report

Chapter 2

Background

2.1 Technologies

Ontologies are stored in a *.owl* file, which is similar to an XML. I initially used Protege to get used to working with ontologies, as it is a good environment for working with them. The user interfaces are written in Java as it has the useful and comprehensive OWL API[**?**]. I useed Java Swing to create the user interfaces themselves. The project is built using Maven and produces a runnable .jar file that should be placed in the same folder as the .owl file.

2.2 Ontologies

An ontology is a knowledge-based way of storing information. It works in a object oriented kind-of way, where everything is a class, and is a subclass of *"Thing"*. For example, the class *"Ingredient"* is a subclass of *"Thing"*, and has *"CarbohydrateIngredient"*, *"MeatIngredient"* and *"VegetableIngredient"* are it's subclasses. Then, the specific *"LettuceIngredient"* is a subclass of *"VegetableIngredient"*. Everything that holds true for a superclass also holds true for all the subclasses. It is convention to name a subclass with a suffix to make it clear what it is a subclass of (other than when creating a subclass of *"Thing"*), for example all dishes would have the suffix *"Dish"* to differentiate them from an ingredient that could have the same name.

Ontologies contains facts, properties and rules, and uses those rules and properties to infer more facts. The reasoner is the tool which infers these facts. While many are available, I have used Hermit[**?**] as it is the most up to date.

There are two kinds of properties, object properties and data properties. An object

property relates two classes. For example, the object property *"containsIngredient"* would link the *"Dish"* class to the *"Ingredient"* class. A data property links a class to a literal value. For example, the data property *"hasCalories"* could be used to link an *"Ingredient"* class to an integer value. All the standard types are available, including booleans, integers, strings etc. These properties are also stored in an object oriented fashion, so if the *"containsTopping"* property was a subproperty of *"containsIngredient"*, and it is stated that *MargheritaPizzaDish containsTopping TomatoIngredient*, then the reasoner would be able to infer that *MargheritaPizzaDish containsIngredient TomatoIngredient*. Properties can be Transitive, Functional.....

Defined Classes are classes that contain rules. For example, a *"VegetarianDish"* could be defined as a dish that contains no *"MeatIngredient"*. In an ontology, you would say that a *"VegetarianDish"* is a subclass of *'Dish'* and a subclass of *"not(containsIngredient some MeatIngredient)"*.

Instances are just that, an instance of a class. You can have many instances of a class, each with different object properties and data properties. This way, you could create two instances of an ingredient but with different numbers of calories. This would be useful if you have different quantities of the same ingredient in two different dishes.

Enumerations.

2.2.1 Open World Assumption

Ontologies have an open world assumption, so all constraints on classes must be declared. It is not sufficient to say that a dish contains some ingredients, but also that it contains only these ingredients. Otherwise, a dish with those ingredients plus more would be valid. This is important when calculating things like a vegetarian dish. Take the example of a Margherita pizza, which contains a pizza base, tomato sauce and cheese. If the ontology only states that it contains at least these ingredients, then while a Margherita pizza would be classed as a margherita pizza, so would a meat feast pizza that contains those three ingredients plus more. It is for this reason why the reasoner cannot deduce that a margherita pizza is a vegetarian dish, unless the ontology states that a margherita can only contain those three ingredients.

2.3 The OWL API

2.4 Java Swing

I chose Java Swing because it has a lot of components already set up that can be used. JFrames are the windows that contain JPanels (a view). JPanels have a layout, and you can add many types of Components to display, such as a JTextField which allows the user to enter text, a JButton which makes actions for the user to take obvious, JLists to display data and JCheckboxes to select specific options. A JSlider can be used to choose a particular value from a set range, and a JOptionPane which displays a message in a pop-up window - which is useful for giving informative feedback. They also allow for clear consistency between objects of similar types.

Chapter 3

Design & Methodology

3.1 Requirements

To evaluate the system, I have come up with a list of functional and non-functional requirements that the system should comply to. These are broken down into those which are compulsory, and those which would be good to include if time permits.

3.1.1 Functional Requirements

Functional requirements for the system:

- The system needs to be able to store all dishes from the menu of a restaurant, along with the ingredients that make up that dish.
- The system should be able to produce a list of these dishes to the customer.
- The owner of a restaurant should be able to add ingredients and dishes to the system.
- A customer needs to be able to query the system for dishes they desire.
- Therefore, the system should filter the dishes by various dietry requirements, such as diets, allergies and filter by calories.
- The system should have the option to hide calorie information to those who do not want to see it.

Possible additions to the system:

- The system could filter dishes by how a meal has been prepared and cooked.
- The system could calculate new information for a dish based on whether a customer would like to add or remove an ingredient.
- The system could allow for a restaurant owner to register new allergens and diets to filter by.

3.1.2 Non-Functional Requirements

The non-functional requirements of the project are that it must:

- The system must be intuitive, easy to use, and provide appropriate feedback when changes are made to the system.
- The interface must conform to Ben Shneiderman's 8 golden rules [3].
- A restaurant owner should require minimal training to use the system.
- A customer should need no training to use the system.
- The system should allow customers to search for dishes within an appropriate response time. Performance is important as the restaurant and the customer will want the customer to make a decision quickly on what dish they would like to order.

3.1.3 Filters

Dietary Requirements to filter by:

- Vegetarian
- Vegan
- Pescetarian
- Religious diets e.g. Halal / Kosher preparation, Hindu and Sikh.

The most common allergies [1]:

- Peanut

- Ceoliac (gluten)
- Wheat
- Cow's milk
- Eggs
- Fish
- Shellfish
- Tree nuts
- Soybeans

3.2 My Ontology Design

My first ontology design was laid out as can be seen in Figure X and was heavily inspired by [2]. It consisted of Dishes, Components, Ingredients and Allergens with an obejct property *"contains"* to link them in that order. A Dish contains Components, which contains Ingredients which contains Allergens. Then there are Customers, who *"canEat"* dishes - another obejct property. Calorie information was stored on an ingredient level, with a data property *"hasCalories"*. On top of this, a *"PreparationMethod"* class would be created to store whether a dish was suitable for halal or kosher diets based on how it was prepared. This could also be extended to encompass unhealthy cooking methods, such as deep frying.

After reading the paper that inspired that diagram[2], I realised how easy it was to create redundant data. For example, the Customer class is not required at all - we have all we need from the defined dish classes.

3.3 Evaluation Plan

To test and evaluate the ontology system, I plan to conduct user studies, where I will ask people that I know to carry out certain tasks using the system, such as adding or removing a dish from the system, and searching for dishes with certain parameters. These tasks will cover each aspect of the functional requirements of the system to completely evaluate how successfully the system meets the criteria. I will then ask a

few short questions to cover the non-functional requirements and assess the success of the system at meeting those.

The user studies will be caried out in two rounds. The first pilot study will be carried out much earlier, so I can gauge what improvements need to be made. This is important as I will be able to identify whether I am on the right track with the project, and gain feedback on what does and doesn't work. Then, I will carry out a second user study after development on the system has finished to see how the system fits the requirements set out. Ideally, I would carry out a series of continuous feedback sessions - where users can repeatedly give feedback on the system for me to improve.

Each user study will be split into two smaller evaluations, one of the user interface for the customer, and one of the user interface for the restaurant owner. The participants were split in half and instructed to act the role of either a customer or restaurant owner. Each study will be tailored towards what those two roles should be able to achieve from their separate interfaces. For example, for the customer study, the participant will not be given any instruction on how to use the system, as real customers will not and should not need to require training to use the system. However, the owner of the restaurant will be told how the system works initially, and then not during the second phase of the final study. This is because in the real world, they will only recieve training initially, and should not need to relearn every time they want to use the system. A restaurant doesn't change it's menu that often, so they will not be using the system frequently.

It is important that the participants of the study are comfortable and give an honest review of the system. To do this, I aim to avoid putting them under any pressures. I will not place them under any time constraints to complete the tasks, and make sure that when I propose the study to them that the time estimation is accurate, while also a slight overestimate. It is also important that the participant doesn't feel any pressure to falsely support the system, when it is in fact failing. To do this, before the study I made it clear to the participants that they were not being judged based on whether they were able to complete the tasks, and that the results are purely reflective of how well the system is built and not their abilities. I also made sure they knew that I wanted to hear about problems as I want to improve the system where it is failing. Especially during the pilot I emphasized that there are improvements that need to be made and encouraged the participant to point out problems, as there was still plenty of time left to fix them.

I chose to have 3 participants per role for my study, as I found in the pilot that this number was the most appropriate. The big issues that I expected were brought up by

all participants, and everything else was small or personal preferences that were low priority issues to fix.

3.3.1 Ethics

I will carry the studies out in an ethical way. All participants will be over 18 and able to fully consent to taking part. No participant will be coerced into completing the study, and they are free to withdraw at any point. Also no personal information will be stored about them, or any information that can be used to identify them. The only information that will be collected will be whether they were able to complete all the tasks, and the feedback given, both qualitative and quantitative.

On top of this, there will be no harm inflicted on the participants, and I will make it clear what I am using the information I collect is for.

3.3.2 Interpreting the Results

With the results of the user study, I adopted a mildly agile workflow to move forward. I listed all of the problems that were identified during the study, and then a list of solutions. I scored each solution from 1-5 with an urgnecy score - based on how important it was that this change was made - and a time score - how long I estimated it would take to implement the solution. I then made sure to tackle the high urgency issues first, especially those with a small time estimate. This way, I could make sure to get as many of the more impactful problems solved within the time I had left.

3.4 Design Choices

Initially, I started creating the user interface entirely in one class, with a method that would create and return a JFrame for each of the different pages. I would then change Frame by closing the old one and opening the new one. After making a few pages I quickly realised this was a bad idea. I ended up making 19 different pages, and this would create one very difficult to read and convoluted class. After doing some research[**?**], I found the better way to approach this was to make each page it's own class which extends JPanel, and then use a CardLayout to switch between each page. The CardLayout gives takes each page with a String that can be used to identify it (I made this a class variable of the page), and then you can show the panel you desire

by using it's identifier. Despite the time it took to learn about, it made my code much more navigable and saved me time in the long run.

One design choice that was made after my pilot study was to change the way I showed how things in a list were selected. Previously, users had a list of everything provided, and had to click on an element to select it. If a user wanted to select multiple elements in the list, they would have to hold the Ctrl button to do so. If Ctrl was not held, any previously selected elements would be unselected. This is similar to how file selection works in File Explorer, and the default selection model for a JList. However, as many of lists were long and had to be scrolled, it wasn't obvious that the previously selected elements were now unselected. My first thought was to solely change the selection model so that you only needed to click on each element to select it, and the only way to unselect an element was to select it again. However I ultimately decided on a different method, where there is one box to select elements from, and another box which contains all the selected elements. To unselect an element, click on it in the second list and it will automatically update. This makes it much more clear what has been selected as you can see them without seeing everything that hasn't been selected. It also supports the ability to select an element multiple times, which could be desirable in the case where a customer wants to add an ingredient twice to a dish.

One big consideration I had at the end was the purpose of the Component. With the particular menu I have chosen, components aren't particularly necessary. A burger is a whole meal and it doesn't consist of definitive sections, just ingredients. This caused some confusion at what components to make during the user study when the user was given free reign to make a dish. Other menus lend themselves more to the idea of a Component however. If you have pie with mash potatoes and roasted vegetables - you have three very clear components to the final dish made up of many ingredients themselves. It lends itself to larger dishes that contain multiple parts, rather than menus where you order lots of small bits or somethiing that could be a Component itself. This is the reason I decided to keep Components in, as it gives the system flexibility to be used in different kinds of restaurants.

Chapter 4

Results

4.1 Deliverables

4.1.1 Ontology

4.1.2 Restaurant UI

4.1.3 Customer UI

4.2 User Studies

4.2.1 Pilot User Study

4.2.2 Final User Study

Chapter 5

Evaluation

5.1 Results Summary

5.2 Did I Meet My Objectives?

5.3 Improvements

5.4 Future Work

Bibliography

[1] Wesley Burks, Ricki Helm, Steve Stanley, and Gary A Bannon. Food allergens. *Current opinion in allergy and clinical immunology*, 1(3):243–248, 2001.

[2] Pablo López-García, Eduardo Mena, and Jesus Bermudez. Some common pitfalls in the design of ontology-driven information systems., 01 2009.

[3] Ben Shneiderman. The eight golden rules of interface design.

Appendix A

Example of operation

An appendix is just like any other chapter, except that it comes after the appendix command in the master file.