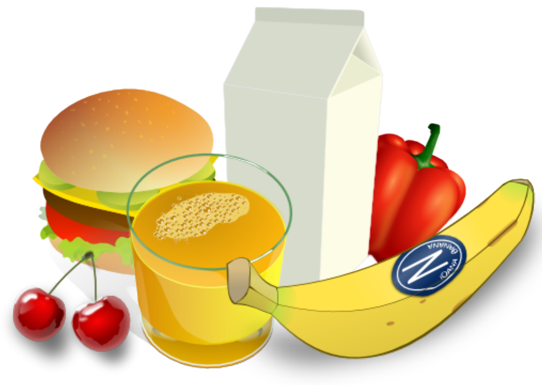
Foodolini

DAT1 Project - Group d101a
Computer science, Aalborg University
Autumn semester 2009

Authors:
Karsten Jakobsen
Anne K. Jensen
Jonas F. Jensen
Christina Lillelund
Sabrine Mouritsen
Thomas Nielsen
Lars Kærlund Østergaard

Supervisor:
Carmen Ruiz Vincente



Foodolini
Project report

Abstract

This report documents the analysis, design and implementation of the system Foodolini. Foodolini is designed to administrate food storage in the private household. This information can then be used for suggesting recipes and creating a shopping list.

This project was conducted using the OOA&D method for analysis and design of the functionality and class structure, the ADRIA method for user-interface design and the IDA method for usability evaluation. This report documents how these methods were applied to create Foodolini, from the analysis where the functionality was determined, through the design phase and finally the test and usability evaluation where the final result was evaluated. This report extensively documents this process with documents, diagrams and pictures.

Theme:
System Development

Project period:
DAT1, autumn semester 2009

Project group:
D101A

Concluded:
16th December 2009

Supervisor:
Carmen Ruiz Vincente.

Prints: 9

Report page count: 155

Appendix page count: 34

Group members:

Karsten Jakobsen	------------------------.
Anne K. Jensen	------------------------.
Jonas F. Jensen	------------------------.
Christina Lillelund	------------------------.
Sabrine C. H. Mouritsen	------------------------.
Thomas S. Nielsen	------------------------.
Lars Kærlund Østergaard	------------------------.

The department of computer science at Aalborg University

Preface

This report was written as part of the Dat1 project at Aalborg University by group D101a, autumn semester 2009.

The reader is expected to be familiar with the OOA&D method (see Mathiassen et al. [2009]), including UML. In addition, it is also expected that the reader has knowledge of the ADRIA method (see Dolog and Stage [2009]) with regard to design of user interfaces. Later, in the report, the reader will be introduced to more UML diagrams and sections discussing the subject of object-oriented design, along with the C# programming language. For that reason, an understanding of these subjects is necessary. Lastly, familiarity with unit testing and the IDA evaluation method (see Kjeldskov et al. [2004]) is required.

How to Read this Report

This report is organised into five parts. Each part focuses on a specific subject matter. The following briefly highlights the topic of each main part.

Analysis

The first part of this report covers and documents the analysis work done in relation to the OOA&D software development method. In short, this includes the system definition, problem domain, application domain and recommendations for further development.

Design

In this part, the design activity, proposed by the OOA&D method, of the system is explained. This includes the technical platform, architecture, components, evaluation and test plan.

Evaluation

The evaluation part brings up the evaluation of the system developed in connection with this project. It describes the method used and findings from the evaluations carried out.

Test Document

The test document details the unit tests performed, as well as the results.

Study Report

This report explains and discusses the realisation and organisation of the development project in its entirety. The analysis and design process, including the tools used, are reflected upon. Finally, suggestions for future work are covered.

Litterature Used in this Report

The analysis and design parts of the report follow the document standards, which are based on the guidelines put forward in Mathiassen et al. [2009] with minor modifications, according to the DAT1 course. For that reason, references to Mathiassen et al. [2009] have been omitted.

Obtaining the Source Code and Documentation

The source code and documentation for the program are located on the CD attached to the last page of this report. The source code exists in two versions; `foodolini/src/linux` and `foodolini/src/windows`, the first configured to be compiled on a Linux-based operating system, the latter compiles on a Windows-based system. Binaries for each operating system are also included on the CD, and are located in `foodolini/bin/linux` and `foodolini/bin/windows`, respectively. The HTML documentation, compiled by Doxygen, is located in the folder: `foodolini/docs`. Log files from the user evaluation are also on the disk in the folder: `foodolini/evaluation`.

Contents

List of Figures

Part I

Analysis

CHAPTER 1

Task

1.1 Purpose

The overall purpose of this system is to keep track of the user's food supply. In addition, the system can be used to suggest recipes, track food consumption and log exercises. This will allow the user to monitor their nutritional intake and expenditure, making it possible to use the system as a tool for dieting.

1.2 System Definition

Foodolini is a system designed to assist persons of a household in keeping track of the contents of their food storage. Furthermore, it will be capable of suggesting recipes based on stock, expiration date, rating and diet. Foodolini will be able to help manage grocery lists for restocking and recipes. The system must be efficient to use by a varying user group, with different needs in terms of diet and desires. Foodolini will be written in C# using the .NET/Mono platform, however, the system must be platform independent. Foodolini should be able to supervise dieting in relation to excercise and nutrional needs.

FACTOR

- **Functionality**
  - Suggest recipes from current content and diet (caloric expenses/exercise)
  - Create shopping list
  - Keep track of expiration dates
- **Application Domain**
  - The people living in the household are primary users of the system.
  - Guests who dine in the household are secondary users.
  - The system is used in the kitchen.
- **Conditions**
  - Must be efficient to use
  - Users have varying computer experience
- **Technology**
  - Must work on commodity hardware
  - C# and .NET/Mono

  - Platform independent
- **Objects**
  - Refrigerator, user, recipe, food item, ingredient, shopping list, diet and exercise
- **Responsibility**
  - A system that assists the user in the daily food consumption
  - All users are equal
  - Voluntary surveillance system

1.3 Context

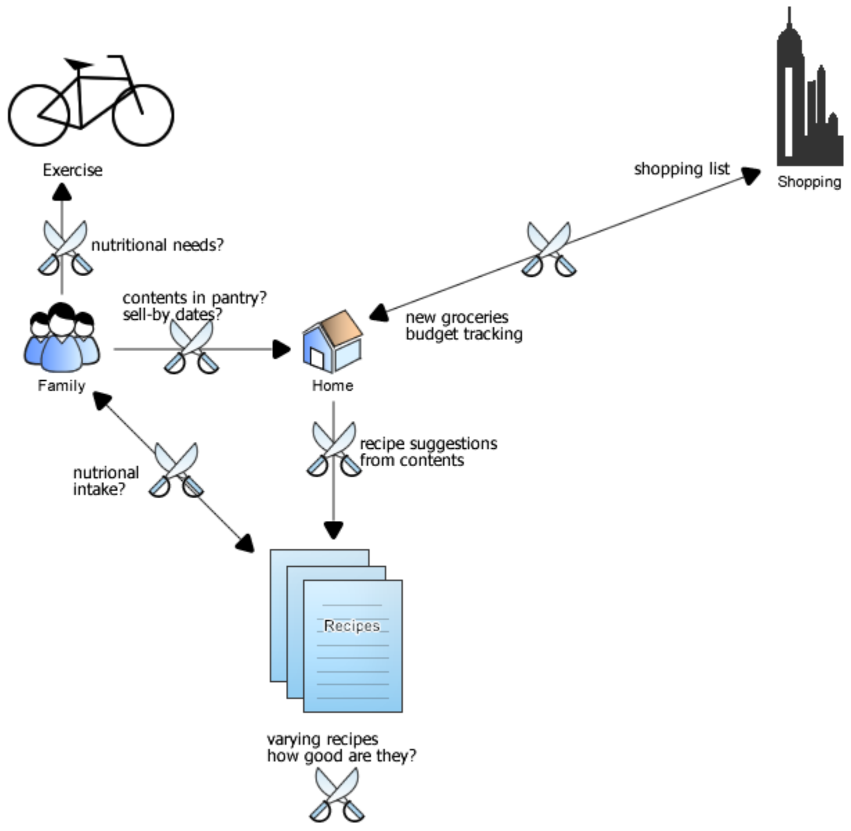


Figure 1.1: Rich picture illustrating the daily problems of the user.

The rich picture in figure 1.1 describes a family setting inside their home. A family will often purchase and consume a lot of food items. Knowing what food items currently in stock can become an overwhelming task, especially if shopping is spread amongst several family members, which can also make budget tracking difficult. Shopping in itself can be a problem, especially for certain items only a few of the family members enjoy.

Futhermore, members of the family may wish to exercise regularly, but may be unaware of the increased demand in their diet due to this activity. Some of the household members may also dislike certain recipes and in a household family it can be difficult for the cook to remember the likes and dislikes of each individual family member. Finally, some recipes are better than others, and sorting the bad ones from the good ones is desirable.

1.4 Problem Domain

The problem domain focuses primarily on cataloging food stored in a household. This includes monitoring expiration dates of foodstuff, finding recipes based on what food a user already has in stock and tracking daily calorie and nutritional intake.

1.5 Application Domain

The application domain involves people in an ordinary household, who require an overview of their food, diet, recipes and exercise. Typical tasks revolve around registering newly purchased groceries in the storage, modifying the contents of the storage when food is consumed and entering exercises performed by the user. Furthermore, a user may need a recipe that includes certain ingredients currently in stock and a shopping list with the missing ingredients.

CHAPTER 2

Problem Domain

2.1 Cluster diagram

Figure 2.1 is a cluster diagram of the system, Foodolini. The three main classes - Person, Storage and Cookbook - have been used to create the clusters.

The first cluster, Person, contain classes that relate to Person. Therefore, the classes: Person, ShoppingList, Diet, Exercise and SportsActivity are in the cluster Person. Each Person has an individual ShoppingList, which means that Person and ShoppingList are closely connected. Diet is used in two places: a Person always has a Diet, which can be used to search for Recipes in the Cookbook. When the Cookbook is searched with the Diet, it is always based on a search by the current user, who is registered as a Person. For that reason, Diet belongs in the Person-cluster. Exercise is performed by a Person and an exercise has an SportActivity. So, these are also in the Person cluster.

The Storage cluster contains classes that relate to the Storage and registration of food. Thus, the classes: Storage, FoodItem, Ingredient, and BarCode are in this cluster. Ingredient is also used in the Cookbook cluster, but given that the main feature of Foodolini is storage management, it is regarded as more important in Storage than in Cookbook.

The last cluster, Cookbook, contains the classes: Cookbook, Recipe and ShoppingList. These classes all deal with how to cook meals, and belong in a common cluster.

2.2 Class Diagram

The class diagram of Foodolini can be seen in figure 2.2.

The main class is Storage, which is associated with the Cookbook, with the purpose of the Cookbook knowing what is in Storage when searching for Recipes. Both classes will only be created once, and will never be deleted. The Cookbook consists of two-to-many Recipes, while Storage consists of zero-to-many FoodItems. FoodItem is associated with Ingredient in an Item-Descriptor Pattern, which means that Ingredient is a describer of FoodItem. Recipe aggregates one-to-many Ingredients, while an Ingredient is aggregated to zero or more Recipes. This is done because a Recipe consists of one or more Ingredients, yet an Ingredient does not necessarily have to be in any Recipe. An Ingredient is associated with zero-to-many BarCodes, as an Ingredient contains a collection of BarCodes. For example, there can be many different brands of an Ingredient, all having a different BarCode, but still being the same Ingredient. The ShoppingList is aggregated by zero-to-many Ingredients and is associated to Person. Each instance of a Person must have a ShoppingList, which implies a one-to-one association.

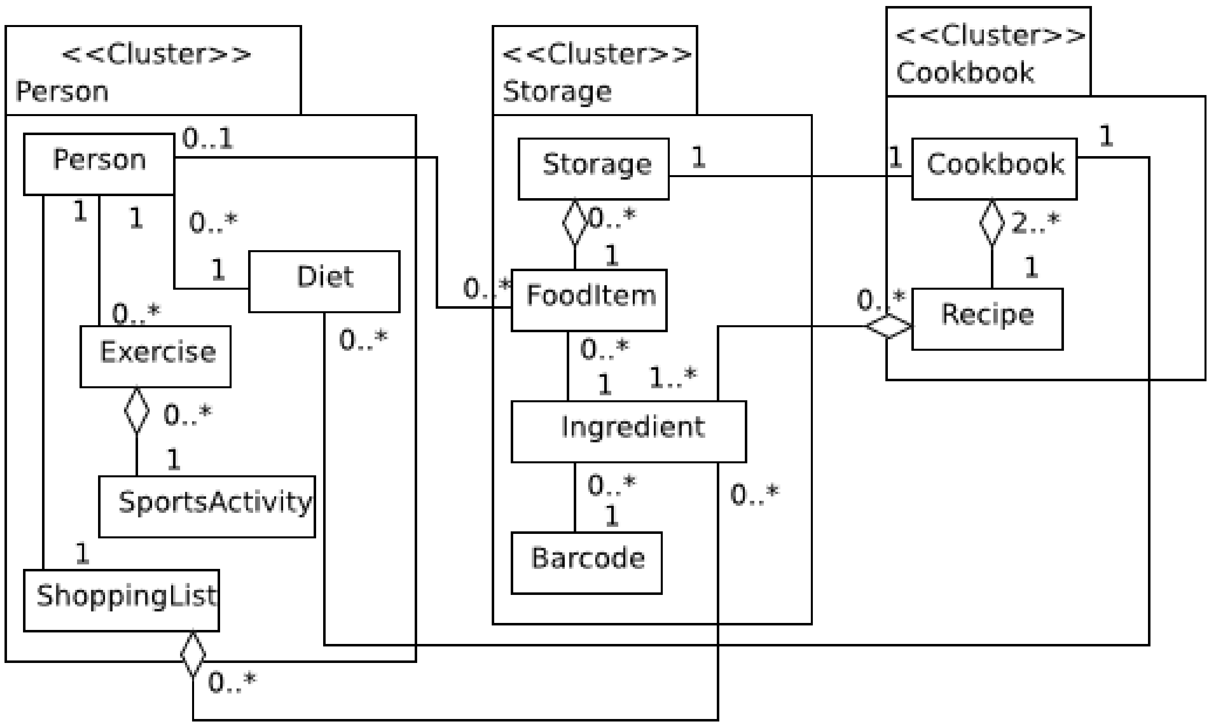


Figure 2.1: Cluster diagram

The Person class has an aggregation with FoodItem that tells what the Person has consumed, and is associated with Diet and Exercise. A Person can only have one Diet, but a Diet can have many Persons, so that a household can share Diets. Diet is also associated with Cookbook, because the Cookbook needs Diet to suggest Recipes. A Person can have zero-to-many Exercises, but an Exercise can only have one Sports-Activity. On the other hand, a SportsActivity can be associated with zero-to-many Exercises, while an Exercise only can be associated with one Person. The relationship between SportsActivity and Exercise is also an Item-Descriptor Pattern, where Sports-Activity is a describer of Exercise.

2.3 Class Description

In this section the classes from the class diagram (see figure 2.2) will be described. The classes will be briefly explained, followed by a state chart diagram for the events of that class.

2.3.1 Storage

The Storage class is one of the central classes of this system. Storage is a list of all FoodItems in storage. That being in the refrigerator, the cupboard, or the freezer. Even though the Storage can represent any kind of storage place, the system has no knowledge of where each FoodItem is. So, it is the user's own resonsibilty to keep track of

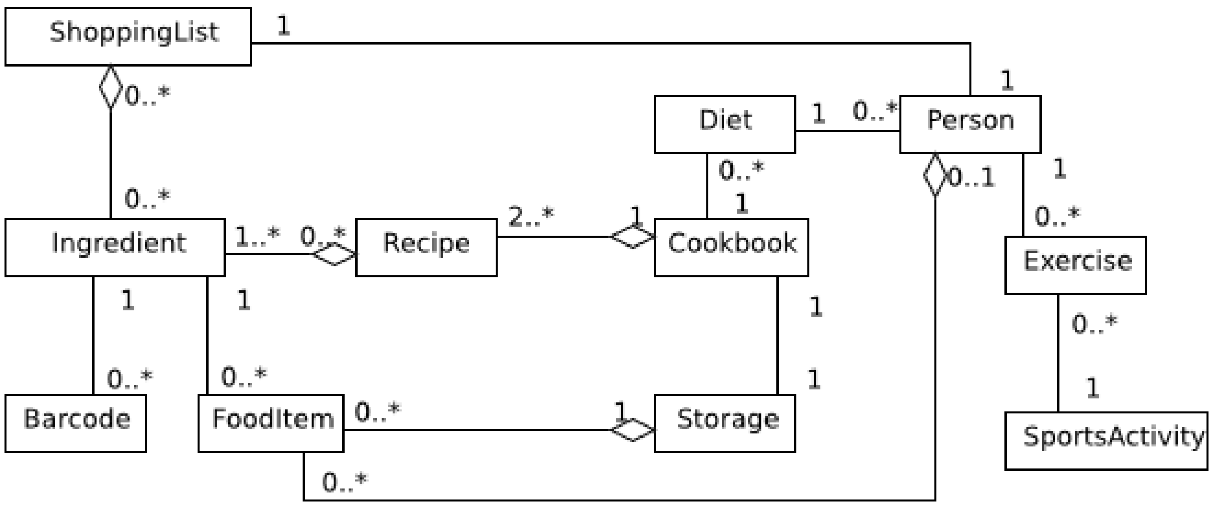


Figure 2.2: Class diagram

this. When the user then asks for a Recipe, the storage can state which FoodItems are available.

Behavioral Pattern for Storage

The content of Storage is updated by the user when he or she enters a new FoodItem. If the FoodItem has been bought, and hence has a BarCode (See section 2.3.11), this is used to identify the product from a database which contains information about the nutritional facts of a wide variety of products. Hereafter, it is the users own responsibility to enter the remaining information, such as expiration date and quantity. Some items may not exist in the database, and the user must therefore enter all information. If the user has created the FoodItem from other FoodItems, the user must again enter all information. When the FoodItem has been used, the user must delete it from Storage, to maintain the correct contents. It is, of course, not possible to delete any FoodItems from Storage if it is empty. Figure 2.3 is a state chart diagram for the Storage class.

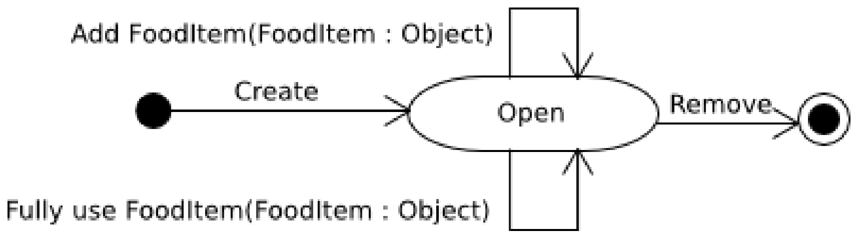

Add FoodItem(FoodItem : Object)
Create
Open
Remove
Fully use FoodItem(FoodItem : Object)

Figure 2.3: State chart diagram for Storage

2.3.2 FoodItem

In Foodolini a FoodItem is the food available in Storage. It can either be something bought in a supermarket or something made from other FoodItems. This opens up the possibility to, for example, bake a bread, and then use it the following day for lunch. This also implies that a FoodItem can be partially consumed, and leftovers are saved for later use.

A FoodItem contains information about the shelf life, expiration date after opening, the quantity and the nutritional values as an Ingredient (see section 2.3.3). The shelf life belongs to the Ingredient and is the life span of a FoodItem. The expiration date after opening is when the FoodItem expires after it has been opened.

The expiration date is provided by the user when the FoodItem is scanned. It is set to a default value of two days if made from a Recipe. The expiration date will be estimated by Foodolini if the associated ingredient has a shelf life assigned to it. As the Storage does not know where the FoodItem is physically stored, the expiration date will not always up-to-date. If the FoodItem is, for example, put in the freezer it will be the user's own responsibility to alter the expiration date.

Behavioral Pattern for FoodItem

After a FoodItem has been added to the Storage, see section 2.3.1, it can either be partially consumed, and then still be in the storage, or it can be fully consumed, and hence removed. When a FoodItem is partially consumed it is also considered open. When a FoodItem is open it is no longer the shelf life that determines when a FoodItem expires, rather it is the expiration date after opening.

The expiration date can also be passed while the FoodItem is in Storage, however the system allows one to continue using FoodItems that have expired. If the user chooses to do this, it changes state, but can still be fully consumed or partly consumed. From either state the FoodItem can be removed and thereby not be used to cook meals. A FoodItem can be removed without having to consume, or it being expired, as it can be necessary to account for situations where outside influence, not tracked by Foodolini, alters the contents of the storage. Figure 2.4 is a state chart diagram for the events of FoodItem.

2.3.3 Ingredient

An Ingredient contains the information about a certain type of food. This includes information about the nutritional values, such as calories, protein, carbohydrates, fat, salt vitamins and minerals, for every 100 grams.

Behavioral Pattern for Ingredient

After an Ingredient has been created, a Recipe (See section 2.3.4) can request information from it, as well as the user can modify its information. Moreover, an Ingredient can be added to the ShoppingList, where it is associated with a quantity indicating much to buy.

In figure 2.5 the events of Ingredient are shown.

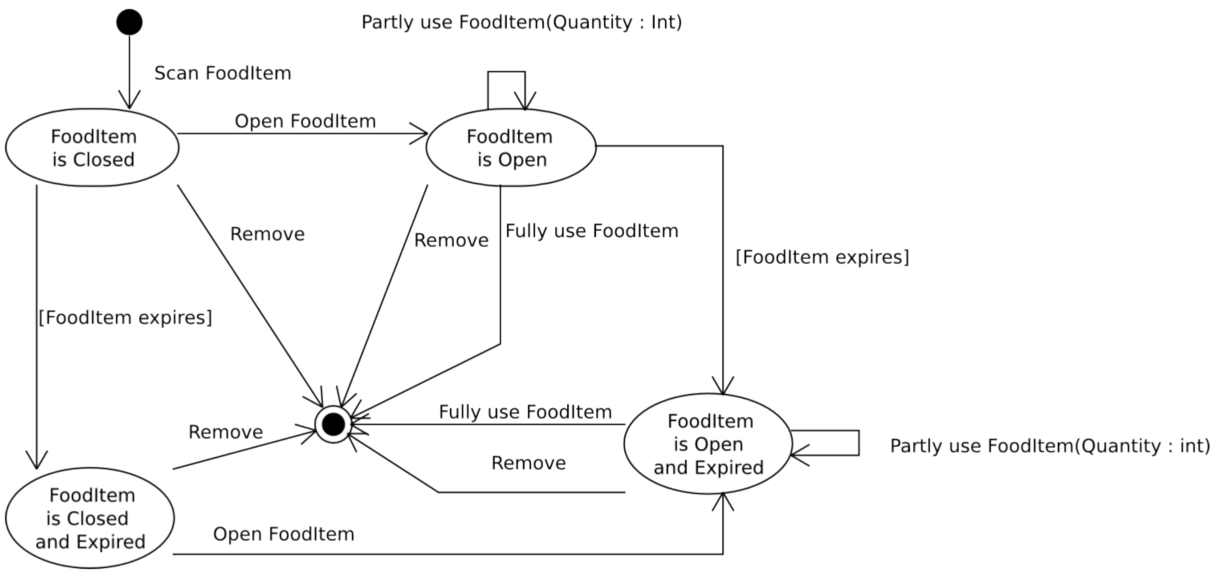


Figure 2.4: State chart diagram for FoodItem

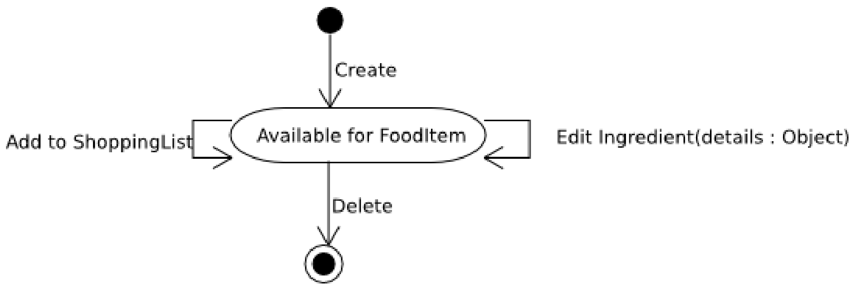


Figure 2.5: State chart diagram for Ingredient

2.3.4 Recipe

A Recipe contains a list of the required Ingredients to make a dish and the cooking instructions. Since a FoodItem is an amount of an Ingredient, the Cookbook can suggest a Recipe from the FoodItems currently in Storage (See section2.3.6).

Behavioral Pattern for Recipe

A Recipe has several events after being added to the Cookbook. The Cookbook can be suggest a Recipe, which can be rejected by the user. The user also has the ability to edit a Recipe, if, for example, incorrect information were entered, or another Ingredient is preferred. The ingredients of a Recipe can be added to a ShoppingList. The user is able to rate a Recipe at any time, and thereby change the occurrence of it. All these events are diagrammed in figure 2.6.

2.3.5 ShoppingList

The ShoppingList contains the Ingredients that need to be purchased.

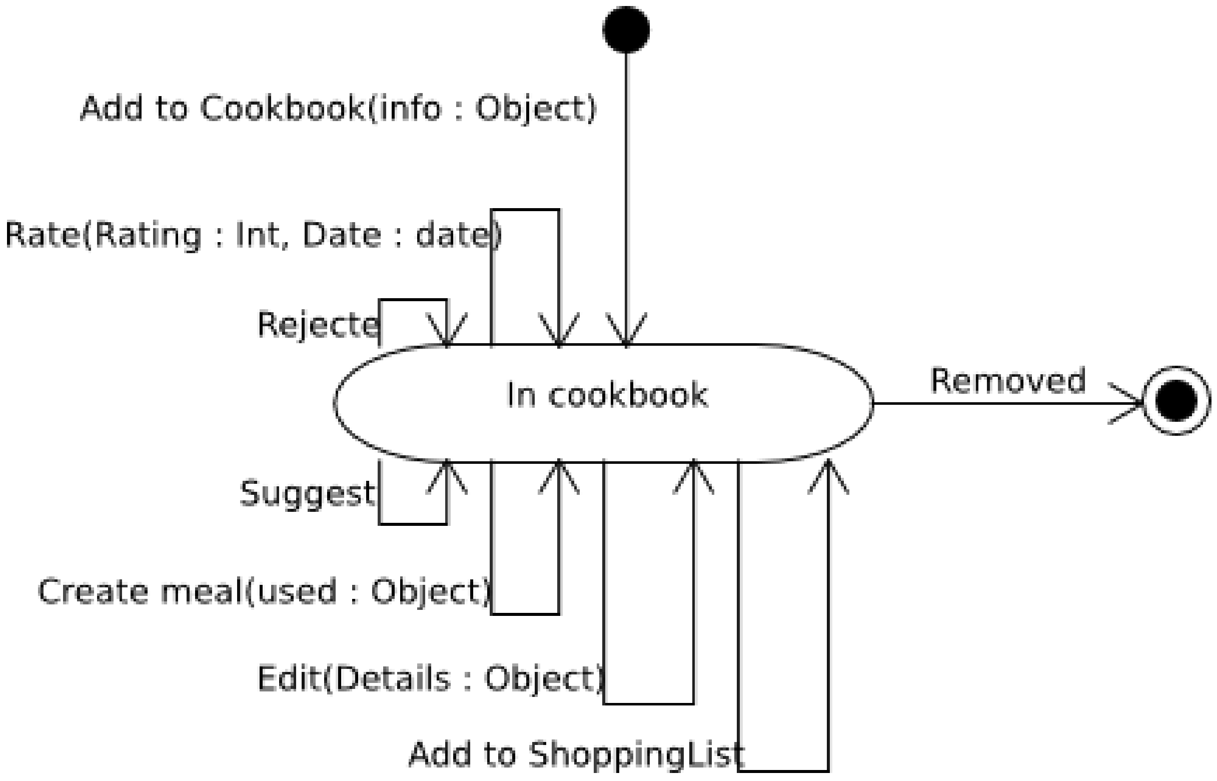

Add to Cookbook(info : Object)
Rate(Rating : Int, Date : date)
Rejecte
In cookbook
Removed
Suggest
Create meal(used : Object)
Edit(Details : Object)
Add to ShoppingList

Figure 2.6: State chart diagram for Recipe

Behavioral Pattern for ShoppingList

The state chart diagram for ShoppingList can be seen in figure 2.7. Recipes can also be added to the ShoppingList, or rather, the Ingredients listed in the Recipe are added. Ingredients can either be removed one by one or all together by clearing the ShoppingList.

2.3.6 Cookbook

The Cookbook contains all of the Recipes, and is responsible for suggesting Recipes to the user, based on the contents of Storage and the active Diet (See section 2.3.8).

Behavioral Pattern for Cookbook

The Cookbook can be searched for Recipes. Recipes can be created, edited or deleted from the Cookbook.

2.3.7 Person

A Person is a registered user of the system, which contains information about the name, height, weight, age, gender and current diet. These details are used for calculating the daily caloric needs in connection to the Exercises (See section 2.3.9) performed by the person.

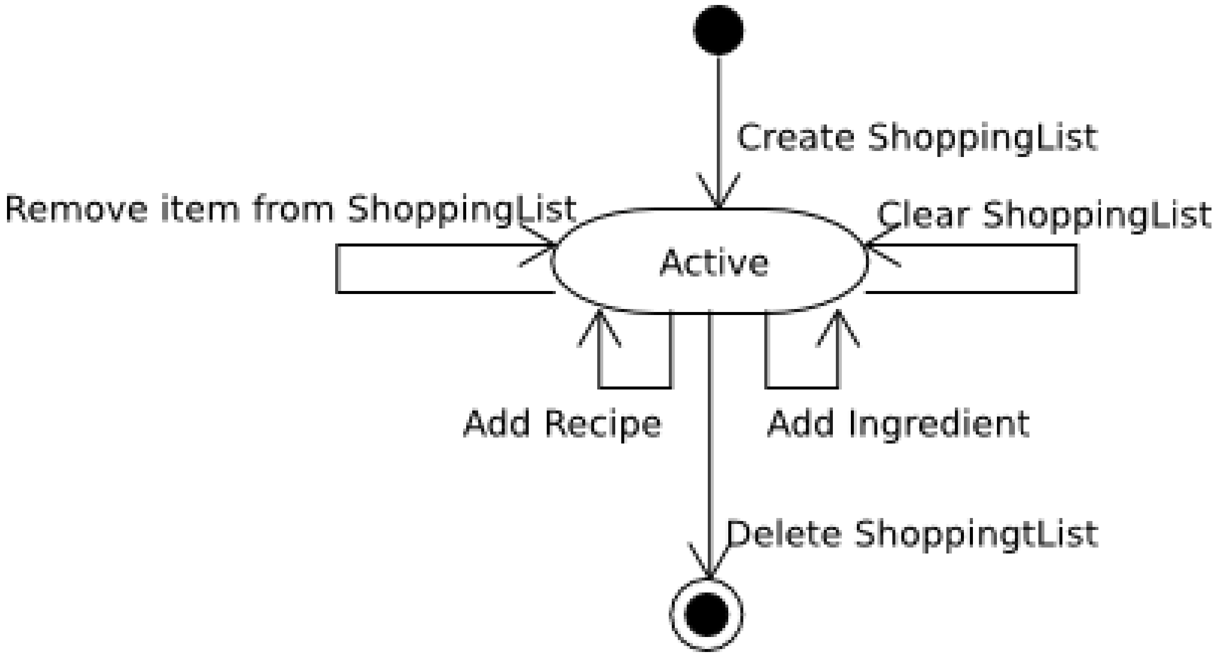


Figure 2.7: State chart diagram for ShoppingList

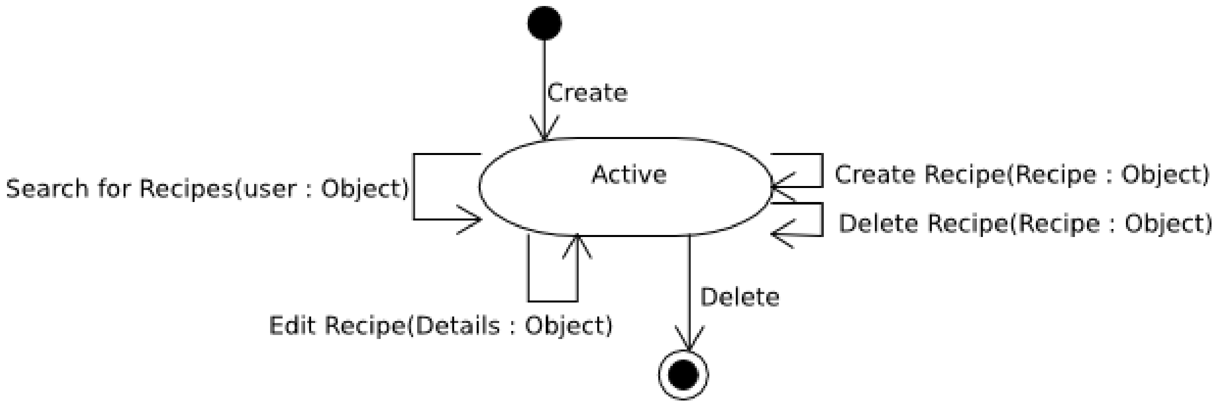

Create
Active
Search for Recipes(user : Object)
Create Recipe(Recipe : Object)
Delete Recipe(Recipe : Object)
Edit Recipe(Details : Object)
Delete

Figure 2.8: State chart diagram for Cookbook

Behavioral Pattern for Person

After registering the Person, the Person is active. After that the abovementioned details can be changed, for instance by changing the height or selecting a new Diet. The user can register any Exercises performed. The mentioned events can be seen in state chart diagram 2.9

2.3.8 Diet

A Diet contains information about a Persons daily nutritional needs, in terms of total calories, proteins, carbohydrates and fat. It is not a list of food that the user should eat at a specific time, nor food which the user may not eat, as the name could imply. When a Person, is created, a default Diet is selected, and hereafter it is possible to create new

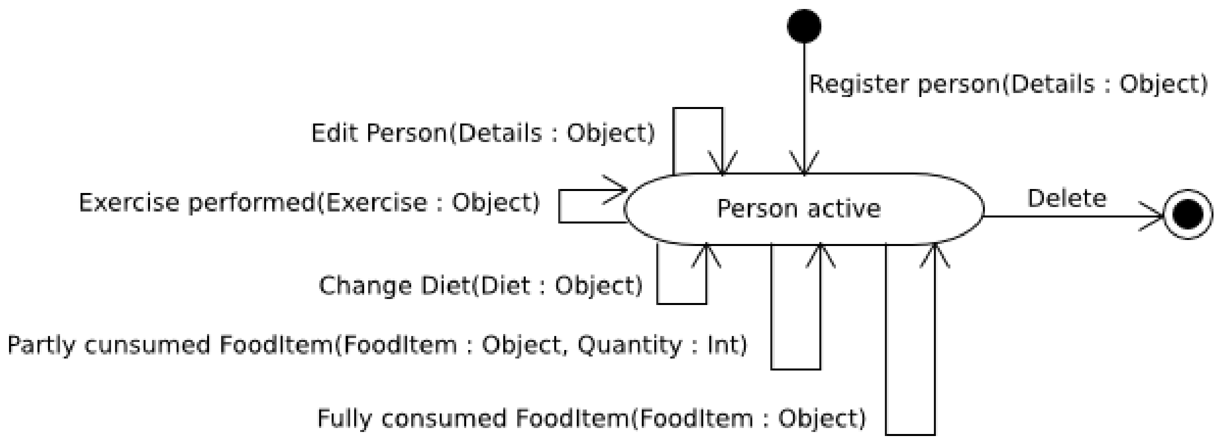

Register person(Details : Object)
Edit Person(Details : Object)
Exercise performed(Exercise : Object)
Person active
Delete
Change Diet(Diet : Object)
Partly cunsumed FoodItem(FoodItem : Object, Quantity : Int)
Fully consumed FoodItem(FoodItem : Object)

Figure 2.9: State chart diagram for Person

Diets. However, a Person can only have one Diet active at once, yet multiple Persons can share the same Diet.

Behavioral Pattern for Diet

The system allows the user to select one of the Diets in the system, but they can at any time switch to another, or create a new Diet. When a user chooses another Diet, they will automatically deselect the old one. This will make the process faster and more usable for a user who changes Diet often. It is possible to change the details of the Diet, in case some values were wrong entered. Then again, since users can share the same Diet, the effect of a change will effect all of the other users of that particular Diet. The state chart diagram for the Diet class can be seen in figure 2.10.

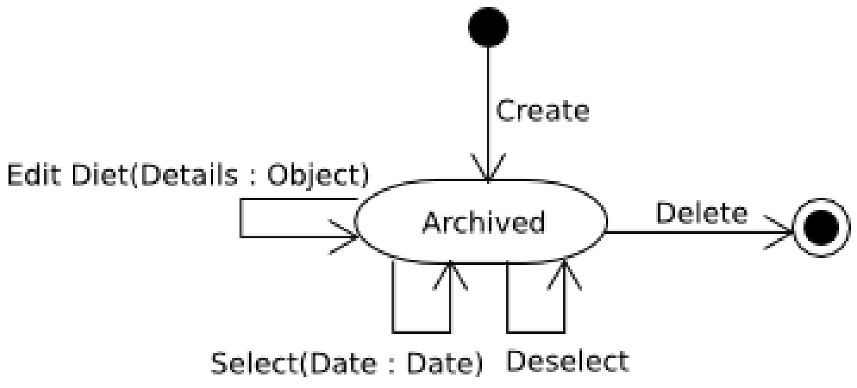


Figure 2.10: State chart diagram for Diet

2.3.9 Exercise

The user has the ability to enter which Exercise the user has performed, and the system will incorporate it when searching for Recipes, since performing exercises will change the caloric needs of a user. An Exercise is an activity the user has done. It consists of a certain SportsActivity (See section 2.3.10), which has been performed, such as running, as well as the duration, date and total calories burned.

Behavioral Pattern for Exercise

When creating an Exercise, the information mentioned in 2.3.9 is sent along, hereafter the class does not have many events as displayed in figure 2.11. The Exercise can be edited, if something needs changing.

If the owner (a Person) of the Exercise is deleted, so will all of the Exercises registered by that person.

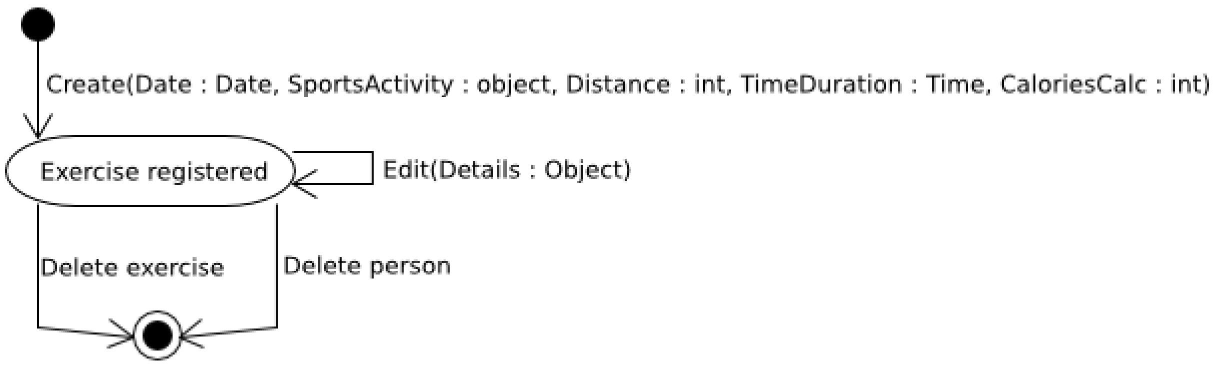

Create(Date : Date, SportsActivity : object, Distance : int, TimeDuration : Time, CaloriesCalc : int)
Exercise registered
Edit(Details : Object)
Delete exercise
Delete person

Figure 2.11: State chart diagram for Exercise

2.3.10 SportsActivity

A SportsActivity is a type of sport - much like an Ingredient is a type of food. It has a predefined amount of calories burned per minute of execution, and the total caloric expenditure can then be calculated in Exercise.

A SportsActivity can be activities such as running, cycling, swimming, etc.

Behavioral Pattern for SportsActivity

SportsActivities can be created, edited and deleted, as seen in figure 2.12.

2.3.11 BarCode

As mentioned in section 2.3.2, a FoodItem can either be added through its bar code or by entering all of the information about the FoodItem. A BarCode is always associated with an single Ingredient. But an Ingredient can have several BarCodes, as an example the Ingredient, milk, can have several brands with different BarCodes pointing to it. In other words, there exists a one-to-many relationship from Ingredient to BarCode.

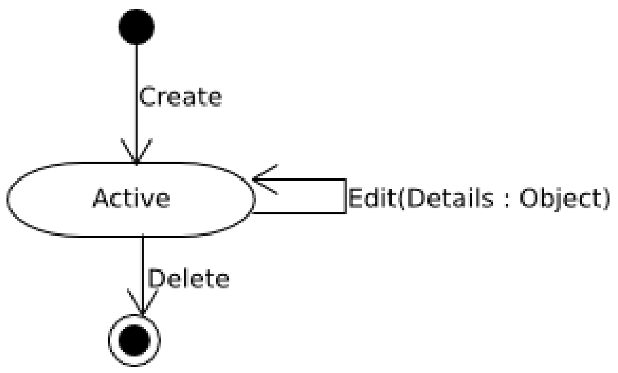


Figure 2.12: State chart diagram for SportsActivity

Behavioral Pattern for BarCode

If the system identifies a new BarCode, it will be added to the catalogue. The BarCode does not have a termination state, since the system does not allow deletion of BarCodes.

2.4 Event Table

Table 2.1 is an event table including all of the events mentioned in the above section. If more than one class is marked for an event, the event is common. This means that all of the marked classes participate in that particular event. However, in the case of the grey areas, the marked classes are not all necessary for the event; these events are not common, but merely an event that all of the marked classes have. To avoid repeating these events for every single class, they have been consolidated into one.

Events / Classes	Storage	FoodItem	Ingredient	Recipe	Cookbook	Person	Diet	Exercise	SportsActivity	BarCode	ShoppingList
Register FoodItem	*	+								+	
Fully use FoodItem	*	+									
Partly used FoodItem		*									
Fully consume FoodItem		+				*					
Open FoodItem		+									
Partly consume FoodItem		+				*					
Remove FoodItem	+	+									
FoodItem expried											
Edit Ingredient			*								
Add Ingredient to Recipe			*	*							
Add Ingredient to FoodItem		*	*								
Add Ingredient to ShoppingList			*								*
Create Recipe				+	*						
Rate Recipe				*							
Search Recipe				*	*						
Reject Recipe				*							
Edit Recipe				*							
Add Recipe to shoppingList				*							*
Cook meal	*	*		*	*						
List Recipes				*	*						
Create Person						+					
Delete Person						+		+			
Create Diet							+				
Edit Diet						*	*				
Selcted Diet							+				
Deselected Diet							+				
Register Exercise						*		+	*		
Edit Exercise								*			
Edit SportsActivity									+		
Create/Register			+			+			+		
Delete		+	+	+			+	+	+	+	

Table 2.1: Event table

CHAPTER 3

Application Domain

3.1 Overview

Two actors are identified as users of the Foodolini; a person and an administrator. Table 3.1 illustrates the identified actors and their respective use cases.

	ACTORS	
USE CASES	User	Administrator
Register FoodItem	✔	
View FoodItem	✔	
Edit FoodItem	✔	
Delete FoodItem	✔	
Create Ingredient	✔	
Add Ingredient	✔	
View Ingredient	✔	
Edit Ingredient	✔	
Delete Ingredient	✔	
Create Recipe	✔	
View Recipe	✔	
Edit Recipe	✔	
Delete Recipe	✔	
Rate Recipe	✔	
Create meal from Recipe	✔	
View ShoppingList	✔	
Add Recipe to ShoppingList	✔	
Remove Item from ShoppingList	✔	
Edit Person	✔	
Register Exercise	✔	
View Exercise	✔	
Edit Exercie	✔	
Delete Exercise	✔	
Create Diet	✔	
View Diet	✔	
Choose Diet	✔	
Delete Diet	✔	
Edit Diet	✔	

	ACTORS	
USE CASES	User	Administrator
Add SportsActivity	✔	
Create Person		✔
Edit Person	✔	
Delete Person		✔
Create SportsActivity	✔	
View SportsActivity	✔	
Edit SportsActivity	✔	
Delete SportsActivity	✔	

Table 3.1: Actor table for the Foodolini.

The role as a user is performed by any person in a household using Foodolini. The administrator role may also be performed by any person from the household using Foodolini, yet the administrator role is the only one that allows access to create or delete persons of the household. This is to prevent destructive actions. Editing a user can, in theory, be as destructive as deleting a user, though editing needes much more dedication, and hence not considered equally destructive.

3.2 Personas

Personas can be used to specify some stereotypes of the target group, so that the system developers know who to design for. Making detailed personas will often help to understand the users, and make sure that the designers all have the same view of the target group.

3.2.1 Professional with Children

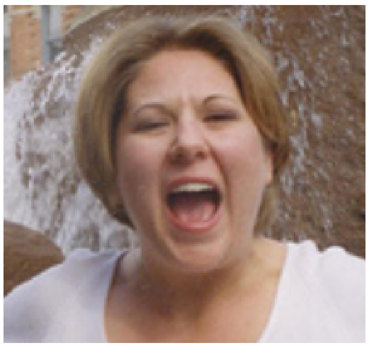


Figure 3.1: Linda Hansen [Adam, 2006]

Biography

Linda Hansen is a 39-year-old woman, working as a kindergarden teacher. She is married to Ulf (44) and they have three children: Emilie (12), Sandie (9) and Ernst (2). They live in a house in Spøttrup where Linda works. Ulf is a middle manager at an entrepeneuring company in Århus - he works 42 hours per week and spends an hour per day commuting. Thus most of the cooking and shopping fall on Linda, as well as picking up the children after school and kindergarten. Linda is of average height and slightly chubby. She has blonde hair and brown eyes. Linda often feels stressed, and would like to spend less time wondering what she should make for dinner and how much milk is left in the refrigerator. Linda has a bit of trouble remembering which recipes are a success with all members of the family, and would like a sort of rating system to help her remember which.

Computer Experience

Linda would not describe herself as a computer whiz, but she has some experience. She uses the family computer for checking her e-mail, which she uses for staying in contact with friends all over the country, and checking if she won on her Lotto numbers. Linda buys most of her and the children's clothes online, as she feels it is quicker. The family has a digital camera, which Linda administrates.

Storage Management Experience

Linda has previously attempted to maintain a registry of what she has in her freezer, by counting everything in it and writing it in a spreadsheet. She found it to be tedious work to update her spreadsheet every time she bought or stored something. Emilie takes things from the freezer some times as well, and she did not update the spreadsheet. Thus, the registry was quickly out of date, and Linda dropped the idea again.

3.2.2 Person Who Throws Away Too Much Food

Figure 3.2: John Petersen [Ingorrr, 2007]

Biography

John Petersen is a 25-year old student, who lives alone in Århus. He studies hospitality and tourism management. His problem is that he has never learned to cook and knows next to nothing about cooking healthy food. He has lived off pizzas, pasta and oatmeal since moving away from home, but since his father had a stroke half a year ago he wants to lower his fat intake and eat a more varied diet. He has tried solving the problem on his own by buying more vegetables and fruit, but ends up throwing most out because he does not know what to do with it. John would like to use the food in his refrigerator and learn some simple, healthy recipes. John has black hair, brown eyes and is of average, slightly skinny build. He is farily interested in his appearance.

Computer Experience

John uses his computer for Facebook, chatting with friends, surfing on the internet and finding information for school. He takes a bit of interest in his computer - he knows how to install programs and download things from the Internet, but does not care how things work, as long as they work.

Storage Management Experience

John has an idea of what foods he has in storage, but has never attempted to manage it. He would very much like an efficient and easy way of ensuring he eats the healthy things he buys.

3.2.3 Man Who Wants to Gain Muscle

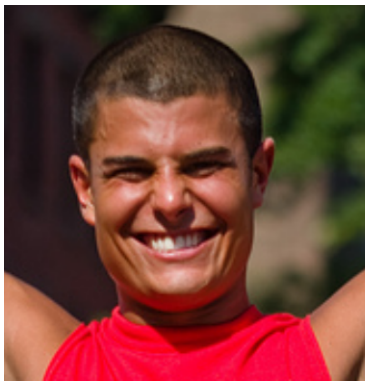


Figure 3.3: Knud Obel Hentze [ming Lee, 2007]

Biography

Knud Obel Hentze is a 24-year old apprentice mason. He lives in Aalborg with his pet spider Bob (3). He has a girlfriend, Maria (22), who comes over for dinner and sleeps over a few days per week. Knud is brown-haired and has blue eyes. He is fairly muscular of build, but wants to gain even more muscle. Knud is fond of his work, and feels he will look more competent with bigger muscles, which could help him excel in his career. In his spare time Knud exercises in a fitness center, runs, and plays football with his friends every Thursday. Knud would like his food intake to match his exercise, and help him build larger muscles. He does not know anything about nutrition, though, nor does he like searching for recipes.

Computer Experience

Knud has a laptop, which he uses for watching movies, checking his Hotmail and updating his MySpace and Facebook pages. He holds no particular interest in computers.

Storage Management Experience

Knud tends to shop for groceries every day, as he forgets what he has in the refrigerator. He only remembers what he lacks, when he needs it.

3.2.4 Woman Who Wants to Lose Weight

Figure 3.4: Helle Markussen [Art, 2009]

Biography

Helle Markussen is a 30-year old woman living with her husband Ove (29), his son Frederik (5) of an earlier relationship and two cats, Poul (4) and Kjøller (4), in a terrace house in Silkeborg. She bicycles the three kilometers to work every day, but is otherwise sedentary in her job and spare time. Helle has brown hair and brown eyes and wears glasses. She feels she has grown chubby over the years - she would like to lose some weight. She has dieted many times before to no avail, as the diets have either not worked, or she has fallen off the wagon. Helle does not know how to distribute the number of calories she is allowed in a day. She would like a system to tell her what to eat, so that she can just do that.

Computer Experience

Helle uses different accountancy programs for work and also administrates the boss' calendar and e-mails. Otherwise she is inexperienced with computers and does not use one in her spare time.

Storage Management Experience

Helle has never used or thought or using a system to manage the stock in her storage. She likes going shopping every day for the things she needs.

3.3 Scenarios

In this section the scenarios for the personas in section 3.2 are written. The scenarios are to illustrate the use of Foodolini by the targeted persons. The personas and the scenarios are imaginary and an expression of what type of persons we think would benefit from a system, such as Foodolini.

3.3.1 Linda Hansen

Linda Hansen, a working mother with 3 children in the age of 12, 9 and 2 years and a husband commuting, is the person responsible of cooking and shopping in the Hansen family. To plan the cooking of the family's meals, that being both breakfast, lunch and dinner, Linda finds a great help using Foodolini. Every evening, when the two older children are washing up after the dinner and her husband is bathing the youngest child, Linda goes to the family's joint PC in the corner of the kitchen. She turns on the PC and starts Foodolini. While waiting for the system to start up, one of her daughters, Sandie, tells her a story from school. When Foodolini is started Linda chooses to look through the list of recipes. To each recipe a rating from 1 to 5 is attached. The rating is based on the family's opinion of the recipe. When a recipe has been chosen, the system will check if all ingredients are stored in the household. Linda adds the recipe to the shopping list. Her oldest daughter reminds her that some apples also should be added to the shopping list. And the younger daughter asks for grapes. When Linda has read through the shopping list, she finally push the button "Print" to print out the shopping list and she puts the list in her handbag, ready to shop the following day.

3.3.2 John Petersen

John Petersen is a young man at twenty years. He lives in a small one-room flat, not far away from his college where he studies Hospitality and Tourism Management. As most young people of today, he has a laptop which he uses both in his study and in his spare time. After living half a year off pizza, he has now chosen to make his eating habits more healthy. He has installed Foodolini on his laptop. In his lunch break at college, he starts up Foodolini. He has only had Foodolini for a month, and there is still a lot of recipes John has not yet tried to cook. To support his goal of having better eating habits, he has, in the search criteria for recipes, stated how the energy distribution should be. He also still has some carrots, he would like to use. The carrots are also added to the search criteria and John push the button "Search". His study fellows look curiously over his shoulder as Foodolini lists the recipe suggestions. A girl recognises one of the recipe suggestions and recommends it to John, because it is quickly prepared and, if there are leftovers they are suitable for lunch the following day. John follows the girl's advice and chooses the recipe. Foodolini checks for missing ingredients and John adds the recipe to the shopping list, and saves it as a simple text file. John thinks it is a waste of paper to print the shopping list, he prefers to mail the list to his mobile phone, and then read the shopping list from the mobile phone display while shopping. When John comes home from college and shopping, he opens his laptop and starts up Foodolini. All of the groceries, he has bought, must be registered to the Foodolini storage. John chooses the option "Scan grocery" and holds the bar code of the food item up in front of the built-in web cam of his laptop. John thinks it is a bit fun to watch himself on the screen, before holding up the food item. When all the food items have been registered to the storage, John finds the name of the recipe in the list again, and selects it to view the cooking instructions. John begins to cook the meal while reading the instructions on the screen of his laptop.

3.3.3 Knud Obel Hentze

Knud is twenty-four-years-old and lives in a two-room flat in the centre of Aalborg with his pet spider, Bob. He has a girlfriend, Maria, but they have not yet decided to move in together. They both like to be alone at times and think they may enjoy each other's company more when it is not daily. Knud's work as an apprentice mason requires a strong body, but Knud also wants to look strong and gain muscle. Knud has installed the Foodolini system on his laptop, to supplement his weight training with appropriate foods. An early Friday morning, where Knud sits in his girlfriend's kitchen, he wants to make a decision regarding what to eat for dinner the same evening, at his own place. Knud starts up Foodolini and decides to search for recipes, based primarily on food items in the storage soon to expire. Knud likes the prospect of using food before it goes bad. Foodolini suggests a recipe with pasta, and Knud finds it tempting. As his girlfriend is also joining for dinner, Knud doubles the number of servings. Foodolini checks if all ingredients are available in his storage and notifies him that he needs to buy pasta. Knud is almost certain that he has some rice in his cupboard and opts for rice instead of pasta. To see if he has enough rice, Knud first selects to edit the recipe, by replacing pasta with rice. He asks his girlfriend how much rice she thinks is suitable per serving. He adds the amount of rice per serving and saves the modified recipe with a new name. Foodolini checks again whether any ingredients need to be bought and notifies that all ingredients are available.

3.3.4 Helle Markussen

Helle Markussen is just about to modify the diet related to her user profile in Foodolini. Helle is a thirty-years-old and Ove, her husband, is twenty-nine-years-old. Although, it is five years since she gave birth to a son, she has not yet managed to loose all of the excess weight gained during her pregnancy. Helle has installed Foodolini on her and the husband's common PC. An external webcam, bought in the supermarket, is plugged in. Helle thinks her husband will also benefit from a more concious approach to eating meals, based on how the energy is distributed in it, as well as how many calories it contains. The kitchen in their house is narrow and long. It does not offer room for a dinning table, nor room for a PC, and they do not own a laptop. Consequently the PC is placed on a small table in the corner of the living room, just before the entrance to the kitchen. Until now, Helle and Ove have used the default diet as a guideline for better eating habits, but after a visit to a nutritional counsellor, both Helle and Ove need to adjust their diets. Helle has to lower her daily energy intake by 500 kcal, in order to support her desired weight loss. Helle has been told by the nutritional counsellor that she might want to choose low-fat foods, to reduce the calorie intake. Helle starts up Foodolini, to modifiy the diet related to her profile. She changes the energy distribution in the diet and the daily maximum energy intake. Helle is now sure that she will only be suggested low-fat foods and meals by the system.

3.4 Use Cases

Description of the non-trivial use cases.

3.4.1 Manage Person

Create New Person

When initiating "Create new Person", the system will request the Administrator password (see figure 3.5). If the user enters the correct password, the system will list all existing Persons, showing the names and user names of these. The user then chooses to "Create Person", the system will request personal information for the new Person. Which consists of the following: name, user name, birth date, gender, weight, height and activity factor. Subsequently, the system assigns the default Diet to the Person. The user either selects "Cancel", which terminates the "Create New Person" use case, or they select "Save Person", after which the system saves the new Person and ends.

Edit Person

The details entered about a person may be changed at any time by the user (see figure 3.6). This is done using a simple user interface accessed through the person manager interface - it should prevent the user from entering invalid data. The user starts in the "List Persons" screen. Here, the user chooses which Person to edit. The system the shows previous Person details: name, user name, birth date, gender, weight, height and activity factor. The user enters new information about the Person and chooses "Save Person". Next the system saves the Person.

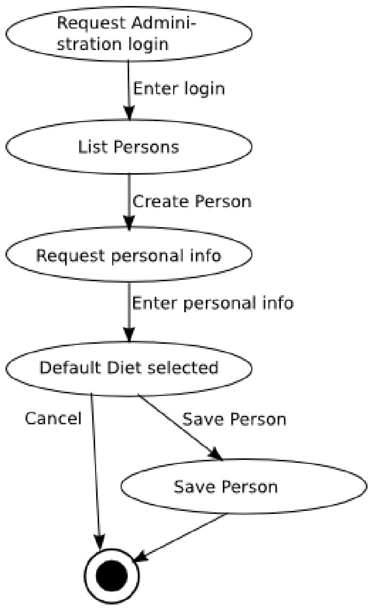


Figure 3.5: Use case diagram for creating a Person.

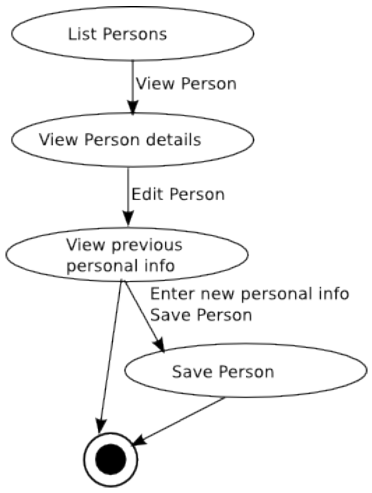


Figure 3.6: Use case diagram for editing a Person.

Delete Person

To delete a Person, the user will enter the Administration-screen and be prompted for a password (see figure 3.7). If the user enters the correct password, the system will show the ”List Persons”-screen, in which the user can see the names of all registered Persons. The user then chooses to view a specific Person. Then the system displays the Person. When the user chooses to ”Delete Person” the system will prompt for a confirmation. If the user chooses to cancel, it will exit. Provided the user chooses to delete, the system deletes the Person and exits.

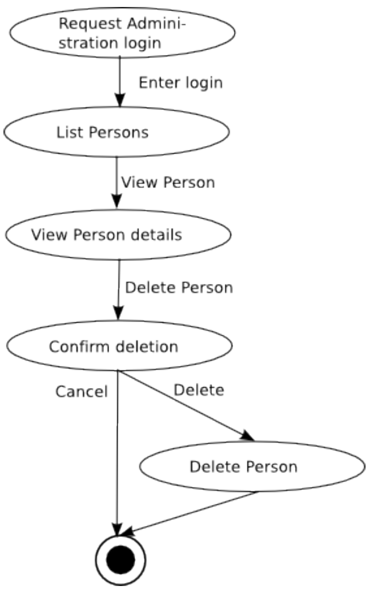


Figure 3.7: Use case diagram for deleting a Person.

3.4.2 Manage Recipe

Create Recipe

A new Recipe can be added to the system at any time using the Cookbook interface. As seen in figure 3.8, the Cookbook displays a list of Recipes, showing the names of all available Recipes. The user chooses to create a new Recipe. The system prompts for details for the new Recipe: name, meal type, categories, Ingredients, amount of the Ingredients, categories, picture, and cooking instructions. When adding Ingredients to the Recipe, the system displays a list of Ingredients. When the user chooses an Ingredient, the system shows that particular Ingredient, which the user then adds to the Recipe (see section 3.4.4). The recipe can only be saved if a title, meal type, directions and at least one category and ingredient have been given. The user can either add more Ingredients, cancel or choose to save the Recipe. If the latter is selected, the system saves the Recipe. When wanting to add a new Ingredient to the Recipe, it is also possible to create a new Ingredient (see 3.4.4).

View Recipe

If the user wishes to view a specific Recipe, they must begin by searching by criteria (see figure 3.9). The system displays the Recipes corresponding to these criteria. The user selects a Recipe. The system prompts for number of servings, that will change the quantity of the ingredients, in the Recipe the user wishes to make. If not all Ingredients are present in the Inventory, the user is informed of which are missing. The system proceeds to show the details for the Recipe. From here it is possible for the user to rate the Recipe (see section 3.4.2), edit the Recipe (see section 3.4.2) and delete the Recipe (see section 3.4.2).

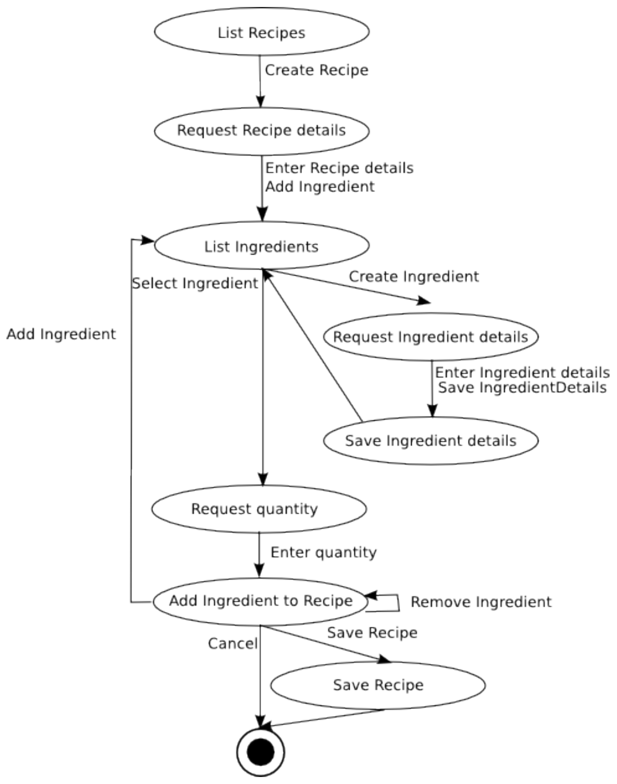


Figure 3.8: Use case diagram for creating a Recipe.

Edit Recipe

A Recipe may be edited at any time (see figure 3.10). This is initiated through the Cookbook interface, choosing ″Edit Recipe″. A list of all of the Recipes in the system will be displayed, and the relevant Recipe may be selected from this list. Editing a Recipe works the same way as when creating a new Recipe (section 3.4.2) - except for the last step. From the state ″Add Ingredient to Recipe″ the user can also choose to ″Save as new″ - which leaves the original Recipe unchanged, and saves the changed version as a new Recipe.

Delete Recipe

A user may choose to completely remove a Recipe from the system (see figure 3.11). This is done by choosing ″Delete Recipe″ in the Cookbook interface. A list of the names of all of the Recipes in the system will appear and the user can hereby select the desired Recipe for removal. The system will ask for confirmation upon removal.

Rate Recipe

In the case that the user wants to rate a Recipe, as seen in figure 3.12, the user must perform the same initial actions as when viewing a Recipe (see section 3.4.2). The user then gives the Recipe a new rating. Afterward the average rating from all of the users is re-calculated and the rating is saved.

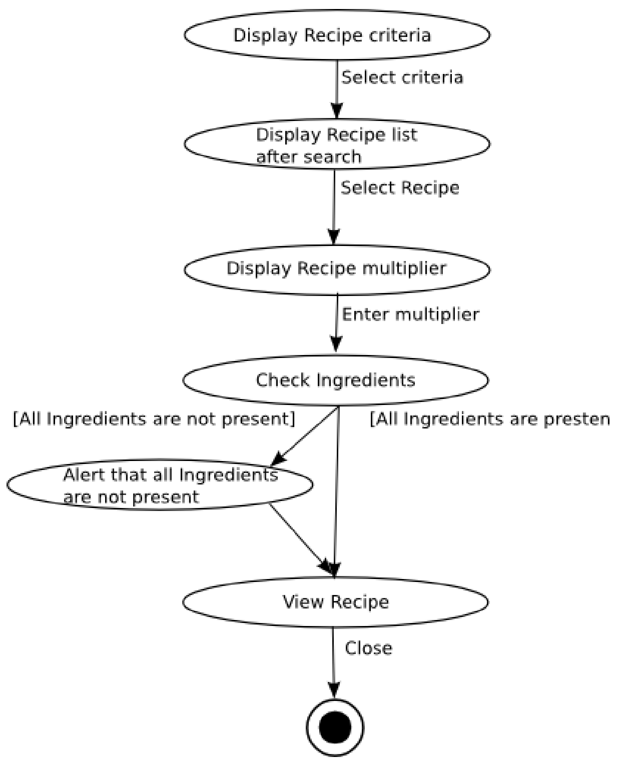

Display Recipe criteria
Select criteria
Display Recipe list after search
Select Recipe
Display Recipe multiplier
Enter multiplier
Check Ingredients
[All Ingredients are not present]
[All Ingredients are presten
Alert that all Ingredients are not present
View Recipe
Close

Figure 3.9: Use case diagram for viewing a Recipe.

Create Meal from Recipe

When a user wants to cook a meal from a Recipe, they must begin by finding a Recipe (see figure 3.13). The system lets the user search by entering their search criteria, e.g. dinner, beef or potatoes. The system then displays the results from the search, sorted by the criteria given. Then the user selects a Recipe. The system asks the user for the number of servings of the Recipe, they wish to make. The system then displays the Recipe details: name, Ingredients, amount of the Ingredients, categories, picture, and cooking instructions. Meanwhile, the system checks if all of the necessary Ingredients are present in the Inventory. If they are not, the system informs the user that certain Ingredients are missing. The user can then either choose to store the meal as a Food-Item, or to consume it, if all Ingredients are present. If the user chooses to consume it, the system asks which Persons have consumed it. The user then fills in the user names of the relevant users. The system registers the amount consumed by each Person. If the user chooses to store it, the Recipe is saved as a FoodItem.

3.4.3 Manage FoodItem

Register FoodItem

When the user wants to add a FoodItem to the Storage, the system must have the scanner ready (see figure 3.14). The user then scans the bar code of the FoodItem, which is then processed by the system. The system either recognises the bar code or

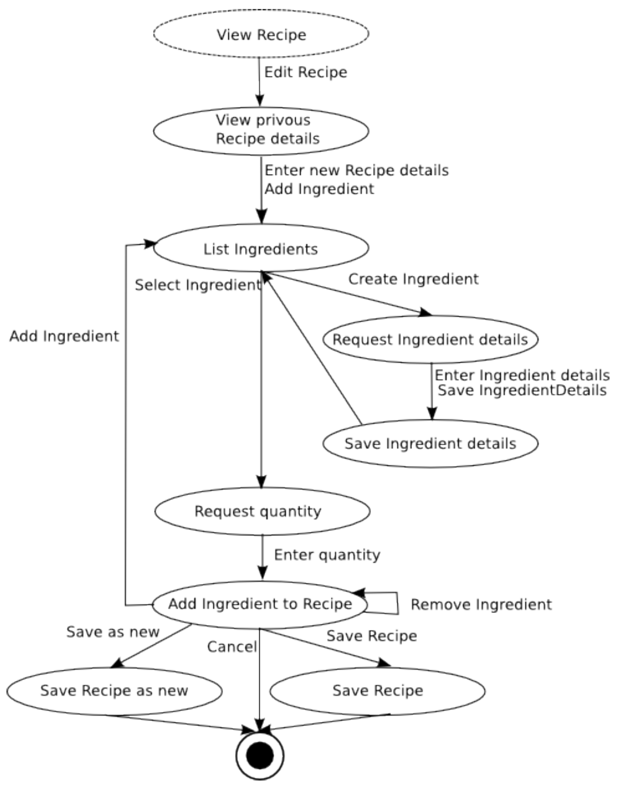


Figure 3.10: Use case diagram for editing a Recipe.

not. In case of the former, the system will display the information associated with this specific Ingredient: name, amount, category, and expiration date and the ability to edit the Ingredient (see section 3.4.4). The user then enters the missing information (for example, expiration date) and saves the FoodItem. The system adds the FoodItem to the list of registered FoodItems in the Inventory. If the system does not recognise the bar code, it will display a list of Ingredients to choose from. The user can either view a specific Ingredient, and then select it, or create a new Ingredient (see section 3.4.4). In both cases the user ends up at the screen for entering information for this FoodItem, which is then added to Storage.

View FoodItem

If the user wishes to view a FoodItem, they will choose it from the list of FoodItems. The system will then display the details of the FoodItem: name, amount, category, and expiration date.

Edit FoodItem

If the user wants to edit a FoodItem, for example, if a wrong expiration date has been entered, the system will show the list of FoodItems in Inventory (see figure 3.15). The user chooses to view the FoodItem (see section 3.4.3). The user chooses to edit the FoodItem. The system displays the previous details of the FoodItem in editable fields.

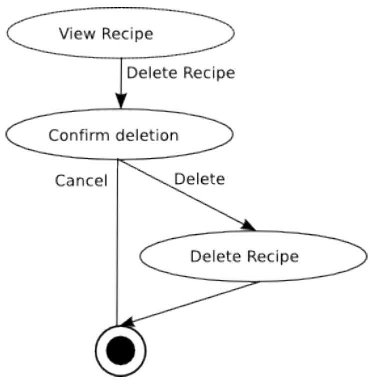


Figure 3.11: Use case diagram for deleting a Recipe.

The user enters new information. The system shows the list of all Ingredients, from which the user can either view an Ingredient (see section 3.4.4) or create a new Ingredient, which can be selected (see section 3.4.4). The system saves the new details.

Delete FoodItem

The user chooses to view a FoodItem in the list of FoodItems in the Inventory. The system displays the FoodItem, and the user chooses to delete it. The systems prompts for confirmation. If the user confirms deletion, the system deletes the FoodItem.

3.4.4 Manage Ingredient

Create new Ingredient

If an Ingredient is not already in the system, it can be added (see figure 3.16). The system will prompt for the name of the new Ingredient, the category of the Ingredient, shelf life and expiration date, and its nutritional facts - e.g. grams of fat per 100 grams. Then the Ingredient is saved.

Add Ingredient

An Ingredient can be added to both a FoodItem, ShoppingList, and a Recipe. The procedure is the same, so they shall all be generalised in this use case, as seen on figure 3.17.

The Ingredient is selected from a list of Ingredients and then added.

View Ingredient

To view an Ingredient, the user first selects an Ingredient from a list of Ingredients and then selects ”View Ingredient details” (see figure 3.18). The details of the Ingredient are then shown: name, amount, category, and expiration date.

Edit Ingredient

To edit an Ingredient, first follow the View Ingredient use case (see section 3.4.4). While viewing an Ingredient, the user selects "Edit Ingredient", and is presented with the Ingredient data, and is in turn allowed to edit the information: category, shelf life and expiration date, as well as the nutritional facts of the Ingredient. The changes are then saved.

Delete Ingredient

To delete an ingredient, first follow the "View Ingredient" use case (figure 3.4.4). While viewing an ingredient the user selects "Delete Ingredient" and is required to confirm deletion or cancel.

3.4.5 Manage ShoppingList

View ShoppingList

To view the ShoppingList, the user has to expand the ShoppingList sidebar. The ShoppingList then displays the Ingredients added to the ShoppingList and how much needs to be bought. It is possible to clear the ShoppingList (see section 3.4.5) and delete an item from the ShoppingList (see section 3.4.5).

Add Recipe to ShoppingList

To add a Recipe to the shopping list, the Recipe must first be selected by searching with matching criteria, and then selecting the Recipe from the search results (see figure 3.19). Upon viewing the Recipe, the user must specify the number of servings they wish to make. The system will then check the Storage to see if the Ingredients are present. If all Ingredients are not present in their specified quantity the system will alert the user. While viewing the Recipe, the user can then add the Recipe to the ShoppingList and the system will add the Ingredients, including their respective quantities, and display the ShoppingList.

Clear ShoppingList

To clear the ShoppingList, the user must expand the ShoppingList sidebar and select Clear ShoppingList.

Remove Item from ShoppingList

To remove an item from the ShoppingList the user must first expand the ShoppingList sidebar. The user can then mark the item for deletion.

3.4.6 Manage Diet

Create Diet

As seen in figure 3.20 a user begins in the ”List Diets” screen. Here they choose to ”Create new diet”. The system prompts the user for nutritional information for the new Diet: name of the Diet, percent of fat, protein and carboydrates, maximum energy intake per day and a description. The user enters information about the Diet and clicks ”Save”. The system saves the Diet.

View Diet

When a user wishes to view the specifics of a single Diet, they choose to view a specific Diet from the ”List Diets” screen. The system then displays the Diet. Here the user can choose a new Diet (see section 3.4.6) or close the view, which terminates this use case.

Choose Diet

When the user wants to choose another Diet than the one they are already associated with, they view the details (name, energy distribution, maximum caloric intake per day and description) of a specific Diet (see section 3.4.6) in the Diet list. Here the user can either select the Diet, they are viewing, edit the Diet (see section 3.4.6), or create a new Diet (see section 3.4.6 for creating a new Diet). Then the system replaces the previously selected Diet with the chosen one.

Edit Diet

As seen in 3.21, when the user wishes to edit a Diet, they will choose to edit a Diet from the ”List Diets” screen. The system will display the previous Diet details: name, energy distribution, maximum caloric intake per day and description. The user enters new details and chooses to save the Diet. The system saves the details.

Delete Diet

The user chooses to delete a Diet from the ”List Diets” screen or View Diet (see section 3.4.6). The system prompts the user to choose ”Delete” or ”Cancel”. If the user chooses to ”Cancel” the system terminates the ”Delete Diet”-action. If the user chooses to ”Delete” the system deletes the Diet.

3.4.7 Manage Exercise

Register Exercise

When the user wishes to register an Exercise, they are presented with a list of sports activities to choose from. If the desired SportsActivity is not available, they can create a new one following the use case for creating a SportsActivity (see section 3.4.8). Once the SportsActivity is chosen, the user must input the duration and the date of which the Exercise was done. The system will then apply the user input and calculate energy expenditure, and save the Exercise. The user can also choose to cancel, discarding any Exercise data.

View Exercise

To view an Exercise, the user must select the Exercise from a list and select "View Exercise". The details for the Exercise will be displayed: SportsActivity, date and duration. From here it is possible to choose to edit the Exercise (see section 3.4.7).

Edit Exercise

To edit an Exercise, the user must view the Exercise (see section 3.4.7), and select "Edit Exercise" (see figure 3.22). The user is then presented with a list of SportsActivities, the previously chosen SportsActivity already being highlighted. The user can then choose to continue with the same SportsActivity, choose a different SportsActivity, or create a new SportsActivity (see section 3.4.8). Once the SportsActivity has been chosen, the user can then input new exercise data (i.e. duration, and date), and submit the data for a new energy expenditure calculation. As with creating an exercise, the user can choose to save the modifications or discard them.

Delete Exercise

Begin with viewing the Exercise (see section 3.4.7) and select "Delete Exercise". A confirmation is required to delete the Exercise. The system then deletes or cancels the deletion.

3.4.8 Manage SportsActivity

Create SportsActivity

To create a new sports activity, the user opens the screen that lists the sports activities and selects "Create new Sports Activity". The system will then request the name and MET value of the SportsActivity, which the user must enter so the SportsActivity can be saved. The user can also cancel.

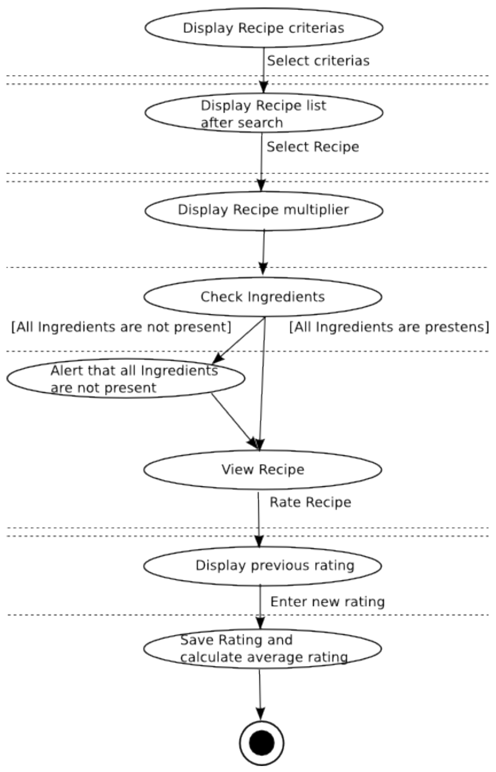

Display Recipe criterias
Select criterias
Display Recipe list after search
Select Recipe
Display Recipe multiplier
Check Ingredients
[All Ingredients are not present]
[All Ingredients are prestens]
Alert that all Ingredients are not present
View Recipe
Rate Recipe
Display previous rating
Enter new rating
Save Rating and calculate average rating

Figure 3.12: Use case diagram for rating a Recipe.

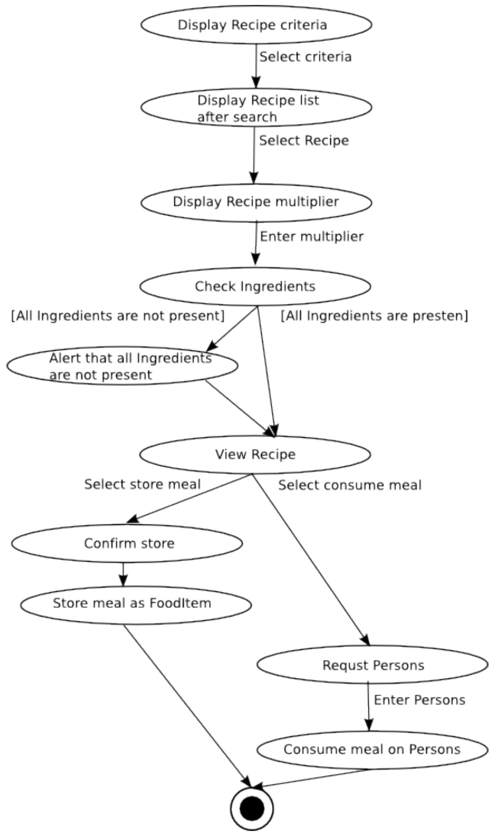
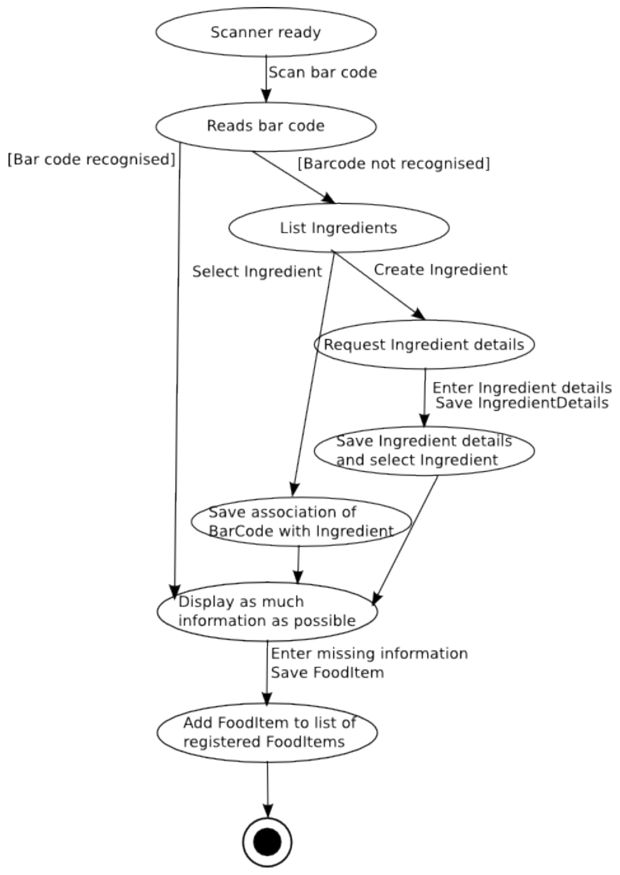


Figure 3.13: Use case diagram for creating a meal from a Recipe.



Scanner ready
Scan bar code
Reads bar code
[Bar code recognised]
[Barcode not recognised]
List Ingredients
Select Ingredient
Create Ingredient
Request Ingredient details
Enter Ingredient details
Save IngredientDetails
Save Ingredient details and select Ingredient
Save association of BarCode with Ingredient
Display as much information as possible
Enter missing information
Save FoodItem
Add FoodItem to list of registered FoodItems

Figure 3.14: Use case diagram for registering a FoodItem.

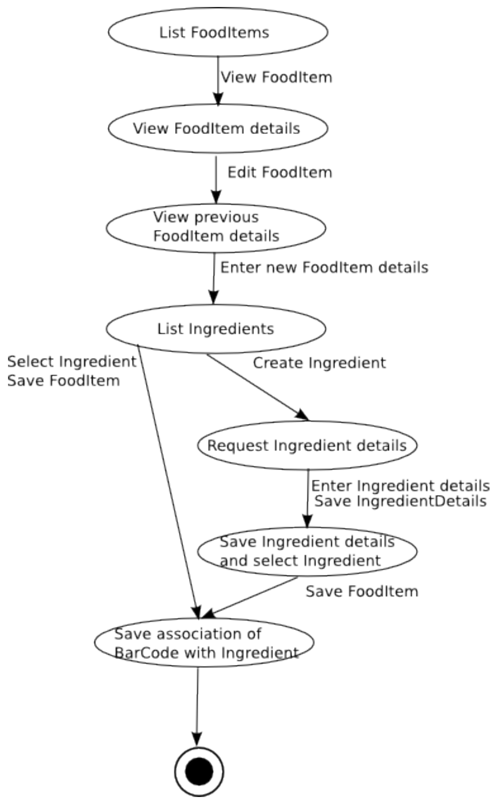

List FoodItems
View FoodItem
View FoodItem details
Edit FoodItem
View previous FoodItem details
Enter new FoodItem details
List Ingredients
Select Ingredient Save FoodItem
Create Ingredient
Request Ingredient details
Enter Ingredient details Save IngredientDetails
Save Ingredient details and select Ingredient
Save FoodItem
Save association of BarCode with Ingredient

Figure 3.15: Use case diagram for editing a FoodItem.

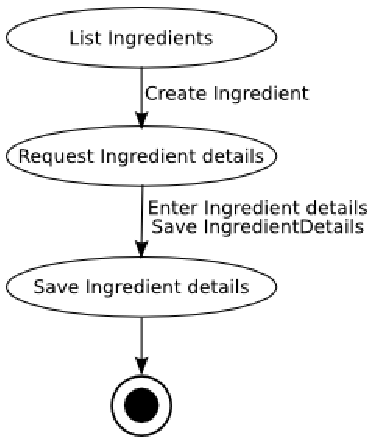

List Ingredients
Create Ingredient
Request Ingredient details
Enter Ingredient details Save IngredientDetails
Save Ingredient details

Figure 3.16: Use case diagram for creating a new Ingredient.

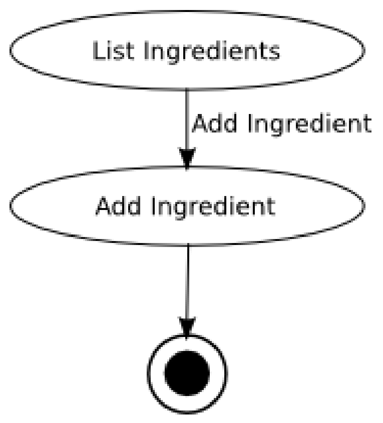


Figure 3.17: Use case diagram for adding an Ingredient to FoodItem, ShoppingList, or Recipe.

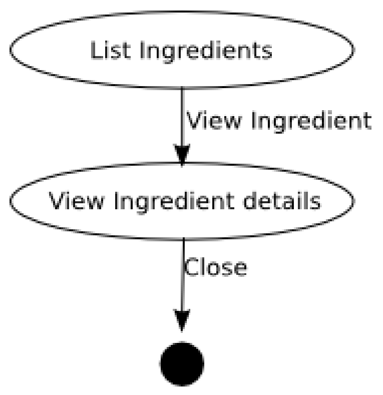


Figure 3.18: Use case diagram for viewing an Ingredient.

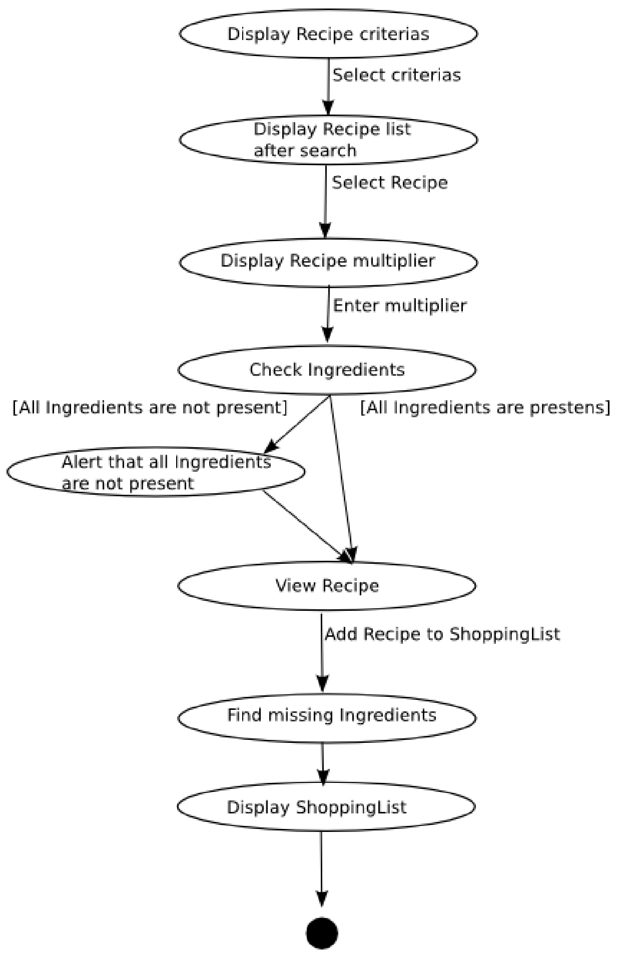

Display Recipe criterias
Select criterias
Display Recipe list after search
Select Recipe
Display Recipe multiplier
Enter multiplier
Check Ingredients
[All Ingredients are not present]
[All Ingredients are prestens]
Alert that all Ingredients are not present
View Recipe
Add Recipe to ShoppingList
Find missing Ingredients
Display ShoppingList

Figure 3.19: Use case diagram for adding a Recipe to the ShopppingList.

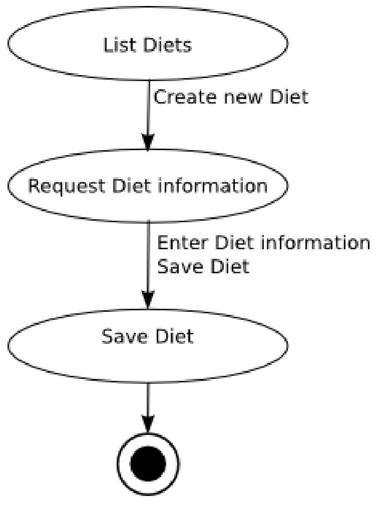


Figure 3.20: Use case diagram for creating a new Diet.

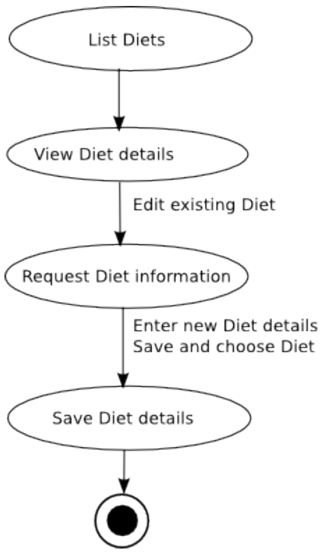


Figure 3.21: Use case diagram for editing a Diet.

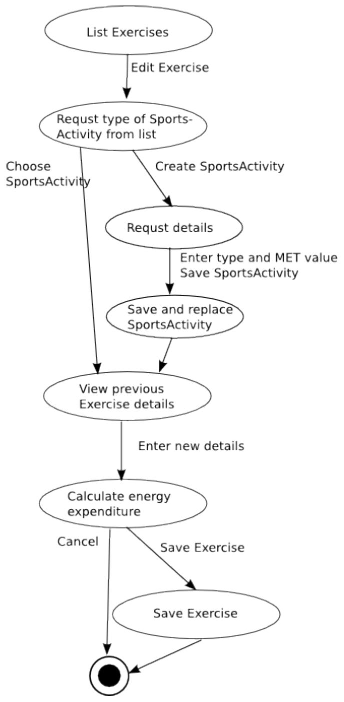


Figure 3.22: Use case diagram for editing an Exercise

View SportsActivity

To view a SportsActivity, it is found from the list of sports activities and "View Sports Activity" is selected. From here the user can edit (see section 3.4.8) or delete (see section 3.4.8) the SportsActivity.

Edit SportsActivity

To edit a SportsActivity, the user must first view the SportsActivity details (see section 3.4.8). From there the user can modify the name and MET value on the SportsActivity, and save the changes.

Delete SportsActivity

To delete a sports activity, the user must view the SportsActivity (see section 3.4.8), and select "Delete SportsActivity". The user must then confirm the deletion of the SportsActivity before it is deleted by the system.

3.5 Functionality

The following is a list of functions Foodolini will include. The purpose of making this list is to establish the functionality of the application.

Functions are split into three subgroups: Storage management, Recipe management and Administration. Functions with a complexity of *complex* or above will be further specified with a hierarchy of functions; the dash signaling where in the hierarchy the specific function is positioned.

Many functions are intuitively named, though a few particular functions will be described in more detail after the table.

Bar Code Scanning

It is important that purchased groceries can be registered efficiently, otherwise the system would easily become a burden. In order to easily register many groceries Foodolini must be able to recognize bar codes using commodity hardware, i.e. a webcam. This is a especially complex feature, as it involves capturing, scanning and displaying a video stream. In addition, it is also likely that a background thread must be used to pull images from the capture device, with the aim of ensuring a responsive UI. Whilst it may be possible to interface a third party library to do most of the work, e.g. scan images for bar codes, no such library is readily available for .Net/Mono. And it would still be necessary to handle thread synchronization, as well as image drawing.

Calculate New FoodItem

The purpose of this function is to create a FoodItem that is the product of a Recipe, and therefore has no bar code nor any database entry associated with it. Nutritional information will be calculated by Foodolini, based on the Ingredients in the Recipe and

Function	Complexity	Type
Register FoodItem	Complex	Update
-Bar code scanning -Edit grocery info -Add to upload list	-Very complex -Simple -Simple	
Edit FoodItem	Simple	Update
Delete FoodItem	Simple	Update
Calculate new FoodItem	Medium	Update
-Calculate new Ingredient	-Simple	
Consume FoodItem	Simple	Update
Store FoodItem	Simple	Update
Calculate calories for meal	Medium	Update
FoodItem expired	Simple	Signal
Create Ingredient	Simple	Update
Edit Ingredient	Simple	Update
Delete Ingredient	Simple	Update
Add to ShoppingList	Simple	Update
Remove from ShoppingList	Simple	Update
Print ShoppingList	Complex	read

Table 3.2: Storage management

saved as a new ingredient. The FoodItem will, once created, thereafter be handled as a normal FoodItem.

Consume FoodItem

This function symbolises the act of eating a food item. It will make the Recipe into a FoodItem, and list it as consumed by the users that participated in consuming the FoodItem.

Store FoodItem

This function saves the FoodItem, created from a Recipe, as in section 3.5, in Storage.

Add Recipe to ShoppingList

When adding a Recipe to the ShoppingList, all its Ingredients are then added to the ShoppingList.

Print Shoppinglist

Printing the shoppinglist will involve formatting of the information in the shopping list into a layout, which can be printed onto a standard sheet of A4 paper. Moreover, Foodolini will have to access the printers available to the PC and request them to print the list.

Calculate Calories for Meal

This function, as the name implies, calculates the calories of a meal, in order to allow improved suggestion of possible Recipes by Foodolini. It differs from the information that *Consume FoodItem* calculates, since it is used prior to even suggesting a Recipe. This is to help determinine if the Recipe is a valid choice, based on the Diet.

Function	Complexity	Type
Create Recipe	Simple	Update
Edit Recipe	Simpe	Update
Delete Recipe	Simple	Update
Search Recipes	Very Complex	Read
-Search for keywords -Search for meal type -Prioritise after expiration date -Diet citeria –Nutritional demands -Rating	-Simple -Simple –Simple -Medium –Simple -Simple	

Table 3.3: Recipe management

Search Recipes

Searching for a Recipe is one of the central functions in Foodolini. The objective of this function is to suggest a recipe which fits the user's search criteria. Criteria can include expiration date, rating, diet or keywords. Each can be searched by if needed. If any keywords are entered, only Recipes containing these particular keywords will be suggested. The list will be sorted so that the oldest FoodItems, with the highest rating and fitting the Diet best, will be in the top of the list.

Function	Complexity	Type
Create Person	Simple	Update
Edit Person	Simple	Update
Delete Person	Simple	Update
Create Diet	Simple	Update
Edit Diet	Simple	Update
Delete Diet	Simple	Update
Change Diet	Simple	Update
Create SportsActivity	Simple	Update
Edit SportsActivity	Simple	Update
Delete SportsActivity	Simple	Update
Register Exercise	Medium	Update
Edit Exercise	Medium	Update
Delete Exercise	Simple	Update

Table 3.4: Administration

3.6 User Interface

The goals of use, the conceptual model, the interaction forms and the general interaction model are discussed in this section.

3.6.1 Goals for Use

The goals for use includes two perspectives; the usability goals which are specific usability criteria, and the usability experience goals which are goals concerning the users overall experience when utilising the system. Table 3.5 illustrates the goals for usability and the usability experience for Foodolini. Below the table a short description of each goal is given.

		Very important	Important	Less important	Irrelevant
Usability	Effectiveness	✔			
	Efficiency	✔			
	Safety			✔	
	Utility		✔		
	Learnability			✔	
	Memorability			✔	
Experience	Satisfying			✔	
	Enjoyable			✔	
	Fun				✔
	Entertaining				✔
	Helpful	✔			
	Motivating		✔		
	Aesthetically pleasing		✔		
	Supportive of creativity				✔
	Rewarding				✔
	Emotionally fulfilling				✔

Table 3.5: Goals for use

Usability Goals:

Effectiveness: Effectiveness is an expression of how well the system solves the tasks it is supposed to. This is rated as very important.

Efficiency: Efficiency concerns how well the system supports the user when carrying out tasks. This is rated as very important as the system is to be used many times a day and some times in stressed situations. The user must not have to do unnecessary actions to obtain their goals.

Safety: Safety concerns the consequences of the system not working correctly. This is rated as less important since the system does not hold any critical information or information hard to recreate.

Utility: Utility has to do with the functions available to the user. This is rated as important. For example the system must provide the option of changing the number of servings in a recipe.

Learnability: The learnability of a system is the time the user has to spend learning the systems functionality just by trying out. This is rated as less important. Some features of the system relies on the user possessing special knowledge in advance, thus willing to invest some time getting to know the system.

Memorability: Memorability is about how well the user will remember how the system works, once learned. This is rated as less important as the system is to be used every day.

Usability Experience Goals:

Satisfying: That the system is to be satisfying to use is rated as less important. Foodolini should just be part of the daily routine and not require any further attention.

Enjoyable: Enjoyable is similar to satisfying. It is not relevant for Foodolini.

Fun: Foodolini is not supposed to entertain the user, and is therefore irrelevant.

Entertaining: See Fun.

Helpful: Foodolini is meant to be helpful. This has therefore been rated as Very Important.

Motivating: Motivating is rated as important, as the user should feel encouraged to use the system.

Aesthetically pleasing: Aesthetically pleasing is rated to be important as the system is aimed for a wide range of users.

Supportive of creativity: Supporting creativity is irrelevant to the system. Foodolini's primary concern is assisting the user with managing the kitchen contents

Rewarding: Because Foodolini is a management system and not, for instance, a game, Rewarding has been considered irrelevant.

Emotionally fulfilling: It is not relevant that Foodolini is emotionally fulfilling.

3.6.2 Conceptual Model

The task of the system is to manage the contents of a Storage. The user will have to update information if any changes are made in the Storage. If a new FoodItem needs to be registered, the user will first have to scan the bar code of that specific FoodItem. The user may need to add additional information through the user interface using the mouse and keyboard. When FoodItems have been used, the user will again have to enter information regarding how much was consumed, using the mouse and keyboard. The system is then able to display the contents of the Storage in the user interface, helping the user to determine if something is missing or in low supply. If the user decides to do this, the system can also help the user selecting different Recipes to cook from the Cookbook. When doing this, the user can input search criteria based on what he would like to eat, and the system will return Recipes, which the user can browse.

When entering additional information about the users height, weight and type of diet, the user will use a combination of the mouse and keyboard. The system can then help the user by selecting Recipes which are consistent with the diet and display them, so that the user can choose between these. The system is also able to keep track of the user's Exercises and adjust the search for Recipes based on changes in Diet, due to performing exercises. Since doing exercises will change the dietary needs of the user, primarily in terms of calories. This requires the user to input every Exercise performed into the system by filling out a form.

3.6.3 Interaction Forms

Foodolini will feature a graphical user interface. Icons and menus will represent the functions available to the user. The system will be build with the WIMP interaction form in mind. Yet, some functions, such as adding a food item, require the user to input specific data, which means that a schematic form will have to be used.

3.6.4 General Interaction Model

The purpose of this section is to identify the interaction spaces in the general interaction model. The interaction spaces will be identified through the use cases discussed in section 3.4 and divided into subtasks. In the end these will be collected in the general interaction model.

A naming convention is used to identify the purpose of an interaction space. A viewer just shows the information about the item, while a browser lists these items with the most important information. The browsers always contain a button with which the user can add a new item. In the viewer, the user can edit and delete, among other actions, with regard to a particular item. An editor is an interaction space which can be used to enter information. When used for creating a new item, the fields will be empty, and if used to edit the information, the fields will be filled with the previous information.

Ingredient

The task model for creating a new Ingredient can be seen in figure 3.23. When creating a new Ingredient the user will go through two interaction spaces. The first is the IngredientBrowser, which lists all Ingredients. The next is the IngredientEditor, which contains input fields for information about an Ingredient. Only, the IngredientEditor is in the subtask, Create Ingredient, since the browser merely is a way of gaining access for creating a new Ingredient.

Adding an Ingredient to a FoodItem, Recipe or ShoppingList uses the Ingredient-Browser that lists the Ingredients, and a QuantitySelector, where the amount to add can be selected.

For deleting and editing an Ingredient, the Ingredient must first be viewed, as in figure A.1. When deleting an Ingredient a ConfirmDialog will ask the user to confirm the deletion. When editing an Ingredient the IngredientEditor will be used. The task models for delete (figure A.3) and edit (figure A.2) can be seen in appendix A.

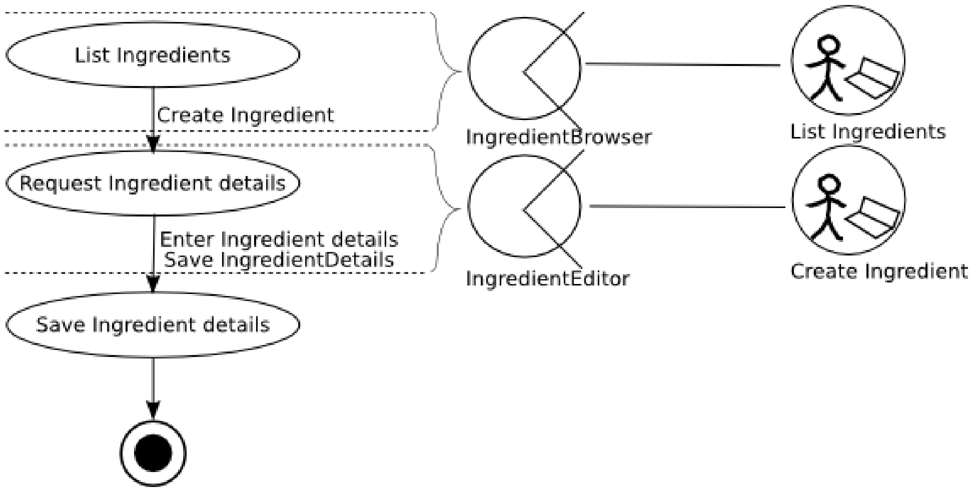


Figure 3.23: Task model for creating a new Ingredient

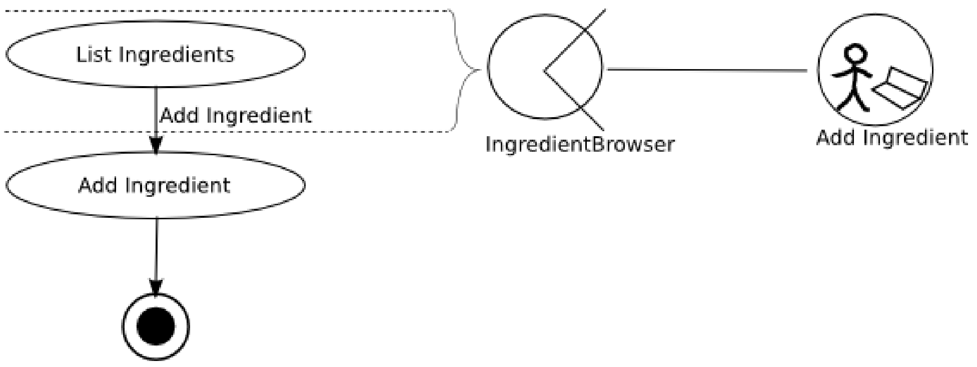


Figure 3.24: Task model for adding an Ingredient to FoodItem, Recipe or ShoppingList

Recipe

Creating a new Recipe involves a RecipeBrowser, RecipeEditor, IngredientBrowser, QuantitySelector and can be seen in figure 3.25. The RecipeBrowser lists Recipes and the RecipeEditor is used to enter the details. The IngredientBrowser is used to select the needed Ingredients, and the QuantitySelector to select the quantity of each Ingredient. From the IngredientBrowser a new Ingredient can be created using the IngredientEditor.

Creating a meal from a Recipe can be seen in figure 3.26 and involves first viewing the Recipe. This involves a CriteriaSelector, a RecipeBrowser, a MultiplierSelector and a RecipeViewer. The CriteriaSelector is used to select the criteria for searching and a RecipeBrowser that shows the search results. The MultiplierSelector changes the number of servings and the RecipeViewer shows the details of the Ingredient and notifies the user if any Ingredients are missing. Creating a meal from a Recipe can

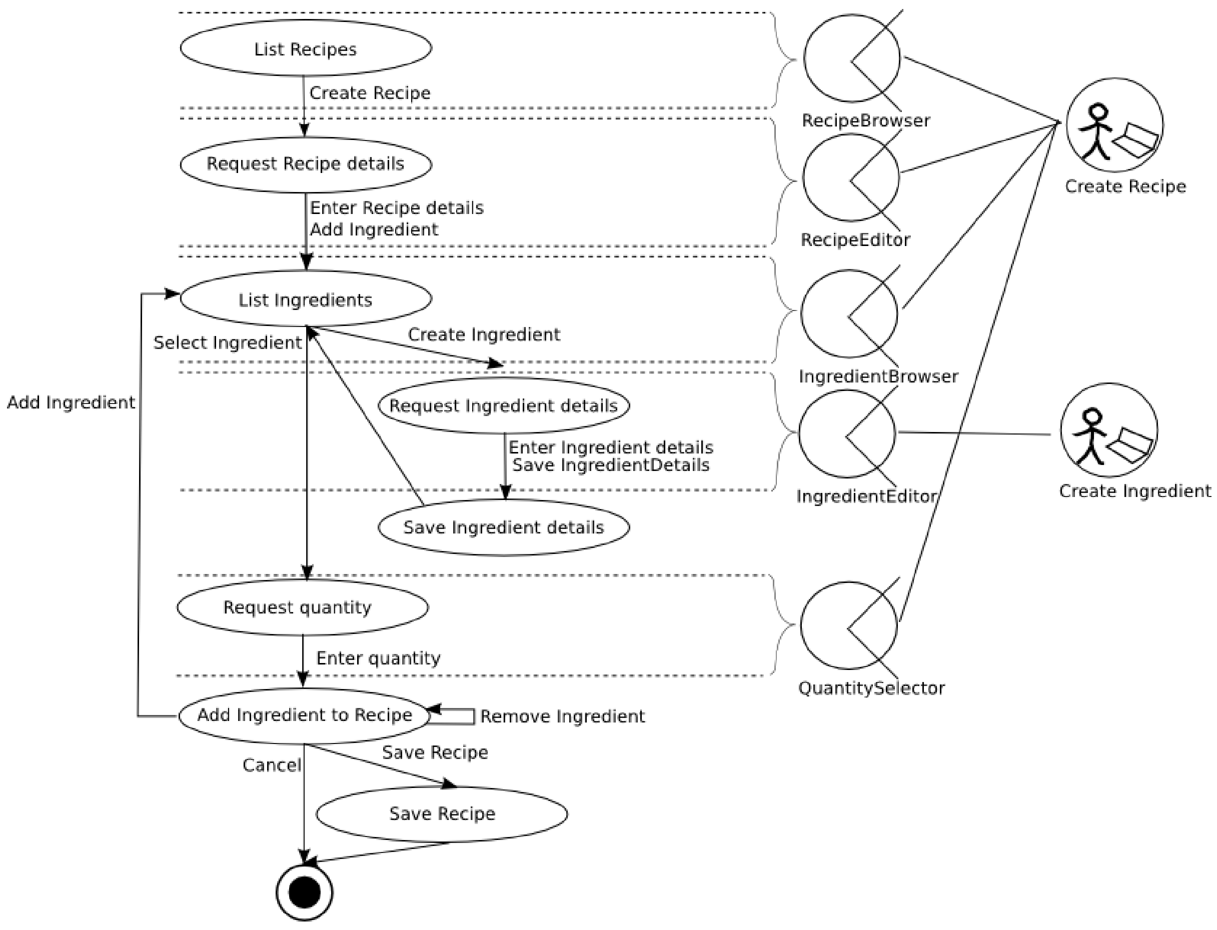


Figure 3.25: Task model for creating a new Recipe

either be storing it as a FoodItem, which uses a ConfirmDialog, to confirm the action, or consuming it as several FoodItems, by using a PersonSelector, where the people, who attended the meal, as well as their individual consumption, can be entered.

Viewing a Recipe applies the interaction spaces seen in figure A.4 in the appendix, which are the same as the beginning of figure 3.26, for creating a meal from a recipe. These are the CriteriaSelector, RecipeBrowser, MultiplierSelector and RecipeViewer. Editing a Recipe uses the same interactions spaces as creating a Recipe, apart from the Recipe needing to be viewed first. Figure A.5 in the appendix shows the task model for editing, not including viewing. Rating a Recipe uses the RecipeRater, after viewing the Recipe. In the RecipeRater a new rating can be given, and the average rating is calculated. Deleting a Recipe also uses the interaction spaces from viewing a Recipe, and a ConfirmDialog to confirm the deletion. The same applies for adding a Recipe to the ShoppingList, only the ShoppingListViewer, is used after adding the Recipe instead of the ConfirmDialog. Rate (figure A.6), Delete (figure A.8) and Add to ShoppingList (figure A.7) can be seen in the appendix.

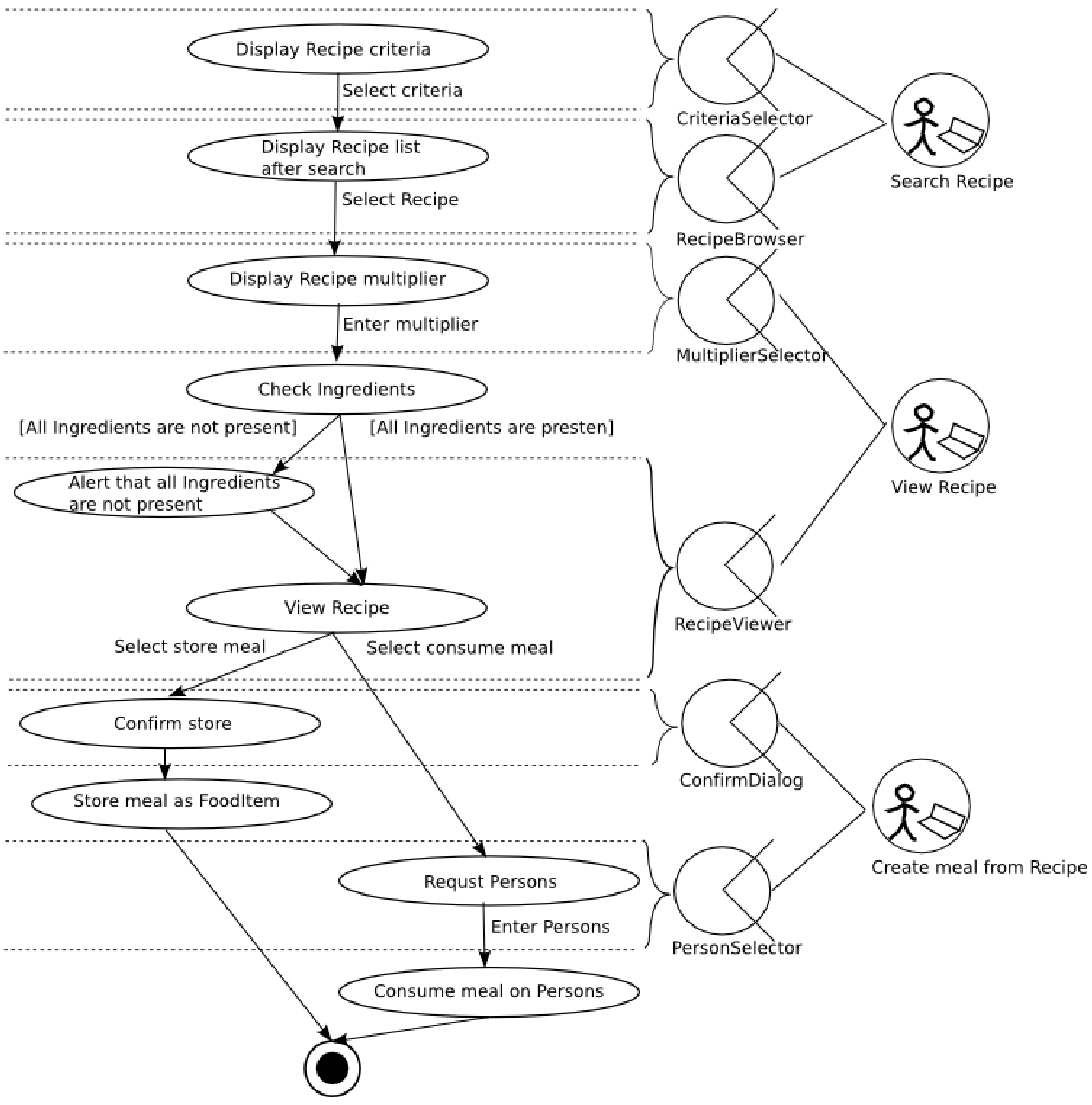


Figure 3.26: Task model for creating a meal from a Recipe

ShoppingList

The task model for viewing the ShoppingList can be seen in figure A.9 in the appendix. Viewing the ShoppingList involves the interaction space ShoppingListViewer. The ShoppingListViewer displays a list of FoodItems to the user.

The task model for removing an item from the ShoppingList can be seen in figure A.10 in the appendix. Remove item from ShoppingList involves two interaction spaces: ShoppingListViewer and ConfirmDialog. The ConfirmDialog prompts the user for confirmation, before the system removes the FoodItem from the ShoppingList.

The task model for clearing the ShoppingList can be seen in figure 3.27. Clear ShoppingList involves two interaction spaces: ShoppingListViewer and ConfirmDialog. The ConfirmDialog prompts the user for confirmation, before the system removes all of the FoodItems from the ShoppingList.

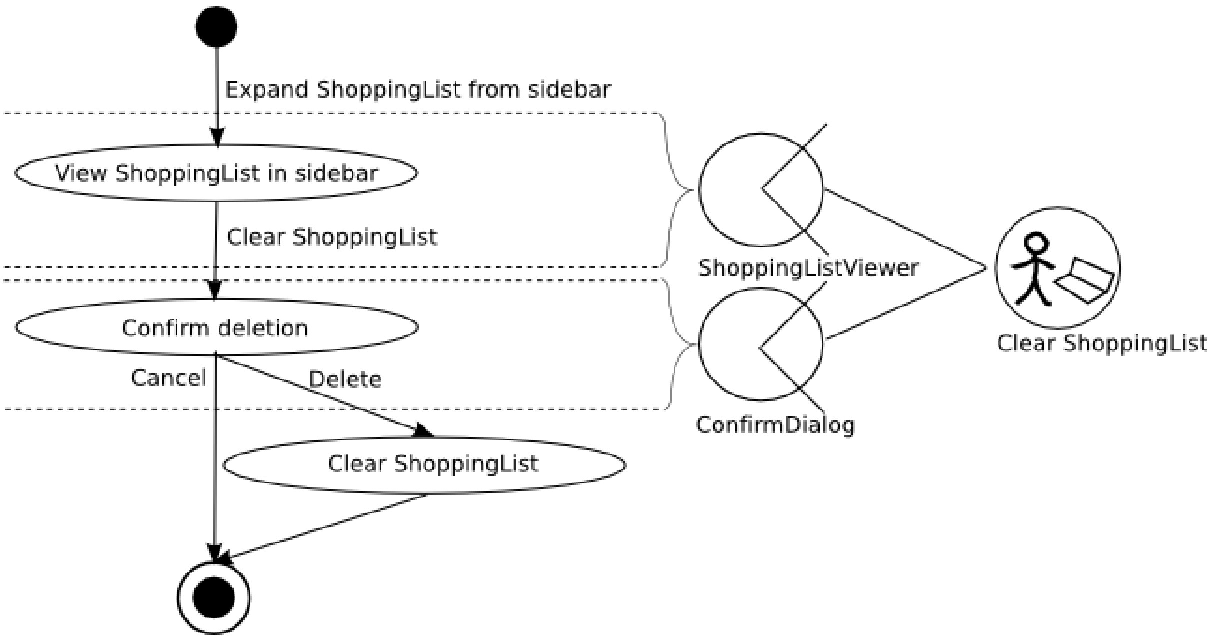


Figure 3.27: Task model for clearing the ShoppingList

FoodItem

The task model for viewing a FoodItem can be seen in figure A.11 in the appendix.

Viewing a FoodItem involves the interaction spaces: FoodItemBrowser and FoodItemViewer. In the FoodItemBrowser the system displays FoodItems for the user to select or view. In the FoodItemViewer the system displays details about a specific FoodItem for the user to view.

The task model for registering a new FoodItem can be seen in figure 3.28.

Registering a FoodItem involves the interaction spaces: BarCodeScanner, IngredientBrowser, FoodItemBrowser and FoodItemEditor. The BarCodeScanner displays a picture from the camera, where the user can scan a bar code. The IngredientBrowser displays ingredients for the user to select. The IngredientEditor allows the user to edit the details about a specific Ingredient. The FoodItemBrowser displays FoodItems for the user to select or view. The FoodItemEditor allows the user to edit details about a specific FoodItem and enter the quantity.

The task model for consuming a FoodItem can be seen in figure 3.29.

Consuming a FoodItem involves the three interaction spaces: FoodItemBrowser, FoodItemViewer and QuantyitySelector. The FoodItemBrowser allows the user to select a FoodItem. The FoodItemViewer displays details about a specific FoodItem. The QuantitySelector allows the user to select how much of the FoodItem to consume.

The task model for editing a FoodItem can be seen in figure A.12 in the appendix.

Editing a FoodItem uses the same interaction spaces as registering a FoodItem, except it makes use of the FoodItemBrowser and the FoodItemViewer, instead of the

BarCodeScanner. In the FoodItemBrowser the user can select the FoodItem to edit. In the FoodItemEditor the FoodItem details are displayed for the user to edit.

The task model for deleting a FoodItem can be seen in figure A.13 in the appendix.

Deleting a FoodItem includes the FoodItemBrowser, FoodItemViewer and the ConfirmDialog interaction spaces. The ConfirmDialog prompts the user for confirmation before the system delete the FoodItem.

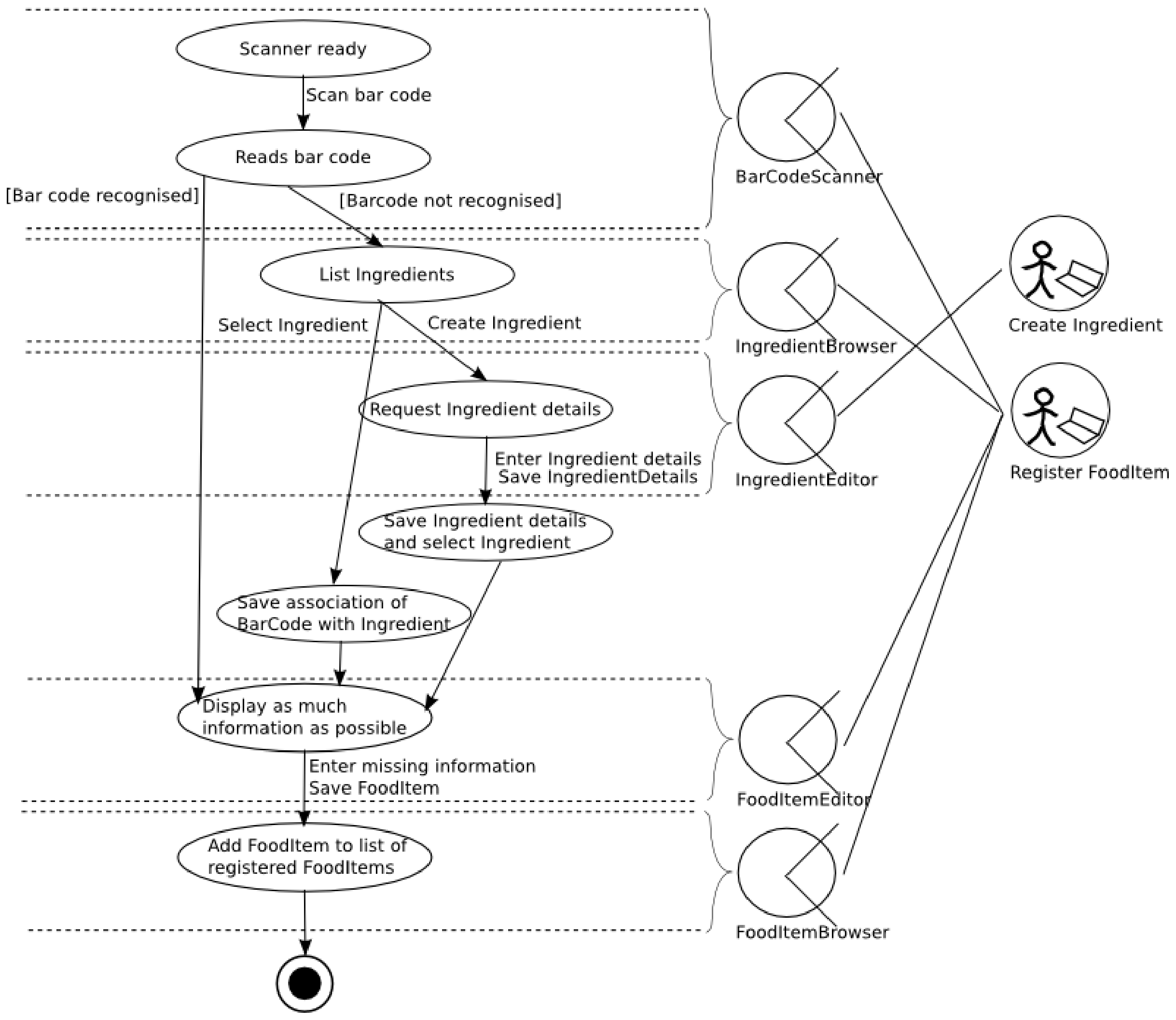


Figure 3.28: Task model for registering a new FoodItem

Person

The task model for creating a new Person can be seen in figure 3.30.

Creating a new Person involves the three interaction spaces: PersonBrowser, PersonEditor and Login. Login demands the user to log in as Administrator, where the user can enter the password. PersonBrowser shows all users registered in the system. The PersonEditor requests the user to enter the information for the new Person and pick save or cancel.

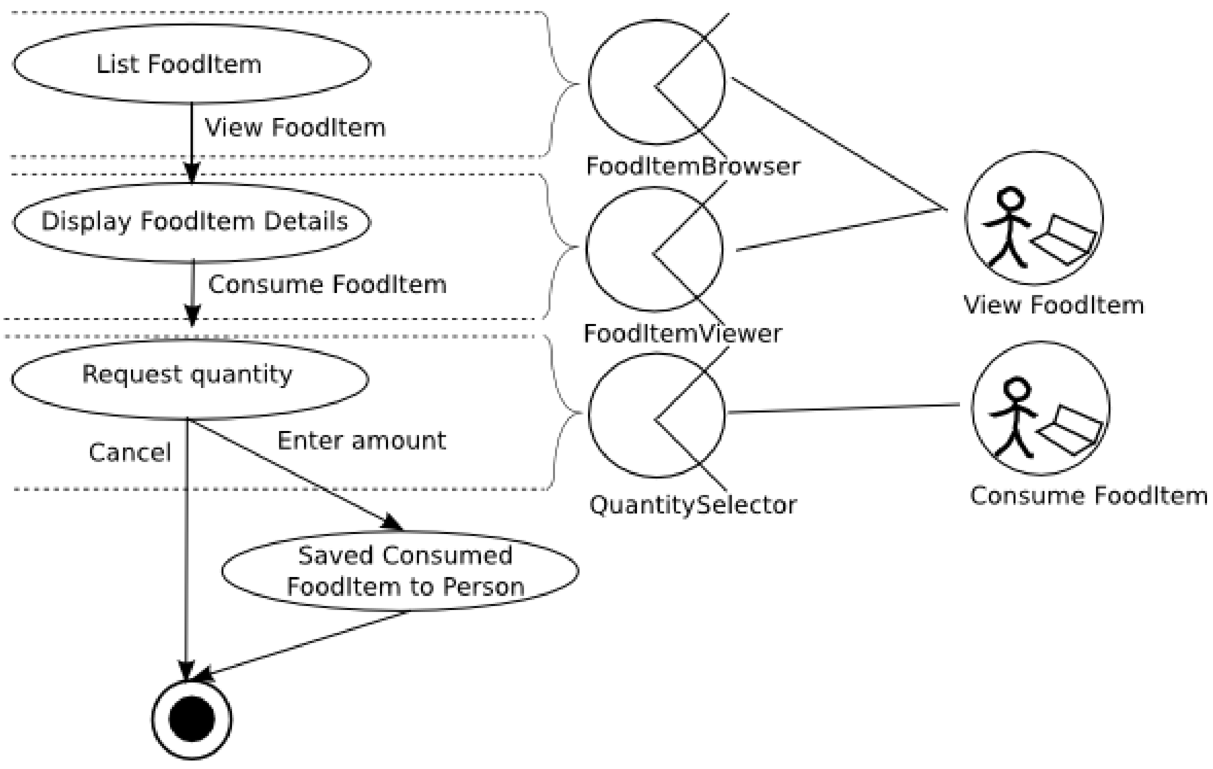


Figure 3.29: Task model for consuming a FoodItem

The task model for editing a Person can be seen in figure A.15 in the appendix.

Editing a Person involves the three interaction spaces: PersonBrowser, PersonViewer and PersonEditor. PersonViewer display the details about the Person to the user. PersonEditor display the information of the Person, the user can to edit.

The task model for viewing a Person can be seen in figure A.14 in the appendix.

Viewing a Person involves the two interaction spaces: PersonBrowser and PersonViewer, which have already been explained.

The task model for deleting a Person can be seen in figure A.16 in the appendix.

Deleting a Person involves four interaction spaces: Login, PersonBrowser, PersonViewer and ConfirmDialog. Login, PersonBrowser and PersonViewer have been previously described. The ConfirmDialog prompts the user for confirmation to delete the Person.

Exercise

Editing an Exercise involves three interaction spaces: ExerciseBrowser, SportsActivityBrowser and ExerciseEditor. The task model can be seen in figure 3.31.

The ExerciseBrowser lists all of the previous Exercises, where an Exercise can be selected for editing. The SportsActivityBrowser lists all of the SportsActivities where one can be selected, and the ExerciseEditor is used to edit the other aspects of the Exercise.

A new SportsActivity can also be created when editing an Exercise, and this is done using the SportsActivityEditor, where the details can be entered.

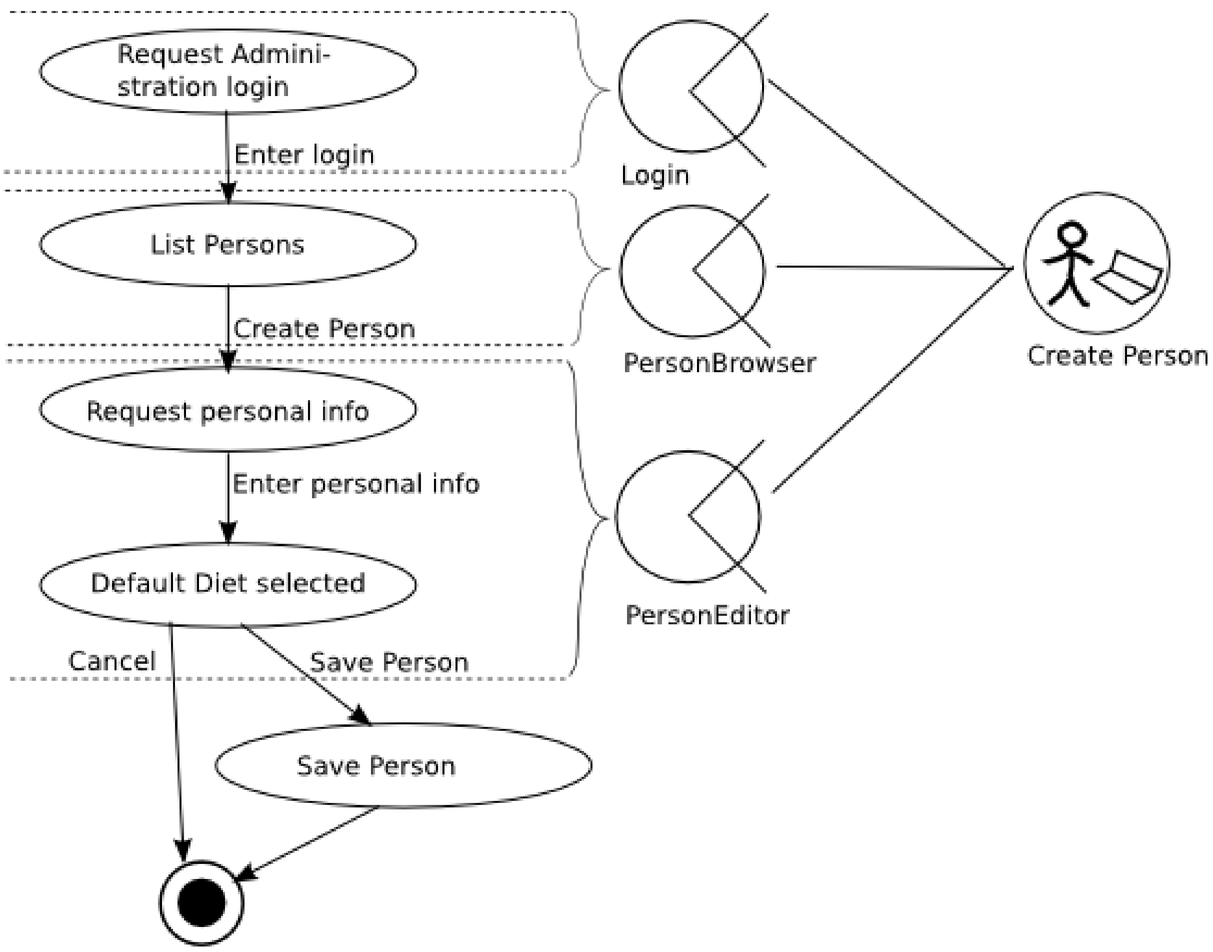


Figure 3.30: Task model for creating a new Person

The task model for registering a new Exercise can be seen in figure A.18 in the appendix. It involves two interaction spaces: SportsActivityBrowser and ExerciseEditor, which is similar to editing an Exercise.

Viewing an Exercise involves the ExerciseBrowser. The task model can be seen in figure A.17. The Exercise is displayed in the ExerciseBrowser for the user to view.

Deleting an Exercise involves two interaction spaces: ExerciseBrowser and ConfirmDialog. The task model can be seen in figure A.19. Deleting an Exercise can be done by selecting an Exercise in the ExerciseBrowser and clicking delete. The ConfirmDialog will then prompt the user for confirmation.

SportsActivity

Creating a SportsActivity involves two interaction spaces, SportsActivityBrowser and SportsActivityEditor, as in figure 3.32. The SportsActivityBrowser lists all of the SportsActivities, and in the SportsActivityEditor the details can be entered.

Editing a SportsActivity can be seen in figure A.21 in the appendix, and is the same as creating a SportsActivity.

Deleting a SportsActivity involves two interaction spaces, SportsActivityBrowser

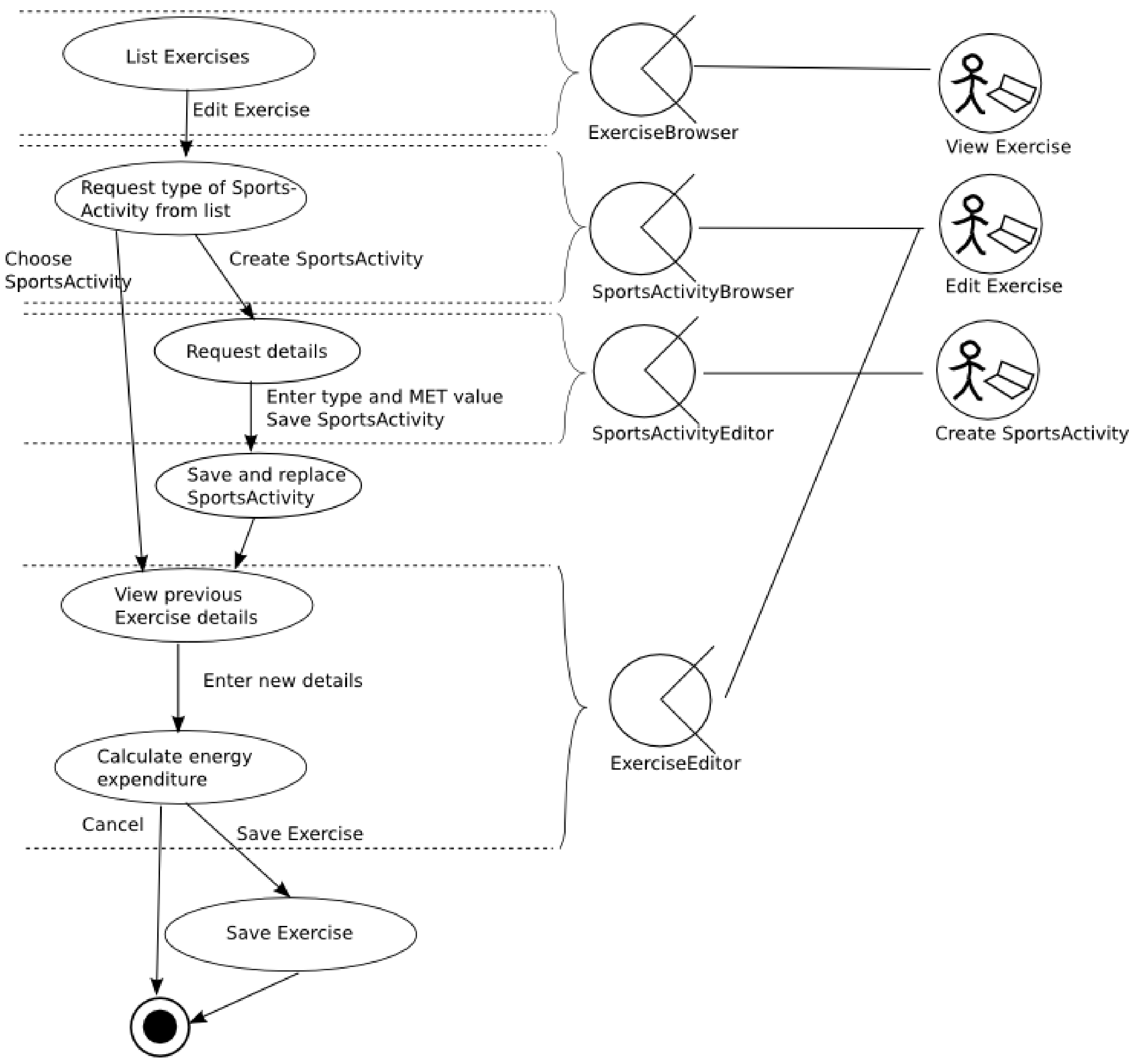


Figure 3.31: Task model for editing an Exercise

and ConfirmDialog, as in figure A.22 in the appendix. The SportsActivityBrowser lists the SportsActivities and the ConfirmDialog confirms the deletion.

Viewing a SportsActivity only uses the SportsActivityBrowser, as there is not a lot of information to display. It can be seen in figure A.20 in the appendix.

Diet

In figure 3.33 the task model for editing a Diet is displayed. Three interaction spaces are involved; DietBrowser, which lists all of the Diets registered in the system, DietViewer in which the details of the diet are shown, and the DietEditor, where the new Diet details are entered. When Creating a Diet only two of the aforementioned interaction spaces are involved. These are DietBrowser and DietEditor. See figure A.23 in appendix A. When viewing a Diet and choosing a Diet, the interaction spaces Diet-

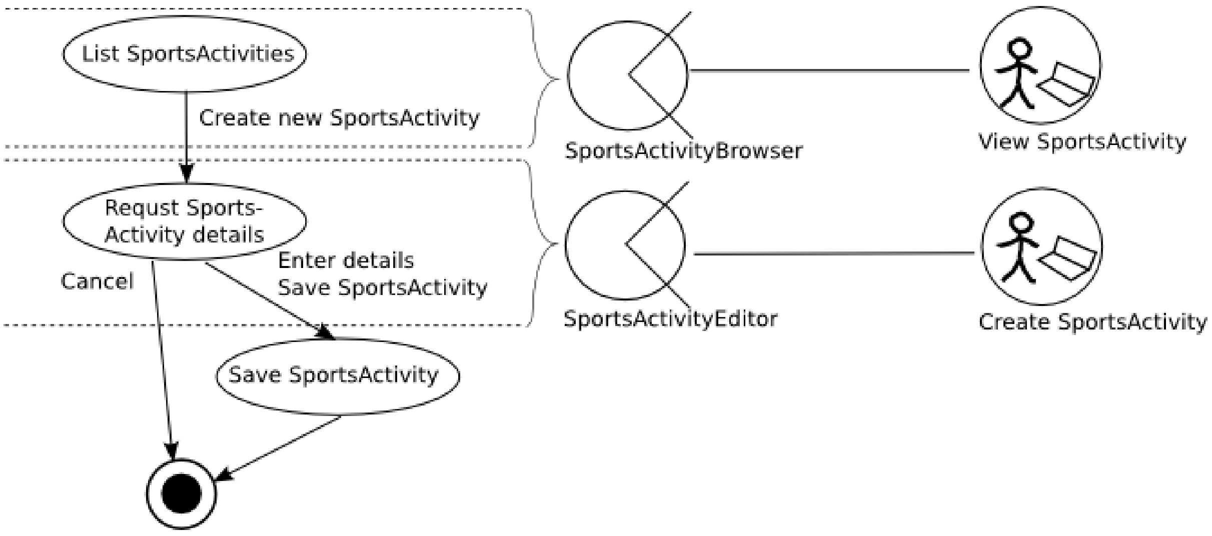


Figure 3.32: Task model for creating a SportsActivity

Browser and DietViewer are involved. Viewing a diet is illustrated in figure A.24 and choosing a Diet is illustrated in figure A.25, both in the appendix A. Deleting a Diet involves the interaction spaces DietBrowser and ConfirmDialog, this is illustrated in figure A.26.

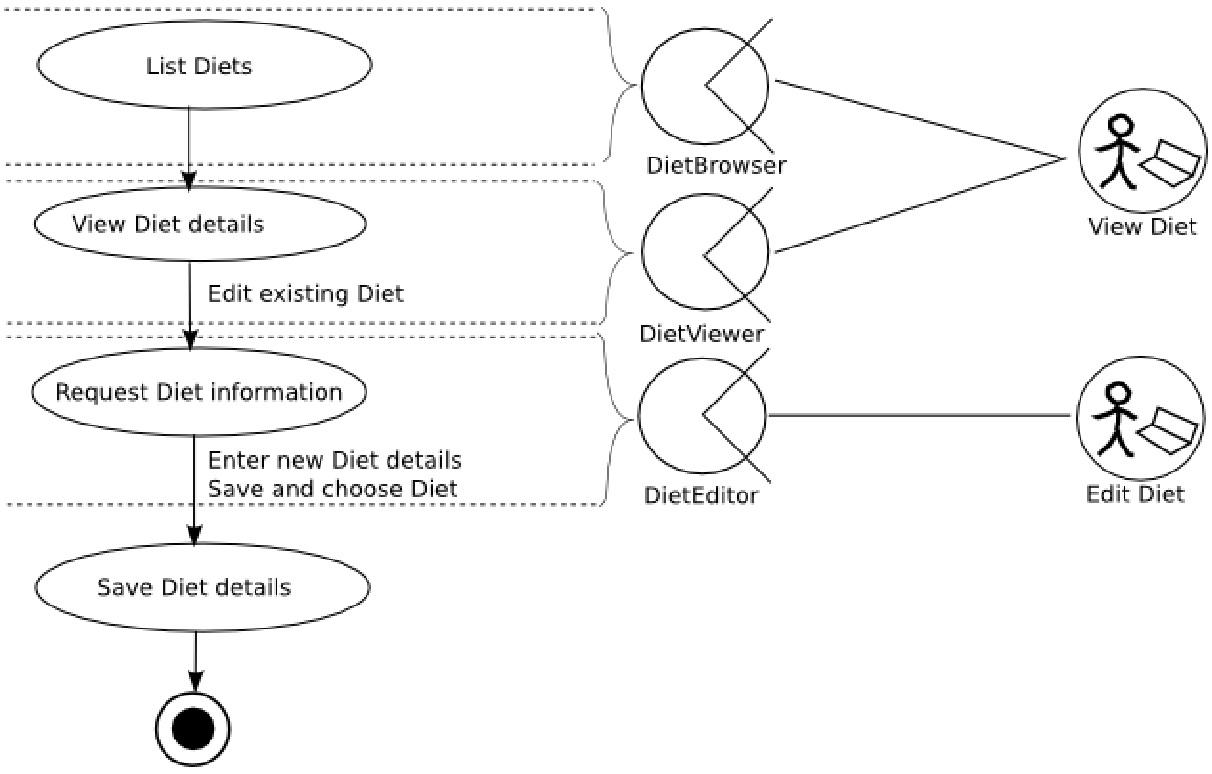


Figure 3.33: Task model for editing Diet

General Interaction Model

With all the interaction spaces defined, the general interaction model can be made. The general interaction model is, as mentioned, a collection of all interaction spaces and subtasks, and how these are connected. Figure 3.34 and 3.35 shows the general interaction model.

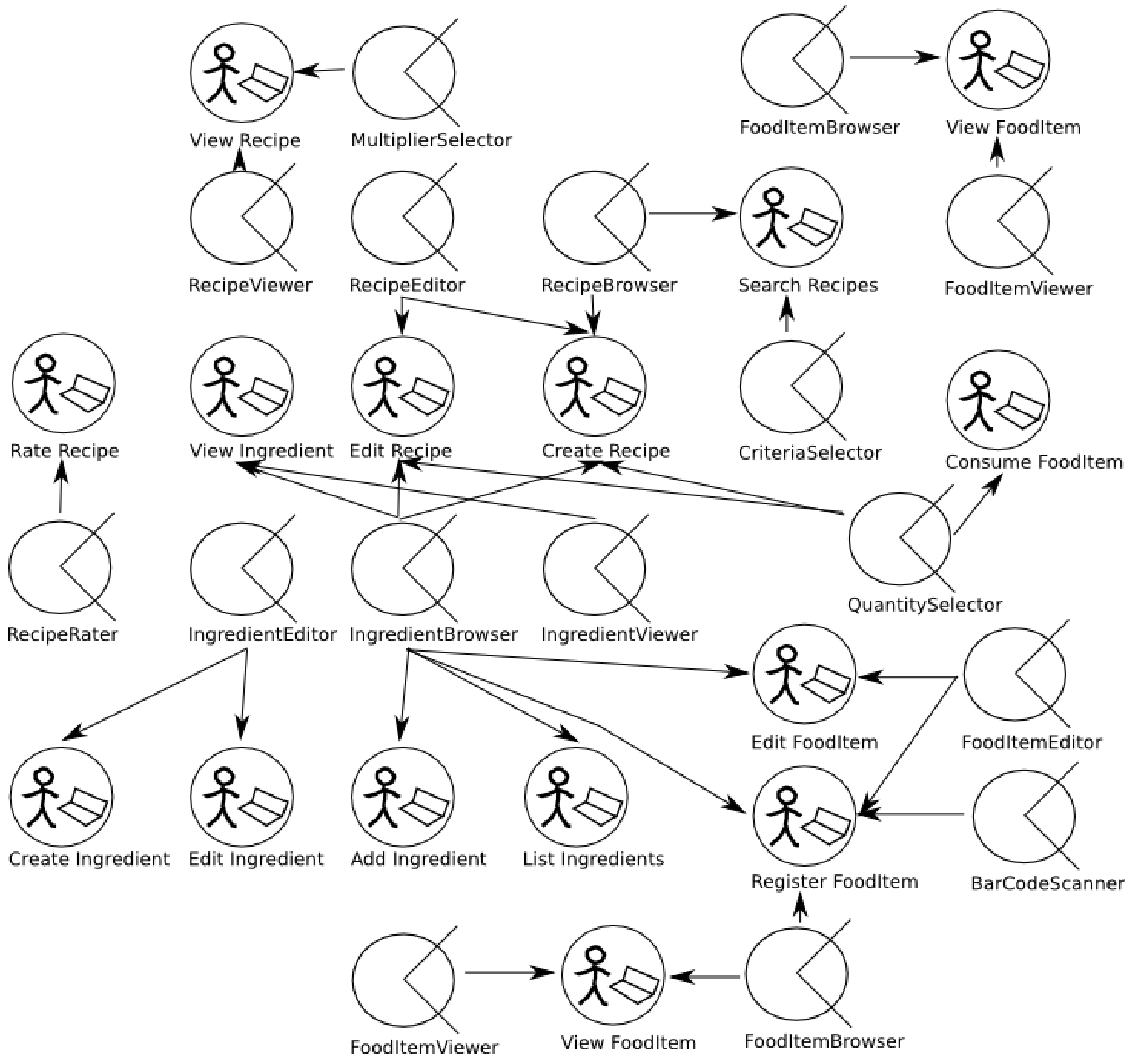


Figure 3.34: General Interaction Model part 1

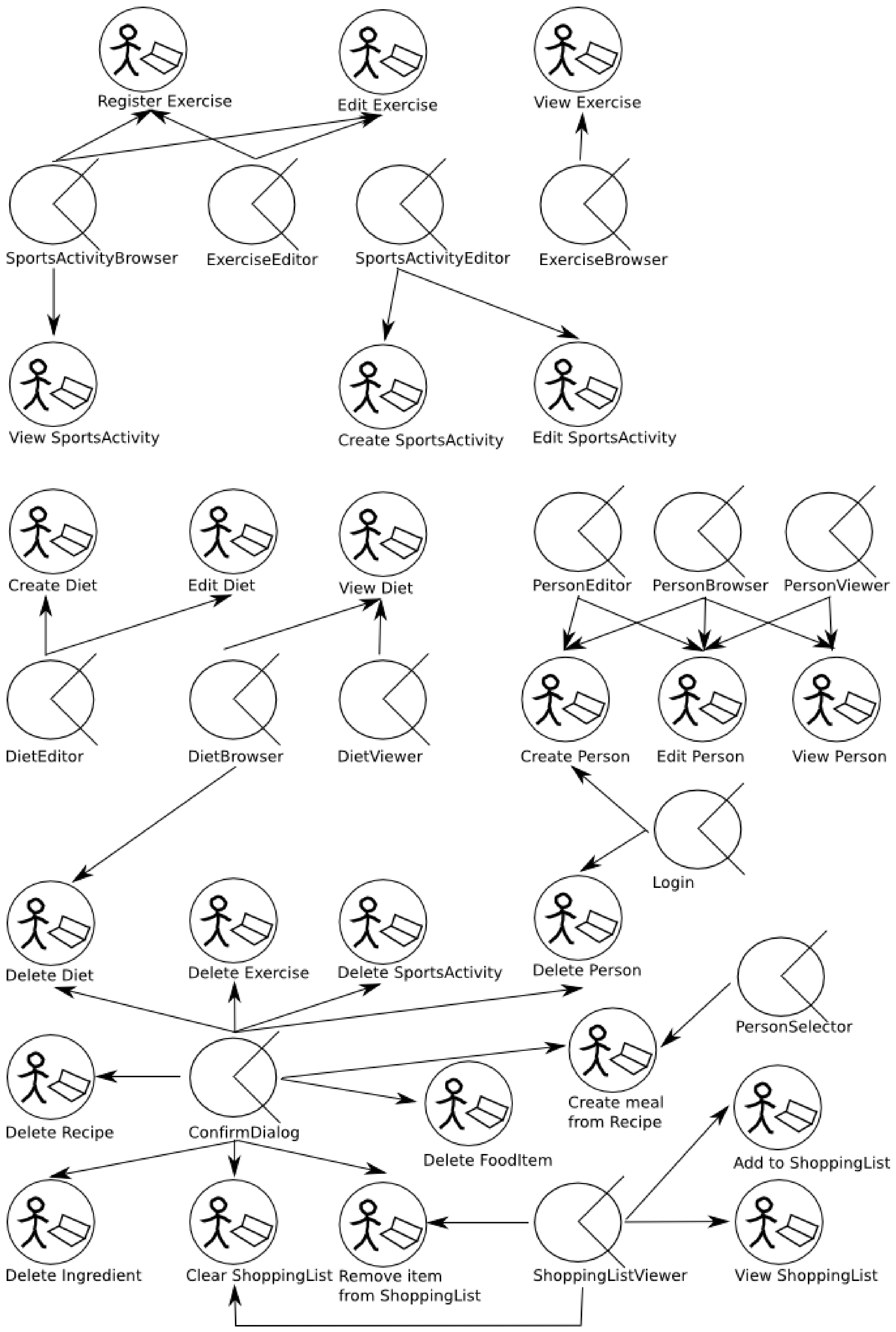


Figure 3.35: General Interaction Model part 2

CHAPTER 4

Technical Platform

The system will be designed to run on a normal desktop computer, as a desktop application, using a mouse and keyboard. It will be written in C# on .Net/Mono with Gtk# as the front-end. The system will use an SQLite database to manage data, and nutritional information will be imported from the USDA National Nutrient Database for Standard Reference. For bar code scanning, a managed wrapper for a subset of libzbar will be written, and libzbar will be used to scan bar codes and interface video devices as it wraps V4L (Video 4 Linux) and V4W (Video 4 Windows) on Linux and Windows, respectively.

CHAPTER 5

Recommendations

Recommendations for design and development.

5.1 Usefulness and Feasibility

The system should be useful, and even if not implemented in its entirety. The system should be able to function as a recipe and food manager, without support for features, such as webservices, exercise and diet management.

It should be fairly possible to implement the system. Some research into bar code recognition with webcams has been done and it appears to be achievable. This is the most questionable part of the system. If it is possible, the system will most likely be realisable.

5.2 Strategy

It is recommended that important parts of the user interface be designed and tested before actual implementation. This is to ensure that the user interface will be efficient, since an inefficient user interface could render the entire system useless. Therefore, important parts of the user interface should be tested during development.

5.3 Development Economy

The development team consist of seven people and there is about 1-2 months to implement the system. The skills and experience of the team members vary a lot. We believe that it should be possible to implement the entire system, however, it may be necessary to only implement a subset of the system, if documentation and other tasks require too many resources.

Part II

Design

CHAPTER 6

Task

6.1 Purpose

The purpose of this development project is to create an application to assist in maintaining the contents of the refrigerator and various cupboards in the kitchen. The application will act as an assistant in the kitchen, monitoring food stocks and approaching expiration dates. Furthermore, the application will be able to assist the household user in maintaining a diet.

It does this by allowing the user to create and maintain certain diets. The application will then be able to use this information to help suggest recipes that fullfil the dietary requirements amongst the recipes that it has stored. More functionality will be added to assist this capability, such as being able to add new recipes. The user can also bypass the personal diet system by using the default diet, thereby only using the storage assisting capabilities of Foodolini, and/or recipe suggestion without regard for dietary needs.

6.2 Quality Goals

Our primary goal for Foodolini is for it to be usable in an efficient way. This is important because of the inherent nature of the application. The application requires the user to go out of his way to update the system, in order for it to be of assistance. Foodolini must therefore offer incentives of sufficient value in order to offset this significant downside. This downside can also be reduced in magnitude by ensuring that Foodolini allows these activities to be performed quick and painlessly.

Other criteria are more related to development of the application. For instance, Foodolini must be testable and flexible. Foodolini must also work on multiple platforms, and be able to connect to external databases for information such as recipes and information regarding food items.

6.3 Analysis Corrections

The analysis describes the search capability of Foodolini as a complex algorithm which integrates the dieting, allowing the user to search for recipes based on their diet. It is out of the scope of this project to develop and implement an efficient algorithm for taking everything Foodolini has to offer into account, so searching recipes will be down prioritised.

CHAPTER 7

Technical Platform

The system will be designed in UML and the application will run on a normal desktop computer. It will be developed in C# for the .NET/Mono Framework. Gtk# is used for the user interface. Visual Studio 2008 and MonoDevelop are the integrated development environments of choice for this project.

To persist data, the system will run on a portable SQLite database. This database will primarily hold nutritional data from the USDA National Nutrient Database, as well as recipes from various sources. The database will be accessed using our own database metadata mapping component, which supports CRUD (Create-Read-Update-Delete) operations.

A wrapper for an unmanaged bar code library, called Zbar, will be used to enable the user to scan bar codes. Zbar runs on both Linux and Windows, since both Video 4 Linux and Video 4 Windows are supported.

7.1 Equipment

A regular laptop/personal computer as well as a webcam is required for the system to run. The webcam is required, in order to add new `FoodItems` to the storage.

7.2 Programs

The UML diagrams for the system will be designed in a tool called ArgoUML. Class diagrams will be created using NClass. To develop the system, Visual Studio and MonoDevelop will be used. Source control will be managed using SVN. Unit tests will be created and conducted with NUnit. Finally, Doxygen will be used to generate code documentation in several different output formats, such as CHM.

7.3 System Interface

A component, called libzbar will allow the system to interface a webcam via the USB port. This component will serve to both capture the image stream from the device, in addition to identifying bar codes from the images.

The system will interface the database with the aforementioned database metadata mapping component, which encapsulates various methods of interaction with the database. All communication with the database will therfore be channeled through this component.

7.4 Design Language

The design language used for this project is UML, since it is a general-purpose modeling language that encompasses class, use case, state, sequence and activity diagrams. All of which are required for this project. There are also many UML modeling tools available, such as ArgoUML and nClass. Finally, UML is part of the design process in this project.

CHAPTER 8

Architecture

8.1 Design Criteria and Architectural Demands

Features and flexibility that the architecture must accommodate.

8.1.1 Design Criteria

	Very important	Important	Less important	Irrelevant	Easily fulfilled
Usable	✔				
Secure			✔		
Efficient			✔		
Correct			✔		
Reliable			✔		
Maintainable			✔		
Testable		✔			✔
Flexible		✔			✔
Comprehensible			✔		
Reusable			✔		
Portable		✔			
Interoperable		✔			

Usable: As this system will be used a lot during the day, it is very important that it is easy and fast to operate.

Secure: This has been given low priority due to the relatively small consequence of the system being compromised.

Efficient: This has been given low priority because the hardware the system will run on, will be much bigger than the system will ever use.

Correct: This has low priority as it is not critical that the system runs exactly as described.

Reliable: As the system does not handle any critical operations it is less important if the system crashes or miscalculates.

Maintainable: As the system will run on existing hardware it is less important to locate system defects.

Testable: As the system has many features and multiple developers it is important that it is easy to test for errors.

Flexible: Because system elements might need to be changed flexibility is important.

Comprehensible: As this system is used a lot during the day the user will over time become familiar with the user interface and as the number of different tasks the user can perform is limited it is less important that the user interface is consistent.

Reusable: The system is a specially designed program not intended to be used for anything else, therefore this has low priority.

Portable: As the development is going to take place on different platforms it is important that the system is portable.

Interoperable: The system needs to connect to external databases therefore this is important

8.1.2 Architectual Demands

Table 8.1 displays the factors and their respective priorities.

8.2 Generic Design Decisions

Decisions that affect the design and enables the architecture to deliver the desired flexibility.

8.2.1 User Interface

The user interface will consist of a main UI component, which contains an interchangeable widget for a specific use case. Widgets will encapsulate functionality with regards to a certain use case, for instance a `FoodItem` registration widget would be responsible for scanning bar codes and adding `FoodItems` to the storage.

8.2.2 Model

Each class in the model contains functionality that enables mapping from the data access layer. For example, an `Ingredient` requires more than one data access class in order to be fully instantiated, since the database is normalised accross more than one table for this class in particular.

8.2.3 Persistence

Data for the application will be stored in a local SQLite database. This is due to the ease of storing, filtering and retrieving data from a database.

8.2.4 Data Access

Database querying is handled via a set of database-centric classes, which are used by a mapping component. Calls to this layer will be performed through the model layer. This method ensures that the database platform is interchangeable.

8.3 Component Architecture

The components for Foodolini will be separated into three layers: front-end, logic and database. These layers should ideally be closed-strict, though if a minor violation would ease the development significantly, it would be acceptable. Such a circumstance could arise because of asynchronous operations in the logic or database layers. However, no such circumstance seems likely at the moment - so unless it requires an unreasonable amount of work, this architecture should be respected. And if violated, it should only be so in a few very specific places.

The database layer is responsible for persistence and the ability to efficiently perform simple queries. The model component is responsible for providing thin classes for representing the tables in the database. The repository component is responsible for encapsulating any database specific information such as SQL queries. This component will feature a minimalistic ORM[1]. It is the goal that this component may easily be patched so that it may interface another backend database. This feature is desirable, since large datasets may alter the requirements for the database.

On top of the database layer, the logic layer is to be implemented. This layer is supposed to offer actual functionality on top of the persistence from the database layer. The business logic component mainly contains classes that are used to represent the problem domain. This component will be based on top of the database layer and use objects from this layer.

The user interface is in the front-end layer, which is based on the logic layer and may not interface the database layer directly. The UI is built using a simple plug-in architecture, where the main component hosts plug-ins found in the activities component. The activities component contains an interface the plug-ins must implement and an attribute they must apply. The plug-ins will then be loaded dynamically at runtime, using reflection. This approach makes it possible to implement the plug-ins of the activities component in as many different sub-projects as needed in order to facilitate parallel development.

A plug-in in the activities component will be a major widget that almost fills the entire main windows when it is active. Such plug-in will be created for each major use case, such as registration of food items using bar codes.

The libzbar-cli component is suppose to contain bindings for the ZBar library, a cross platform free software library for scanning bar codes. The bindings in the libzbar-cli component are placed in a separate component so that dependencies to other libraries may be kept to a minimum and the code may be given back to the community for use elsewhere.

[1]ORM: Object-Relational Mapping.

The point of the three layer architecture is that the front-end and database layers may be replaced without the need for replacing the logic layer. This might be interesting if the system were to be ported to an embedded device, or if a web service for bar code association exchange is to be implemented. As the database engine may be replaced with an engine that is capable of handling larger data sets and the front-end replaced by a SOAP web service.

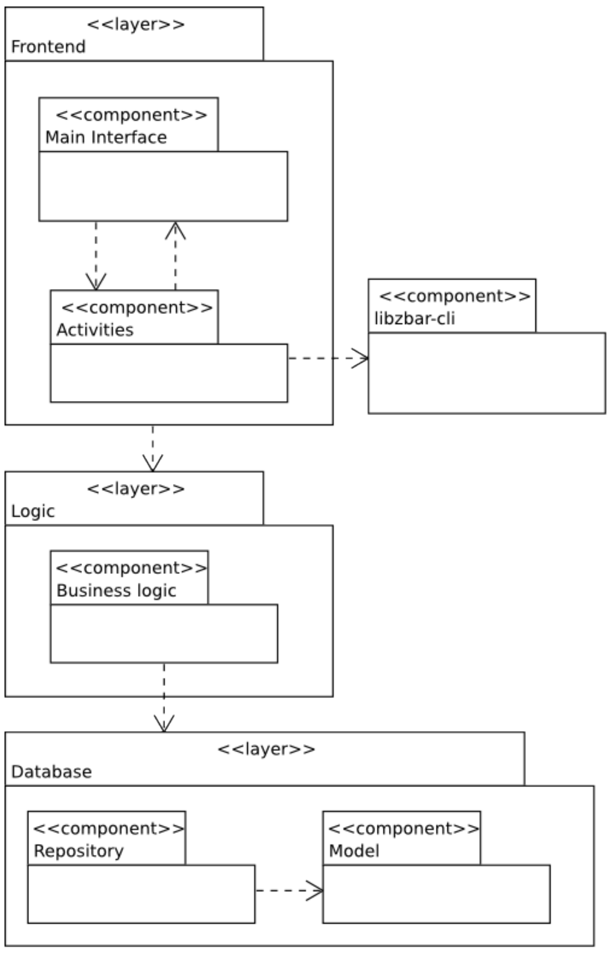


Figure 8.1: Architecture for Foodolini.

	Goal	**Variability**	**Impact of Factor**	**Priority of Success (1-5)**	**Difficulty or Risk**
Reliability - Persistence					
Database recovery after crash	The database shouldn't be lost on crash		If the computer or the application crashes the database shouldn't be affected	5	Medium
Persistence	Data must be persistent from session to session	The database engine platform can be replaced, without complete rewrite.	The information must be saved in order to make the system usable	5	Hard
Interoperability					
Bar code information distribution	Bar codes should be uploaded to a shared database, so that a specific bar code only has to be defined by one user		Lowers the need to manually define new bar codes	2	Hard
Import/export of recipes	Recipes should be imported and exported from various formats	It should be possible to add new formats later	Will make it easier to get new recipes and share them with others	2	Medium
Usability					
Efficient to use	With 10 minutes of training a computer literate user should be able to register 10 known FoodItems within a minute		The system will be aggravating and slow to use if not	5	Medium
Speed	The system must not take longer than 500 milliseconds for the first recipe suggestion and 10 seconds for the 10th. recipe suggestion.		The used algorithms must be efficient for many data input	1	Medium
Other					
Extendable	Easy to add new features			1	Medium

Table 8.1: Factor Table

CHAPTER 9

Components

9.1 Model Component

The model component offers thin classes for the tables in the database.

9.1.1 Structure

This component offers a thin class for each table in the database layer. An instance of such a class represents a single row of a database table, thus providing strongly typed access to the fields of the specific row. The repository component (see section 9.2) is reponsible for adding, updating, deleting and loading these thin classes to/from the database. By following this pattern the classes in this component will only consist of automatic properties with public getters and setters, and the types of these properties will be rather primitive. For instance the `Picture` class is used for the table called `Pictures` with a primary key called `PictureId` and a column called `Image`. The thin classes can be seen on figure 9.1, the relations in this diagram indicates relations in the database. For example the relation from `FoodDescription` to `FoodGroup` indicates that an instance of `FoodDescription` has an identifier that can be used to request a `FoodGroup`, in this case that is the `FoodGroupId` field in a `FoodDescription`.

9.1.2 Classes

Description of some of the classes on figure 9.1 that are not immediately intuitive.

RecipeFoodMapping

An instance of `RecipeFoodMapping` maps between a `RecipeRow` and a `FoodDescription`, that is it maps a recipe to its ingredients, with a quantity that says how much of a given ingredient should be in the recipe.

RecipeTag

An instance of `RecipeTag` represents a row that applies tags to a recipe. The `Title` property holds the title of the tag and the `RecipeRowId` references the `RecipeRow` this tag is applied to. Tags could also be applied to a recipe using a comma separated string property, however, this would make it harder to search by tag.

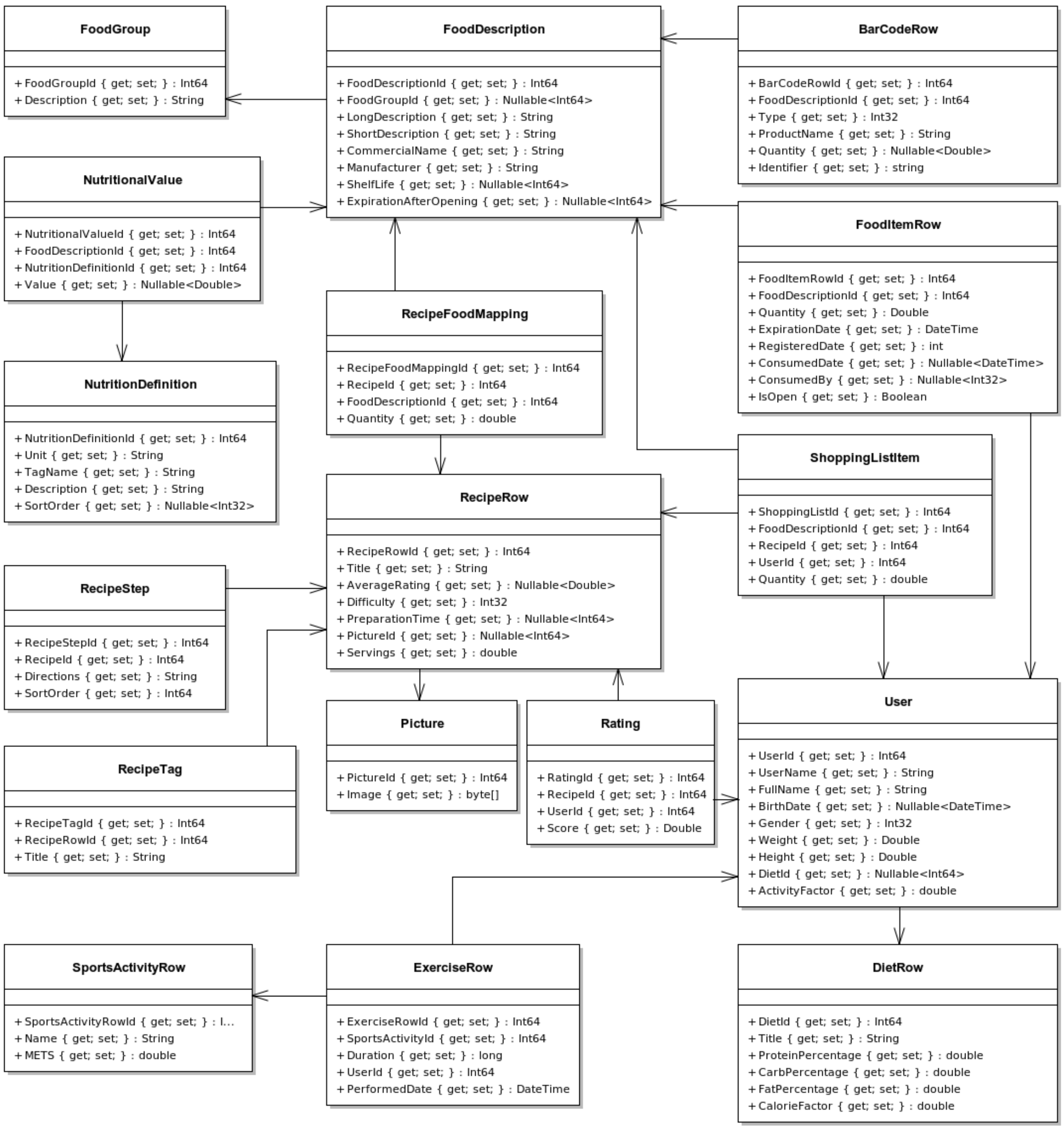

FoodGroup
+FoodGroupId { get; set; } : Int64
+Description { get; set; } : String
FoodDescription
+FoodDescriptionId { get; set; } : Int64
+FoodGroupId { get; set; } : Nullable<Int64>
+LongDescription { get; set; } : String
+ShortDescription { get; set; } : String
+CommercialName { get; set; } : String
+Manufacturer { get; set; } : string
+ShelfLife { get; set; } : Nullable<Int64>
+ExpirationAfterOpening { get; set; } : Nullable<Int64>
BarCodeRow
+BarCodeRowId { get; set; } : Int64
+FoodDescriptionId { get; set; } : Int64
+Type { get; set; } : Int32
+ProductName { get; set; } : String
+Quantity { get; set; } : Nullable<Double>
+Identifier { get; set; } : string
NutritionalValue
+NutritionalValueId { get; set; } : Int64
+FoodDescriptionId { get; set; } : Int64
+NutritionDefinitionId { get; set; } : Int64
+Value { get; set; } : Nullable<Double>
FoodItemRow
+FoodItemRowId { get; set; } : Int64
+FoodDescriptionId { get; set; } : Int64
+Quantity { get; set; } : Double
+ExpirationDate { get; set; } : DateTime
+RegisteredDate { get; set; } : int
+ConsumedDate { get; set; } : Nullable<DateTime>
+ConsumedBy { get; set; } : Nullable<Int32>
+IsOpen { get; set; } : Boolean
RecipeFoodMapping
+RecipeFoodMappingId { get; set; } : Int64
+RecipeId { get; set; } : Int64
+FoodDescriptionId { get; set; } : Int64
+Quantity { get; set; } : double
NutritionDefinition
+NutritionDefinitionId { get; set; } : Int64
+Unit { get; set; } : String
+TagName { get; set; } : String
+Description { get; set; } : String
+SortOrder { get; set; } : Nullable<Int32>
ShoppingListItem
+ShoppingListId { get; set; } : Int64
+FoodDescriptionId { get; set; } : Int64
+RecipeId { get; set; } : Int64
+UserId { get; set; } : Int64
+Quantity { get; set; } : double
RecipeRow
+RecipeRowId { get; set; } : Int64
+Title { get; set; } : String
+AverageRating { get; set; } : Nullable<Double>
+Difficulty { get; set; } : Int32
+PreparationTime { get; set; } : Nullable<Int64>
+PictureId { get; set; } : Nullable<Int64>
+Servings { get; set; } : double
RecipeStep
+RecipeStepId { get; set; } : Int64
+RecipeId { get; set; } : Int64
+Directions { get; set; } : String
+SortOrder { get; set; } : Int64
Picture
+PictureId { get; set; } : Int64
+Image { get; set; } : byte[]
Rating
+RatingId { get; set; } : Int64
+RecipeId { get; set; } : Int64
+UserId { get; set; } : Int64
+Score { get; set; } : Double
User
+UserId { get; set; } : Int64
+UserName { get; set; } : String
+FullName { get; set; } : String
+BirthDate { get; set; } : Nullable<DateTime>
+Gender { get; set; } : Int32
+Weight { get; set; } : Double
+Height { get; set; } : Double
+DietId { get; set; } : Nullable<Int64>
+ActivityFactor { get; set; } : double
RecipeTag
+RecipeTagId { get; set; } : Int64
+RecipeRowId { get; set; } : Int64
+Title { get; set; } : String
SportsActivityRow
+SportsActivityRowId { get; set; } : I...
+Name { get; set; } : String
+METS { get; set; } : double
ExerciseRow
+ExerciseRowId { get; set; } : Int64
+SportsActivityId { get; set; } : Int64
+Duration { get; set; } : long
+UserId { get; set; } : Int64
+PerformedDate { get; set; } : DateTime
DietRow
+DietId { get; set; } : Int64
+Title { get; set; } : String
+ProteinPercentage { get; set; } : double
+CarbPercentage { get; set; } : double
+FatPercentage { get; set; } : double
+CalorieFactor { get; set; } : double

Figure 9.1: Class diagram for the model component.

Rating

An instance of `Rating` maps a rating and the user who gave that rating to the recipe that was rated.

NutritionDefinition

An instance of `NutritionDefinition` defines a nutrient, for instance protein, fat or one various vitamins. The idea is that instances of `NutritionalValue` are used map a nutrients with an amount (`Value`) to a `FoodDescription`. This way it is possible to have a `NutritionDefinition` that defines what fat is, a `NutritionalValue` that tells how much fat is in the `FoodDescription` instance that represents milk. It is possible to replace `NutritionDefinition` with a simple enum, however, the database with nutritional information we have got from USD [2008] uses a similar approach and this would make the data conversation less complicated. At last but not least, using an enum is also less flexible, in case new nutrients were to be added in the future.

9.2 Repository Component

The repository component offers a database abstraction layer that is easily portable.

9.2.1 Structure

The purpose of the database abstraction layer is to isolate database specific code, such as SQL statements, so that underlying database can be easily replaced, while creating an easy to use API. This can be achieved by mapping the rows of a table to objects. The classes that these objects are instances of can then be generated from the database table scheme, or vice versa. For this project we will be generating the database tables based on the classes which represent the rows of these tables. This pattern is called metadata mapping [Fowler et al., 2002] and will be implemented using reflection. It is highly inspired by the SimpleRepository of SubSonic[1] which offers some of the same features, and which probably could replace this component if it were to become stable in the future.

The database interaction takes place in the `Repository` class. This class has several type parameterised methods, such as `Add<T>(T item)`, which creates a table to store instances of type `T`, if such table does not already exist, and adds `item` to the table that is generated from `T`. Similarly, it has a method called `All<T>()` which returns all the rows in the database table, which were generated from `T`. These methods use reflection to find and modify the properties of the parameterised type `T`. This approach works sufficiently, as long as, the parameterised type only has primitive properties. The `Repository` class is not supposed to handle type conversation beyond wrapping in `Nullable<>` and simple conversation using
`Object.ChangeType(Object, Type)`.

In order to isolate SQL generation completely, a strategy pattern defined by `SqlStrategy` will be used, see figure 9.2. This means that when the `Repository` class needs to generate an SQL statement it will invoke a method on an instance of a `SqlStrategy` implementation. Thus, the only thing necessary to port the `Repository` to another database is to make a new implementation of `SqlStrategy` for this database. For example, in order to use MySQL as a database backend for the `Repository` class, an implementation of `SqlStrategy` for MySQL would be needed. An instance of such an implementation, along with a database connection to a MySQL database, could be passed to the constructor of the `Repository` class [Gamma et al., 1998].

[1]ORM tool by Rob Connery, SubSonicProject.com

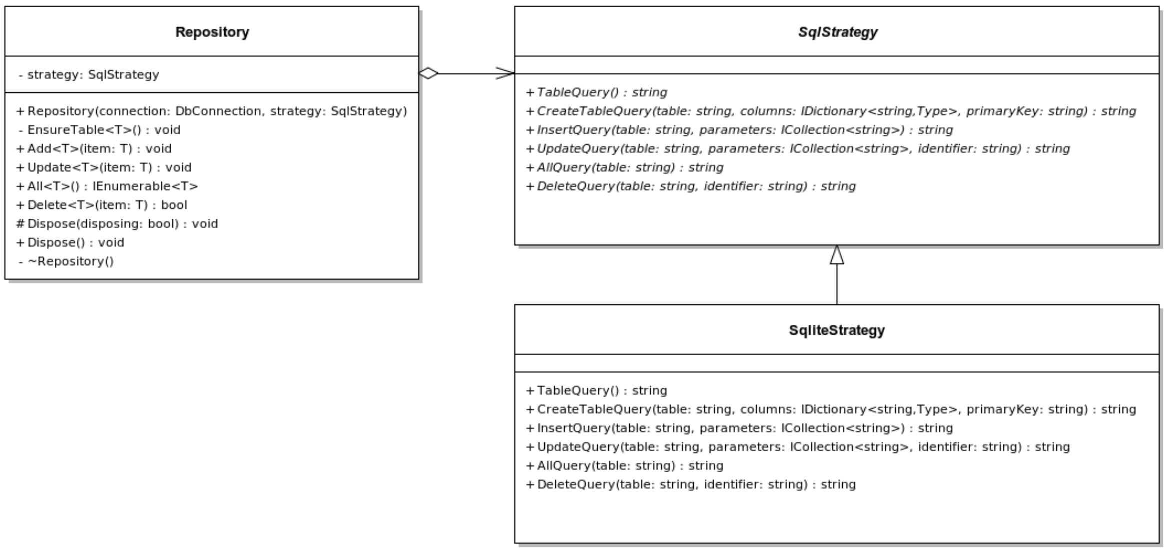

Repository
- strategy: SqlStrategy
+ Repository(connection: DbConnection, strategy: SqlStrategy)
- EnsureTable<T>() : void
+ Add<T>(item: T) : void
+ Update<T>(item: T) : void
+ All<T>() : IEnumerable<T>
+ Delete<T>(item: T) : bool
# Dispose(disposing: bool) : void
+ Dispose() : void
- ~Repository()
SqlStrategy
+ TableQuery() : string
+ CreateTableQuery(table: string, columns: IDictionary<string,Type>, primaryKey: string) : string
+ InsertQuery(table: string, parameters: ICollection<string>) : string
+ UpdateQuery(table: string, parameters: ICollection<string>, identifier: string) : string
+ AllQuery(table: string) : string
+ DeleteQuery(table: string, identifier: string) : string
SqliteStrategy
+ TableQuery() : string
+ CreateTableQuery(table: string, columns: IDictionary<string,Type>, primaryKey: string) : string
+ InsertQuery(table: string, parameters: ICollection<string>) : string
+ UpdateQuery(table: string, parameters: ICollection<string>, identifier: string) : string
+ AllQuery(table: string) : string
+ DeleteQuery(table: string, identifier: string) : string

Figure 9.2: Class diagram for the repository component.

However, this pattern also means that whenever the `Repository` needs to support a new type of query, corresponding methods for SQL generation must be implemented to all implementations of `SqlStrategy`. This may be slightly inflexible, however, it is not expected that this project will need a large set of complex queries. Also as the `SqlStrategy` is an abstract class, and not an interface, it can evolve. A default implementation for a new method can be provided on `SqlStrategy` either by attempting to generate portable SQL or by throwing a `NotSupportedException` and providing a boolean property for telling if the feature is supported, as suggested in Abrams and Cwalina [2008] section 4.3.

9.2.2 Classes

Description of the most important classes. Thin classes representing database tables are omitted.

Repository

`Repository` is a simple minimalistic object-relational mapper, that maps objects to database tables using reflection. This class is responsible for type conversion and execution of SQL queries. While SQL generation is delegated to an instance of `SqlStrategy`, this class also has a method for executing SQL queries on the database, so that not all queries desired by the clients need to be supported by this class and the `SqlStrategy` class. This class is inspired by the SimpleRepository class of SubSonic.

SqlStrategy

The `SqlStrategy` class is abstract, and declares abstract methods for generating the SQL statements needed by the `Repository` class. This means that the `Repository` is highly dependant on this class.

SqliteStrategy

This class is an implementation of `SqlStrategy` for Sqlite, which is designed to generate SQL statements supported by Sqlite 3.

9.3 Business Logic

The business logic component offers easy-to-use classes which represent the problem domain.

9.3.1 Structure

The business logic component wraps around the model component that represents rows in the database (See description of the model component 9.1). These classes are supposed to make the properties of their rows available as properties and methods with a reasonable level of validation. They should also absorb and hide the rows that relate to rows in the database, and expose these rows as lists or dictionaries between objects of classes of the business logic component. Last but not least these classes should also know how to load and save or update themselves from the database.

For instance `Ingredient` is supposed to wrap around a `FoodDescription`, and make its name, description and other properties available. It might throw an exception if it is assigned to a non-existing category or if it is assigned a negative `ShelfLife`. `Ingredient` should also lazily load the relations, e.g. the `NutritionalValue` rows, which tell how much of which `NutritionDefinition`s the `Ingredient` contains. These relations should be hidden in a dictionary of `Nutritent`s and `double`s. Such a dictionary should add, update or delete the relation rows appropriately when the `Ingredient` is saved. This way data presentation, validation and serialisation to the database is encapsulated in the `Ingredient` class.

Some of the classes in this component have a limited set of different identities. For instance there only exists a limited set of `NutritionDefinition` in the system. So when a `Nutritent` is loaded, a reference to it is stored in a static field, so that whenever the same `Nutritent` is requested again, a new object does not need to be created. This form of caching will be used for simple classes with a very limited set of different identities, such as `Nutritent` and `Person`.

In this component a singleton class called `Settings` will hold a reference to an instance of `Repository`, from the database layer. This means that the business logic component can only be connected to one database at a time. This may be somewhat inflexible, however, this will make it possible to place static methods on the classes in this component that lists/queries the database. For instance `Person` may have a static method called `ListUsers()`.

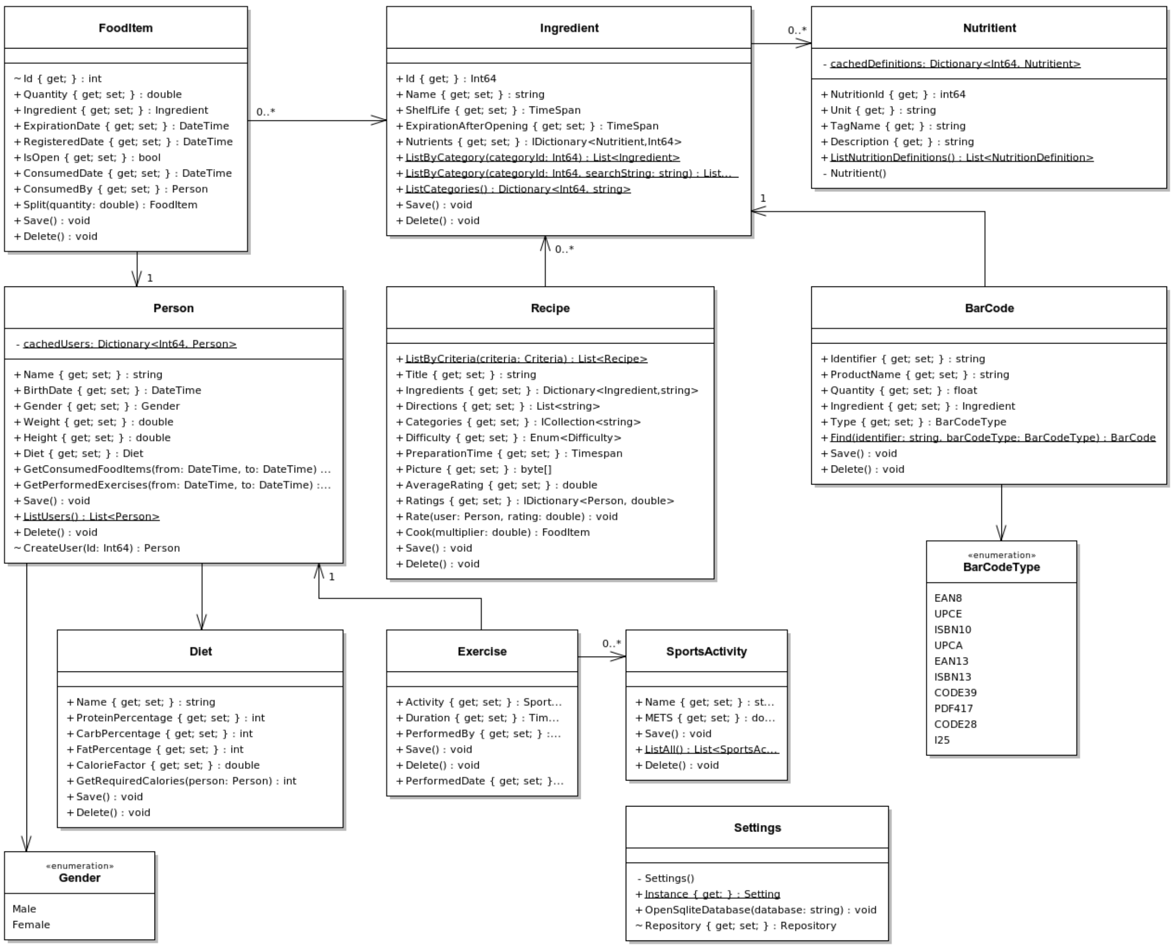

FoodItem
~ Id { get; } : int
+ Quantity { get; set; } : double
+ Ingredient { get; set; } : Ingredient
+ ExpirationDate { get; set; } : DateTime
+ RegisteredDate { get; set; } : DateTime
+ IsOpen { get; set; } : bool
+ ConsumedDate { get; set; } : DateTime
+ ConsumedBy { get; set; } : Person
+ Split(quantity: double) : FoodItem
+ Save() : void
+ Delete() : void
0..*
Ingredient
+ Id { get; } : Int64
+ Name { get; set; } : string
+ ShelfLife { get; set; } : TimeSpan
+ ExpirationAfterOpening { get; set; } : TimeSpan
+ Nutrients { get; set; } : IDictionary<Nutritient,Int64>
+ ListByCategory(categoryId: Int64) : List<Ingredient>
+ ListByCategory(categoryId: Int64, searchString: string) : List...
+ ListCategories() : Dictionary<Int64, string>
+ Save() : void
+ Delete() : void
0..*
Nutritient
- cachedDefinitions: Dictionary<Int64, Nutritient>
+ NutritionId { get; } : int64
+ Unit { get; } : string
+ TagName { get; } : string
+ Description { get; } : string
+ ListNutritionDefinitions() : List<NutritionDefinition>
- Nutritient()
1
0..*
1
Person
- cachedUsers: Dictionary<Int64, Person>
+ Name { get; set; } : string
+ BirthDate { get; set; } : DateTime
+ Gender { get; set; } : Gender
+ Weight { get; set; } : double
+ Height { get; set; } : double
+ Diet { get; set; } : Diet
+ GetConsumedFoodItems(from: DateTime, to: DateTime) ...
+ GetPerformedExercises(from: DateTime, to: DateTime) :...
+ Save() : void
+ ListUsers() : List<Person>
+ Delete() : void
~ CreateUser(Id: Int64) : Person
Recipe
+ ListByCriteria(criteria: Criteria) : List<Recipe>
+ Title { get; set; } : string
+ Ingredients { get; set; } : Dictionary<Ingredient,string>
+ Directions { get; set; } : List<string>
+ Categories { get; set; } : ICollection<string>
+ Difficulty { get; set; } : Enum<Difficulty>
+ PreparationTime { get; set; } : Timespan
+ Picture { get; set; } : byte[]
+ AverageRating { get; set; } : double
+ Ratings { get; set; } : IDictionary<Person, double>
+ Rate(user: Person, rating: double) : void
+ Cook(multiplier: double) : FoodItem
+ Save() : void
+ Delete() : void
BarCode
+ Identifier { get; set; } : string
+ ProductName { get; set; } : string
+ Quantity { get; set; } : float
+ Ingredient { get; set; } : Ingredient
+ Type { get; set; } : BarCodeType
+ Find(identifier: string, barCodeType: BarCodeType) : BarCode
+ Save() : void
+ Delete() : void
«enumeration»
BarCodeType
EAN8
UPCE
ISBN10
UPCA
EAN13
ISBN13
CODE39
PDF417
CODE28
I25
1
Diet
+ Name { get; set; } : string
+ ProteinPercentage { get; set; } : int
+ CarbPercentage { get; set; } : int
+ FatPercentage { get; set; } : int
+ CalorieFactor { get; set; } : double
+ GetRequiredCalories(person: Person) : int
+ Save() : void
+ Delete() : void
Exercise
+ Activity { get; set; } : Sport...
+ Duration { get; set; } : Tim...
+ PerformedBy { get; set; } :...
+ Save() : void
+ Delete() : void
+ PerformedDate { get; set; }...
0..*
SportsActivity
+ Name { get; set; } : st...
+ METS { get; set; } : do...
+ Save() : void
+ ListAll() : List<SportsAc...
+ Delete() : void
Settings
- Settings()
+ Instance { get; } : Setting
+ OpenSqliteDatabase(database: string) : void
~ Repository { get; set; } : Repository
«enumeration»
Gender
Male
Female

Figure 9.3: Class diagram for business logic.

9.3.2 Classes

Description of the classes in the business logic component.

Settings

The settings class provides internal[2] global access to the repository. This class uses the singleton pattern [Gamma et al., 1998] to provide a global point of access to an instance of `Settings` (Notice the static `Instance` property on figure 9.3). While it from an architectual point of view might be more appealing to not have one `static` instance of `Settings`, it simplifies the rest of the code in this component. Since this permits all other classes in this component, such as `FoodItem`, not to have an instance reference to a `Repository`. Also classes can have static methods for listing, searching and otherwise create queries on the underlying database, without the need for explictly providing a reference to `Repository`.

However, this design also limits this component, since this component cannot be

[2]The internal modifier in C# means: only accessible to classes in the same assembly.

connected to more than one database simultaneously. Therefore, this component cannot be used to move data from one database to another. Nevertheless, this is a small sacrifice, in exchange for an API that is easy to use from the front-end layer.

Nutrient

The `Nutrient` class wraps around the `NutritionDefinition` class from the model component. As there is a limited set of these in the database. These will be lazily loaded to a `static` list, so that an instance of a `Nutrient` representing a specific `NutritionDefinition` can be quickly returned. This approach is quite similar to the flyweight pattern described in Gamma et al. [1998].

Also, since instances of this class are immutable, any inconsistencies will be prevented. Conversely, this could happen if they were mutable and the client got two references (by different means) to what would be the same instance, modified one, chose not to save it and then started looking at what the client might believe was another instance. Such an inconsistency would be a minor malfunction.

Ingredient

An `Ingredient` has a name, expected shelf life, etc. and a dictionary of `Nutrients` and quantity of the nutrient. The `Ingredient` class mainly wraps around `FoodDescription` from the model component, but as shown in figure 9.1 `FoodDescription` is referenced by `NutritionalValue`, which references `NutritionalDefinition` and has a field `Value` denoting the quantity of the referenced `NutritionalDefinition` the referenced `FoodDescription` contains. This many-to-many relation is exposed as the lazily loaded `Nutrients` property on `Ingredient`, which is the previously mentioned dictionary of `Nutrients` (representing the referenced `NutritionalDefinition`) and `double` (representing the `Value` field on the relation). The sequence for the static method for loading all `Ingredients` by category is shown in figure 9.4.

FoodItem

A `FoodItem` has an `Ingredient` that describes the content, and a quantity of the specific ingredient associated with the `FoodItem`. These `FoodItems` can be registered as "consumed" or "in storage", in the latter case it could also be expired. FoodItem wraps around the `FoodItemRow` class from the model component, and offers methods `Save` and `Delete`, for saving and deleting a `FoodItem`. The `FoodDescriptionId` field on the `FoodItemRow` is wrapped in an instance of `Ingredient` and exposed as a property. Notice that a `FoodItem` is responsible for (lazily) loading its `Ingredient` and saving/deleting it when `Save` or `Delete` is invoked, respectively. The method `OpenFoodItem` changes the `FoodItem` state to open. This means the `FoodItem` now uses the `ExpirationDateAfterOpening` instead of the `ShelfLife` to calculate when the `FoodItem` expires. This represents the physical opening of the food, which in many cases would shorten the shelf life. The sequence diagram for `FoodItem.Consume` is shown in figure 9.5.

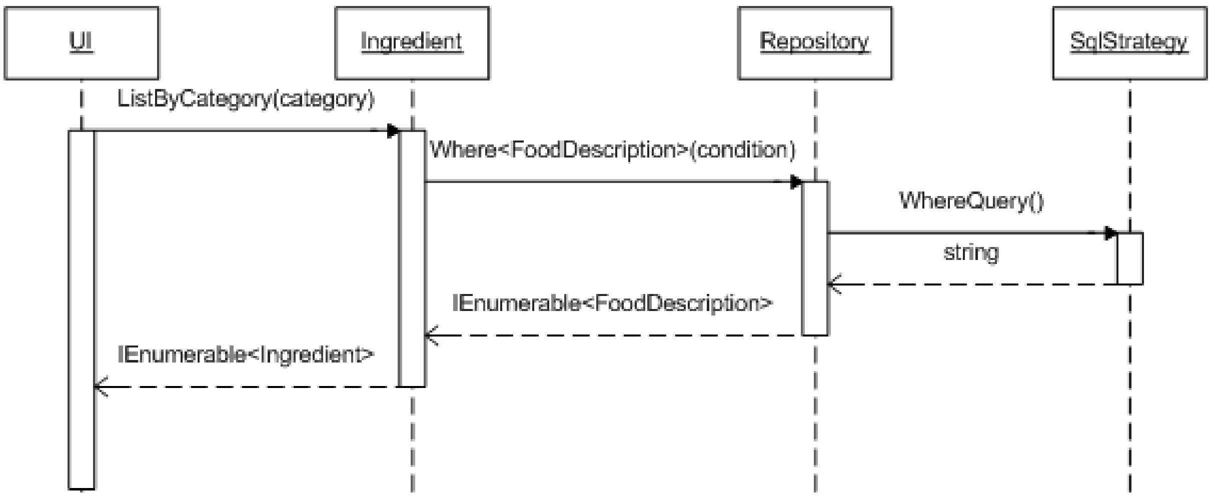


Figure 9.4: Sequence diagram illustrating interaction between classes when listing ingredients by category.

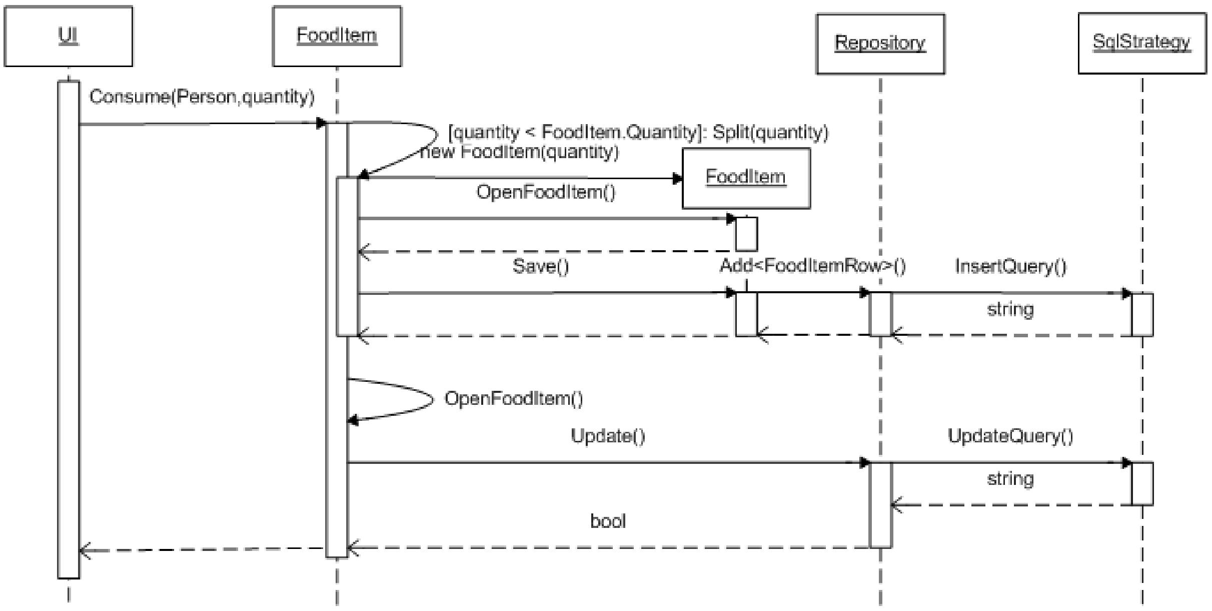


Figure 9.5: Sequence diagram illustrating interaction between classes when consuming a FoodItem.

BarCode

An instance of `BarCode` wraps around a `BarCodeRow` from the model component. It has a `ProductName`, `Identifier` (e.g. bar code number) and a `Type` that is the type of the bar code. The property `Ingredient` lazily loads the `Ingredient` that represents the `FoodDescription` instance, which the underlying `BarCodeRow` refers to. This `Ingredient` defines the contents of a product with the given bar code, the property `Quantity` tells how many grams of this `Ingredient` the product is expected to contain. `BarCode` is responsible for saving and deleting its underlying row if `Save` or `Delete` is called. The `static` method `Find`, finds a matching `BarCodeRow` in the database, creates and returns a `BarCode` instance to represent it.

Recipe

The `Recipe` class is designed to wrap around the `RecipeRow` class from the model component. `Recipe` is reponsible for loading, saving and deleting the underlying row and its relations, such as `RecipeFoodMapping` that defines which ingredients and the amount needed for a recipe. `RecipeStep` defines the step of the recipe, this class is crafted to expose these relations as lists, collections or dictionaries.

The `Recipe` class is also reponsible for handling ratings of a recipe. In the model component these are saved and loaded to/from the database using the `Rating` class. The `Recipe` class exposes these as dictionaries of `Person` and `double` for the rating by the associated user.

Searching Recipes

`Recipe` also includes the static member `ListByCriteria`, which allows the UI to search for recipes, exposing a list of recipes sorted in an order by specifying `Criteria` given by the UI.

The UI lists the recipes after their priority, which is an integer that is incremented every time the recipe matches a search criteria. When searching for keywords, Foodolini checks the title, categories and the recipe description. It will increment the priority if it finds a keyword in either of these. The priority can be incremented three times per keyword, if the keyword is found in all three sections. If the keyword is only found in, for example, the title and description, the priority is only incremented twice.

Foodolini will also prioritise after rating and expiration dates on the ingredient if the two options are selected under the advanced search option. Here the priority is incremented with varying amounts, based on for instance, how high the rating is, or how soon the fooditem will expire. Foodolini will also increment the priority several times, if there are several fooditems which are about to expire. The slide bar which weighs the priority of either rating or expiration dates simply multiplies the value the priority is incremented with the value in the slide bar.

Once `ListByCriteria` has prioritised the individual recipes in the list after the Criteria, it sorts the List, using the following Quicksort algorithm [Cormen, 2009] after priority. Quicksort has a worse worst case time then most other sorting algorithms, however it makes up for this by having an expected running time of $\theta(n \cdot lgn)$.

Algorithm 1 Quicksort(A, start, length)

```
if $start < length$ then
    $q = Partition(ref A, start, length)$
    $Quicksort(A, start, q - 1)$
    $Quicksort(A, q + 1, length)$
end if
Return $A$
```

Algorithm 2 Partition(ref A, start, length)

```
1: $x = A[length]$
2: $i = start - 1$
3: for $int j = start$ to $length - 1$ do
4:     if $A[J] \leq x$ then
5:         $i+ = 1$
6:         Exchange A[i] with A[j]
7:     end if
8: end for
9: Exchange A[i + 1] with A[length]
10: Return $i + 1$
```

ShoppingList

The `ShoppingList` class serves the purpose of holding `ShoppingListItem`s for a specific `Person`. The constructor takes a `Person` and subsequently instantiates a list of `ShoppingListItem`s owned by the provided `Person` from the database. It contains a method to enumerate over the list of items (`GetShoppingListItems`), as well as a couple of mutating methods. Among these methods is `AddIngredient`, which exposes a means of adding a quantity of an `Ingredient` to the list. If the `Ingredient` already exists in the list, the provided quantity will be added to the existing value. Therefore the list is designed to only hold one instance of each `Ingredient`. Similarly, the entire list of `Ingredient`s of a recipe can be added via the method `AddRecipe`.

It is also possible to remove a certain `Ingredient` with `RemoveIngredient`. Finally, the method, `Clear`, deletes all `ShoppingListItem`s in the list.

The class, `ShoppingListItem`, holds the actual data for each individual item in the `ShoppingList`. The `ShoppingListItem` has public properties which can get/set the quantity, `Ingredient` and `Person`. Internally, the `ShoppingListItem` wraps around a `ShoppingListItemRow`. It has one static method, called `GetShoppingListItems`, invoked by the `ShoppingList`, which performs the actual database-lookup for finding items owned by a certain `Person`. This method is `internal`, which ensures that it can only be called via the `ShoppingList` in the front-end.

Person

The `Person` class holds data about a given user of the system and wraps around the `User` class in the model component. The class contains the following information: name, birth date, gender, weight, height and current diet. It has a static method, called `ListUsers`, for enumerating the users of the system. The list of users is cached in a `private static` dictionary after the first call, so that it is not necessary to perform any database-lookups after the first call. When a new `Person` object is created and saved, it is added to this cached dictionary. Likewise, it is removed again, should the `Delete` method be called on the object.

The class also has an instance method for getting performed `Exercises`, named `GetPerformedExercises`. This makes it possible to return a list of `Exercises`. In the same way the class holds a method for listing, which `FoodItems` the person has consumed. It is called `GetConsumedFoodItems`.

Diet

The `Diet` class is a simple representation of a diet. It has a name, calory factor and percentages of protein, carbohydrates and fat. It is simple, since it does not encompass other more complex aspects of a diet, such as allergies, macronutrients, dietary fibre, etc. Though, the level of abstraction used in the class fits the scope of this project.

This class wraps around the `DietRow` class from the model component. It has a single instance method, `GetRequiredCalories`, which calculates how many calories a given person needs. This is based on equation 9.1. In the first term, the maintenance caloric needs are found, or the number of calories needed to neither gain nor loose weight. The second term $caloryFactor$ calculates how many calories to add/subtract from the baseline. For instance, for fat-loss the $caloryFactor$ is -0.2, so 20% of the BMR (Basal Metabolic Rate) is subtracted. The absolute lowest recommended calory deficit for fat-loss is roughly 18 calories per kg, which needs to be taken into consideration [web, 2009].

$$calories = BMR \cdot activityFactor + (BMR \cdot caloryFactor) \tag{9.1}$$

A `Persons` BMR provides an estimate of the resting energy expenditure of a `Person`. The most reliable estimate is calculated by the Mifflin-St Jeor formula shown in equation 9.2. Weight and height are measured in centimetres and age is in years.

$$\begin{aligned} maleBMR &= 10 \cdot weight + 6.25 \cdot height - 5 \cdot age + 15 \\ femaleBMR &= 10 \cdot weight + 6.25 \cdot height - 5 \cdot age - 161 \end{aligned} \tag{9.2}$$

The activity factor corresponds to how active a person is. Common values are listed in table 9.1. The base factor is 1.2 due to the thermic effect of food. In other words, it requires energy to digest food [Roitman and Kelsey, 1994].

Activity factor	Description
1.40 - 1.69	Sedentary or light activity lifestyle
1.70 - 1.99	Active or moderately active lifestyle
2.00 - 2.40	Vigorous or vigorously active lifestyle

Table 9.1: Activity Factors

[FAO/WHO/UNU, 2001]

SportsActivity

`SportsActivity` represents a certain activity that can be performed. It wraps around `SportsActivityRow`, has a name and a MET value. MET stands for Metabolic

Equivalent, which is used to estimate how many calories a `Person` burns throughout the day. MET is the amount of work relative to a `Person`s resting metabolic rate. One MET is defined as 1 kcal/kg/hour, which is approximately the same energy cost as sitting quietly. The MET for a person can be calculated by $BMR/24$. The entire formula for calculating the amount of calories burned by performing a certain activity for a particular duration is shown in equation 9.3. This can also be calculated in minutes by dividing the BMR with the total number of minutes in a day and multiplying with the duration in minutes instead [Roitman and Kelsey, 1994].

$$kcal\,burned = MET \cdot (BMR/24) \cdot duration\,in\,hours \tag{9.3}$$

The class has a static method for listing all activities called `ListActivities`. So, the client is able to select a certain `SportsActivity` when registering an `Exercise`.

Exercise

The `Exercise` class is designed to wrap around the `ExerciseRow` class from the model component. It exposes the `SportActivityId` field on its underlying row as a lazily loaded property called `Activity` that is an instance of `SportActivity`. It also has properties exposing the underlying `UserId` and `Duration` fields, as `PerformedBy` of type `Person` and `Duration` as `TimeSpan`. The `Exercise` class is responsible for saving and deleting its underlying row when `Save` and `Delete` is called respectively.

Apart from a public constructor for creating a new `Exercise` from scratch, Exercise will probably also need an internal constructor used by the `GetPerformedExercises` method on `Person`.

9.4 libzbar-cli

The libzbar-cli component offers C# bindings for ZBar, an unmanaged bar code scanning library.

9.4.1 Structure

ZBar is an open source cross platform C library for reading bar codes from images, video streams and various other sources. ZBar also offers abstractions for interfacing V4L[3] and V4W[4] on Linux and Windows respectively.

The libzbar-cli[5] component is suppose to offer bindings for a subset of ZBar. Though being written in C, ZBar is fairly object oriented, there is even a simple C++ wrapper, header files only, to support this statement. To make good bindings for ZBar requires a few classes that wraps around the unmanaged structures and delegates the logic in their methods to unmanaged functions that operate on these structs.

[3]Video 4 Linux, video capture API for Linux.
[4]Video 4 Windows, video capture API for Windows.
[5]Named according to Debian CLI Policy for GAC libraries.

The wrapper classes in libzbar-cli (see figure 9.6) should implement the IDisposable pattern (See Abrams and Cwalina [2008, Section 9.4]) in order to release their unmanaged resources, or decrement a reference counter on structs that uses reference counting. The wrapper classes should also import the relevant C functions for the struct it wraps as `private static`. The imported method should also, to the extent it makes sense, be exposed as properties or methods on the wrapper class.

ZBar offers both high, mid and low-level interfaces, this component will wrap around the mid-level interfaces as the high-level interface is rather limited in terms of control over the graphical output. The mid-level interfaces offer us direct access to the raw image, while still making it easy to scan and find barcodes in the image. See the sequence diagram figure 9.7 to see how the mid-level abstractions wrapped by libzbar-cli can be used.

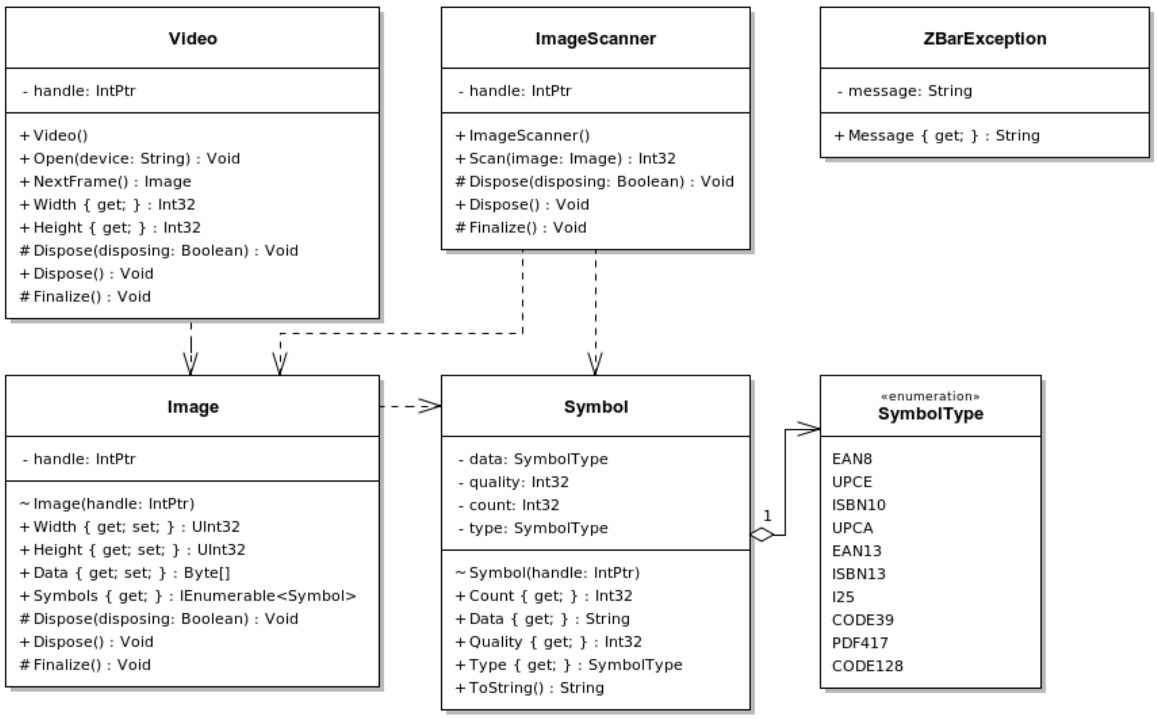


Figure 9.6: Class diagram for libzbar-cli.

9.4.2 Classes

The non-trivial classes in the libzbar-cli component.

Video

The `Video` class wraps around the video interface from ZBar which offers a video source abstraction, that interfaces V4L and V4W on Linux and Windows respectively. That means, an instance of this class shall hold an `IntPtr` to an underlying zbar_video_t [Brow, 2009], and the functions that primarily operates on this struct shall be exposed as instance methods and properties.

The NextFrame method will invoke the unmanaged function for capturing a frame (zbar_video_next_image) and wrap the result, a pointer to a zbar_image_t, in an instance of the Image class, which appropriately has an internal constructor that takes a single IntPtr as parameter.

Image

The Image class wraps around the image interface from ZBar which offers format conversation and bar code association. The Image class will hold an IntPtr to an underlying zbar_image_t and will decrement the reference counter on this structure when it is disposed or finalised.

This class will also make it easy to find all the symboles that is associated with an image. Using the zbar_image_first_symbol function it is possible to get the first symbol, a pointer to an zbar_symbol_t. The next symbol can be found using zbar_symbol_next (The functions will be return a null pointer if no first or next symbol exists). These pointers should be passed to the internal Symbol(IntPtr handle) constructor, and these instances of Symbol should be returned using an IEnumerable<Symbol>.

The Image class will also make the raw image available as byte array, though the Data property. This is a raw image and format may depend upon camera source, thus it is desired that this class also exposes the conversation algorithms the ZBars image interface offers. However, how to do this efficiently is left unspecified for the implementor to decide, and requirements may depend upon GTK drawing abilities.

Symbol

The Symbol class represents a bar code on an instance of the Image class, with a type, data-string, quality and count of how many times the bar code has been detected in a row. While this class has an internal constructor that takes a pointer as parameter, it should not implement the IDisposable pattern and hold a pointer to this symbol, as the data is very limited it is consider easy to just copy it into private managed fields.

ImageScanner

The ImageScanner class wraps around the image scanner interface from ZBar which scans images for bar codes, and associates detected bar codes with the respective images. This class holds a pointer to an unmanaged zbar_image_scanner_t structure and releases it upon being disposed or finalised.

The ImageScanner should have a Scan method that takes an Image instance as parameter. This method should invoke the zbar_scan_image function which takes a pointer to the unmanaged image structure as an argument. How to get this pointer from an instance of Image is left unspecified for the implementor, an internal property or method might do the trick. The ZBar image scanner interface also offers a few settings, how and which of this settings to support is left unspecified for the implementor.

The `zbar_scan_image` function scans an image for bar codes and associates decoded symbols with the image. Thus to scan a camera source for bar codes with this component, a instance of `Video` should be used to capture an image, and an instance of `ImageScanner` should be used to scan the image. Then the `Symbols` property on the image should hold the symbols found. See the sequence diagram figure 9.7.

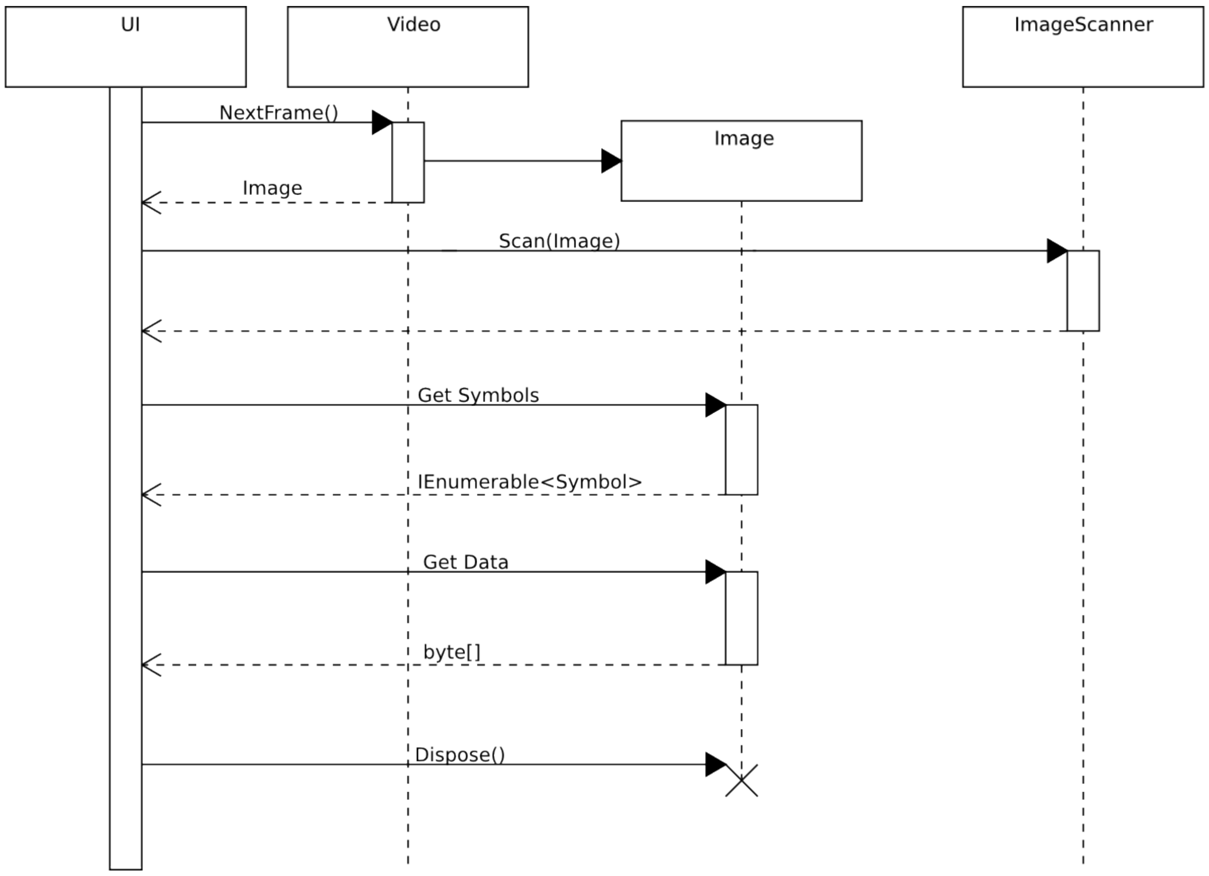


Figure 9.7: Sequence diagram for getting an image and scanning it using libzbar-cli.

ZBarException

The `ZBarException` class wraps around ZBar error codes as a subclass of `System.Exception`, and provides error message. The wrapper classes should, whenever they encounter a ZBar error code, throw this exception with an appropriate error code. Implementation of auxiliary constructors and extra information these error codes might contain is left unspecified for the implementor to decide.

9.5 Main Component

Description of the main component that hosts plug-ins defined in the activities component

9.5.1 Structure

The main window in this component implements the `IOwner` interface of the activities component. And provides functionality to search .dll files for plug-ins at runtime, using reflection. This means that the user interfaces plug-ins or activities that are specified in the activities component need not be implemented in a single binary library, but can implemented as multiple sub-components (or sub-projects) at the discretion of the implementation coordinator.

9.5.2 The Interaction Element

The main window holds the navigation between the different activities. This is done through a tool bar placed in the top of the main window (See figure 9.8 for a mock-up). The navigation bar has a shortcut to the Home page, wherein links to the most used activities are placed. This will give a quick access path for the user, and having a clear starting point for all activities is good for the novice user. The navigation bar also have two selectors consisting of drop down boxes. One - the user selector - contains all the users and an anonymous user from where the current user can be selected. In many of the activities, User Profile for instance, a specific user is needed and that user must be selected from the user selector. The anonymous user is for users who wants to be anonymous or does not have a user account, guest of the households springs to mind. The user selector is placed in the navigation bar, as it must be possible to change the user at any time, and it must be possible to see the selected user at any time. The last part of the navigation bar is the activity selector. This contains all activities - not sub activities - and provides a shortcut to them. Using the activity selector the user can access all activities from any activity or sub activity. Activities are provided as plugins to the main component by the activities component.

The shopping list is also placed in the main window, and is accessible from main at all times. The shopping list is one of the important features in Foodolini, and it can be useful to have quick access to it at all times. Because it is not that useful in every activity, such as register exercise, it can be collapsed or expanded as the user sees fit. The details of the shopping list is explained further in the activities component section 9.6.3.

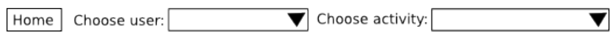


Figure 9.8: The top bar located in main window.

9.6 Activities Component

In this section the navigation between the screens of Foodolini and the functions in each of them will be described. In the section about interaction elements the interaction form, a mock-up and a reference to the relevant use cases will be described for each interaction space.

9.6.1 Structure

This component defines a plug-in interface called `IActivity` (see figure 9.9) and an attribute called `FoodoliniActivityAttribute` that all plug-ins must apply. The `IOwner` interface is intended to be implemented by the main window, and thus offer functionality to the plug-ins. For instance the active plug-in may be notified when the currently selected user changes or request that another plug-in is loaded. If the `PushActivity` method is used, a new instance of the requested plug-in is loaded, and if the `PopActivity` method is called, it will return to the plug-in that was active before `PushActivity` was called. Thus allowing plug-ins to be stacked on top of each other.

The `IActivity` interface defines a `Register` method for passing a reference to an implementer of `IOwner` to the plug-in. The `Unregister` method is used when the plug-in is being closed, in order to remove references to the `IOwner` implementation, including event handlers which usually need to be removed manually. The `FoodoliniActivityAttribute` is used to recognise classes that are plug-ins, and provide the name of the plug-in, which should be displayed in the main window, an icon for the plug-in and lastly a boolean `IsSubActivity` that determines whether or not the activity that this plug-in provides should be available from the activity selector (comobox in main window). Sub-activities are not intended to be available from the activity selector in the main window. Instead sub-activities should be loaded or pushed by other activities. For instance, the ”edit user” activity is available from the ”user profile” activity and from the activity selector in the main window. See a diagram for the interfaces and classes in figure 9.9.

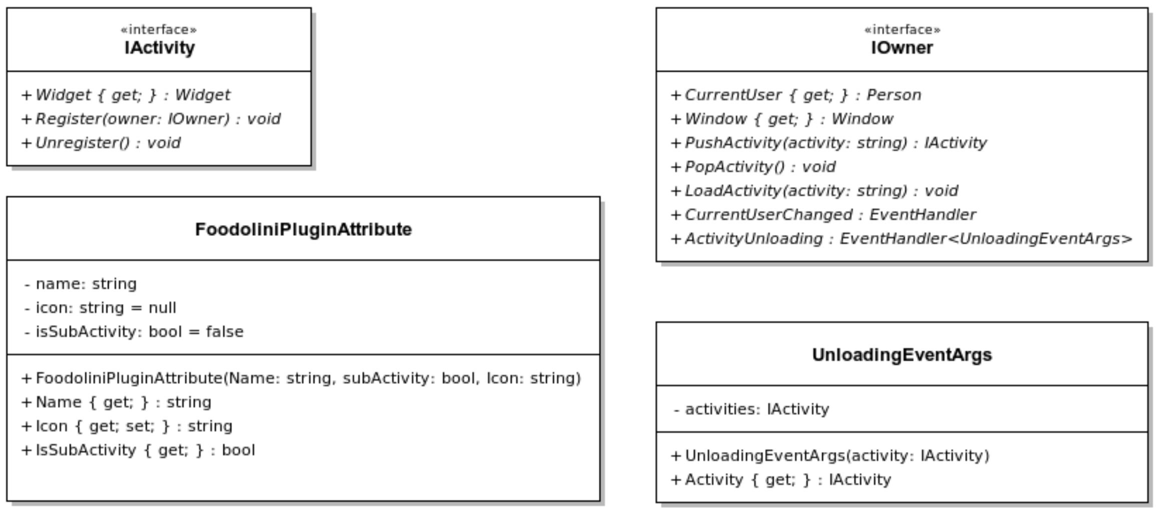

«interface»
IActivity
+ Widget { get; } : Widget
+ Register(owner: IOwner) : void
+ Unregister() : void
«interface»
IOwner
+ CurrentUser { get; } : Person
+ Window { get; } : Window
+ PushActivity(activity: string) : IActivity
+ PopActivity() : void
+ LoadActivity(activity: string) : void
+ CurrentUserChanged : EventHandler
+ ActivityUnloading : EventHandler<UnloadingEventArgs>
FoodoliniPluginAttribute
- name: string
- icon: string = null
- isSubActivity: bool = false
+ FoodoliniPluginAttribute(Name: string, subActivity: bool, Icon: string)
+ Name { get; } : string
+ Icon { get; set; } : string
+ IsSubActivity { get; } : bool
UnloadingEventArgs
- activities: IActivity
+ UnloadingEventArgs(activity: IActivity)
+ Activity { get; } : IActivity

Figure 9.9: Interfaces and classes used for providing plug-ins.

9.6.2 Presentation Model

Figure 9.11 and 9.10 are navigation diagrams for the screens of Foodolini. Activities are placed in the main window and are readily available from the activity-selector in the main window. Sub-activities are either pushed on top of an activity or used in dialogs depending on the context. The usage of dialogues helps the users keep track of where they are, as they can still see the previous screen in the main window. Some of the sub-activities also do not have many details either, and placing them in main would not look very aesthetic.

The shopping list as shown in the mock-ups later can be expanded or collapsed if the user interacts with it through the current activity. The shopping list can also, at any time, be expanded on request of the user. The `ShoppingList` has a button leading to the cookbook, from where a `Recipe` can be added to the `ShoppingList`, as well as a button which opens a dialog containing ingredients, which also can be added to the `ShoppingList`. Thus, you can gain access to these two activities from every screen in the main window. This decision was made to make it easy for the user to add `Ingredient`s or `Recipe`s to the `ShoppingList`, as this feature is expected to be used rather often.

In appendix B full size versions of the mock-ups can be seen.

9.6.3 Interaction Elements

Many of the interaction spaces are either a browser, a viewer or an editor.

The browser is a list of objects and displays the name on the listed objects and additional important information. By clicking an item in the browser, a viewer will appear to the right in the main window, depending on whether the item is designed to be displayed in this manner. By double-clicking an item an editor will appear, enabling information to be edited and saved. An example of a browser can be seen in figure 9.12, displaying a browser listing all users. A list is very useful to get a quick overview of primary information of several objects, and is hence an effective way to display little information about many objects. The browser always contains a 'Create New' button below the list of objects, and is intended for the user to create a new object if not found in the browser list.

The viewer only prints out data, and is not editable. When the need for printing data arises, it is important to do this in a manner so the user will know that the data is merely printed and not editable. The viewer is always opened to the right side of the browser, when an object in the browser is clicked. This gives the user quick and effortless access to detailed information of an object. For all objects the name will be printed in the left upper corner of the viewer and the details will follow below. How the information is displayed depends on the type of object to be viewed, and accordingly the details available. At least two buttons are placed in the bottom of the viewer; a delete-button and an edit-button. The delete-button is for deleting an object from the database, which as a result closes the viewer and removes it from the browser. The object and all of its details will be deleted from the system. The edit-button activates the editor and is used when any details of an object need to be edited. An example of a viewer can be seen in figure 9.13, which displays a `Recipe`.

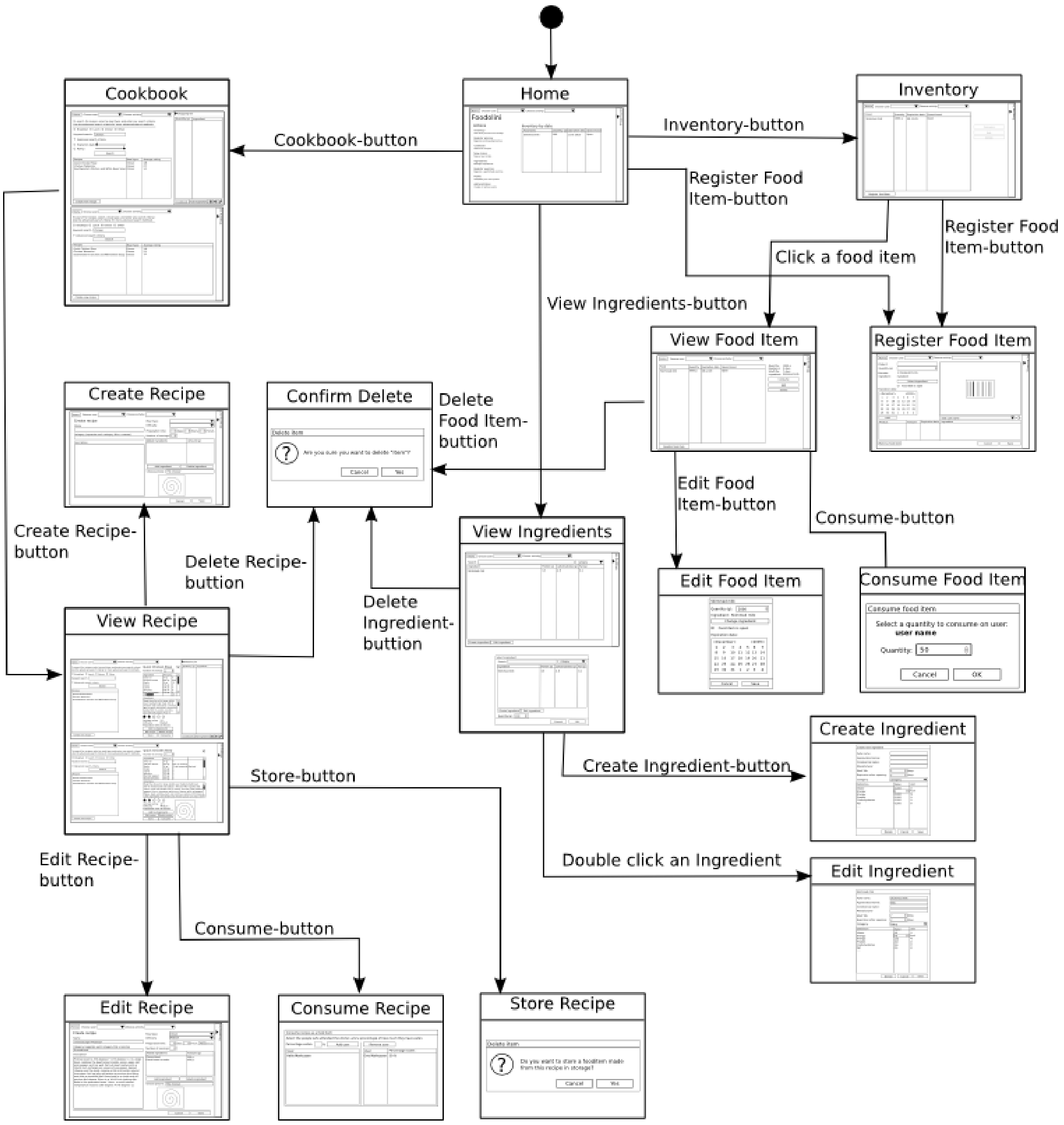


Figure 9.10: Navigation between the screens for Administration and User Profile.

To maintain consistency the editor has, when possible, the same structure as the viewer; the name always is in the upper left corner and detailed information will follow below in the same order as in the viewer. The editor consists of text entry fields since the information may vary a lot. In addition spin buttons and combo boxes may be used, when the user has to enter a number or select an option. For example, the shelf life will be entered using a spin button, which only allows digits in a specified range to be entered, thus preventing the user from entering invalid data. Two buttons are placed in the lower right corner of the editor: a save-button and a cancel-button. The save-button is always placed to the right of the cancel-button. An editor for a

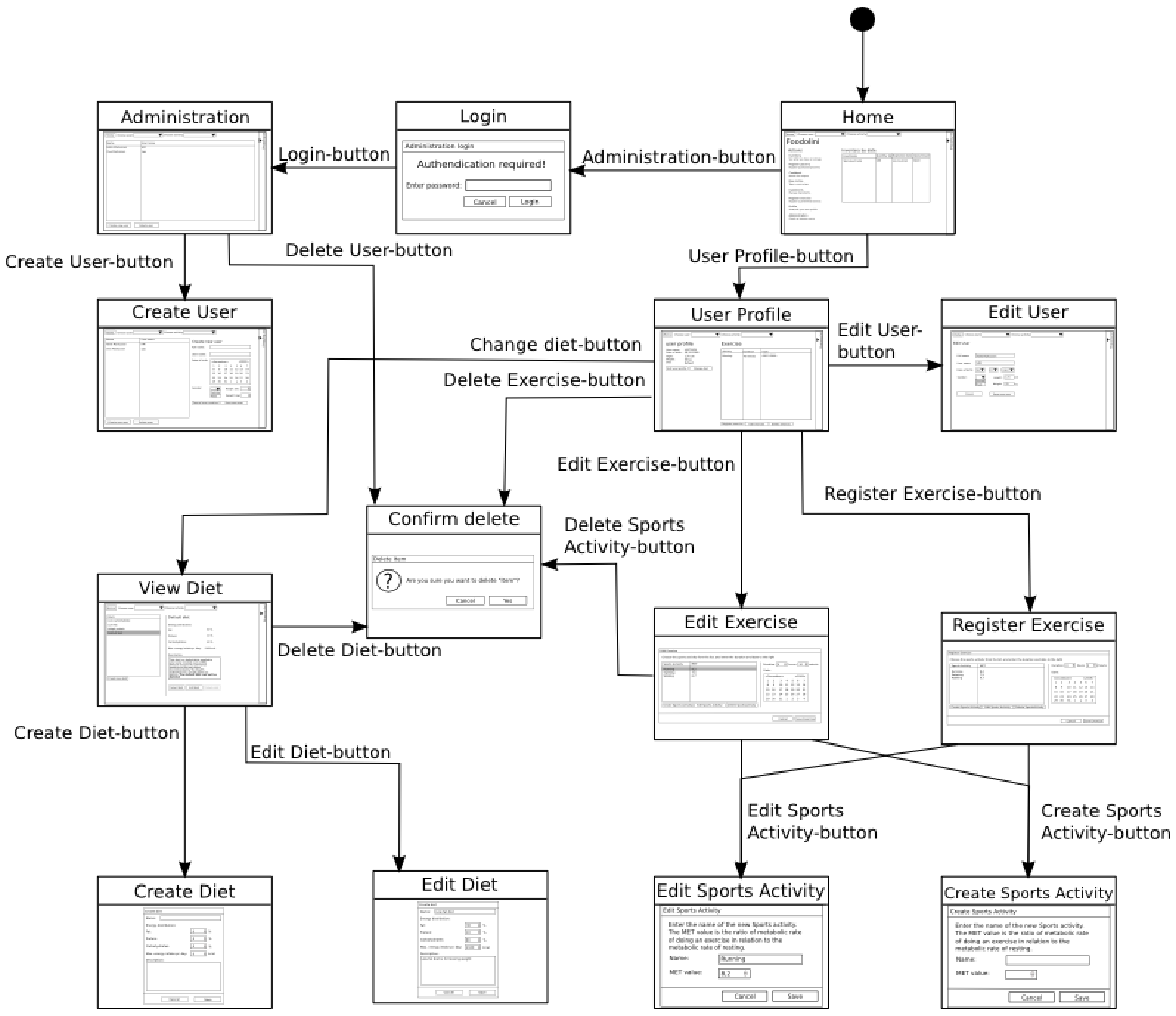


Figure 9.11: Navigation between the screens for Cookbook, Inventory, and View Ingredients.

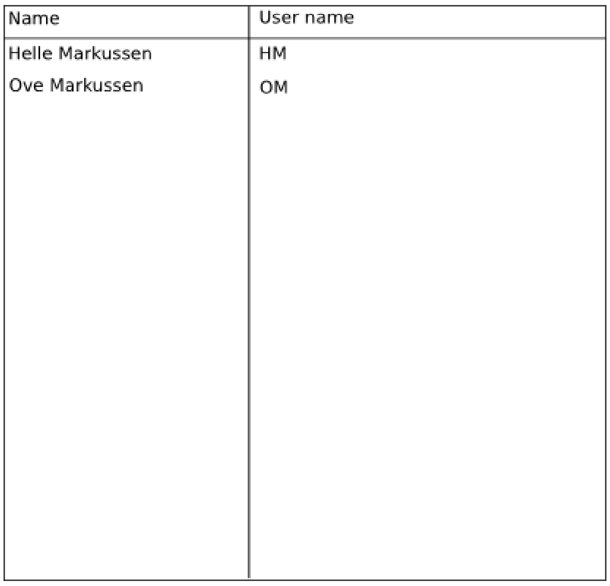
`FoodItem` can be seen in figure 9.14.

As previously mentioned it is possible to create new objects in the browser. The interaction space for creating a new object is the same as in an editor, except for the fact that the text entry fields will be empty initially. Figure B.21 for editing a `Diet` and figure B.20 for creating a `Diet` shows the similarities and differences.

The individual interaction spaces will be explained below, with some reference to the above-mentioned.

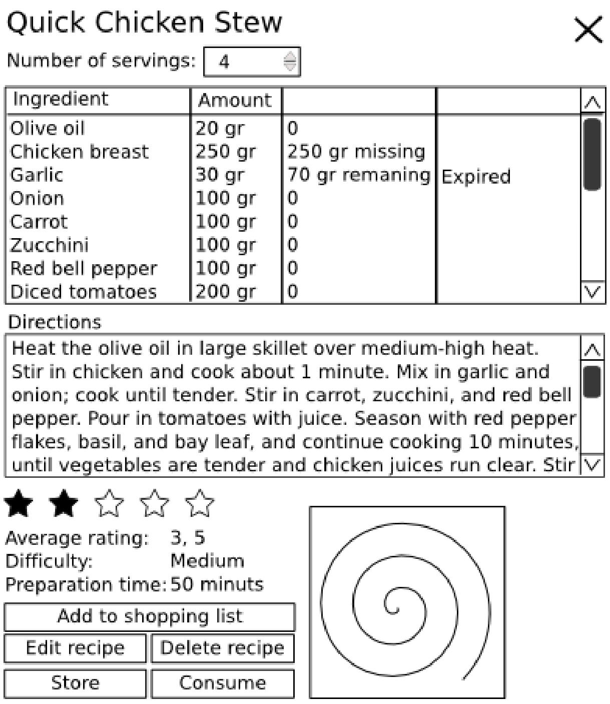
ShoppingList

A mock-up of the `ShoppingList` can be seen in figure B.18 and a print of it is showed in figure B.19. As the `ShoppingList` is useful in many places it is, as described earlier, extendable and collapsable, and both versions can be seen in the figure. The `ShoppingList` consists of the ShoppingListViewer, which primarily consists of a



Name	User name
Helle Markussen	HM
Ove Markussen	OM

Figure 9.12: User browser



Quick Chicken Stew

Number of servings: 4

Ingredient	Amount		
Olive oil	20 gr	0	
Chicken breast	250 gr	250 gr missing	
Garlic	30 gr	70 gr remaning	Expired
Onion	100 gr	0	
Carrot	100 gr	0	
Zucchini	100 gr	0	
Red bell pepper	100 gr	0	
Diced tomatoes	200 gr	0	

Directions

Heat the olive oil in large skillet over medium-high heat. Stir in chicken and cook about 1 minute. Mix in garlic and onion; cook until tender. Stir in carrot, zucchini, and red bell pepper. Pour in tomatoes with juice. Season with red pepper flakes, basil, and bay leaf, and continue cooking 10 minutes, until vegetables are tender and chicken juices run clear. Stir

★★☆☆☆

Average rating:	3, 5
Difficulty:	Medium
Preparation time:	50 minuts

Figure 9.13: Recipe viewer

simple version of the IngredientBrowser, where only the name can be viewed. The ShoppingListViewer contains all the `Ingredients` that have been added to the `ShoppingList`. Buttons to access the cookbook activity and ingredients activity are placed in the bottom, to ease the task of adding `Ingredients` to the `ShoppingList`. There are also buttons for removing one item, and clearing the entire `ShoppingList`. Finally, the `ShoppingList` can be exported to a text file or be printed. Figure B.19 dis-

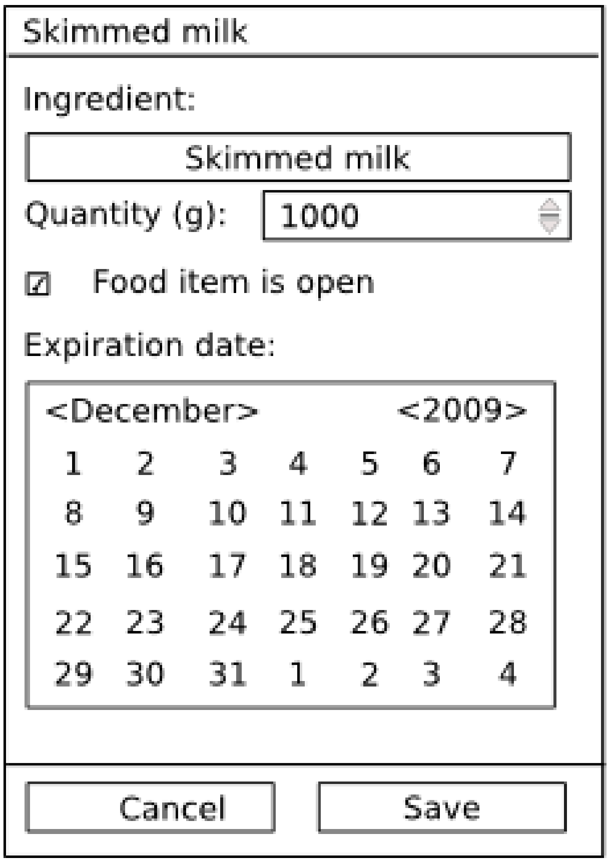


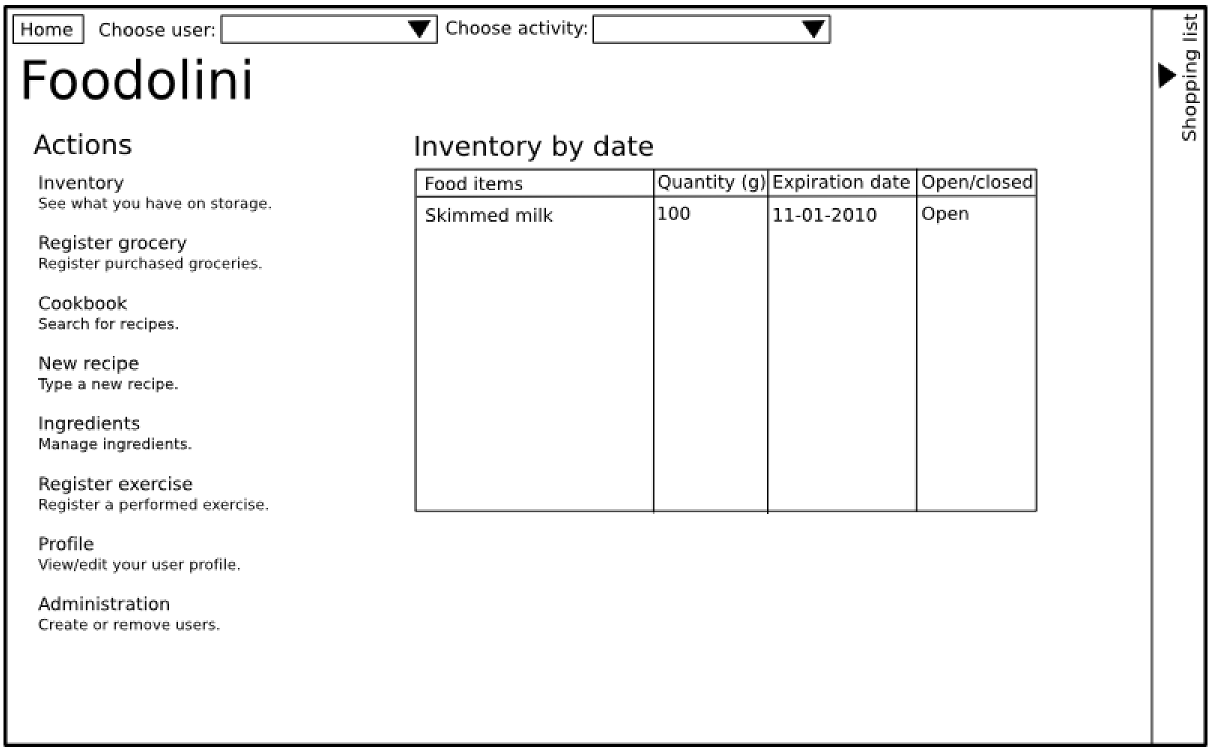
Figure 9.14: FoodItem editor

plays an example of the print result. The printed version makes it possible to take the `ShoppingList` along when buying groceries. The ShoppingListViewer is used when viewing the `ShoppingList`, as well as adding a `Recipe` too, removing an item, and clearing the `ShoppingList`.

Home Activity

The Home activity contains a FoodItemBrowser, listing the `FoodItems` currently in the inventory, including buttons for commonly used functions in Foodolini. Figure 9.15 is a mock-up of the Home activity.

The FoodItemBrowser holds information about the name of the `Ingredient`, the quantity, expiration date, and whether it is open or closed. This is the most important information regarding the `FoodItems`. In contrast to the other browsers, this browser is not clickable, nor does it open a viewer, since it would be out of place in the Home activity. The FoodItemBrowser is used in the use cases for Consume `FoodItem`, Delete `FoodItem`, Edit `FoodItem`, Register `FoodItem` and View `FoodItem`.



Actions

Inventory
See what you have on storage.

Register grocery
Register purchased groceries.

Cookbook
Search for recipes.

New recipe
Type a new recipe.

Ingredients
Manage ingredients.

Register exercise
Register a performed exercise.

Profile
View/edit your user profile.

Administration
Create or remove users.

Inventory by date

Food items	Quantity (g)	Expiration date	Open/closed
Skimmed milk	100	11-01-2010	Open

Figure 9.15: Mock-up of Home.

Login

The login dialog can be seen in figure B.1 and is used for accessing the Administration activity. The login dialog is activated when clicking the Administration button from the Home activity or in the activity selector. Login is required for the use cases for creating and deleting a `Person`, to make sure these functions are only used by an administrator. The password is entered by the user in a masked textentry.

Administration Activity

A mock-up of the Administration activity can be seen in figure B.2 and B.3, and contains a PersonBrowser and a PersonEditor for creating a new user.

The PersonBrowser is a list of all the registered users in the household. The PersonBrowser lists the full name and username. There are two buttons below the list of users; a Create new user button and a Delete user button. The Delete user button activates the ConfirmDialog, which prompts for a confirmation from the user (See figure B.25). The Create user button activates the PersonEditor. The PersonBrowser is related to the use cases, Create `Person` and Delete `Person`. The ConfirmDialog is used for clearing and removing an item from the `ShoppingList`, deleting a `Person`, `FoodItem`, `Ingredient`, `Recipe`, `Diet`, `Exercise` and `SportsActivity`.

The PersonEditor is a form for entering information about the user, which includes the full name, username, date of birth, gender, height and weight. These data are used in relation to exercises, with the objective of calculating the number of calories burned.

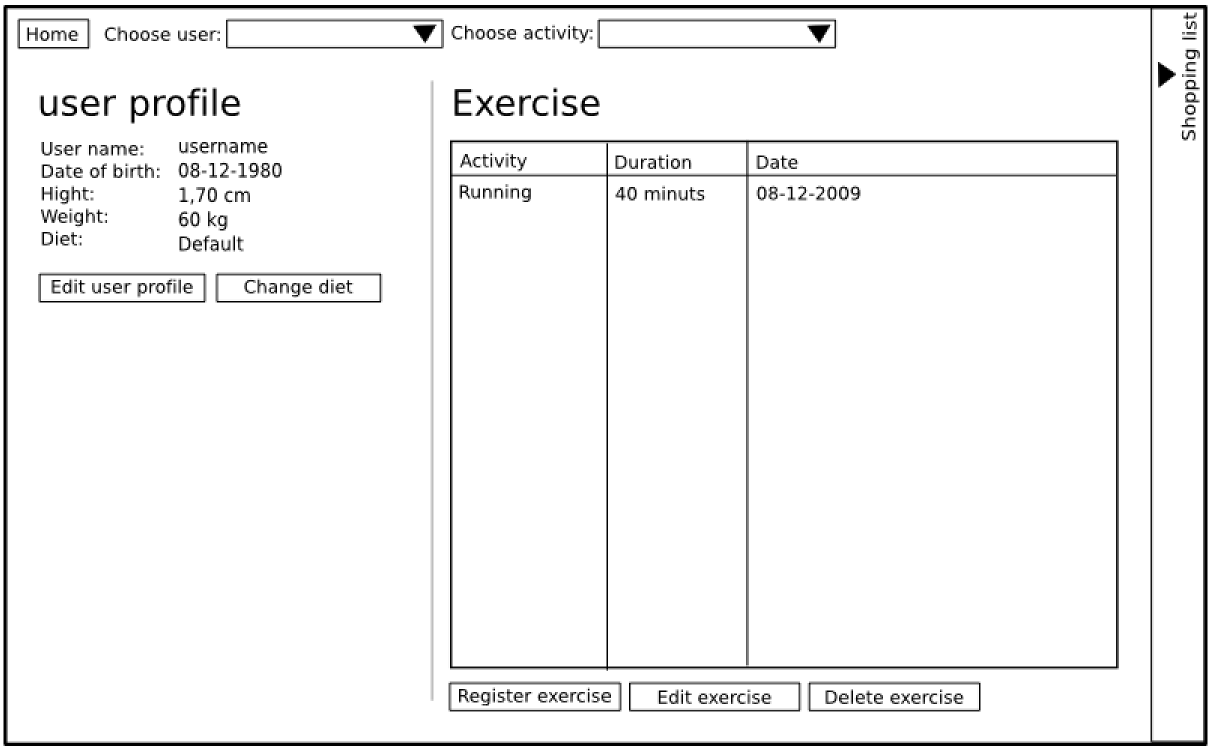
The use cases related to PersonEditor are Edit `Person` and Create `Person`.

User Profile Activity

A mock-up of the User profile activity can be seen in figure 9.16, which contains a PersonViewer and an ExerciseBrowser.

The PersonViewer is positioned to the left and shows the full name, username, date of birth, height and weight of the user, along with the selected `Diet`. The `Person` to be viewed is selected in the User selector located in the top bar. If [Anonymous] is selected, nothing is shown, except the full name and username which are ”Anonymous”. Also, the edit button can not be clicked. By clicking the Change Diet-button the Diet administration activity (See section 9.6.3) will open. The PersonViewer is used in the use cases for viewing, editing and deleting a `Person`.

The ExerciseBrowser is a list of all the exercises registered by that user, and displays the names of the performed `SportsActivities` and their respective duration and performed date. The ExerciseBrowser is placed on the right half of the screen and contains three buttons; one for deleting, one for editing and one for registering an `Exercise`. The edit button opens an ExeciseEditor, and the delete button, deletes the marked `Exercise` from the list and from the users profile. The ExerciseBrowser is related to the use cases, register `Exercise`, edit `Exercise`, delete `Exercise` and view `Exercise`.



user profile

User name:	username
Date of birth:	08-12-1980
Hight:	1,70 cm
Weight:	60 kg
Diet:	Default

Exercise

Activity	Duration	Date
Running	40 minuts	08-12-2009

Figure 9.16: Mock-up of User Profile.

Edit user activity

Creating and editing a user involves the `Person`. A mock-up can be seen in figure B.4. The PersonEditor uses a table to change the information of a `Person`.

This PersonEditor is used to to edit the full name and username, date of birth, weight, height and gender, which is all of the data about a user. The use cases related to PersonEditor are Edit `Person` and Create `Person`.

Cookbook activity

The Cookbook activity contains a CriteriaSelector, a RecipeBrowser and a RecipeViewer and can be seen in figure B.5, B.6 and 9.17. Figure B.6 is with the shopping list collapsed and figure 9.17 with it expanded.

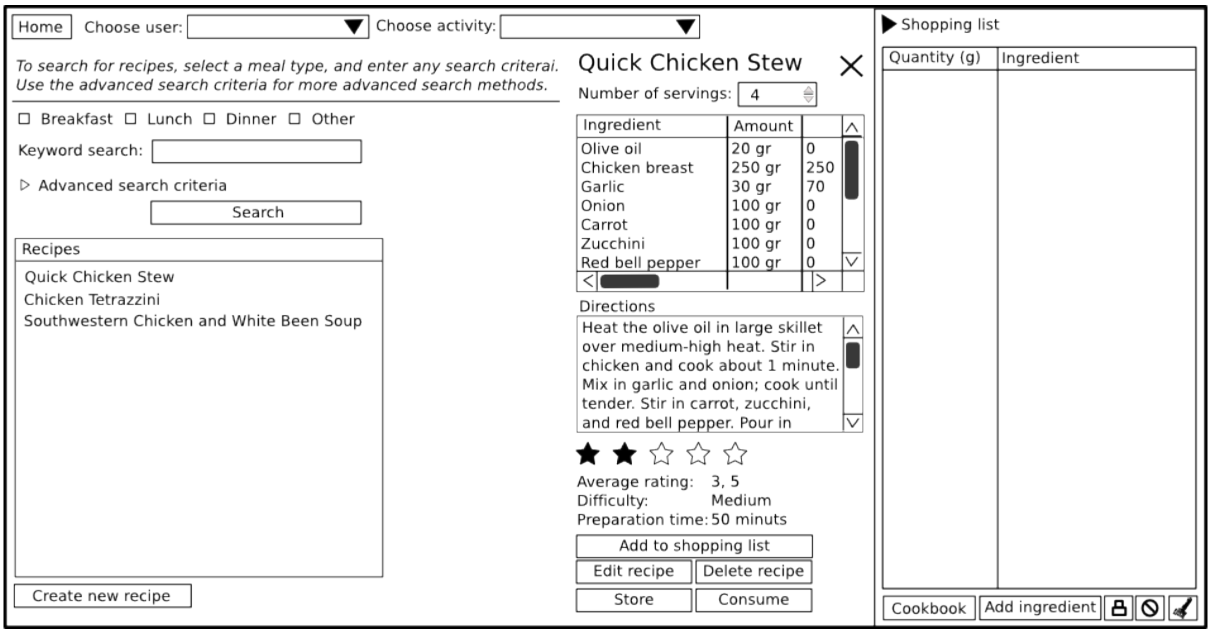
The CriteriaSelector is placed in the top of the screen and is used for searching for `Recipe`s. It consists of a search field, check and scale buttons, for advanced searching, which takes the expiration date and rating into consideration. These search criteria can be used as the user sees fit, and the result will be displayed in the RecipeBrowser. Search fields are widely used to allow the user to search. The users can search for any word in the title, category or directions, however, words with fewer then three characters are ignored. The check buttons can be used to limit the returned meal types, for instance, if only `Recipe`s for dinner are desired. The advanced search criteria can be used to prioritise the recipes found. If expiration date is checked, the `Recipe`s with `Ingredient`s in storage close to expiration date will be ranked higher than those that uses ingredients that are far from their expiration date. The scale can be used to set the importance of the criteria.

The CriteriaSelector is used when searching through `Recipe`s, which is done in the following use cases: Add `Recipe` to `ShoppingList`, Create meal from `Recipe`, Rate `Recipe`, Delete `Recipe` and View `Recipe`.

The RecipeBrowser is a list of `Recipe`s, where the user can open a RecipeViewer by clicking one of the `Recipe`s in the list. The RecipeBrowser shows the name of the `Recipe`, the type of meal and the average rating, as these are the most useful facts about a `Recipe` in this case. The RecipeBrowser is used every time a `Recipe` is to be viewed. The same use case applies to the CriteriaSelector.

The RecipeViewer displays the details of a `Recipe` to the right in the main window when a recipe is selected from the RecipeBrowser. The RecipeViewer lists the `Ingredient`s, the quantities needed, the total remaining quantities in the inventory as `FoodItem`s and whether this `FoodItem` is expired. The number of servings can be changed in the MultiplierSelector, which will change the quantities of the `Ingredient`s. This gives a good overview of whether the needed `FoodItem`s are at hand. The MulitiplierSelector is used when viewing a `Recipe`, adding a `Recipe` to `ShoppingList`, creating a meal from a `Recipe` and rating a `Recipe` At the bottom is the average rating, difficulty and preparation time. It is possible for the selected `Person` to give a new rating by using the RecipeRater.

The RecipeRater consists of five stars, which can be clicked, thereby giving a new rating. The RecipeRater is visible at all times. Buttons for adding the `Recipe` to the `ShoppingList`, editing and deleting the recipe, storing and consuming the `Recipe` as a `FoodItem` are placed below the RecipeRater.



To search for recipes, select a meal type, and enter any search criterai. Use the advanced search criteria for more advanced search methods.

Recipes
Quick Chicken Stew
Chicken Tetrazzini
Southwestern Chicken and White Been Soup

Quick Chicken Stew

Ingredient	Amount	
Olive oil	20 gr	0
Chicken breast	250 gr	250
Garlic	30 gr	70
Onion	100 gr	0
Carrot	100 gr	0
Zucchini	100 gr	0
Red bell pepper	100 gr	0

Directions

Heat the olive oil in large skillet over medium-high heat. Stir in chicken and cook about 1 minute. Mix in garlic and onion; cook until tender. Stir in carrot, zucchini, and red bell pepper. Pour in

Average rating:	3, 5
Difficulty:	Medium
Preparation time:	50 minuts

Quantity (g)	Ingredient

Figure 9.17: Mock-up of View Recipe.

Creating and Editing a Recipe

The RecipeEditor is used for creating and editing a `Recipe`. It can be seen in figure B.7 and B.8, respectively.

The user can enter/edit all the details of a `Recipe` in entries, spin buttons and combo boxes. The `Recipe` data includes a name, categories, directions, meal type, difficulty, preparation time, number of servings, ingredients, and a picture. A `Recipe` requires a unique name, directions and `Ingredient`s assigned to it. The RecipeEditor is used both to create and edit: This makes the interface more consistent, because the user knows where to find an entry. The RecipeEditor is therefore used in the use cases for creating and editing a `Recipe`.

Consume and Store Recipe as a FoodItem

Consuming and storing a `FoodItem` made from a `Recipe` uses a PersonSelector and a ConfirmDialog as can be seen in figure B.9 and B.10, respectively.

The PersonSelector opens up in dialog from which the user can select the persons who consumed the meal from a list of all users, and add them to another list with the percentage they have consumed. The ConfirmDialog only asks for a confirmation to store this as a `FoodItem` in the inventory. The two interaction spaces are used in the same use case as when creating a meal from a `Recipe`.

Inventory Activity

The Inventory activity contains a list of all `FoodItems` in the inventory in a FoodItem-Browser, and a FoodItemViewer with more detailed information. A mock-up of the Inventory activity can be seen in figure B.11 and B.12.

The FoodItemBrowser is the same as the one mentioned in section 9.6.3 about Home. It is a list of all `FoodItems`, with a quantity, expiration date and whether it is opened or closed. However, this one is clickable which enables editing of the selected `FoodItem`.

The FoodItemViewer displays the name, which is the name of the `Ingredient`, the quantity, when it expires and the shelf life. Buttons for consuming, editing and deleting the `FoodItem` are placed below.

The FoodItemBrowser and -Viewer are used when consuming, deleting, editing and viewing a `FoodItem`. The FoodItemBrowser is also used for registering a `FoodItem`.

Register FoodItem Activity

Registering a `FoodItem` involves the BarCodeSelector, FoodItemEditor and Food-ItemBrowser, and can be seen in figure 9.18.

The BarCodeSelector consists of the video feed in the top right corner and is used to scan the bar code of the products. The BarCodeSelector is simply a video feed, streaming frames from the web cam, so that the user can see where the bar code is in the image. If the bar code is successfully scanned, a visual and audible feedback is communicated to the the user. The BarCodeSelector is only used when registering a new `FoodItem` (See use case for registering a `FoodItem`).

The FoodItemEditor is used to get the details of the `FoodItem`. If the bar code is recognised, the information is automatically filled, however, the expiration date must often be adjusted. The `Ingredient` can be selected from a IngredientBrowser in a dialog. The FoodItemEditor is used when registering and editing a `FoodItem`.

When the add button in the FoodItemEditor is selected, the `FoodItem` is added to the FoodItemBrowser in the bottom of the screen. The FoodItemBrowser contains the `Ingredient` name, the quantity, the expiration date, but not whether it is opened or closed, as this is not relevant here. A button at the bottom can be used to remove undesired `FoodItems`.

Edit FoodItem

Editing a `FoodItem` uses the FoodItemEditor and can be seen in figure B.13. This is the same as the FoodItemEditor used when registering a `FoodItem`.

Consume FoodItem

A mock-up for consuming a `FoodItem` can be seen in figure B.14, which contains the QuantitySelector. This allows the user to select how much to consume in grams with a spin button, giving the user a quick way to enter the consumed quantity. The QuantitySelector is used when consuming a `FoodItem`, creating and editing a `Recipe`.

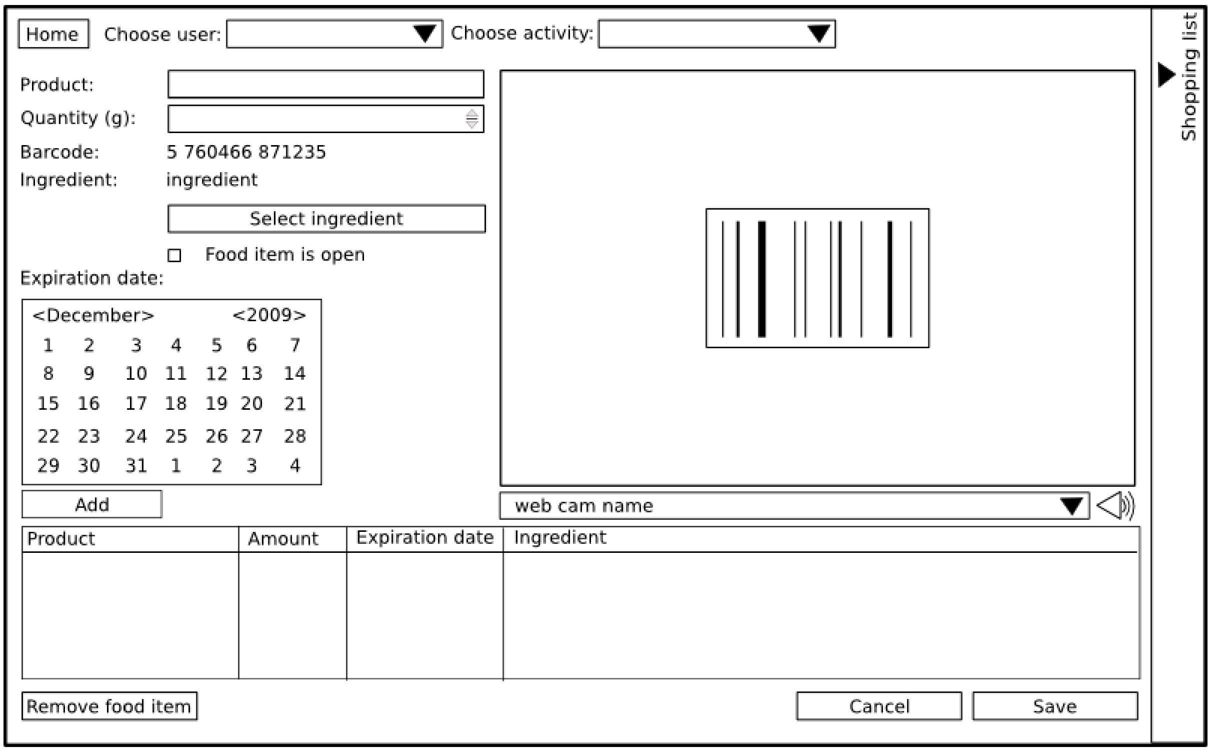


Figure 9.18: Mock-up of Register Grocery.

Ingredients Activity

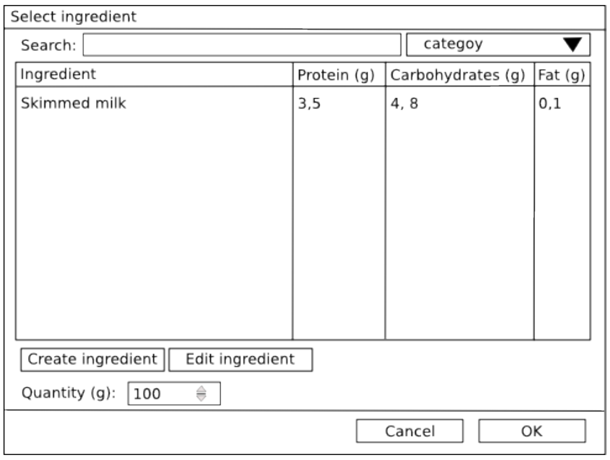
The `Ingredients` activity only consists of an IngredientBrowser and can be seen in figure B.15 and 9.19.

The IngredientBrowser lists all the `Ingredients` and displays their names, total protein, carbohydrates and fat per 100 grams. Since the system holds several thousand `Ingredients`, a search function has been added to the IngredientBrowser which can search in a specific category. This will make it easier and faster for the user to find the desired `Ingredient`. As with the other browsers, it is possible to create a new `Ingredient` using the button in the bottom of the screen.

In figure 9.19 the IngredientBrowser is placed in the main window as well as in a dialog. This is because the IngredientBrowser is used in many situations. Having it open the in main window, on top of the other screens, could cause the user to lose sight of their objective. Some users might think the `FoodItems` were deleted. In addition, it would be preferred not to have to load and close the video scanner every time, from a programmer's point of view.

The dialog version has a QuantitySelector which is placed below the Ingredient-Browser, as in the figure. When using the IngredientBrowser to select an `Ingredient` for a `FoodItem` during the Food Registration, the QuantitySelector is not displayed. This is because the `Ingredient` is often already found when registering a `FoodItem` and it would be tedious and misleading if the user was forced to go to the Ingredient-Browser in order to assign the quantity. On the other hand, it makes sense to place the `QuantitySelector` in the `IngredientBrowser`, when adding an `Ingredient` to the `ShoppingList`.

The QuantitySelector is used when consuming a `FoodItem`, creating and editing a `Recipe`, as described above. The IngredientBrowser is used when creating, editing and deleting an `Ingredient`, when creating and editing a `Recipe`, and finally when registering and editing a `FoodItem`.



Ingredient	Protein (g)	Carbohydrates (g)	Fat (g)
Skimmed milk	3,5	4, 8	0,1

Figure 9.19: Mock-up of Select Ingredient.

Create and Edit an Ingredient

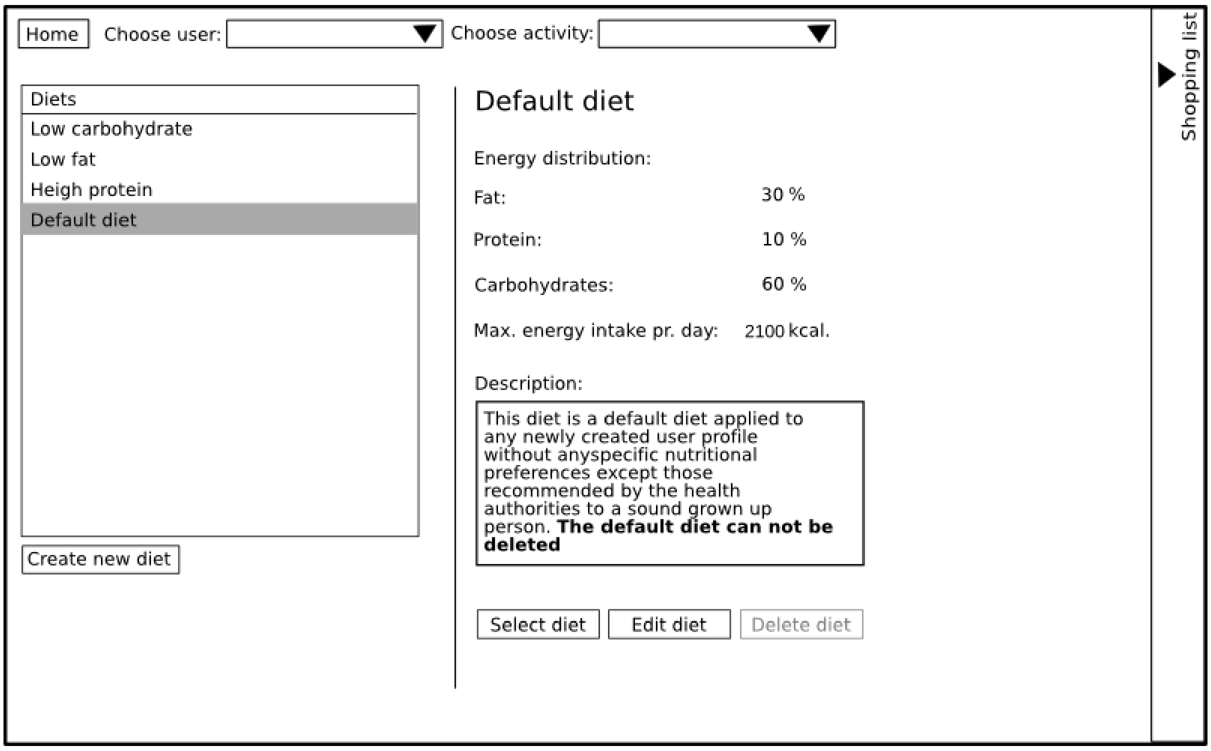
Creating and editing an `Ingredient` uses the same IngredientEditor. The system does not have an IngredientViewer, but only uses the editor. This is because you would rarely want any further information other than the name, amount of proteins, carbohydrates and fat, which are all shown in the browser. The IngredientEditor uses entries, spin buttons and combo boxes to display and assign the information. To add a nutrient, the quantity of that nutrient is double clicked, which changes it into a spin button, where the quantity can be assigned. This is a very compact, inline and simple way of increasing or decreasing nutrients in an `Ingredient`. The IngredientEditor is used when creating and editing an `Ingredient`, registering and editing a `FoodItem`, and when creating and editing a `Recipe`.

Administrate Diets Activity

A mock-up of the Administrate diets activity can be seen in figure 9.20, which contains a DietBrowser and a DietViewer.

The DietBrowser is located to the left, listing all `Diets` by their names. When a `Diet` is clicked, the DietViewer to the right changes to display the selected `Diet`. This makes is easy to search through the `Diets` and find the preferred one. The browser also has a button for creating a new `Diet` at the bottom that opens the DietEditor. The DietBrowser is used when choosing, creating, editing, viewing and deleting a `Diet`.

The DietViewer displays the details of the `Diet`, which are: energy distribution in proteins, carbohydrates, fat, the maximal energy intake per day, and a short description of the `Diet`. The `Diet` can be can be edited, or selected, as well as deleted via the viewer, except the default diet cannot be deleted. Deleting a `Diet` must be confirmed in a dialogue, as described earlier, and editing will open a DietEditor. Selecting the `Diet`, will change the `Diet` of the selected user. The DietViewer is used when choosing, editing and viewing a `Diet`.



Default diet

Energy distribution:

Fat:	30 %
Protein:	10 %
Carbohydrates:	60 %
Max. energy intake pr. day:	2100 kcal.

Description:

This diet is a default diet applied to any newly created user profile without anyspecific nutritional preferences except those recommended by the health authorities to a sound grown up person. **The default diet can not be deleted**

Figure 9.20: Mock-up of Administrate Diets.

Create and Edit a Diet

Creating and editing a `Diet` both use the DietEditor. Mock-ups of them can be seen in figure B.20 and B.21 respectively. The DietEditor contains a text entry field for the name of the `Diet`, and four spin boxes, three of them ranging from 0 to 100 percent to enter the energy distribution between fat, protein and carbohydrates and the fourth spin box is to enter the maximal calory intake per day. Moreover, it contains a text entry field where a short description of the `Diet` may be entered optionally. Lastly there is a cancel and save button located at the bottom of the editor. If the user clicks the cancel button, the DietEditor is closed, and the information is not saved. If the save button is clicked, the user returns to administrate `Diet`s after saving the details. Use cases related to the DietEditor are: Choose `Diet`, Create `Diet` and Edit `Diet`.

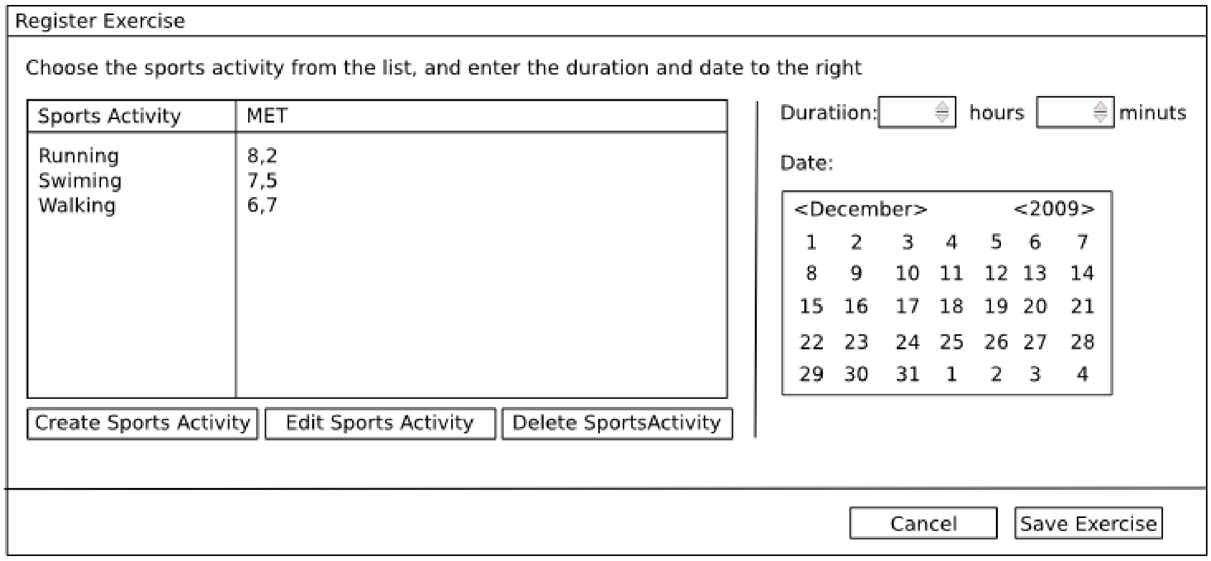
Register and Edit Exercise

Registering and editing an `Exercise` uses the previously described ExerciseEditor, which can be seen in figure 9.21 and B.22, correspondingly.

The ExerciseEditor is divided into a left and right side. The left side contains the SportsActivityBrowser, listing all `SportsActivity`s. In order to associate a `SportsActivity` with an `Exercise`, a `SportsActivity` is selected. Below the SportsActivityBrowser are three buttons: Create `SportsActivity`, Edit `SportsActivity` and Delete `SportsActivity`. The edit and create buttons open up the SportsActivityEditor and the delete button activates the ConfirmDialog, described earlier. The right side of the ExerciseEditor contains an entry field for entering the duration of the `Exercise` and a calendar for selecting the date of the performed `Exercise`. Two buttons are placed in the bottom of the right corner of the ExerciseEditor; Cancel and Save `Exercise`. The cancel button relays the user back to the ExerciseBrowser. The Save `Exercise` button saves the exercise to the user's list of performed `Exercise` and returns to the ExerciseBrowser.

Placing the SporstActivityBrowser within the ExerciseEditor is done because the `SportsActivity` is an integrated part of the `Exercise` class. In addition, it also makes the ExerciseEditor look more balanced.

The ExerciseEditor is used when creating and editing an `Exercise`.


Register Exercise

Choose the sports activity from the list, and enter the duration and date to the right

Sports Activity	MET
Running	8,2
Swiming	7,5
Walking	6,7

Duratiion: hours minuts

Date:

<December>						<2009>
1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29	30	31	1	2	3	4

Figure 9.21: Mock-up of Register Exercise.

Create and Edit a SportsActivity

Creating and editing a `SportsActivity` is activated in the ExerciseEditor, and uses the SportsActivityEditor. A mock-up of creating and editing a `SportsActivity` can be seen in figure B.23 and B.24, respectively.

The SportsActivityEditor is a form, consisting of two text entry fields, one for entering the name of the `SportsActivity`, the other for entering the MET value, as well as two buttons. One is for saving the form and the second for cancelling the data entered. The SportsActivityEditor is used in the following use cases: Create `SportsActivity`, Edit `SportsActivity` and Delete `SportsActivity`.

CHAPTER 10

Evaluation Plan

The main purpose of evaluating Foodolini is to find out whether it will be possible for people like those described in the personas (See section 3.2) to use this system efficiently. As this system is to be used on an everyday basis for things that you do not want to spend much time on, it has to be very efficient to use. This means that the user should be able to scan groceries quickly and easily, and the system should have a low learning curve for daily use. The use of the functionality should also be clear and be easy to find.

Effectiveness is hard to measure, but the time it takes to conduct a use case, and the number of problems found when doing so, can give an estimate on the effectiveness and usability. As the test-subjects will be asked to register a high number of groceries, we will be able to see if this function becomes easier with experience. Nevertheless, the main indicator will be the test-subjects themselves. After the test, the test-subjects will be asked to fill out a small questionnaire on their experience with the system. The questionnaire will be about how easy the functionality was to find, how easy it was to understand what the functions did, etc.? For every question the test-subject will be asked to give examples.

The test-subjects must know how to operate a computer on a basic level. They should use a computer on an every day level for emailing and for searching information, but they do not have to be experts. The system can have a very wide variety of users, and the demands for usability are therefore high. Because the main goal of this project is to learn how to analyse and design a system, we have chosen to lower the demands on usability, and heighten the demands on the users' computer capabilities.

To evaluate the system, three test-subjects will be used. This number is high enough that some usable data should be produced, but low enough that the IDA method will be efficient to use for inexperienced usability evaluaters. The test-subjects are found through acquaintances of the group members. Two are males at the age of 21 and students at Aalborg University, and one is a female at the age of 37, married, and has two children. All test-subjects have the desired computer experience mentioned above. They will fulfil the personas from the analysis, and is in our target group.

10.1 Method of Evaluation

There are many different methods of evaluating a system, where usability tests in a laboratory, field studies, or getting experts opinion, are but some. The evaluation of this system will be done as a usability test in the field. The test will be carried out in the flat of one of the group members, which will make the scenario more realistic as the assignments will be carried out in a realistic environment. As oppose to doing the test in a laboratory, we will get an idea of how the system is used in a real kitchen. Getting experts opinion of the system is not something that will be done, because it will not give an idea of how users will use it. During the test there will be 12 groceries, which the test-subjects must use during the assignments, as using real groceries is important to get the right feel of the program.

10.2 Method of Analysis

For evaluating the system the Instant Data Analysis (IDA) method will be used. This method has been developed at Aalborg University and its main purpose is to evaluate a system within a limited time frame. The method should only take a day to conduct, where the first half of the day will be spent on testing, and the second half of the day on analysing and documenting. Thus it is only possible to have four to six test-subjects - this results in a qualitative evaluation. Instead of recording video of test-subjects and analysing it later, IDA relies on notes taken during the test. This limits how much information can be extracted from the evaluation - however, it also limits the time spent watching the video. The IDA method differs from other methods of evaluation in having an additional person, called an IDA facilitator, who leads the analysis. This person is not present during the test, and does not know anything about the test data. The analysis is primarily a brainstorm of experienced problems where the facilitator writes down and asks clarifying questions. Once all problems have been discussed each is rated and categorised. After 2-2½ hours the brainstorm will be documented in a ranked list of problems.

When using the IDA method we will need a facilitator, a test monitor, and at least two data loggers. Notes will be the only reference point for the analysis brainstorm. This test will consist of one facilitator, one test monitor and two loggers. A third person will be present as a informal logger - that means the person will take notes but not as a logger, more as an evaluater of the system.

10.3 Assignments

The following is a list of the information and assignments the test-subjects will be given during the test. Two user profiles are listed, one of each gender. After the whole test the test-subjects will be asked to fill out a small questionnaire.
Introduction Yesterday you installed your new system, Foodolini, which can manage the content of your refrigerator, cupboard and other food storages. Foodolini has a web camera by which bar codes may be read and food items registered in the system. To fully enjoy the features of Foodolini, you must create a user profile.
In the profile a user name is created and personal details as date of birth, weight and height are entered. When creating a user profile enter the personal details as below (Choose according to your gender):

Full name: Sofie Hansen.

User name: Sof.

Date of birth: 30th of may, 1974.

Weight: 68 kg.

Height: 170 cm.

Gender: Female/ male.

Full name: Bjarne Thomsen.

User name: BJ.

Date of birth: 30th of may, 1987.

Weight: 87 kg.

Height: 185 cm.

Gender: Male.

Assignment 1: Create User profile

a) Create new user. Fill in the personal details from above.

b) Edit your user profile by changing the month of your birth date to December. It is only the month that must be changed.

c) Delete John Johnson from the system.
Please say when you think you have accomplished the assignment 1.

Assignment 2: Register food The registration of food items is based on bar codes scanned with the web camera. When a barcode is recognised a sound will be played. If the barcode is not recognised you may have to register the food items manually.
Today you have bought 7 items and must now register them in the system.

a) Register the first 5 items.

b) Register the 4 bananas to the system. (Each banana has an average weight of 200 g.)

c) Register the milk to the system. You must create the milk as a new food item.

d) Turn of the sound which is played every time a barcode is recognised.

e) Delete the tin of tuna from the list of registered food items.
Please say when you think you have accomplished the assignment 2.

Assignment 3: Recipes Foodolini offers a recipe feature where you can search, edit and create recipes. Searching recipes may be done from different criteria.
Tonight you are to cook a dinner for 6 people. As first course you will have onion soup. As main course leg of lamb with vegetables and for dessert ice creme with hot chocolate sauce.

a) Find a recipe on onion soup and view all the ingredients needed.

b) Edit the recipe to the correct amount of servings (that is 6), and change butter to olive oil.

c) Save the recipe you just edited.

d) Check if you have any food items soon to expire.

e) Create a recipe on how to make meringues.
Please say when you think you have accomplished the assignment 3.

Assignment 4: Shopping list The shopping list feature is to be used when food items are not present in the inventory.

a) Add dark chocolate to the shopping list.

b) Check if there is 1/2 liter of whipping creame in storage, if not add it to the shopping list.

c) Register to the system that you consume 1 glas (200 g) of milk.

d) Check that all the ingredients for the Leg of Lamb recipe is present in the system. If some ingredients are not in the inventory then add them to the shopping list.

e) Print the shopping list to a file on the desktop. Please say when you think you have accomplished the assignment 4.

Assignment 5: Speed scanning As a last assignment we would very much like to test how quick you can register 10 food items to the system. So please scan in the 10 items on the table.

10.4 Data from Assignments

The overall goal of the assignments is to find out whether the system is understandable and efficient to use. If the system is understandable, the user knows which functions to use to achieve their goals, and it is efficient if the user can do this quickly. The purpose of assignment 1 is to test if it makes sense for the user to differentiate between creating and deleting a new user which is only done in Administration compared to editing a user which is done in User Profile. One of the main functions that needs to be efficient is the registration of food and this is tested in assignment 2a and 2b. The purpose of assignment 2 is to evaluate the registration of foods. The registration of a food item has three variations:

- The bar code is recognised and the only information the test person has to add is the expiration date

- The bar code is not recognised but the ingredient is known by the system. The test person has to find the ingredient in the list of ingredients and then add the expiration date and amount and finally associate the ingredient with the bar code.
- The bar code is not recognised nor is the ingredient listed in the system. The test person will have to create a new ingredient and then enter the nutritional facts, such as the energy distribution, the amount and the expiration date. Finally associate the newly created ingredient with the bar code.

The test-subjects will go through all three variations, which are portrayed in assignments 2a, 2b, and 2c. This is to evaluate the systems learnability. We may assume some confusion regarding the naming conventions grocery and ingredient. Through the second assignment it will be possible to see how efficient and intuitive one of the most fundamental parts of the system works. The goal is as mentioned in the factor table, page 79, to scan 10 food items within 1 minute. The rest of the assignments are mainly to test whether the functions are clear about their goals. Through the test every assignment will be timed when the test-subject begins, and when the test-subject ends. That way we will know how long they took to finish an assignment. The loggers will also note when the test-subjects makes positive and negative comments. This will give us an understanding of what the test person thinks about the system.

10.5 The Evaluation Document

The results of the evaluation test is documented in the evaluation document. Below is a list of subjects addressed.

- Method: This section describes the purpose and procedure of the usability test, and a more detailed description of the test subjects.
- Usability Problems: This section contains the problem list, and describes the usability problems in Foodolini, discovered during the test.
- Issues addressed: This section describes the changes made to Foodolini, based on the results of the tests.
- Questionnaire and Answers: This section contains the Questions given to the test subjects, and their answers.
- Conclusion: This section sums up the usability evaluation.

10.6 Questionnaire

When each test-object has accomplished all assignments they will be asked to fill out a questionnaire with the following 5 questions:

- How easy was it to find the functions you needed? Please give examples.

- Was the naming of the functions understandable? Please give examples.
- Did the system do what you expected? Please give examples.
  If not: What did the system that was unexpected?
- Did you like the shopping list function? Please explain why.
  If not: What did you not like about the shopping list function.
- Do you have any additional comments?

The questionnaires and answers may can be seen in 12.4.

CHAPTER 11

Test Plan

This chapter will describe the planning and usage of tests for the program.

11.1 Types to be Tested

The database component and business logic component classes will be unit tested. All database model types will be tested, to ensure that they are stored and retrieved correctly from the SQLite database.

The complex classes of the business logic component will be tested in the most common scenarios, such as when mutating classes (save and delete), as well as all non-trivial functions.

These two layers have been chosen, since they play a central role in the application and because they fit in the scope of unit testing. In contrast, the user interface components can not be easily tested with unit testing.

Table 11.1 contains a list of the classes that will be unit tested.

Class Name	Test Description
Foodolini.Database	
Repository	Create table for all model types
Repository	All methods with all model classes
Foodolini.BusinessLogic	
Person	All methods
FoodItem	All methods
Ingredient	All methods
Nutrient	All methods

Table 11.1: Types to be tested

The tests will primarily be performed in order to assert that data is correctly mapped from the database and up into the business logic layer, as well as back again, when new objects need to be saved to the database.

11.2 Unit Testing

The test cases are written during and after the classes and structs have been implemented. This means that a portion of a class is implemented and subsequently tested. When the class gets revised, the associated test cases are either modified or new cases are added.

11.3 Testing Tools

NUnit will be used for unit testing. A NUnit plugin for MonoDevelop enables tests to be carried out within the IDE. NUnit was chosen, since it easy to write unit tests and has the aforementioned integration with MonoDevelop. Furthermore, it is widely used in the .NET/Mono development community.

11.4 Black Box vs White Box Testing

Both black box and white box testing will be used where it makes sense. Sticking to one testing method may limit what is possible to test in the program. One could also argue that unit testing is a type of white box testing, since it can be used to ensure that a certain function produces the expected result in several different test cases.

Rather, grey box testing will be used, since it allows more targeted tests than with the black box method. Moreover, internal knowledge of the program will be readily available, since it is the group that will be performing the tests.

11.4.1 List of Tests

The following is a list of the tests that will be conducted in the database layer. `FoodItemRow` will be tested extensively, while only the save and load methods will be tested in the other classes.

- Test of `FoodItemRow`
  - Test save and load from database
  - Test deleting from the database
  - Test updating from database
  - Test `where` method in database
- Test of `FoodDescription`
  - Test save and load from database
- Test of `Exercise Row`
  - Test saving and load from database
- Test of `FoodGroup`
  - Test save and load from database
- Test of `NutritionDefinition`
  - Test save and load from database
- Test of `Picture`
  - Test creating a table in the database
  - Test save to the database
  - Test saving and loading from database
  - Stress test

- Test of `Rating`
  - Test save and load from database
- Test of `RecipeRow`
  - Test save and load from database
- Test of `User`
  - Test save and load from database

This list contains the unit tests that will be used to test the business logic layer. Again FoodItem is the primary focus.

- Test of `FoodItem`
  - Test creating a `FoodItem`
  - Test loading a `FoodItem`
  - Test consuming a `FoodItem`
  - Test partial consuming a `FoodItem`
  - Test splitting a `FoodItem`
  - Test exceptions when splitting a `FoodItem`
  - Test opening and closing a `FoodItem`
- Test of `Ingredient`
  - Test creating a new `Ingredient`
  - Test the search function on `Ingredient`
- Test of `Person`
  - Test creating a new `Person`
  - Test save and load of a `Person`
  - Test deleting a `Person`
  - Test editing a `Person`
- Test of `Recipe`
  - Test creating a new `Recipe`
  - Test saving a `Recipe`
  - Test loading a `Recipe`
  - Test cooking a `Recipe`

Part III

Evaluation

CHAPTER 12

Evaluation Document

Summary

This chapter documents the results of the usability test of Foodolini, which was performed on December 8th 2009. The test was based on the "think-aloud" method and performed in the field - that is, in the flat of one of the group members of D101a. Three test users participated. The test users were asked to carry out tasks concerning the User Profile activity, Administration activity, Register Grocery activity, Cookbook activity, and the Shopping List. A test of how quickly they could scan 10 groceries was also conducted to test the effectivness of one of more advanced activities. The test users were also asked to fill out a questionnaire after completing the usability test.

Based upon the questionnaires and the usability test, the following weaknesses were deemed the most important:

- The workflow of registering FoodItems.
- The categorisation of Ingredients.
- Video feed was not reversed.
- Inventory was not sorted according to expiration date.
- The activites selector opened upwards, hiding some options.

12.1 Method

In this section the test and evaluation method will be explained.

12.1.1 Purpose and Focus

The purpose of this usabillity test and evaluation is to identify realistic usability problems in order to improve the design. The focus of the usability tests are the activities User Administration, User Profile, Register Grocery, Cookbook, and Shopping list as they all belong to the main purpose of the system, namely to keep track of a user's food supply. Also the effectiveness, efficiency and helpfullnes as stated in the Goals of Use page 55 will be evaluated.

12.1.2 Procedure

The usability test was performed as a field study utilising the "Think-aloud"-method and the analysis of the data collected was done using the IDA-method. Both methods are discussed in 10.1 and 10.2 respectively.

12.1.3 Test Subjects

Three test users were chosen; One female at age 37, married, and with two younger children, and two male students at Aalborg University. The female test user works as a stage manager at a theatre and her job demands a daily use of email, information searching and use of the Microsoft Office Package. Otherwise she has the approach that things must just work, if not she will ask for help. One of the male test users study Chemistry and the other Informatics at the Basic year. Both are experienced computer users, but not experts. The day before the usability test was to be conducted, the Informatics student cancelled. He was then replaced by a male student from Software Engineer 3 who may be close to an expert user. The change of one of the test users does not affect the purpose and focus of the test, as the experience of an intuitive user interface must also apply to expert users.

12.1.4 Equipment

The equipment used to perfome the usability test were:

- An IBM ThinkPad.
- An USB web cam.
- 12 groceries.

12.1.5 Procedure of the Test

The three usability tests were performed separately, thus the test users did not meet, and could not exchange their experience with one another. Each test user was welcomed and explained the main purpose of Foodolini as a food storage management system. A deep explanation of the system was not given, as the purpose of the test was to identify unexplained functions and routines lowering the efficiency as stated in the evaluation plan (see 10). The test users were informed that the system evaluation was a part of a study project at DAT1 and before the evaluation started, each test user was asked to sign a Statement of Consent regarding the evaluation and publishing of the found results. The test was divided into 4 main assignments each divided into 3-5 subtasks. The main assignments involved the use of Administration and User Profile, Register Grocery, Cookbook and Shopping list. An additional fifth test were performed only focusing on how fast 10 groceries could be registered to the system, where the bar code and grocery of all 10 items has been associated in advance. The test assignments are described in section 10.3. The test users were asked to fill out a questionnaire after accomplishing the user test. The questionnaire contained questions regarding the over all impression of the system, but also questions explicitly directed to certain features of the system. The questionnaire is explained in section 10.6 and the answers can be seen in section 12.4.

12.1.6 Procedure of Evaluation

An hour after the completion of the usability test, the test loggers, the test monitor and the facilitator met to make a brainstorm over the usability problems noted. The brainstorm was supported by the notes of the test loggers and the test monitor and within 2½ hours a ranked problem list 12.2.1 was conducted. The test log file of test subject A may be viewed in the appendix C the remaining 5 test logs may be read on the CD accompanying the report.

12.1.7 Identification and Categorisation of Problems

The location of problems were based on observations such as when:

- The test user verbally or by body language expressed irritation, frustration or insecurity.
- The test user did not act as expected, that is the test user did not fill in all the needed information when supposed to.
- The test user chose a more complex activity pattern when solving an assignment, than actually necessary.

The categorisation of problems are based on definitions by Molich and Nielsen and illustrated in table 12.1.

	Delay	**Irritation, user thinks program is irrational**	**Expectation vs actual response of the system**
Cosmetic	< 1 min.	Low	Small difference
Serious	Several minutes	Medium	Significant difference
Critical	Total (User stops)	Strong	Critical difference

Table 12.1: Table of the criteria used for categorising the user problems.

12.2 Usability Problems

The purpose of this section is to analyse the usability problems found in the evaluation of Foodolini. A problem list has been compiled in tables 12.2 and 12.3, in which the problems are explained. They are categorised as either cosmetic, serious or critical, it is noted which test subjects experienced the problems and in which assignments the problems presented themselves.

12.2.1 Problem List

Probl. nr.:	**Usability problem:**	**Category:**	**Exp. by:**	**Assign- ment:**
	Assignment 1			
1	Tried "User Profile" instead of "Administration" for creating a new user	Serious	a, b, c	1a
2	Ordering of activities in the combo box is different from the activity list in "Home"	Cosmetic	c	1a, 1b
3	The activity combo box opened upwards, hiding a lot of the selectable activities. Test subjects did not think there were more options available than the visible ones.	Critical	a, b, c	1a-1c
4	Took some time to find the "Create new user"-button	Cosmetic	a	1a
5	A lot of clicking necessary in the birth date calender when going back to 1972	Cosmetic	b	1b
6	Test subject expected a right-click function to lead to a delete/edit-functionality of users	Cosmetic	b	1b-1c
	Assignment 2			
7	Editing functionality was expected in food item registration for when erroneous information had been entered, but test subjects could not find one	Serious	b	2a
8	Missing a "Search all categories" in ingredient browser - users searched in "Baby foods"-category first	Serious	a, b, c	2a
9	Cannot find "quantity" when registering FoodItems	Cosmetic	a, b, c	2a
10	It is unclear how bar codes are associated with ingredients - user is not certain a bar code has been associated	Critical	c	2c

Table 12.2: Problem list. *(To be continued on page 125)*

Probl. nr.:	**Usability problem:**	**Category:**	**Exp. by:**	**Assign-ment:**
	Assignment 2 (*continued.*)			
11	Impossible to see which bar codes are associated with which ingredients	Serious	c	2c
12	Ingredient editor does not say that nutritional information is per 100 grams	Cosmetic	a	2c
13	User cannot find ″Edit″ in Ingredient browser	Cosmetic	c	2c
14	Expected ″Muting of camera″ to be in ″Settings″ instead of in the ″Register Food Item″-screen	Serious	a, b, c	2d
15	Subject unsure of the ″Clear″-button's functionality in food registration. Did not dare press it.	Serious	b	2
16	The video feed was not mirrored	Cosmetic	a, b	2
	Assignment 3			
17	User doubts search works, when they receive no results for a faulty search.	Cosmetic	b	3a
18	Ingredient list in Cookbook does not show all information in one view - scrolling is necessary. Annoys the user.	Cosmetic	b	3a
19	It is not possible to write a number of serving manually, instead of using the arrows on the entry	Cosmetic	a	3b
20	User could not sort Inventory by expiration date	Serious	c	3d
21	Ingredients missing for making a Recipe are not obvious to the user	Cosmetic	a	3
	Assignment 4			
22	It takes time to locate the shopping list button	Cosmetic	a, b, c	4a
23	Information about the marked FoodItem in Inventory does not update upon editing - e.g. consumption	Cosmetic	c	4c
24	User confused as to what ″Add Recipe to Shopping List″-button inside a Recipe does	Serious	c	4d
25	User expected a button in Inventory which would add the Ingredient for the marked Food Item to the Shopping List	Cosmetic	b	4

Table 12.3: Problem list. *(Continued from page 124)*

12.2.2 Problems

The problems in the problem list are addressed in the following sections. Problems experienced in the same activity will be discussed together. For the problems that will not be fixed, solutions will be proposed in these sections, while the problems that will be fixed is addressed in section 12.3

User Administration

No users realised they needed to go to the ”User administration”-screen to create a user - they all went to the User Profile. There might be several reasons for this - but a major one could be that it was not readily apparent to the subjects that such a screen existed. In the Home menu, the user administration feature was at the bottom of the list of activities - and in the top of the activities selector in the top bar. The activities selector opened and scrolled upwards, which none of the subjects were used to. It also hid the top-most entries in the box. They all needed to be prompted to look for an administration-feature to find it. A possible solution of this problem could be to add a ”Create new user”-button in the User Profile, if the user is not logged in. This will not be done. The layout of the activities selector is based on Gtk#, which has been used to design the user interface. Thus it is not a readily solvable problem. The ordering of the entries in the activities selector is generated automatically, because the activities are loaded as plug-ins dynamically. The activities could be ordered, but will not be done.

Test subject *b* expressed annoyance that a calendar is used for entering a birth date, because it takes too long to click back to 1972. This could be solved by making three different drop-down boxes for entering a birth date: date, month and year. This will not be done.

A single test subject expected the right-clicking of a user in ”Administration” to lead to an ”edit”-/”delete”-functionality. This could be added - but then the entire program would need to have this functionality for consistency. Therefore this functionality will not be added.

Food Registration

The function of the ”Clear”-button is unclear: test subject *b* expressed that they were unsure of what the button did. They would not press it, for fear of what would happen. Test subjects *c* and *b* (though only after being prompted by the monitor) pressed the ”Clear”-button, and were surprised that it cleared all fields, and not only the bar code. This was not the intention of the button, but a bug.

Test subject *c* had problems with associating a bar code with an Ingredient to create a FoodItem. Non of them seemed to understand that the association is created when the FoodItem is added to the Inventory. This could partly be solved by changing the layout and work-order of food registration. Furthermore, the difference between Ingredient and FoodItem are not obvious to the user. It can be discussed whether or not it should be, but actions will not be made for solving this.

Only one user showed an interest in how bar codes are associated with Ingredients. They expected it to be necessary to explicitly add the bar code to the Ingredient. This resulted in them scanning the same bar code several times, and entering the Ingredient several times, as they were not certain it had worked. A restructuring of the work-flow could fix this problem.

Furthermore it is not possible to see which Ingredients are associated with which bar codes. This functionality could help the user to understand the association between BarCode and Ingredient better - but could be redundant if the change of work-flow fixes problem nine.

Test subjects *b* and *c* expected an edit-functionality on Ingredient when registering a FoodItem, which should let the user edit erroneous information. This slows the user considerably, when they need to correct some information on the FoodItem. It leads the user to search for an edit-feature, as it seems logical for there be one. The test subject never realised that such a feature does, in fact, exist - but it is only accessible by double-clicking the FoodItem in the list of FoodItems to be added. A solution to this would be to simply add an ”Edit Ingredient”-button to the Ingredient selector.

No test subjects realised that there are several categories for Ingredients - they all began searching in the default category: baby foods. They all quickly realised that something was wrong, and saw that the Ingredients were sorted in categories. There were some doubt as to which categories some FoodItems belonged in - and it slowed the test subjects as they had to read through all available categories to decide which was fitting. It would ease the use of the program if it was possible to search the entire Ingredient-database, instead of searching by category.

When creating a new Ingredient, the Ingredient editor does not tell the user that nutritional information is per 100 grams. This is the default for most, if not all, products - but should still be mentioned for clarity.

During testing of Foodolini, the feed from the webcamera was not reversed to behave as a mirror. This slowed the test subjects, as it seemed counter intuitive that the camera did not work as a mirror. Flipping the video feet could make it more intuitive to use the bar code scanner.

All test subjects expected the information about quantity, shelf life etc. to be associated with the bar code, so when registering a new FoodItem, all information would be given. But this did not slow them for long, and a change in the work-flow might solve this.

When a bar code is successfully scanned, Foodolini plays a sound. This sound is mutable inside Food Registration. All users expected the muting of this sound to be in a ”Settings”-menu, which they sought for in the activities selector. Moving the button for muting might help the users, but as this function is not an often used function, no actions will be taken to correct this.

The most important of these problems is that the it is necessary to press the ”Add”-button after scanning each FoodItem. The test subjects all expected the FoodItem to be added below as soon as it had been scanned. If it was fixed to correspond better to the subjects’ expectations, the expiration date would also be set to the general shelf life for such an Ingredient, and the user would have to be aware of changing it themselves. When changing the work-flow it could be an option to remove the need for the ”Add”-

button.

Cookbook

When requested to search for a Recipe, test subject *b* accidentally entered the wrong word to search for. The search function did not return any feedback if it did not find anything fitting the search criteria. Thus the user was unsure if the search was in progress, or if they had done something wrong. Subject *b* then realised that something wrong was entered, and corrected it - resulting in only a minor delay. This could be solved quite simply by displaying a message as "No recipes found".

When viewing a Recipe, subject *b* expressed annoyance that all the information available about the Ingredients was not viewable on the screen without the user having to scroll. This hid the column informing the user how much they were lacking of a certain Ingredient - which subject *a* experienced and had a bit of trouble finding out which Ingredients were missing. This could be fixed by either making the IngredientViewer wider and making room for more columns, making some of the columns slimmer or changing the colour of the expired Ingredients. This issue will, however, not be addressed.

When adjusting the number of servings the user wishes to cook, it turned out to be impossible for the user to enter a number manually - it could only be done by using the arrows on the entry box. This only slowed test subject *a* a bit, but could be very annoying for a user, who wishes to cook 50 servings of something for a party. The problem is a bug, and can be fixed.

Inventory

When editing a FoodItem in Inventory, the FoodItem is selected from the list of FoodItems in Inventory, and further information can be seen in the right side of the screen - from here it is possible to edit the FoodItem. After editing, the Inventory list updated with the new information, but the detailed FoodItemViewer did not. This was a bug, and is fixed.

Test subject *c* was confused as to what the "Add Recipe to Shopping List"-button when viewing a Recipe would do. This can be solved by renaming the button "Add Ingredients to Shopping List".

Test subject *c* expected to be able to sort the Inventory by expiration date - as it is in the Home page. This can easily be changed and will be.

None of the test subjects saw the shopping list at first. As the shopping list is collapsed as default, the button for viewing the shopping list is not very wide - it could very well interfere with the aesthetics of the program if it were. But it does not seem to be wide enough for the users to spot immediately. This problem would possibly only be relevant once, though, as the users would then know where it was. Any actions to change this will not be made.

Test subject *b* expected a button to add a FoodItem in the Inventory to the ShoppingList - instead of having to go to the ShoppingList first. This button could be added, and may seem logical for most users, but it will not.

12.3 Issues Addressed

This section describes how some of the found usability problems have been addressed.

12.3.1 FoodRegistration

It was decided to change the work-flow of registrating a FoodItem. In the new version of Food Registration, the user can either choose to add an item with or without a bar code. If the user chooses to add an item without a bar code, an Ingredient selector will pop up. The search of Ingredients has been changed to include all Ingredients per default. When the user has selected an Ingredient, it will be added to the list of scanned FoodItems with a default shelf-life and a quantity of 100 grams. The user can then change the shelf-life and name of the food item, and the list will update as necessary. If the user scans a bar code instead, Foodolini will check if it already knows this bar code. If it does, the item will be added to the list of scanned food items. Again - the shelf life, name, quantity, and Ingredient can be changed, when the FoodItem is selected in the list. If the bar code is unknown, the Ingredient selector will pop up, and the user will choose the appropriate Ingredient. Foodolini will then associate this BarCode with the Ingredient - and save the quantity if it is changed. A tool-tip has been added to explain the function of the "Clear"-button. Hopefully this will fix most of the problems in FoodRegistration - specifically problem 7, 8, 9, 10, and 15.

An "Edit"-button has been added to the Ingredient browser, as users did not realise that Ingredients could be edited by double clicking.

In the new layout of FoodRegistration, it is possible to reverse the camera view, accordinig to the user's own taste.

12.3.2 Cookbook

To solve problem 17, a Recipe with the name "No recipes found" will appear as a search result, to inform the user that there was no matches to the search criteria. This Recipe can ofcause not be viewed.

The number of servings can now be added by using the keyboard as well as the mouse, fixing problem 19.

12.3.3 Inventory

When updating the specifics on a FoodItem in Inventory, the FoodItemViewer now updates according to the new information.

The FoodItems have also been sorted in an assending order as in the Home activity.

To help the user, the button in a Recipe called "Add Recipe to Shopping List" has been renamed "Add Ingredients to Shopping List", to reflect the effect the user sees - it does the same as before.

12.4 Questionnaire and Answers

The answers to the questionnaire can be read below. As the test subjects are native Danish, the questionnaires were answered in this language. The Danish answers will be written first, with regular type, and the answers translated to English, will be written with italic type after the Danish answers. The question is indicated by " Q: " and the answer from the test subjects are indicated by (a), (b) or (c) respectively.

1. Q: How easy was it to find the functions you needed? Please, give an example.
   (a) : En smule svært nogle steder. Create user og Shopping list var lidt svære at finde. *A bit difficult in certain places. Create user and shopping list were a little hard to find.*
   (b) : Da jeg først fik overblikket var det nemt. *After getting an overview, it was easy.*
   (c) : Det var umiddelbart meget nemt, det krævede at jeg lige kiggede rundt på skærmen, men ellers vil jeg mene at jeg kunne finde det selv. Dog var registrering af mælkens stregkode ikke åbenlys. *Immediatly it was quite easy, but I had to take a look around on the screen. I would say I was able to find it my self. Though registering a milks bar code was not obvious.*

2. Q: Was the naming of the functions understandable? Please, give examples.
   (a) : Inventory, Register Grocery og Ingredients kan let forveksles. *Inventory, Register grocery and Ingredients are easy mixed up.*
   (b) : Ja. *Yes.*
   (c) : Ja, det var let at regne ud hvad funktionernes egenskab var. I hvert fald den hovedsagligt. *Yes, it was very easy to figure out what the purpose of the functions were. At least the primary.*

3. Q: Did the system what you expected? Please, give examples.
   (a) : Ja, dejligt at have elektroniske opskrifter. *Yes, nice to have electronic recipes.*
   (b) : Overvejende, og nogle gange lidt ekstra. *Mainly, and some times a little more.*
   (c) : Ja, på nær i Inventory hvor jeg ikke kunne få varerne vist i en sorteret rækkefølge efter udløbsdato. *Yes, except in Inventory where I could not get the groceries sorted by expiration date.*

4. Q: If not: What did the system do that was unexpected?
   (a) : Ikke meget. Det fungerede meget godt. *Not much. It all worked very well.*
   (b) : F.eks. at tilføje ingredienser til indkøbslisten fra en opskrift, tog den alt det manglende med over. *When adding ingredients from a recipe to the shopping list, it added every missing ingredient*
   (c) : Se ovenstående spørgsmål. *Se answer above.*

5. Q: Did you notice shelf life for each food item changing when scanning? Please, give examples.
   (a) : Nej. *No.*
   (b) : Nej. *No.*
   (c) : Nej. *No.*

6. Q: Did you like the shopping list function?
   (a) : Ja, så behøver man ikke tænke:-) *Yes, then you do not have to think:-)*
   (b) : Ja, den er let at bruge. *Yes, it is easy to use.*
   (c) : Ja, den var rigtig god. Igen skulle jeg lige kigge mig omkring på skærmen, men ingen væsentlige problemer der. *Yes, it was really good. Once again though, I had to look around on the screen, but no significant problems.*

7. Q: If not: What did you not like about the shopping list?
   (a) : At det ikke er et punkt i hovedmenuen. *That it was not an option in the main menu.*
   (b) : (Test-object havde ingen kommentarer). *(blank - test-object did not make a comment).*
   (c) : (Test-object havde ingen kommentarer). *(blank - test-object did not make a comment)*

8. Q: Do you have any other comments?
   (a) : Godt program! *Good program!*
   (b) : Overvejende let at gå til. *Mainly easy to access.*
   (c) : Opdatering i inventory alle steder:-) Godt produkt. *Updating the inventory every where:-) Good product.*

12.5 Conclusion

The purpose of the usability test was to identify problems in Foodolini, so some of them could be fixed.The activities exposed were those related to the main features of the system. Three test subjects participated and were asked to use the "Think-aloud"-method. The evaluation was done according to the IDA-method.

The evaluation identified 25 problems in total. As more than half of the problems were categorised as cosmetic and only 2 as critical, the amount may therefore be considered fair.

One of the critical problems were related to the work-flow when registering FoodItems, as it required superfluous actions from the user. As this is an activity which will be used many times in a day after shopping, it is recommended to redesign the work-flow for food registration and bar code association. The other critical problem identified concerns the activities selector in the top bar. None of the test users were aware that it contained more selectable activities than immediately visible. A redesign of the selector is recommended.

The speed test of how quickly 10 FoodItems could be scanned also consolidated the need for redesigning the work-flow of registering food items. None of the test users were able to register all 10 items within one minute.

The differences between User Profile and User Administration were not immediately obvious to the test users. When asked to create a new user, all test subjects chose User Profile instead of Administration as their first choice. However, after realising a user was to be created utilising User Administration the test users remembered from their first attempt that editing a user's profile should be done in User Profile. After having understood the difference, the system seemed internally logical, resulting in good learnability.

In general all test users expressed excitement over the functionality of the system. But to fulfil the goals of use of the system, both the effectiveness, efficiency and helpfulness needs improvements. If the above mentioned usability problems are corrected, the system could fulfil these goals of use.

Part IV

Test Document

CHAPTER 13

Test Document

The following lists the tests conducted with description of what was tested. The most important tests are those of FoodItemRow (See 13.1.1) and the entire business logic layer (See 13.2).

13.1 Test of the Database Layer

The following test were written to eliminate most bugs in the database layer, prior to implementing the business logic layer. The tests here are all intended for testing the `Repository` class, which serializes objects to the database using reflection. Since this class is very important, it was necessary to test the `Repository` class for all operations with many different combinations of data.

The following tests all derive from the same base class that creates an empty SQLite database and initializes a new `Repository` prior to each test. This ensures that the tests are completely independent of each other. Note that the classes tested below do not contain any logic at all; they are so-called thin classes, which consist of public automatic properties and are exclusively used to represent rows of a database table.

The reason they are all tested is to ensure that the `Repository` class can generate a table for instances of each thin class and save these instances to their respective tables, using reflection. So, all these tests are, in fact, tests of the `Repository` class, with different type parameterization. In each of the test cases saving and loading from `Repository` were conducted, with several objects using different parameters. There was particular emphasis on ensuring that nullable properties were tested with both a null and non-null values. In addition, special cases with regard to the usage of the DateTime class and TimeSpan class, were tested using different values, as to ensure that they were correctly handled by the `Repository`.

13.1.1 Test of FoodItemRow

The `FoodItemRow` was used as the primary test case in the database and used to test all operations in the `Repository`. `FoodItemRow` was chosen as the primary test case, as it was more complex than all the other thin classes, e.g. it had more attributes of different types, thus it was more likely that we would find bugs in `Repository` using `FoodItemRow`, than with any of the other thin classes.

Test Save and Load from Database

Saves a `FoodItemRow` in the database and tests if the object receives an ID. Loads the `FoodItemRow` from the database and tests if the data matches. Since the table for `FoodItemRow` does not exist in the database, creation of this table will also be tested.

Results
This test returned no errors.

Test Deleting using Repository

Inserts a `FoodItemRow` into the database and deletes it again, while checking that the correct data was deleted.
Results
Fixed a crash error occurring when an item was deleted multiple times. The problem was that we checked if the object had an ID different from 0, and threw an exception if that were the case. Instead, the code now does nothing if the ID is 0, or the row, which the object represents, is not in the database.

Test Update on Repository

Tests the `Repository.Update<T>(T item)` method by adding a `FoodItemRow` to the database, change it, update its corresponding database row using update and load it again, to see if the database row was updated.
Results
This test returned no errors.

Test Where Method on Repository

Tests the `Where<t>` and `SingleWhere<T>` methods by inserting rows into the database and then use the `Where<T>` method to load the rows with an SQL condition, finally checking if the correct rows where loaded.
Results
This test helped fix an SQLite syntax error.

13.1.2 Test of FoodDescription

Creates a database table for `FoodDescription`, saves a `FoodDescription` in the database and tests if the object receives an ID. Then it loads the `FoodDescription` from the database and tests if the data is correct.
Results
`TimeSpan` does not implement `IConvertible` and thus conversion in `Repository` failed. This was fixed by representing `TimeSpan` using `long` in the database layer. Which deferred conversion of `long` into `TimeSpan` to the business logic layer.

13.1.3 Test of ExerciseRow

Creates a database table for `Exercise`, saves an `Exercise` in the database and tests if the object receives an ID. Then the `Exercise` is loaded from the database and tests if the data is correct.
Results
This test returned no errors.

13.1.4 Test of FoodGroup

Creates a database table `FoodGroup`, saves a `FoodGroup` in the database and tests if the object receives an ID. Subsequently, the `FoodGroup` is loaded from the database and tests if the data match what was saved.
Results
This test returned no errors.

13.1.5 Test of NutritionDefinition

Creates a database table for `NutritionDefinition`, saves a `NutritionDefinition` to the database and tests if the object receives an ID. Then the `NutritionDefinition` is loaded from the database and tests if the data is correct.
Results
This test returned no errors.

13.1.6 Test of Picture

Database Table Creation Test

Tries to create a `Picture` table in the database.
Results
This test returned no errors.

Test Save to the Database

Save a `Picture` to database.
Results
This test returned no errors.

Test Saving and Loading from Database

Saves a `Picture` in the database and loads the picture from the database. The picture is tested for correctness after being loaded.
Results
This test returned no errors.

Stress Test

Stress tests the database by loading and saving a large `Picture`. This test uses a 5 MiB pseudorandom test vector. Thus, this test may fail at random, if there are any bugs, however it is highly unlikely that the errors will not be consistent.
Results
This test returned no errors.

13.1.7 Test of Rating

Creates a table for `Rating`, saves a `Rating` in the database and tests if the object receives an ID. Then the `Rating` is loaded from the database and tests if the data is correct.
Results
This test returned no errors.

13.1.8 Test of RecipeRow

Creates a table for `RecipeRow`, saves a `RecipeRow` in the database and tests if the object receives an ID. Then the `RecipeRow` is loaded from the database and tests if the data is correct.
Results
This test returned no errors.

13.1.9 Test of User

Create a table for `User`, save a `User` in the database and tests if the object receives an ID. Then the `User` is loaded from the database and tests if the data is correct.
Results
This test returned no errors.

13.2 Tests of the Business Logic Layer

The following tests were written to eliminate most of the bugs in the business logic layer, in advance of implementing the front-end layer. All of the tests here derive from the same base class, which initializes an instance of `Settings`, thus creating a `Repository` and a database connection, prior to each test.

This common base class also ensures that changes to the database are not commited, and rolls back the database after each test. This ensures that the test vectors are not altered. Since the classes in the business logic layer uses static variables for caching data, this business logic test base class also calls the static type initializer on each type in the business logic layer before each test, thus ensuring a clean cache.

13.2.1 Test of FoodItem

FoodItem Creation

Attempts to create a `FoodItem` and save it.
Results
This test returned no errors.

Test Loading a FoodItem

Tests that a `FoodItem` can be saved and loaded by saving a new `FoodItem` and then loaded once more, to check if the loaded data is correct.
Results
This test returned no errors.

Consume FoodItem Test

Tests that the `FoodItem` can be consumed and that a consumed `FoodItem` refers to the `Person`. It also tests that the `FoodItem` is opened after being consumed.
Results
Fixed that the `FoodItem` should be opened once consumed. This was added to the method `FoodItem.Consume()`. An SQL typo in `Person.ListPersons()` was also fixed. It had caused the method to return an empty list. Finally, an SQLite syntax error was fixed in `FoodItem.ListFoodItems()`.

Test Partial Consumption of a FoodItem

Tests that a `FoodItem` can be partly consumed, by creating a new `FoodItem` and partially consuming with a `Person`. All of the remaining `FoodItems` are then tested for correctness. Finally, it tests if the consumed `FoodItem` has been associated with the `Person`.
Results
This test returned no errors.

Test Splitting of a FoodItem

Tests if `FoodItem.Split()` works. Tests that the `FoodItem` is opened after being splitted.
Results
Fixed that `FoodItem.Split()` did not open the split `FoodItems`. This was fixed by calling `FoodItem.Open()`.

Test Exceptions When Splitting a FoodItem

Tests that exceptions are thrown when performing illegal splitting of a `FoodItem`.
Results
This test returned no errors.

Test Deleting a FoodItem

Tests that `FoodItems` can be deleted, by adding `FoodItems` to the database, then removing them again and checking if they have been removed correctly.
Results
This test returned no errors.

Test Opening and Closing a FoodItem

Tests that opening and closing a `FoodItem` works, by creating a new `FoodItem` and attempting to open and close it. It is also checked that the expiration dates are changed.
Results
This test returned no errors.

13.2.2 Test of Ingredient

Test Creating a new Ingredient

Attempts to create an `Ingredient` and save it.
Results
This test returned no errors.

Test the Search Function on Ingredient

Tests that the `Ingredient` search function works by fashioning a number of searches and verifying that the database returns the correct `Ingredient`.
Results
This test returned no errors.

13.2.3 Test of Person

Test Creating a new Person

Makes an attempt to create a `Person` and save it.
Results
This test returned no errors.

Test Save and Load of a Person

Tests that a `Person` can be saved and loaded by saving a new `Person` and then loading it once more, to check if the loaded data matches.
Results
This test returned no errors.

Test Deleting a Person

Tests that a `Person` can be deleted by first creating a `Person` and the deleting it again, and then testing that the `Person` has been deleted.
Results
This test returned no errors.

Test Editing a Person

Tests that the properties of `Person` can be changed. The gender was changed from male to female, the age 10 years younger, the height 50 centimetres taller. It was then checked that the BMR and MET were affected.
Results
This test revealed that BMR was `int` and METS was `double`, instead of `int`. This was fixed by changing the METS to `int`, by performing a cast in the calculation.

13.2.4 Test of Recipe

Test creating a new Recipe

Attempts to create a `Recipe` and save it.
Results
This test returned no errors.

Test Saving a Recipe

Tries to save a `Recipe` with `Ingredients` and `Directions`.
Results
This test returned no errors.

Test Loading a Recipe

Tests that a `Recipe` can be saved and loaded by saving a new `Recipe` and then loading it again, to check if the loaded data is correct.
Results
This test returned no errors.

Test Cooking a Recipe

Tests the `Cook` method on `Recipe` by calling the `Cook` method on a `Recipe` and checking that the inventory, as well as the `Ingredients` match the expected.
Results
Fixed null reference error as `Cook` accessed a private lazily loaded field, instead of the public property, `Ingredients`, which loads the ingredients on demand. Fixed a bug in the calculation of the resulting `Ingredient` when cooking a new `FoodItem`. An error in the calculation of the average of protein, fat and calories occurred, due to an incorrect number used in the divisor of the total number of ingredients used.

Part V

Study report

CHAPTER 14

Study report

14.1 Introduction

This is the study report documenting the development of Foodolini and the accompanying documentation. The report is split into six chapters. The first chapter details our development process, describing our organisation, what we have learned etc. The next couple of chapters will describe the tools we used during the analysis and design. The usability test and evaluation of this will be discussed, as well as the testing of the program. Finally, we will discuss possible future work.

14.2 Process

14.2.1 Organisation

This section describes our organisation during the various stages of development.

Analysis Organisation

During the analysis phase of the project, we chose to divide the group into several smaller groups, each focusing on documenting or adding a specific aspect of the analysis. The bulk of analysis work had been performed before we began writing the report in relation to the SAD course, as well as during group meetings. We had a lot of drafts and work sheets from the analysis that needed to be documented. By assigning separate parts of report authoring and actual analysis work to different people in the group, well-established analysis findings could be integrated into the report while new parts were progressing.

In retrospect, this approach worked well, since we were able to complete a lot of the analysis work in a relatively short period of time. Moreover, various parts of the analysis were progressing concurrently. Questions to prior analysis work would often be raised during this process, which we could act upon and make any changes if necessary.

Design Organisation

As the analysis began nearing completion, a sub-group began the initial design work. However, much of this work went side by side with the implementation. We prototyped some of the sub-systems, such as serialization to database using reflection, pulling webcams and drawing using GDK# to make GTK# widgets. This was to get comfortable using these tools, as well as to find out if they met our requirements, before investing too much time in them and counting on them in the design. Later, when the analysis was complete, the rest of the group moved on to design, creating, for instance, mockups of the user interface. At this time the aforementioned subgroup began the lengthy implementation of the underlying database and business logic layer. This allowed the current design team to move directly to the implementation when they finished designing the various UI components.

Furthermore, individual members often swapped back and forth between design and implementation, as mistakes in the design were discovered during implementation, or new and more effective methods were discovered, which consequently required modifications to the design.

Implementation organisation

During implementation we decided to use pair programming as much as possible, in order to assist less experienced programmers, along with evening out the different levels of proficiency in the group.

It was also decided that a programmer should not be allowed to create unit tests for his own code. This way we ensured that at least two programmers had closely studied the code. This process proved very effective in tracking down bugs, as well as it allowed for discussions in the group, concerning code needing to be changed or rewritten. As the implementation progressed, and most of the database and logic layer were completed, pair programming was used to a lesser extent. There were two reasons for this: First of all, the program needed to be complete for the usability test, which led to some time pressure. Secondly, some of the group members with less programming experience would like to program a part of the program by themselves, in order to make sure that they also were capable of writing code by themselves, rather than just commenting on what others had done.

We implemented the user interface by handing out use case activities to each programmer. That way, we believe we have avoided most of the conflicts which inevitably arise when multiple programmers work on the same UI code. If some UI code was dependent on another programmer, verbal agreements were made on how the dependent methods should be implemented. In addition, documentation was written alongside with the code and the plug-in architecture of the UI layer, which made it easy to get the different parts of the program to work together. Since coding was split up between the programmers in sub-projects, we generally did not have many problems with conflicts nor changes to the code that caused other parts of the program to fail.

During implementation we were inspired by one of the Software Engineering groups' use of a wall of post-its placed in a two-dimensional coordinate system, where X was importance and Y was difficulty. Each label would represent a task to be developed and it would be positioned on the wall based on its importance together with the difficulty of implementation. Each group member would then have a post-it note with his/her name on and place it onto the task label he/she was working on. The idea was to control who was working on what and to ensure that dependencies were implemented in the correct order. It worked to some extent in the early progression of the implementation, especially during unit testing of the database and business logic layer. However, later in the process the wall fell out of use and we abandoned it. We believe that the reason for this was that we from the beginning already had a good idea of who was working on the different parts of the program. Therefore, the labels became just another list that we had to keep updated, instead of a help to see where we were.

14.2.2 To-do and Scheduling

Initially we found it difficult to plan the entire project, as we had little idea what the analysis and design documents were supposed to contain. Through SAD we learned of the analysis and design document standards, which we were to follow. We realised that we had already done a rather large part of the analysis, and only needed to formalise most of the documentation. Thus, for a large part of this project we did not have any detailed plans for when the different part of the project should be done. Nevertheless, such plans were quickly drafted, and followed, when we were given hold of the standards for the design document.

We, as a group, have previously had success with to-do lists as a way of organising the workload - so it was natural for us to attempt to use them in this project again. This time, however, they were far less fruitful, as we ended up creating too many separate to-do lists which created confusion. While it has not had a serious impact on the progress, it has been an annoyance as we had to double-check with each other before beginning to work on anything. This was to ensure that the to-do list was up-to-date and that no one else had begun working on it without updating the specific to-do list entry.

All of this led to a bit of a chaotic project management from a planning and scheduling viewpoint. However, as the group works well together, it has not been a serious issue.

We introduced a new tool for use in this project: `etherpad.com`. The site acts as a multi-user notepad, allowing multiple users to write in a simple text editor. This allowed us to compensate for the reduced number of blackboards, compared to the previous semester project, by providing a platform the entire group could participate in and view at all times. Furthermore etherpad worked better than the blackboards, as it allowed us to access the information from home, and they did not have to be cleared all the time, to make room for new information. And at the end of this project to-do lists on etherpad proved very useful, as these could be updated in real time and accessed from anywhere.

We have also used a small shell script to generate a to-do list based on `TODO` comments in the TEXfiles as we were writing and correcting the report. This was not necessarily something we used to organise the work as much as it was used to mark things that we needed to check out before handing in the report - for this, thiss tool was very useful.

14.3 Analysis

The aspects of the analysis will be discussed in this section. Topics will be the purpose of the analysis and some of the tools used.

14.3.1 Purpose of Analysis

Conducting an analysis of a system is the first part of the system development. In the analysis the system developers analyse the problem and application domain of the system from the end user's point of view. This means that the analysis only concerns what the system should be used for and what it is supposed to be able to do. The goal of the analysis is for the system developers to come to an agreement with the customers. As we have been both developers and costumors for this system, the goal of the analysis was to come to an agreement within the group.

Conducting an analysis is very useful to specify what problem domain the system should try to help, in which application domain the system will be used, and how the system is supposed to work in general. Doing this analysis has helped us a lot in making sure we were all working in the same direction, with the same end product in mind. The course, System Analysis and Design (SAD), gave us the tools to communicate our thoughts and to document our decisions. Some tools worked better than others. Working in such great detail can, in some cases, cause more harm than good, as there is a risk of repetition of obvious points taking too much time.

14.3.2 System Outline

The very first tool used in the analysis is drawing rich pictures. The purpose of these pictures is to get an initial idea of how the system developers see the possibilities of the system. The focus of rich pictures is to identify problems the system might be able to solve, including which objects will be involved, and their communication. We drew our rich pictures as part of an exercise during an SAD lecture. Later we compared and discussed our images, and created a rich image together that we all agreed upon.

After the rich picture was discussed, we documented it in a system definition. This is based on the FACTOR concept, where the Functionality, Application domain, Conditions, Technology, Objects and Responsibility are addressed. These points all refer to topics from discussions of the rich pictures. The system definition is a very useful tool to get an overview of the overall decisions made from the initial discussions. If the system definition is clear, it can be used as a point of reference in future discussions throughout the whole development process.

14.3.3 Classes and Events

The next activity is to find class and event candidates. Based on the problem domain, possible classes and events are found during a brainstorm. After all of the classes and events had been identified, the lists were reduced to the classes and events that the systems should encompass. Using a brainstorm to find all candidates, allows for an open-end discussion about potential classes, including perhaps more untraditional proposals. Some classes and events were removed, because they were not in the problem domain, or because they were overlapping.

When the final list of classes has been found, an analysis class diagram can be crafted. This class diagram shows how the classes are related to each other, as well as how many instances of each class can or must exist. The analysis class diagram is also a tool to get a discussion going while documenting decisions for future work. Creating the analysis class diagram helped us clarify how the system should handle the association between `BarCode`, `Ingredient` and `FoodItem`, with the aim of reflecting the way bar codes are used in everyday food items in the system . It also helped us determine if some of the classes should be merged into one class, instead of two, as they were always dependent on each other.

After agreeing on the classes and their respective cardinalities, the events found previously can be discussed in further depth. The events were assigned to one or more classes, which should handle the events. We created state chart diagrams, which illustrate the states and events of all of the classes. This opened up for discussions, since we found that some classes did not have any events, along with some classes having events that were irrelevant to the system. Creating the state chart diagrams also helped us decide if a class should only have one state, or if it had to have multiple states, and in that case which events should be allowed in each of the states.

An event table summarised the classes and their events. The table contains all classes and events, with each event assigned to the classes it involved. A table is a good way of getting an overview of how classes interact with each other and in what way. In the event table different signs for sequential and selective events further clarify the behavior of each event.

14.3.4 System and User Interaction

A classification of the end users is required before further development of the system. Personas are detailed descriptions of characteristic users, and scenarios are descriptions of how these users would use the system. Personas and scenarios are useful to personalise a target group that, at times, can be very large, and is used when any design decisions must be made. For us, personas and scenarios helped us to discuss our target group and to specify that we would focus on users with some computer experience, and not computer novices.

This is also a good point to discuss the work flow of the system. Use cases are used to describe how the user interacts with the system. It is how the system reacts to an action from the user, e.g. if the user clicks a certain button, the system acts accordingly. Use cases can be explained in text, tables or diagrams, as to what will be most useful for the specific use case. As a diagram version of the use cases was needed in the design document, we decided to rely on diagrams of complicated use cases and text for simple use cases. Diagrams are very useful to get an overview of how a use case works, while a diagram for a very simple use cases would be superfluous and perhaps out of place.

14.3.5 Analyse the Design

At this point the analysis turns its focus to analysing the design. The goals for the use of the system were discussed here. These are factors, such as effectiveness, efficiency, learnability, fun, and helpful. They are rated on how important they are, and is a means of discussing where the focus of the system should be. We all had a clear idea that this system should be very efficient to use, as many of the functions would be used on a day-to-day basis. Therefore, we rated these factors as very important, where factors as fun and entertaining were rated as irrelevant. Determining the importance of these factors helped us a great deal, because whenever we had a design question that would either make it efficient or look good and exclude the other, the decision was clear.

Deciding a conceptual model and the interaction forms is the next step in the analysis. This is how the users should interact with the system physically. Should our system be an integrated part of a refrigerator and use a touch pad, for example, or should it run on a normal laptop? Should it use speech and gestures or the keyboard and mouse to interact with the system? Should the system communicate with external systems? Deciding this is key to the further analysis and design, and the structure of the system is very dependent on the decisions made. Integrating our system in a refrigerator could make sense, as Foodolini often would be used in the kitchen. On the other hand, making the system so stationary could also be a disadvantage. If the system were made to run on any laptop the users could use the features of the system anywhere. They would be able to check the inventory and cookbook and use the shopping list for shopping on the way home from work. As efficiency and effectiveness are very important in this system, we chose to model it to a laptop. We also decided to rely on a keyboard and mouse, as using speech and gestures would create unnecessary obstacles.

The last step in the analysis is to form a general interaction model. This is an overview of which interaction spaces are involved in different subtasks. A subtask could, for example, be to register a food item. Here different interaction spaces, like a BarCodeSelector, FoodItemEditor and FoodItemBrowser, are used. Finding these interaction spaces and where they are used is done while concentrating as little as possible on the design. The purpose is to find repeated interaction spaces in order to create consistency. Discussing these interaction spaces before designing the system helped us plan the layout of the system before designing it. By focusing on three types - browser, editor and viewer - we made sure that all interactions were made the same way. The browser should, for example, be a list containing the name, including additional important information. By clicking an item in the list, a viewer should open, and by double clicking the list an editor should open. As a result of this we were able to make sure that similar things were made in an uniform way. Moreover, the interaction spaces made it much easier in the later stage of the design, because it only was a matter of designing the interaction spaces, and arranging them in the screen.

14.4 Design

14.4.1 Purpose

The purpose of the design process was to find a good class structure and consistent user interface, prior to implementation. This was to ensure that the user interface was consistent, and that the class structure satisfied our needs. Thinking the entire application through before implementation made us capable of solving problems early in the process. Thus we did not have to redo those parts at a time where it would be much more resources consuming, because most of the API would have been designed at this point. It also helped facilitate parallel development.

14.4.2 Design of Architecture

In this project we prioritised the design criteria and settled on an architecture. The prioritisation ensured that we considered all the different features that our architecture could facilitate. Thus we prioritised and decided which features we would focus on. This allowed us to make a formal, stable architecture very early in the process, which helped clarify how dependencies should be isolated - for example that some external dependencies should only have references from a specific layer or component. We believe that this isolation has helped ensure that some of our code may be reusable by third party developers.

If it had not been for our architecture we would probably have referenced libzbar-cli in the business logic layer or vice versa, in order to avoid having two enums representing bar code types. Nevertheless, and likely because of our architecture, we did not, and instead wrote code for converting between the two enums. The result is that libzbar-cli is in no way dependent on our project, and can readily be reused by third party developers. Our business logic layer may also be reused, for example for server side services, because it does not have any ties to libzbar-cli.

Our architecture also made it easy to know where to begin the implementation, and what needed to be implemented before the next layer could be started. Furthermore the plug-in architecture in the UI-layer made it easy to develop the activities component in multiple isolated sub-projects, as the main component did not need to know about the plug-ins before runtime, and was thus not dependent on them.

14.4.3 Component Design

Prior to implementing the different components in Foodolini we designed them. This involved figuring out which classes were needed in a component and what methods, fields, and properties these classes should have. Responsibilities were also assigned to the classes, and structural relations between different classes were determined. This was expressed in classic UML class diagrams, and to the extent it was necessary, the individual classes were described.

This helped us think about the classes and the structure of the components before we implemented them, and it ensured that the extensibility and flexibility we wanted from the components was possible. For instance it was important to us that the API of the database layer was stable, and that the database engine could be replaced in the future. By using design patterns, and designing before implementing, we ensured that the API of the database layer may remain stable even if the underlying database was to be replaced in the future.

Designing the components prior to implementation also helped us ensure consistency between classes, so that they behave similarly. Thus, the APIs were easier to use. For instance most of the classes in the business logic layer act quite similar with respect to their `Save` and `Delete` methods. If we had chosen to make a storage and cookbook class, all objects would have to have a reference to them. In addition they would have to maintain the database connection - making the API more complicated. However, through the design process we realized that we did not need more than one database connection and might as well make it static, making the entire API much simpler and easier to use.

14.4.4 Sequence Diagrams

In this project sequence diagrams were used to illustrate the interaction between a few of the components. For instance, we demonstrated how the user interface interacts with the business logic layer, which in turn communicates with the database component. This type of interaction was, in our opinion, the most essential one to explain, since it shows how the three main components interact with each other.

For creating said sequence diagrams, we experimented with two different tools. The first tool we used was ArgoUML. We quickly discovered that it was not flexible enough for our needs, since we were not able to illustrate complex sequences properly. Due to this, we resorted to manually fixing sequence diagrams in an image editing tool, called InkScape or by creating them from scratch in Microsoft Visio 2007. Visio was good enough for our needs, but the program had various minor, yet frustrating behavioral issues with regard to drawing the actual diagrams. Nevertheless, we were able to produce the diagrams we were looking for in the end.

We found the sequence diagrams to be very useful medium for describing system interactions, since it can give the reader an overview of the essentials, which otherwise might be difficult to gain from text alone. It is almost essential for expressing in what way classes interact in components lacking structural relations. The insight, we gained from using the aforementioned tools, was that we require a sequencing tool, which arranges the diagrams neatly by convention without too much effort, rather than having to resort to tweaking details to get the desired result.

14.4.5 UI Design

The user interface was designed begun by using the interaction spaces from the analysis to make mock-ups of each screen. By making mock-ups before the implementation it was easy to ensure consistency.

We used a drawing program (Inkscape) that is based on scalable vector graphics, which makes it easy to resize and move objects around. Because we did not use a specific program for mock-ups, such as Balsamiq, we were able to use the style we wanted. However, having to draw everything from scratch did make the process more time consuming.

Mock-ups are a very good way of getting the design discussed before spending time on implementing it. If a larger group of programmers works on different parts of a system, they will most often design the parts differently. If consistency is desired, some of the parts must be redesigned. Having to redo this afterwards can be time consuming and cause unnecessary problems.

14.5 Test and Evaluation

14.5.1 Unit-testing

We have described which classes we wanted to unit-test, and to what extent, in the design document. Having decided what tests we wanted to perform prior to implementating helped facilitate parallel development, enabling one developer to work on a class while another developer was writing tests for the class. It also ensured that we performed the tests, and did not just forget about them.

The purpose of these tests was to prevent bugs from propagating upwards through the next layer - and in most cases our tests succeded in ensuring this. Thus, we only experienced minor bugs in the database and logic layers while implementing the frontend layer. Futhermore, our unit-tests help spread familiarity with the code to developers who had not implemented the specific class. We consider our unit-testing of Foodolini a success, as we achieved our goal: not having to solve bugs in the database layer while implementing the frontend.

14.5.2 Evaluation

Testing a system for usability problems will usually be a part of system development. If the developers have been working on both the analysis, design and implementation, they know the thoughts and decisions that have been made, which may lead to trouble seeing which parts of the program are unclear or confusing. Therefore it can be necessary to get outsiders' view of the system. If something is not explained well enough, or works in an unintuitive way, the usability test will reveal this. These problems can then be addressed by the developers. Which problems to address depends on the time frame. Because we had a short time frame, we focused on the critical, as these are the most severe, and the cosmetic, as these are often easy to fix.

On small systems a qualitative evaluation is often preferred, as a larger test group not necessarily will find new problems. Therefore we only used three test users, and could thereby focus on the usability problems, and how to solve them, instead of spending time on tests and evaluation.

The tests were conducted in a kitchen, to make the surroundings more realistic for the test users. It also lowered the formality of the tests, which made the test subjects more comfortable. The surroundings are very formal in a laboratory, and experience from our earlier projects has shown that the two-way mirrors make the test users uncomfortable. If the tests are small, and without observers, conducting the tests in less formal environments will often make the test users more comfortable, and more natural in their interaction with the system.

For the data analysis we used the Instant Data Analysis (IDA) method. We had never tried this evaluation method before, but because of its focus on shortening the time spent finding problems, is was ideal for this project. Solely relying on notes taken during the test was not an obstacle in the evaluation, because the tests were still clear in mind. But excluding video documentation does demand thorough note taking and the evaluation must be conducted on the day of the usability tests. The plan was only to have one facilitator, as the IDA method prescribes, but we actually ended up having two facilitators. During the evaluation two facilitators led the meeting, and one continued with the documentation and was helped by the test monitor.

By conducting this usability test and evaluation we discovered some bugs and usability problems we had not earlier realised were there. Furthermore, we gained experience in the performance of usability tests and evaluation of same. This may both help us anticipate which areas of the user interface could be a problem in later projects - and thus design them from this knowledge, and make us better testers of usability problems.

14.6 Future Work

14.6.1 Import/Export Ability

Given that there are several other sites and applications which manage recipes, various databases with recipes are available. An example of an application with a larger database of recipes is MealMaster, which also allows the user to export the recipes to a file.

There are a large amount of recipes available in this format (100.000+). The ability to import recipes in this format into Foodolini would therefore be a useful feature, especially for a commercial release. The value of this feature lies in the possible benefit to the user of being able to bring existing recipes into Foodolini and search for them, instead of having to enter them all manually.

In addition, the ability to export recipes would allow sharing between friends, as well as taking backups of them. From this point of view, export could also be a worthwhile feature to include in any future work.

Initially, the import/export feature was part of the implementation disposition - yet, it was deprioritised when it became apparent that other features would have to be concentrated on. The actual MealMaster format was parsed and ready, still there was a lack of support for handling ingredient quantities in various units of measurement, such as ounces, cups, etc. This also influenced our decision to put off this feature considerably, since handling ingredient quantities is a fairly complicated endeavour, requiring significant effort and time.

14.6.2 Corrections

We encountered several problems during our usability evaluation which are documented in the evaluation document. Some of these problems were easily addressed, such as responding to the user if a search revealed no results, or changing certain label names. Others were not fixed - either because of time constraints, or because it was deemed unnecessary to change it, as the users quickly found out what to do.

One such problem was that the activity selector opened upwards, hiding some options. This is a drop-down box and is a part of Gtk, and thus not something which we have intentionally designed to behave like this. It could be replaced with another form of menu, or the drop-down box could possibly be tweaked to work.

14.6.3 Enhanced Dieting and Exercise

With regard to dieting, the goal was to give the user an overview of what he/she has consumed in terms of calories, protein, carbohydrates and fat within a given day. This would allow the user to keep track of their calorie consumption. Our current implementation leaves a couple of things to be desired.

For the dieting feature to become more useful, a user should be able to track changes concerning body weight, as well as body fat percentage. This would enable them to stay on track, by determining whether calorie intake is either too high or too low, based on measured weight loss and/or changes in body fat composition. In turn, the user can adjust the calorie intake accordingly. People are different and have diverse levels of metabolism, so it would be natural to assume that the estimated calorie intake might be slightly off. These features, however, require that the user has a scale or a device for measuring body fat percentage, such as a caliper.

There exists different ways of estimating daily caloric needs. We decided to use the Mifflin-St Jeor formula, as mentioned earlier, since it is more accurate than the older, yet still popular, Harris-Benedict formula. A third method, called the Katch-McArdle formula or lean body mass (LBM) formula is more accurate than both of the aforementioned, but it requires that the user knows their LBM, which requires a body fat caliper. The LBM is the total weight of body tissue except fat. We could add an option to use the latter formula to estimate caloric needs, since some users, such as those who are already keeping track of their body fat percentage, may desire higher accuracy.

A helpful way for users to follow their progress would be by visualizing recorded statistics with a chart. The chart could display a series representing actual weight loss and a series showing target weight loss. Likewise, body fat percentage statistics could be visualized. Of course, this will require an additional feature, which makes it possible to define a weight loss goal which spans over a period of time, probably displayed in weeks.

The fundamental basis for the exercise functionality, is that it is for calculating and registering the amount of calories burned by performing a certain exercise. This means that other possible features, which could support or help the users plan and progress in a certain discipline, have not been proposed. For example, a runner may desire the ability to plan trips in the future, indicate a distance and a targeted pace. This would propel Foodolini into the area of exercise planning, rather than solely being a tool for tracking food inventory and dieting.

14.6.4 Improved Search Capability

The nature of the current search algorithm is fairly simple. It merely searches in the titles, categories and descriptions of recipes for keywords, and ranks them after their rating. Ingredients which may be expiring soon can also influence the ranking.

A future addition to the search capability would be to implement the dieting features entirely, and give Foodolini the ability to search recipes based on nutritional needs. Furthermore, the current search algorithm may not prove fast enough if a large database of recipes is to be searched. Yet, its limits have still not been tested.

Searching for ingredients could also be implemented, so users could search for recipes with the highest amount of some particular ingredient.

A last possibility could be to search for variations of the keywords, such as the declension of nouns and conjugation of verbs. For example, if the user searches for ″potatoes″, results will also include recipes with ″potato″ and so on.

14.6.5 User Administration System

The current user administration system is a remainder from past ideas, involving protecting the data of a group of users as a whole, such as a family, against destructive users, such as teens, or playful children - who might decide to delete another user or remove all the recipes containing disliked ingredients, or another user′s diet. To prevent this from happening, we introduced the idea of a recycle bin, which would be able to restore or confirm deleted data.

The recycle bin which were to limit access to the administrator was never implemented and ultimately rejected as a feature. Yet, the administration system remained for the purpose of creating and deleting users.

The feedback gained from the evaluation made it clear that the administration system was actually rather superfluous. This is due to the fact that anyone can select a certain registered user and modify its details. The only actions a user cannot perform are creating and deleting users from the system. One could argue that the ability to edit the details of any given user is just as potentially a destructive act as deleting a user. So, restricting the rights to create and delete users to the administrator does not really aid the interested party, in terms of data protection and system integrity at the end of the day.

The administration system should therefore be completely removed, and in turn allow anyone to create and delete users. This also makes sense due to the open nature of the system. Moreover, the extra menu option only adds additional confusion to the user, especially since the functionality is rarely used.

If the dieting feature were never to be implemented, it would be relevant to completely remove the necessity for users, since there would be no need to tie food consumption to specific users.

14.6.6 Web Service for Bar Code Association

Since no large official database for associating bar codes with products exists, it might be interesting to improve Foodolini by providing a web service where users can share bar code associations automatically. So that when a bar code has been associated with an `Ingredient` in one Foodolini deployment, this association may be shared accross all other Foodolini installations, granted that Internet access is available.

The easiest way to provide such a web service would probably be by making the business logic layer serialisable, most likely using custom serialisation, and then exposing a web service using Simple Access Object Protocol (SOAP). Thus, replacing the front-end layer with a SOAP web service. To facilitate queries on larger datasets it may also be necessary to replace SQLite with MySQL or another database, which can easily be done by writing an implementation of `SqlStrategy` for MySQL. Consequently, no rewrite of the `Repository` class is needed and the public API of the database layer may be preserved.

14.7 Conclusion

The focus of this project has been to learn to apply tools and methods for system development. Throughout this project we have followed the OOA&D method, supplemented with the ADRIA method for user interface design and IDA for usability evaluation. The usage of these methods ensured that we had many discussions and made decisions early on in the process. This was great, as it is easier to change a decision early in the analysis, rather than in the design or implementation phase, as new decisions may have more implications in these stages. We have also experienced that discussing features of the system in an early stage eases the division of large tasks into sub-tasks. In addition, it has allowed similar subtasks to be implemented in a more consistent and effective manner.

We can conclude that with a good analysis and design the implementation process can be fairly painless. In other words, it requires almost no breaking changes, since major decisions have already been made. This is compared to previous projects we have done, where we broke the internal API on multiple occasions, which was a very laborious and time consuming process.

We have also experienced that through design and discussion of the user interface, before implementation, it is possible for a large team to quickly implement a large, consistent user interface. A good example of the consistency the ADRIA method helped facilitate is the IngredientBrowser, which appears several places throughout Foodolini.

Throughout this project we have also acquired experience with UML in connection with both analysis and design. We have found that UML made it easier to discuss functionality (through use cases), class structure and collaboration (using sequence diagrams).

We used the IDA method in our usability tests of Foodolini, which is a slightly different method of evaluating a usability test, compared to what we have previously made use of. It allowed us to perform a quick, yet fairly thorough evaluation of Foodolini.

Finally, we learned to conduct comprehensive unit-tests of the implementation, and used it to find and work out errors quickly and early in the implementation phase.

Part VI

Appendices

APPENDIX A

Additional Task Models

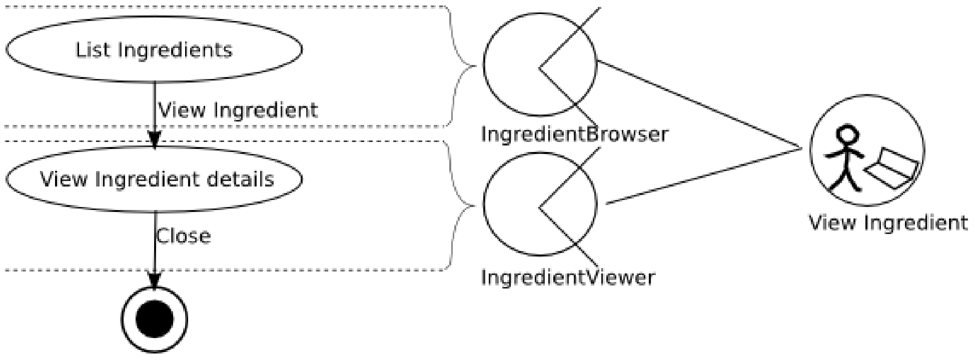


Figure A.1: Task model for viewing a new Ingredient

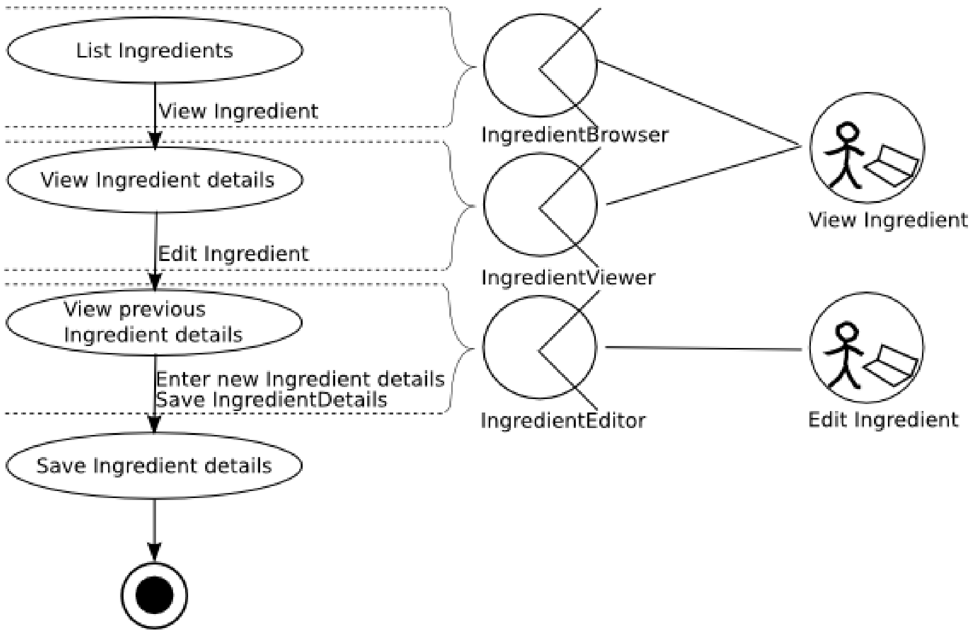


Figure A.2: Task model for editing a new Ingredient

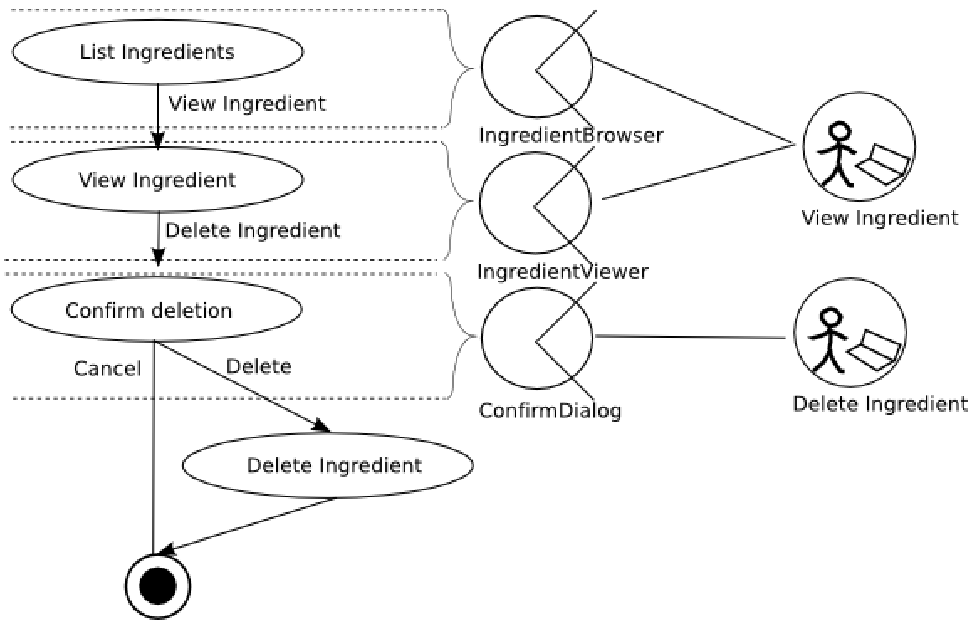


Figure A.3: Task model for deleting a new Ingredient

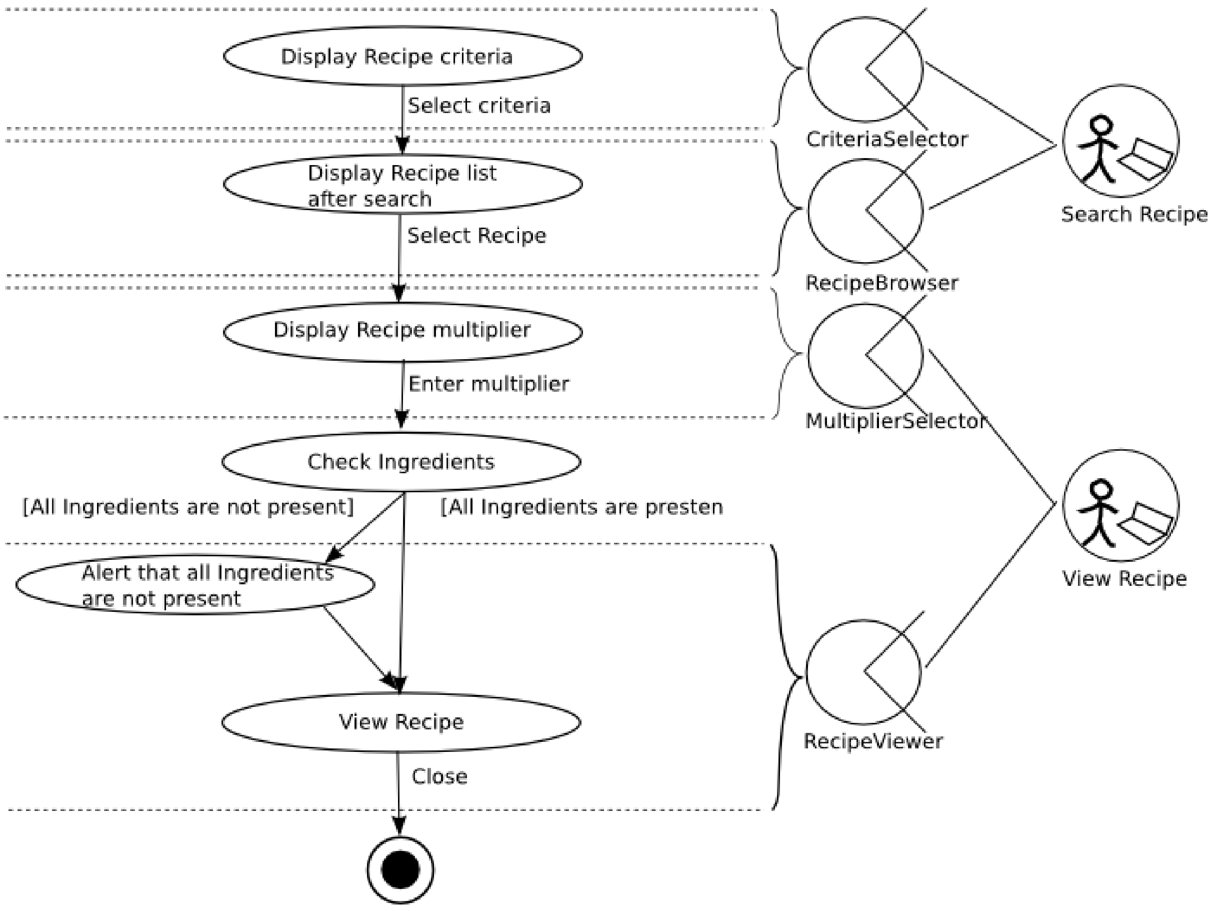


Figure A.4: Task model for viewing a Recipe

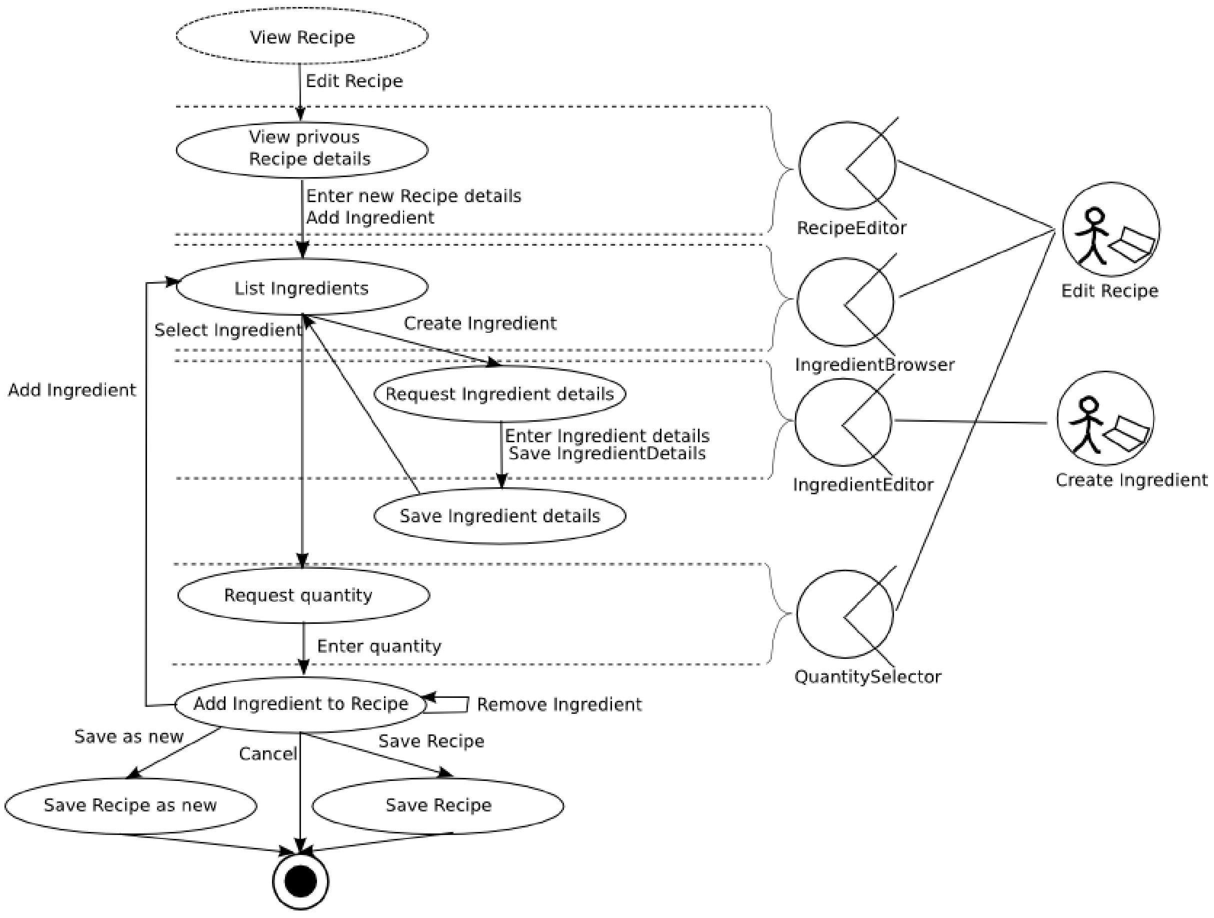


Figure A.5: Task model for editing a Recipe

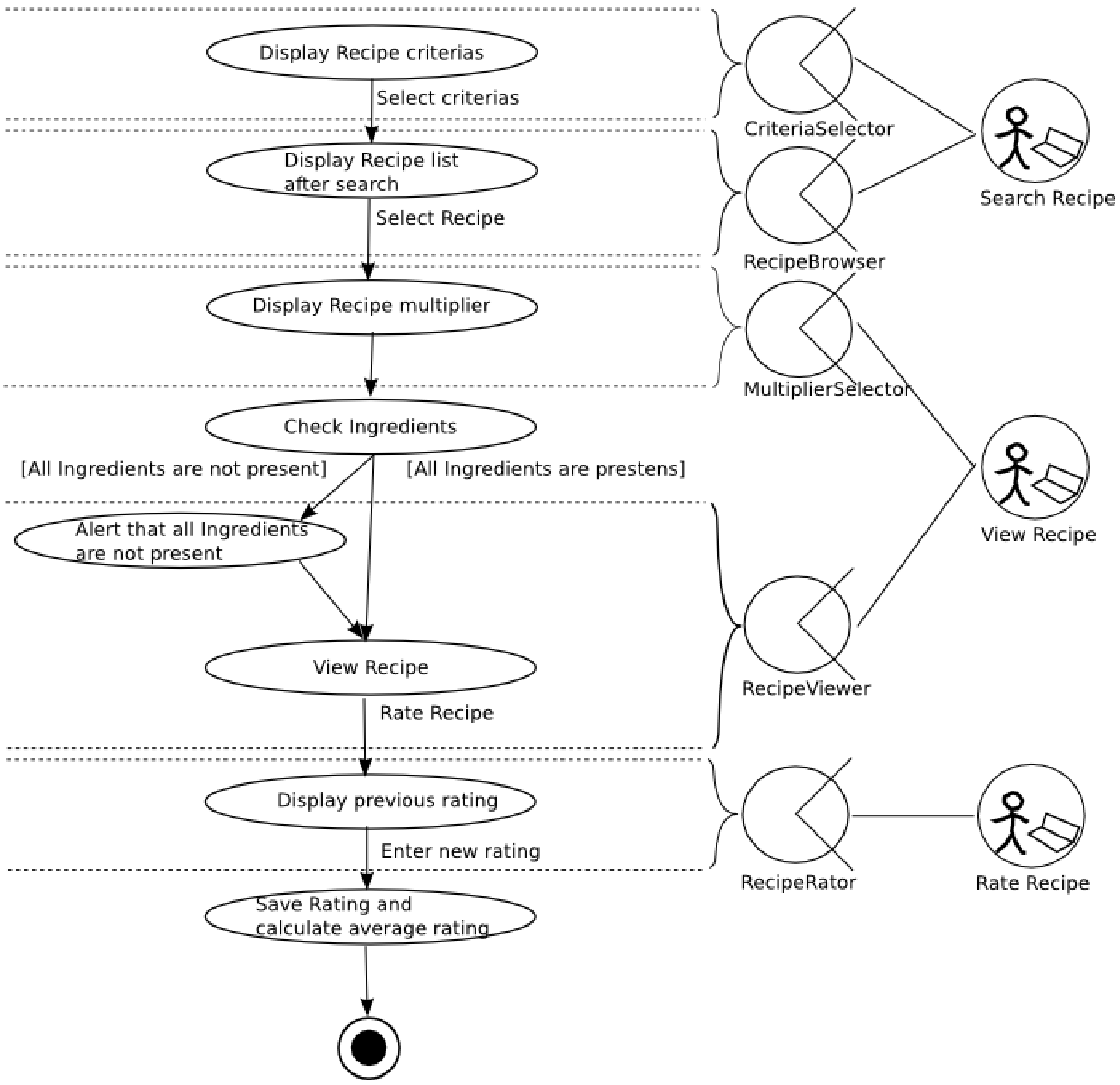


Figure A.6: Task model for rating a Recipe

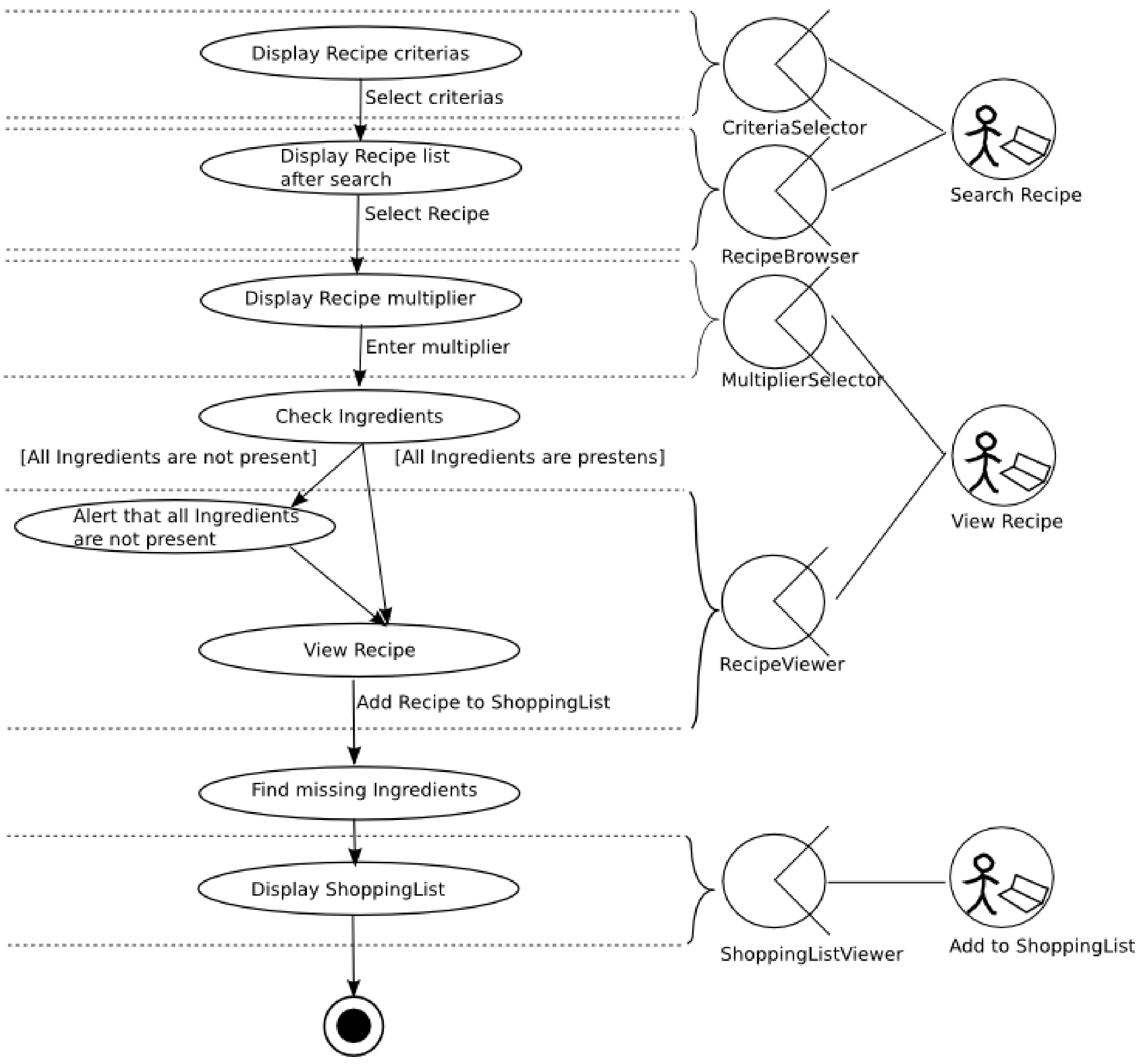


Figure A.7: Task model for adding a Recipe to the ShoppingList

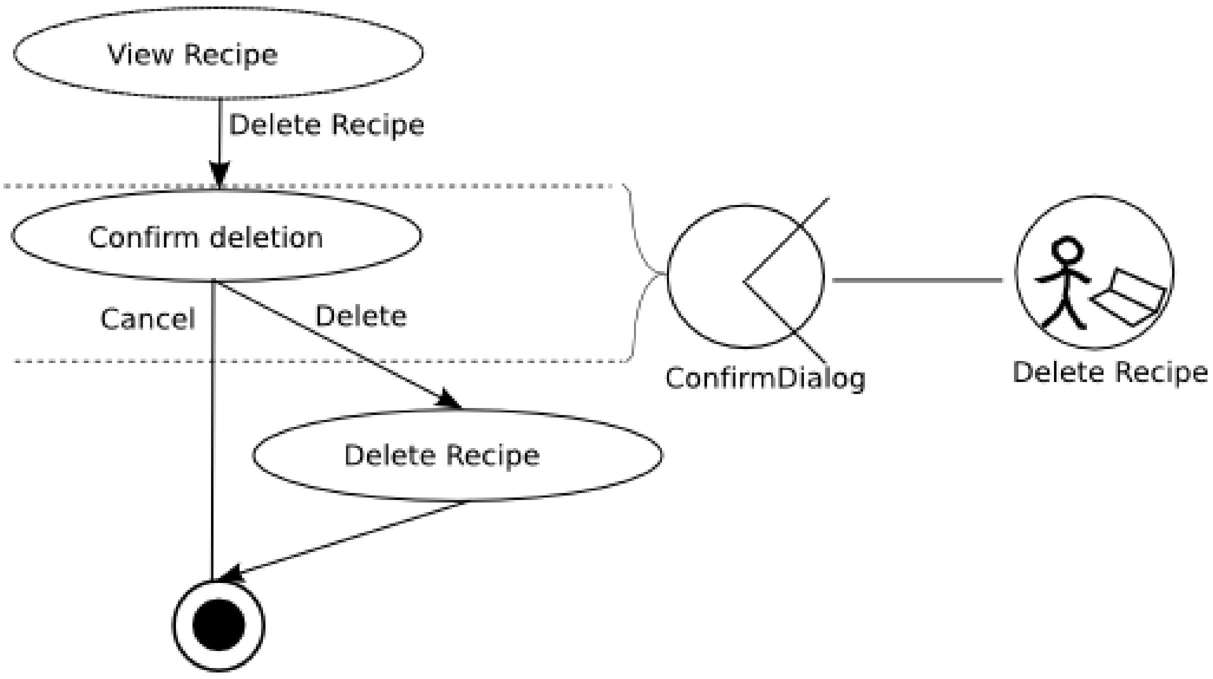


Figure A.8: Task model for deleting a Recipe

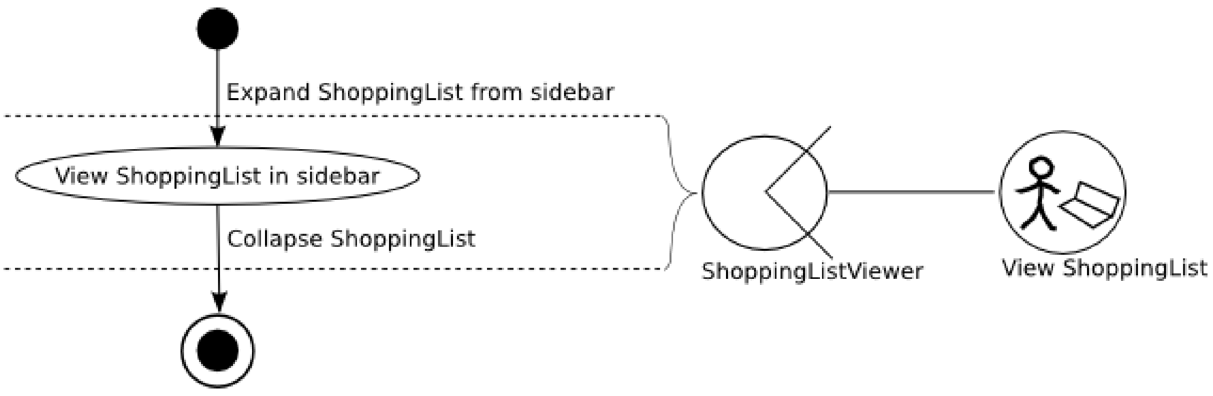

Expand ShoppingList from sidebar
View ShoppingList in sidebar
Collapse ShoppingList
ShoppingListViewer
View ShoppingList

Figure A.9: Task model for viewing the ShoppingList

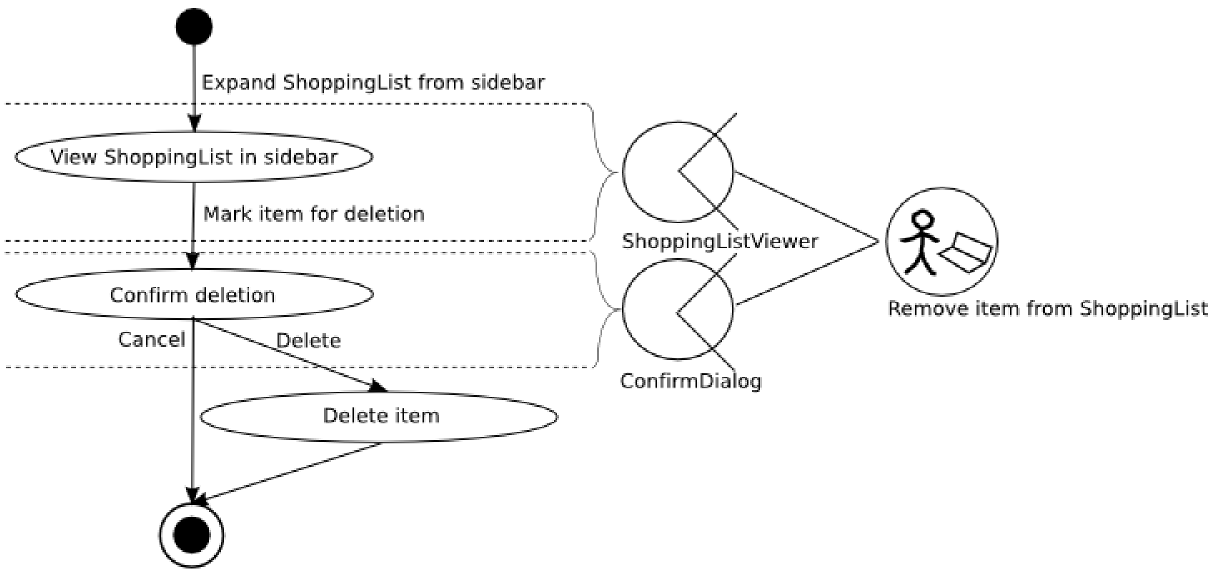


Figure A.10: Task model for removing an item from the ShoppingList

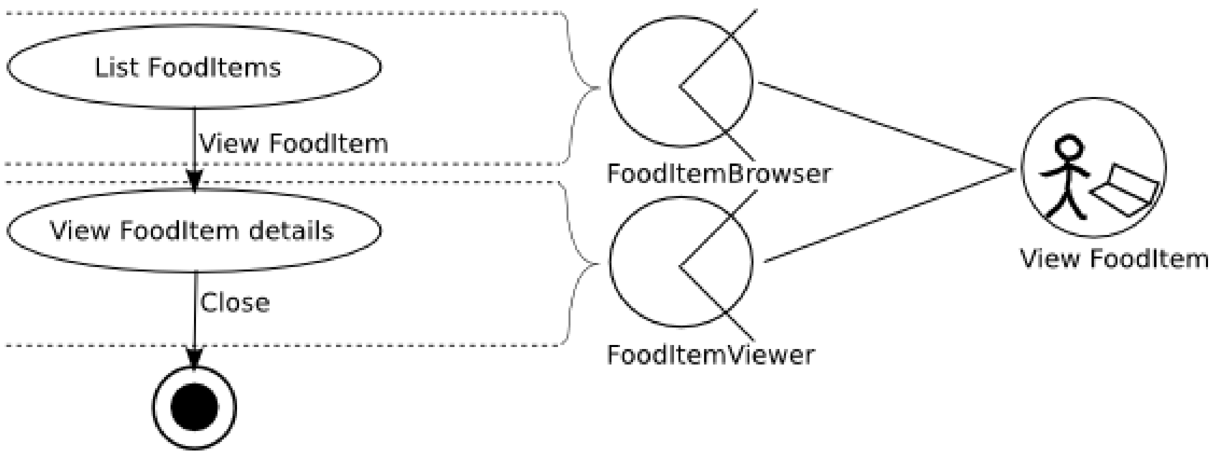


Figure A.11: Task model for viewing a FoodItem

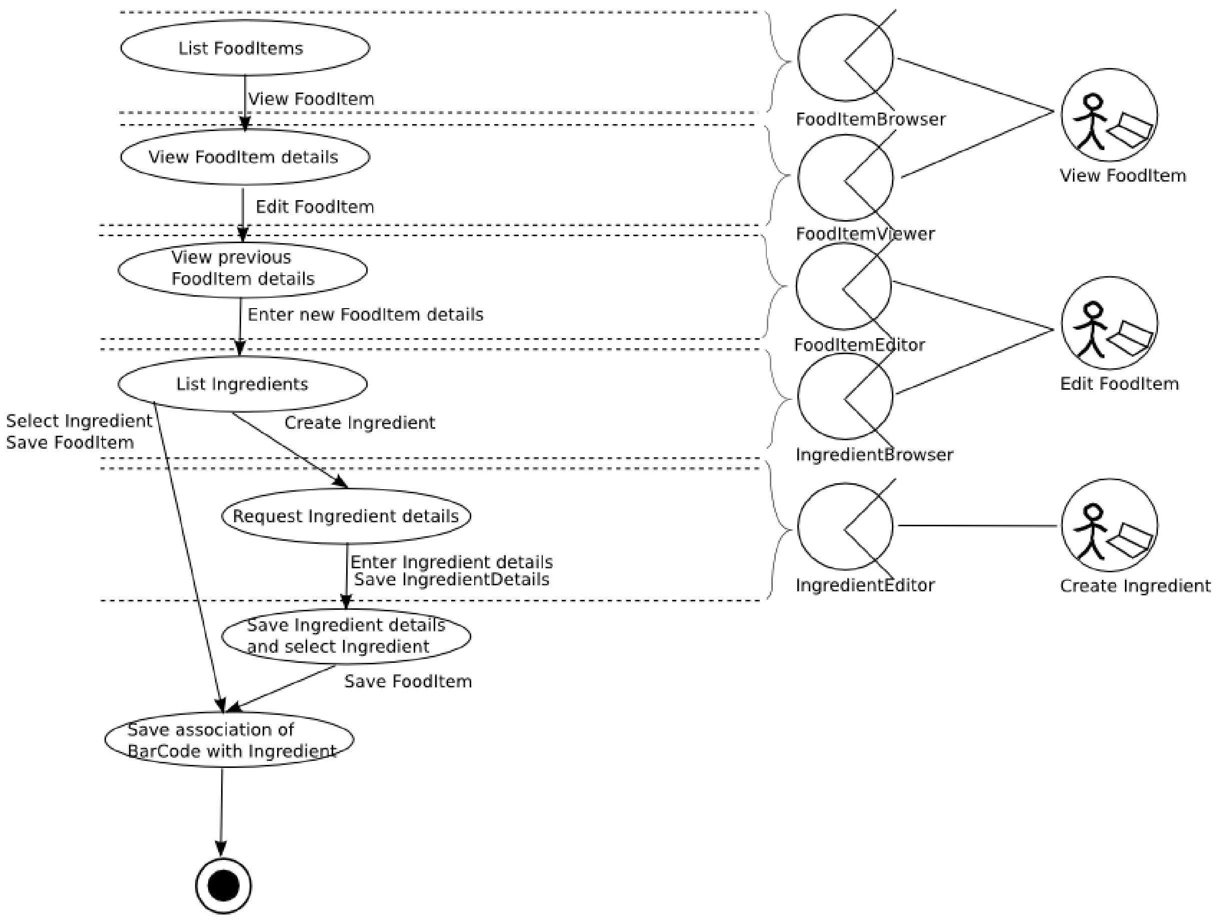

List FoodItems
View FoodItem
View FoodItem details
Edit FoodItem
View previous FoodItem details
Enter new FoodItem details
List Ingredients
Select Ingredient Save FoodItem
Create Ingredient
Request Ingredient details
Enter Ingredient details Save IngredientDetails
Save Ingredient details and select Ingredient
Save FoodItem
Save association of BarCode with Ingredient
FoodItemBrowser
FoodItemViewer
FoodItemEditor
IngredientBrowser
IngredientEditor
View FoodItem
Edit FoodItem
Create Ingredient

Figure A.12: Task model for editing a FoodItem

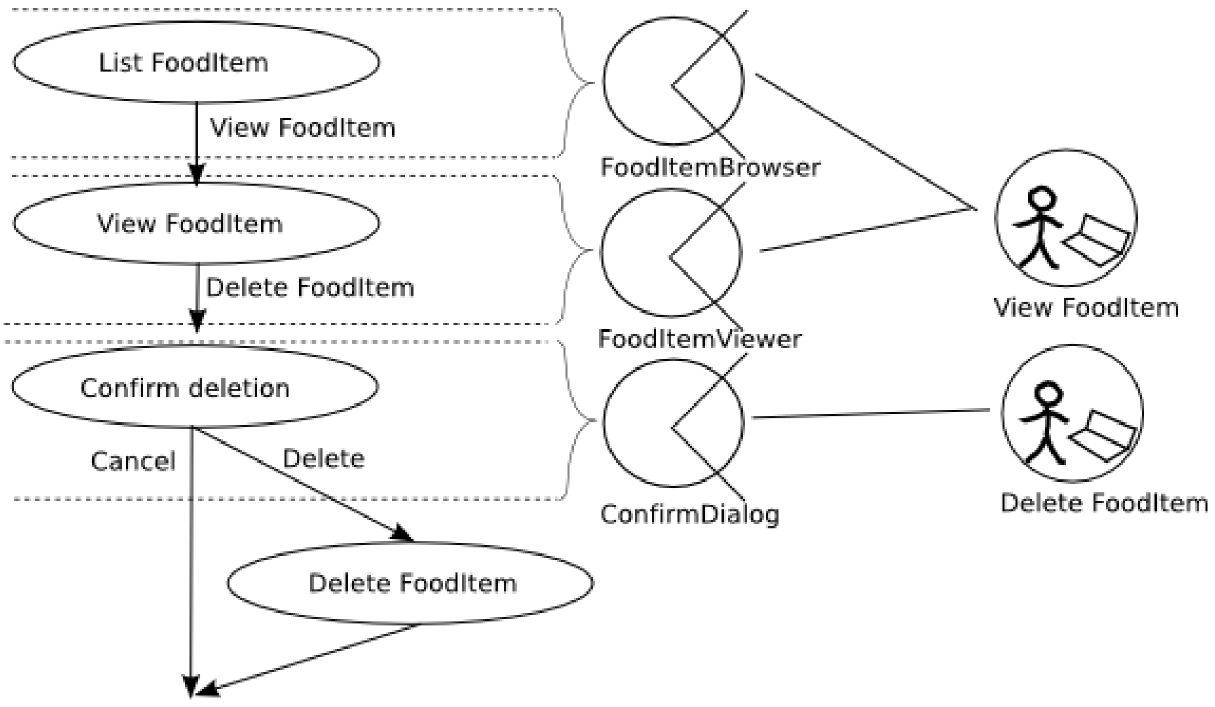


Figure A.13: Task model for deleting a FoodItem

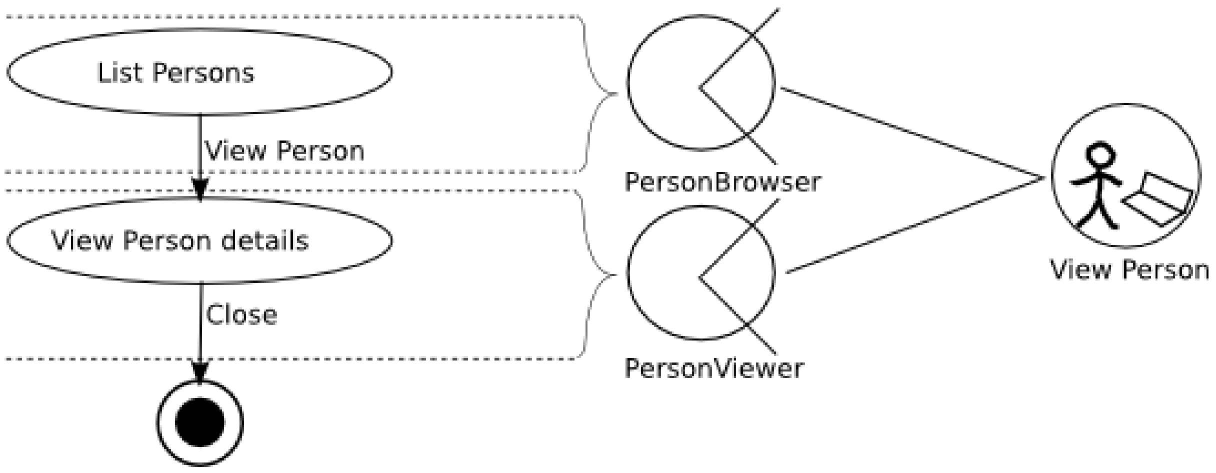


Figure A.14: Task model for viewing a new Person

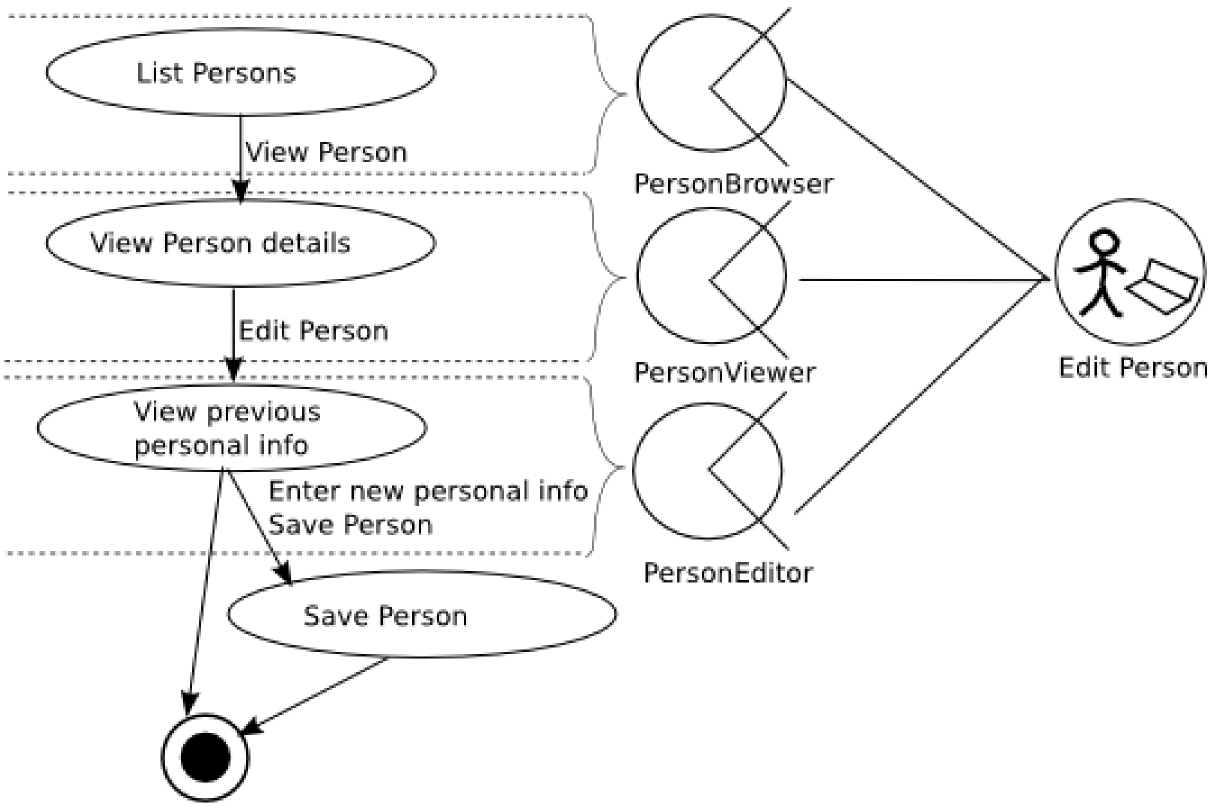


Figure A.15: Task model for editing a Person

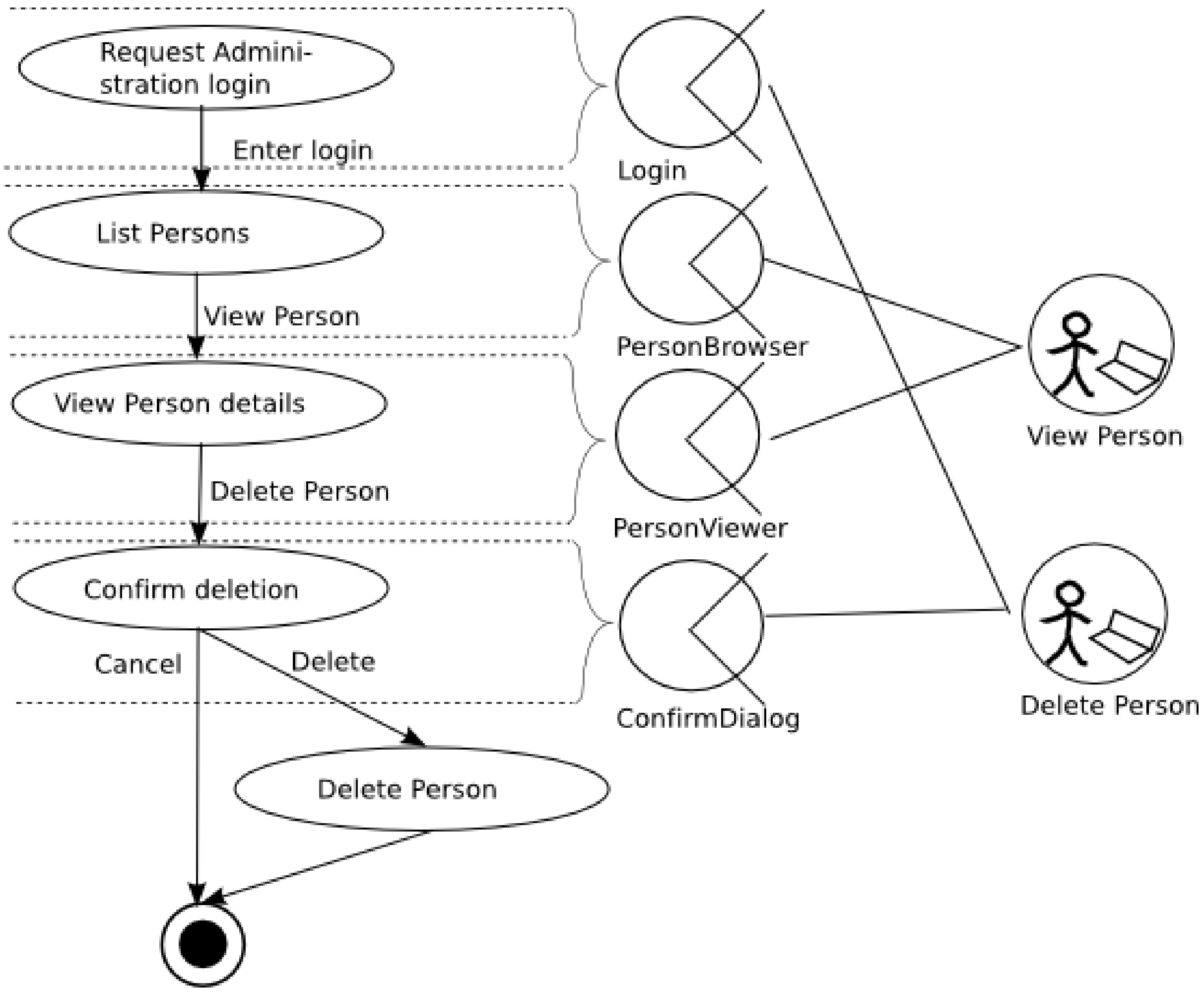


Figure A.16: Task model for deleting a Person

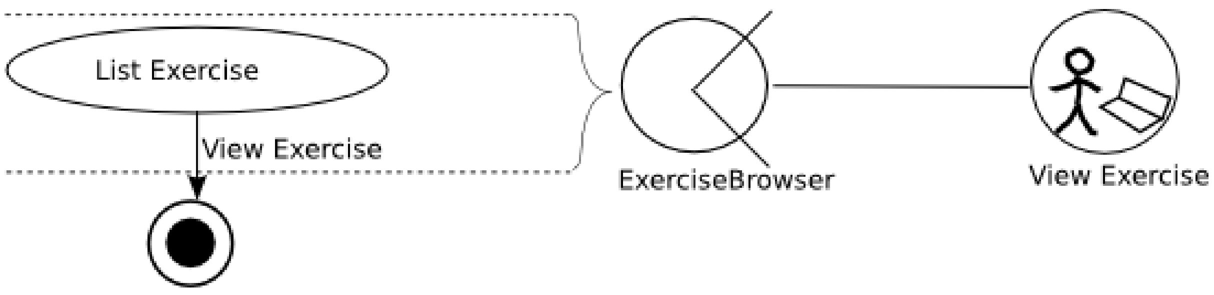


Figure A.17: Task model for viewing an Exercise

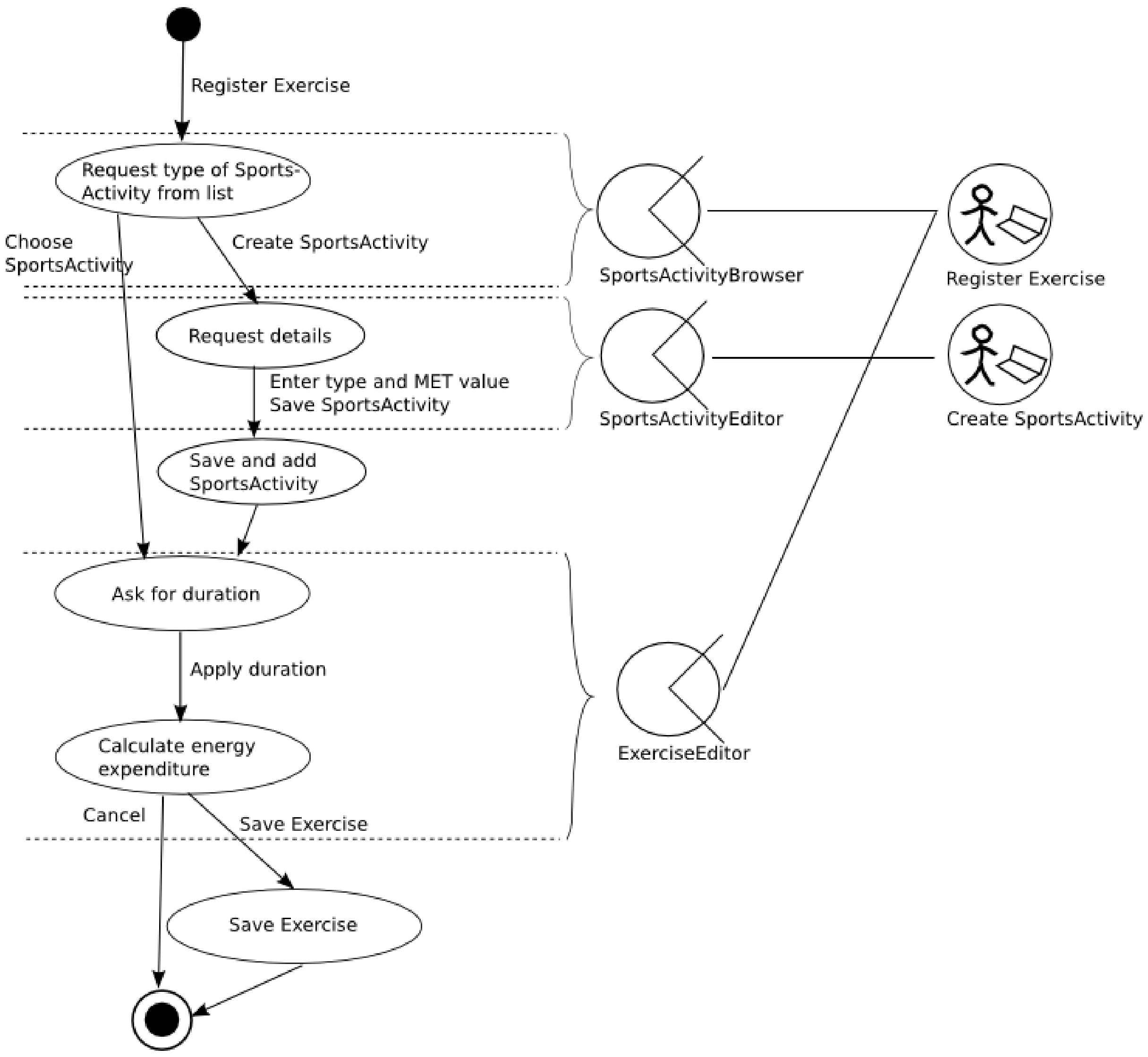


Figure A.18: Task model for registering an Exercise

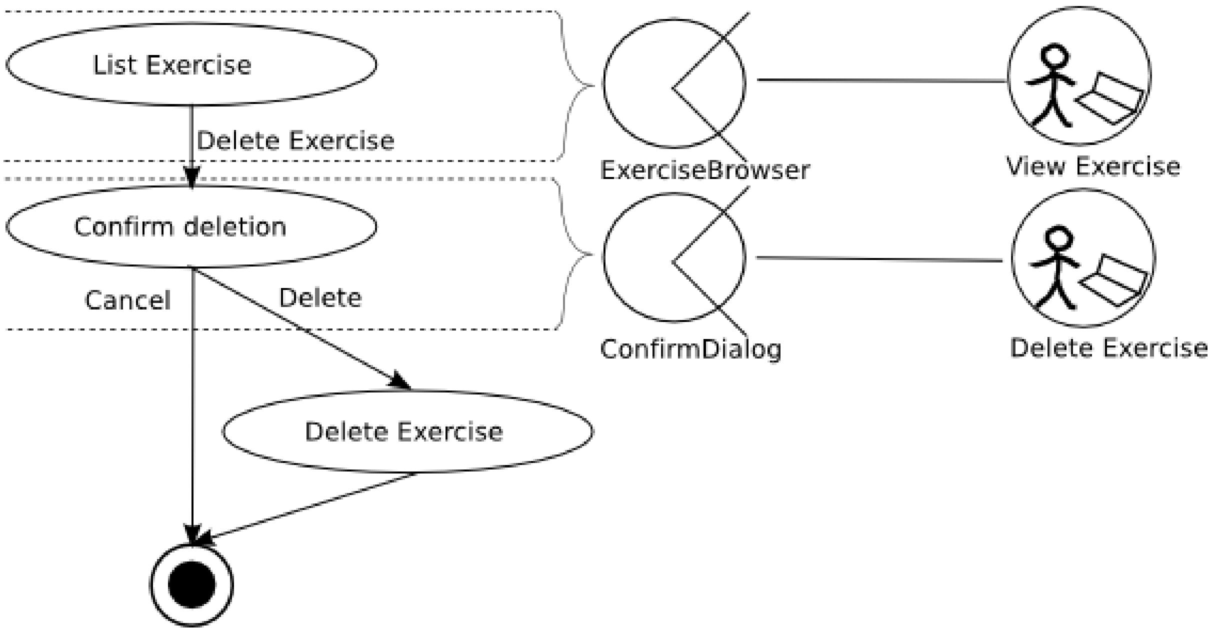


Figure A.19: Task model for deleting an Exercise

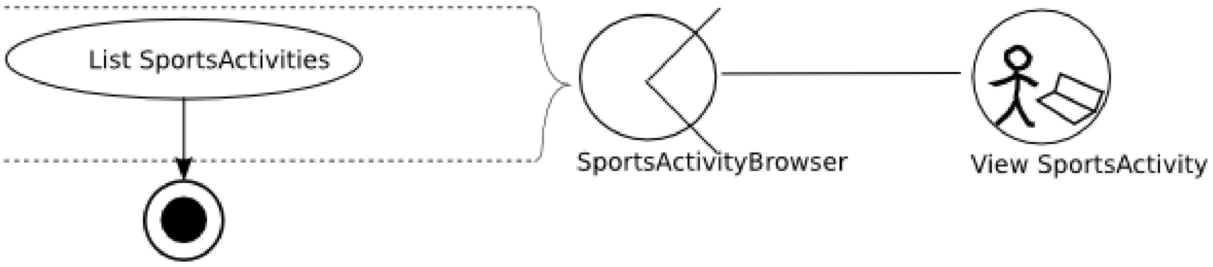


Figure A.20: Task model for viewing a SportsActivity

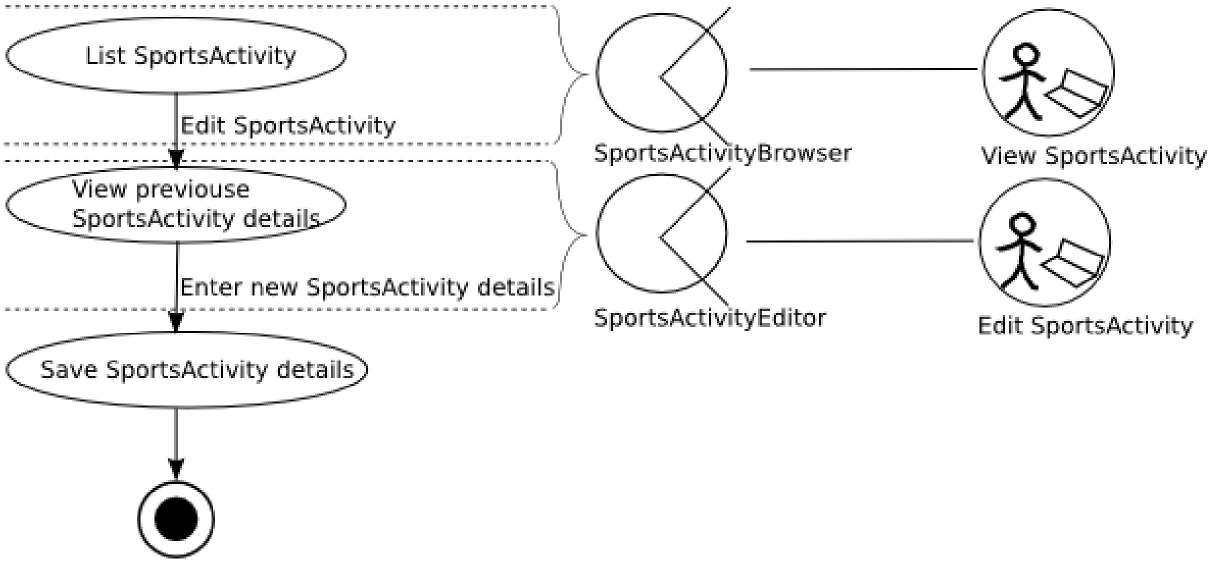


Figure A.21: Task model for editing a SportsActivity

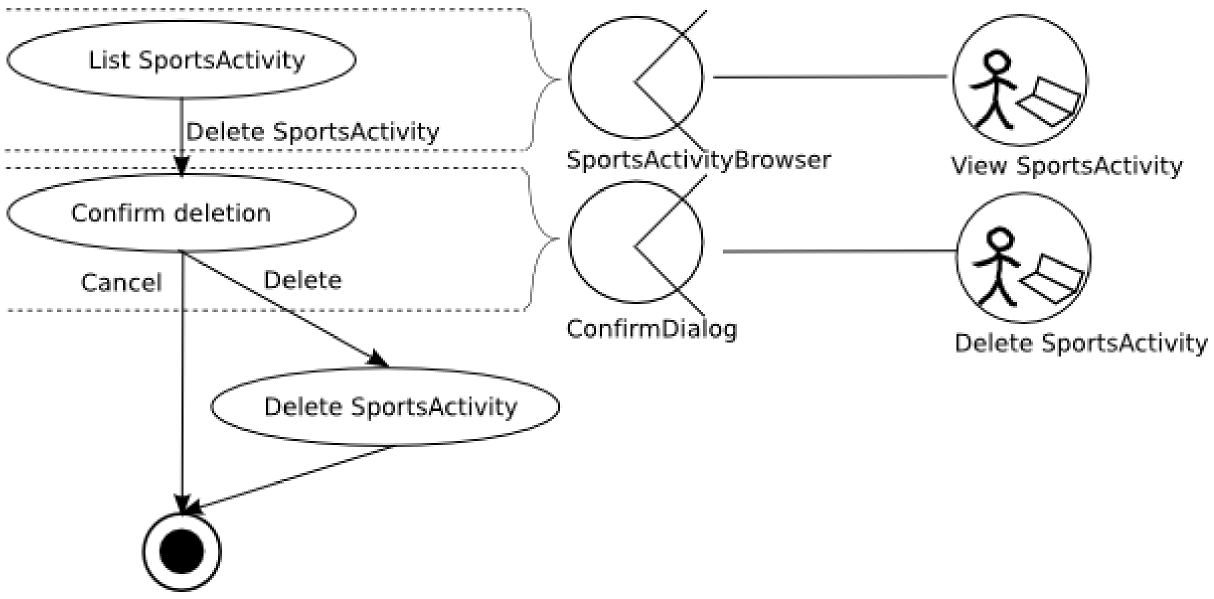


Figure A.22: Task model for deleting a SportsActivity

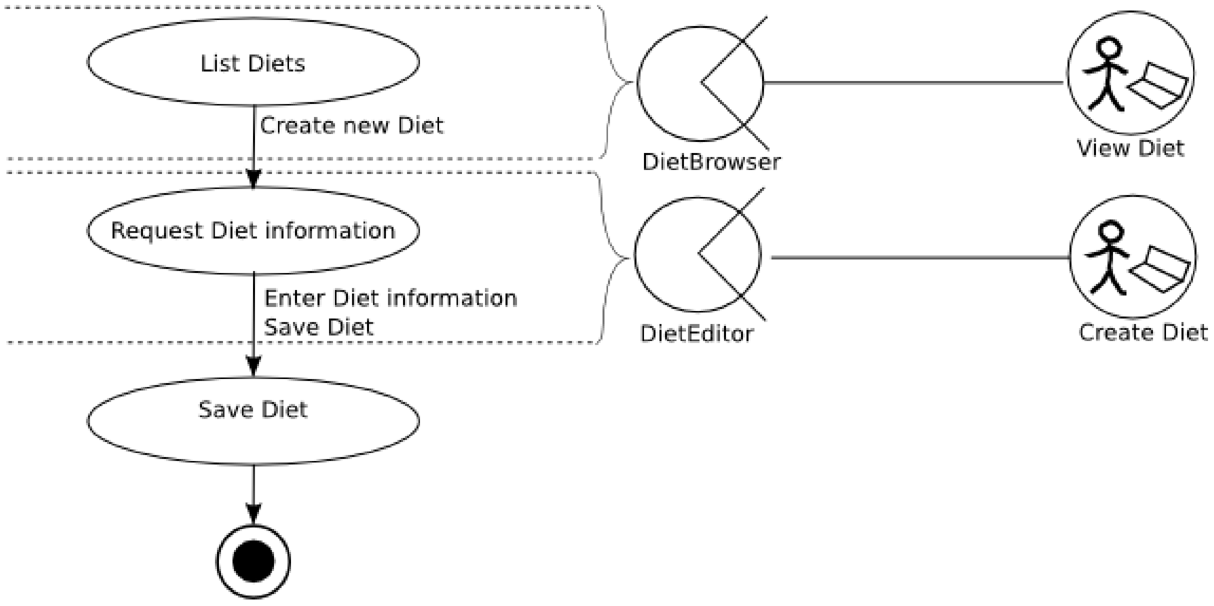


Figure A.23: Task model for creating a new Diet

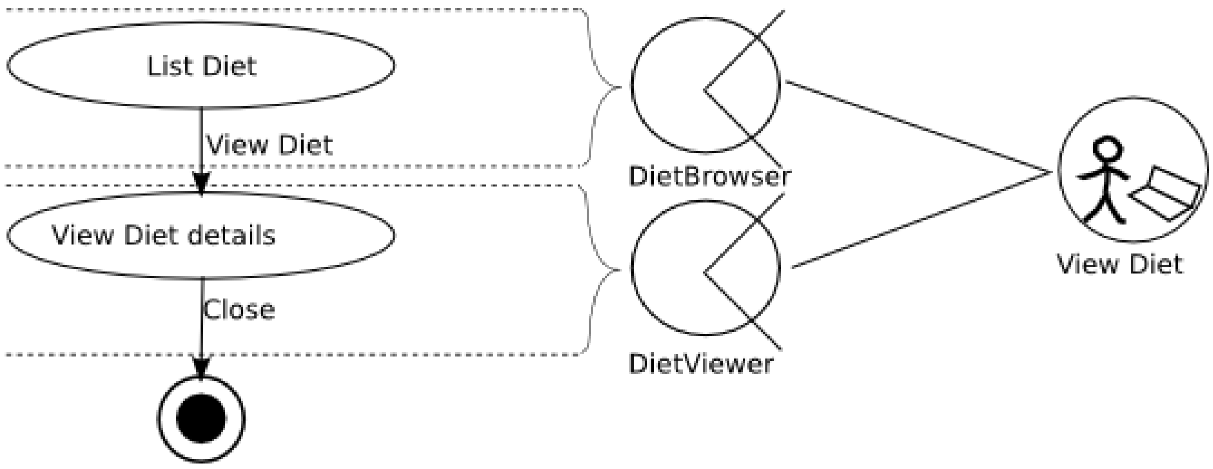


Figure A.24: Task model for viewing a Diet

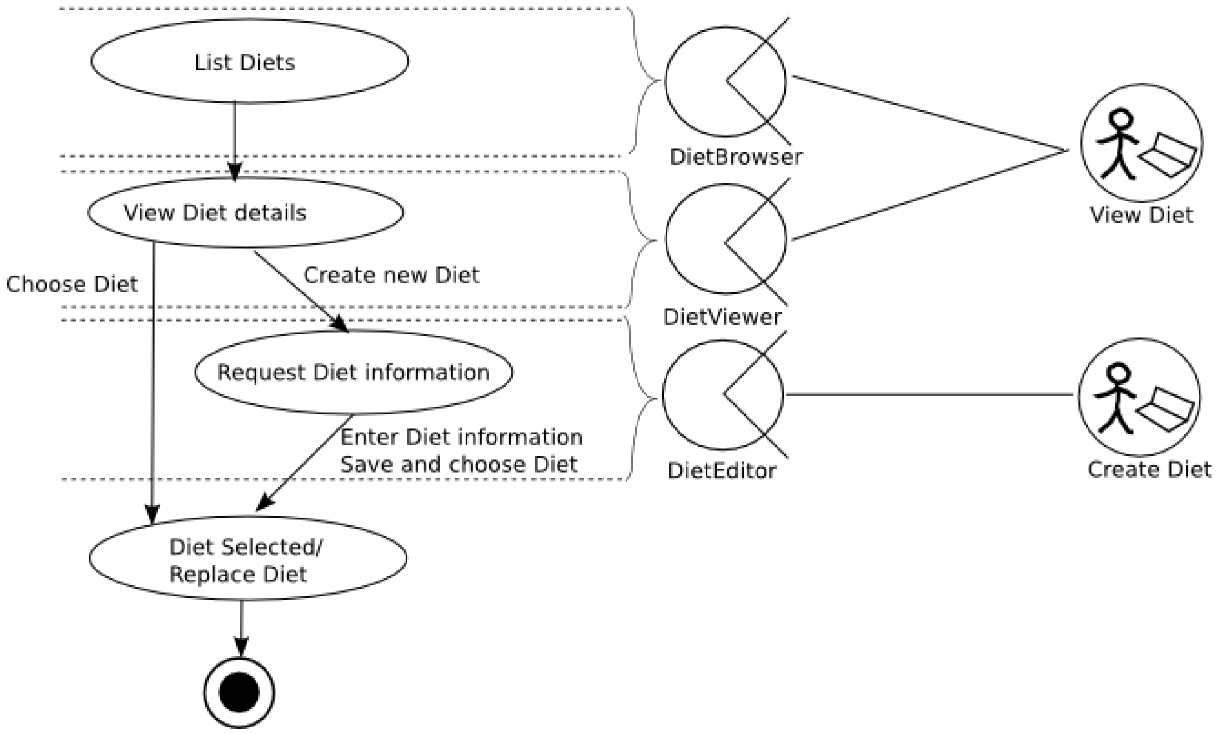


Figure A.25: Task model for choosing a Diet

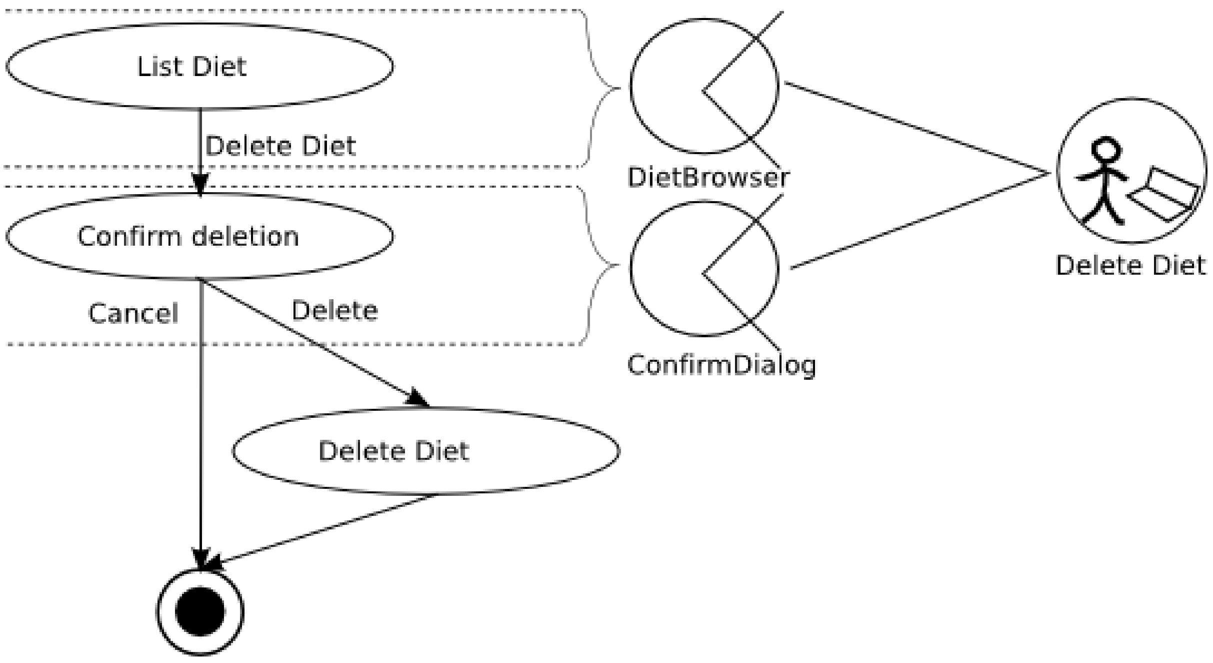


Figure A.26: Task model for deleting a Diet

APPENDIX B

Detailed mock-ups for Foodolini

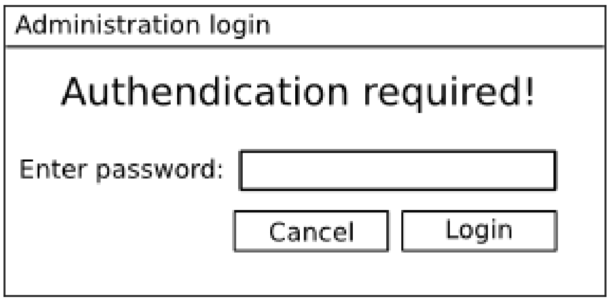
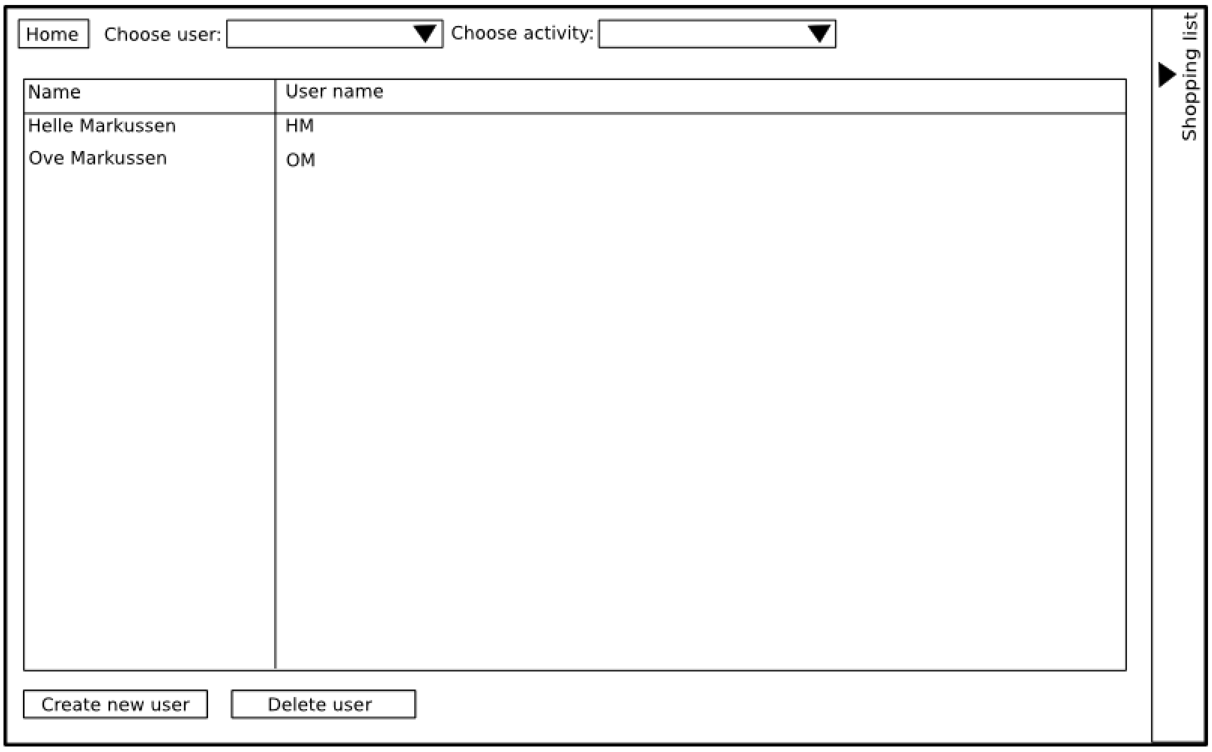
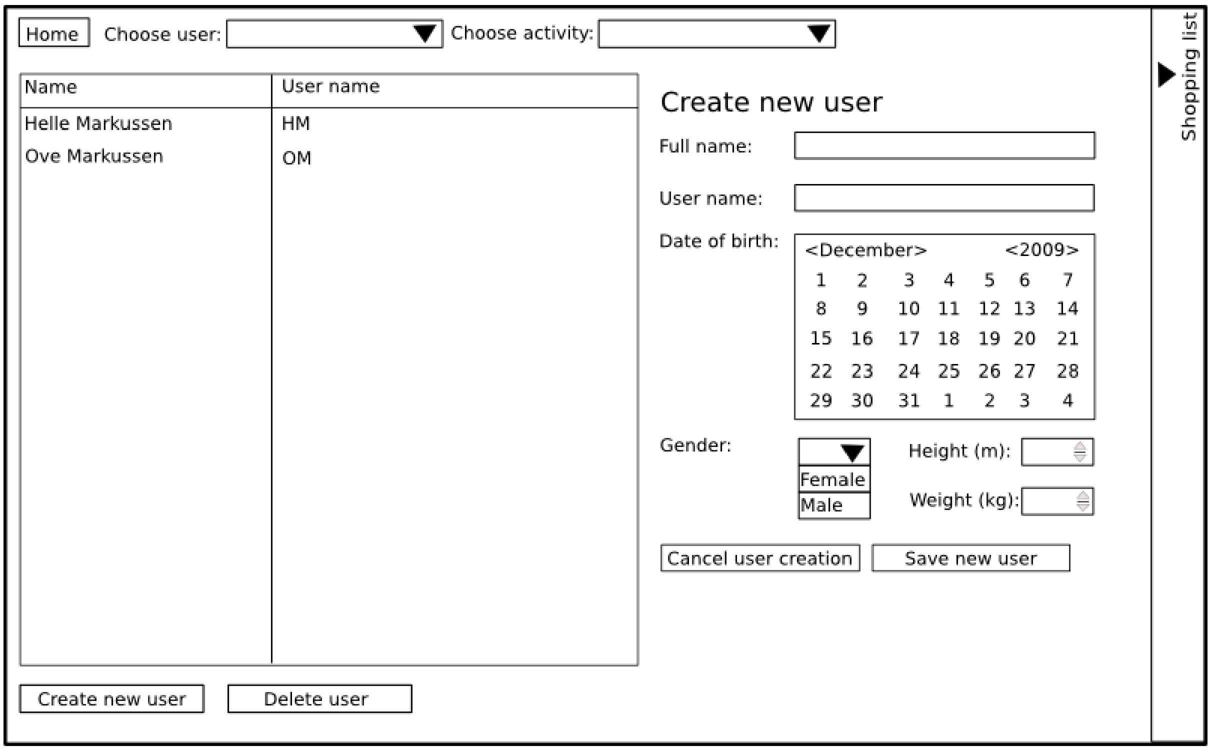


Figure B.1: Mock-up of Login.



Name	User name
Helle Markussen	HM
Ove Markussen	OM

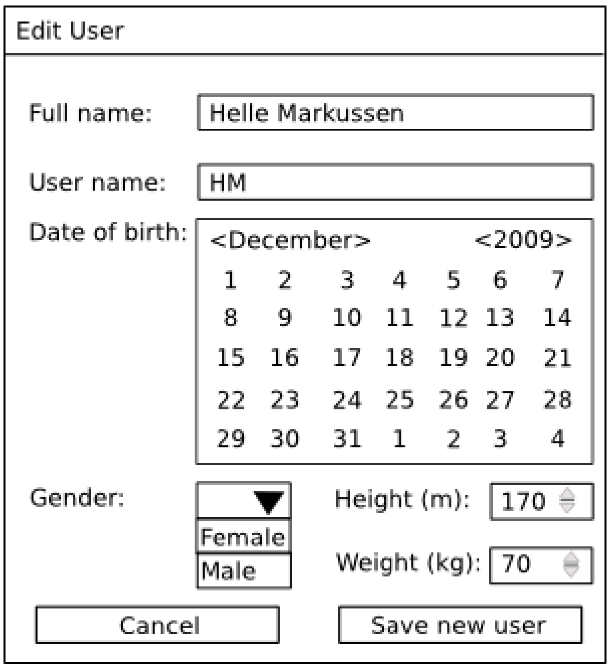
Figure B.2: Mock-up of Administration.



Name	User name
Helle Markussen	HM
Ove Markussen	OM

1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29	30	31	1	2	3	4

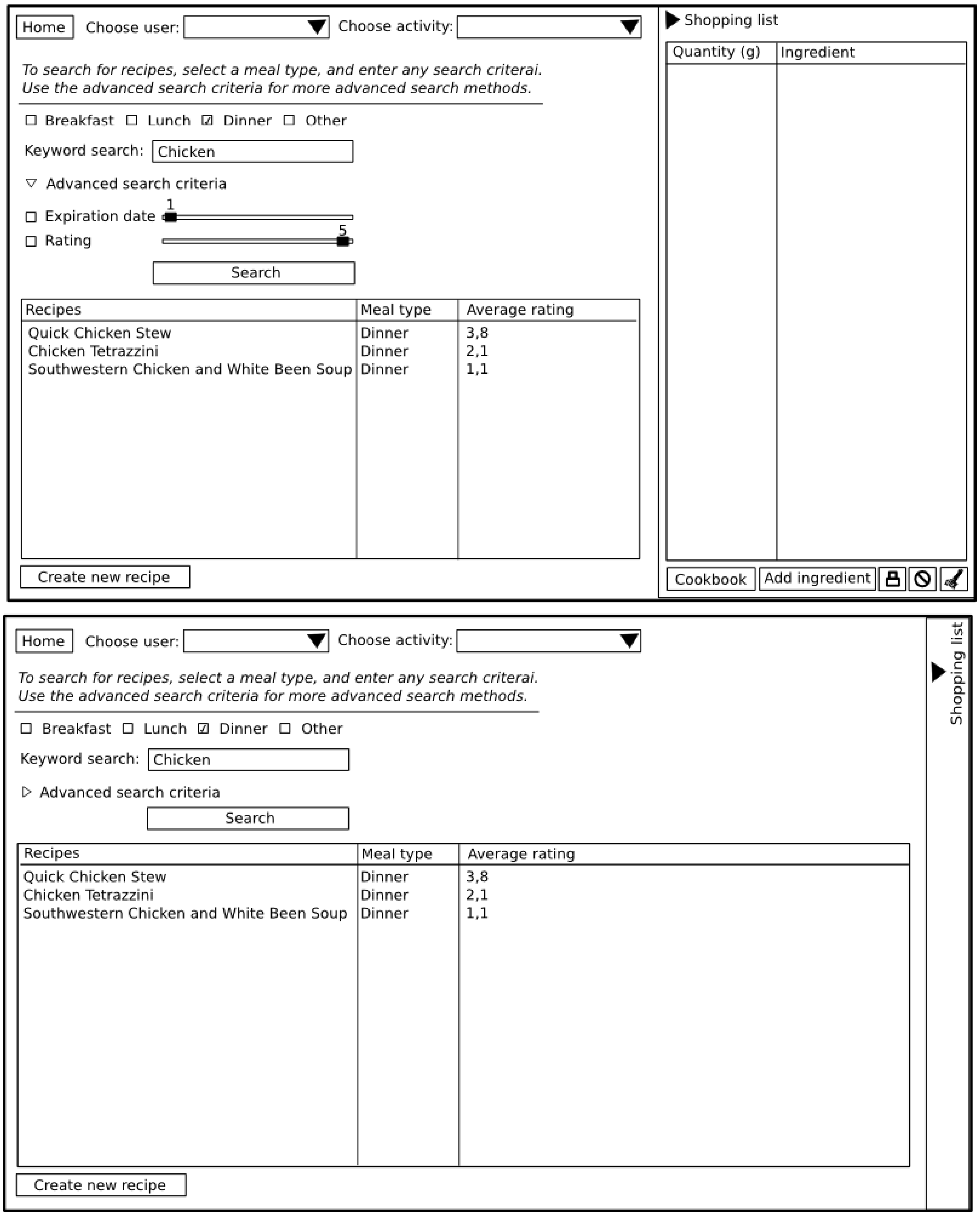
Figure B.3: Mock-up of Create User.



Full name: Helle Markussen
User name: HM
Height (m): 170
Weight (kg): 70

1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29	30	31	1	2	3	4

Figure B.4: Mock-up of Edit User.



To search for recipes, select a meal type, and enter any search criterai.
Use the advanced search criteria for more advanced search methods.

☐ Breakfast ☐ Lunch ☑ Dinner ☐ Other

Keyword search: Chicken

▽ Advanced search criteria

☐ Expiration date 1

☐ Rating 5

Recipes	Meal type	Average rating
Quick Chicken Stew	Dinner	3,8
Chicken Tetrazzini	Dinner	2,1
Southwestern Chicken and White Been Soup	Dinner	1,1

Quantity (g)	Ingredient

To search for recipes, select a meal type, and enter any search criterai.
Use the advanced search criteria for more advanced search methods.

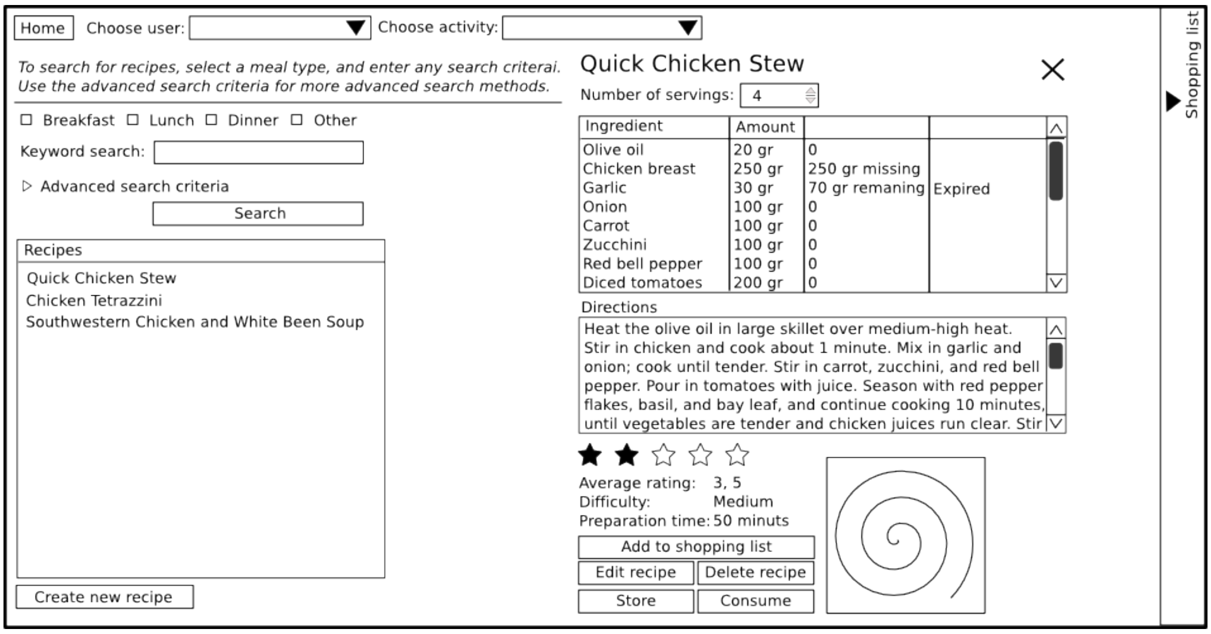
☐ Breakfast ☐ Lunch ☑ Dinner ☐ Other

Keyword search: Chicken

▷ Advanced search criteria

Recipes	Meal type	Average rating
Quick Chicken Stew	Dinner	3,8
Chicken Tetrazzini	Dinner	2,1
Southwestern Chicken and White Been Soup	Dinner	1,1

Figure B.5: Mock-up of Cookbook.



Home Choose user: Choose activity:

Shopping list

To search for recipes, select a meal type, and enter any search criterai. Use the advanced search criteria for more advanced search methods.

☐ Breakfast ☐ Lunch ☐ Dinner ☐ Other

Keyword search:

▷ Advanced search criteria

Search

Recipes

Quick Chicken Stew
Chicken Tetrazzini
Southwestern Chicken and White Been Soup

Create new recipe

Quick Chicken Stew

Number of servings: 4

Ingredient	Amount		
Olive oil	20 gr	0	
Chicken breast	250 gr	250 gr missing	
Garlic	30 gr	70 gr remaning	Expired
Onion	100 gr	0	
Carrot	100 gr	0	
Zucchini	100 gr	0	
Red bell pepper	100 gr	0	
Diced tomatoes	200 gr	0	

Directions

Heat the olive oil in large skillet over medium-high heat. Stir in chicken and cook about 1 minute. Mix in garlic and onion; cook until tender. Stir in carrot, zucchini, and red bell pepper. Pour in tomatoes with juice. Season with red pepper flakes, basil, and bay leaf, and continue cooking 10 minutes, until vegetables are tender and chicken juices run clear. Stir

★★☆☆☆

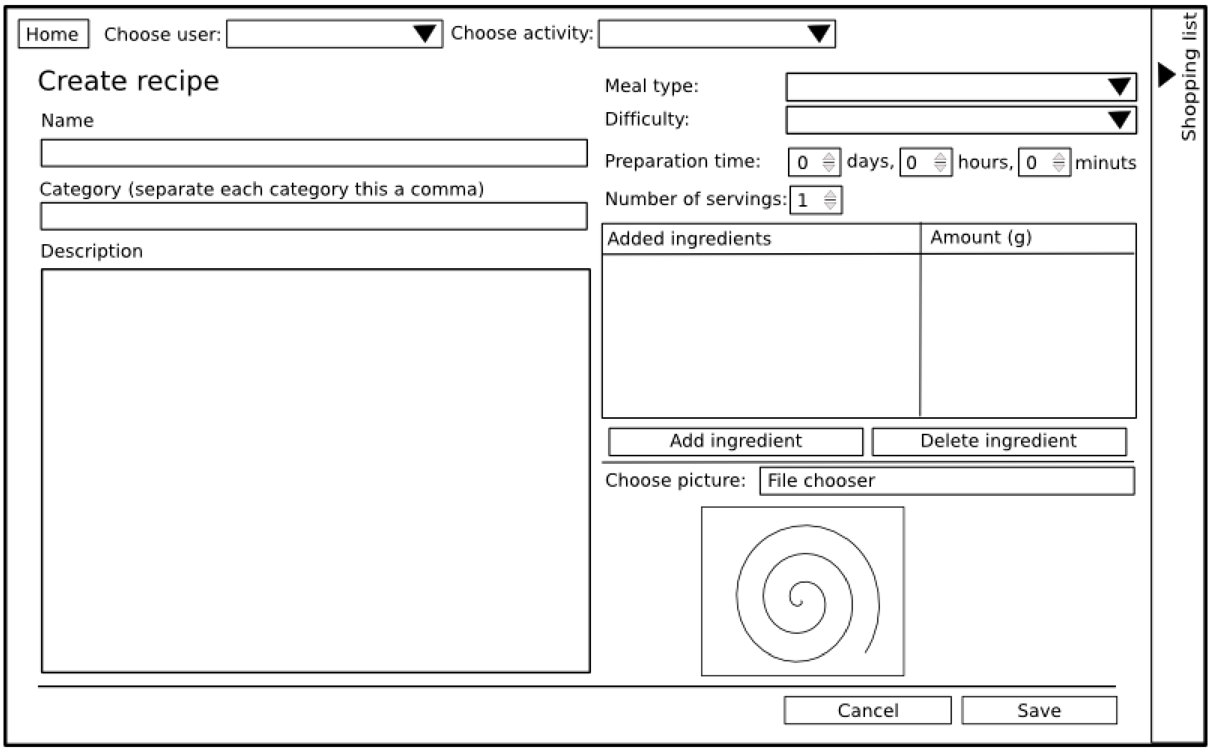
Average rating:	3, 5
Difficulty:	Medium
Preparation time:	50 minuts

Add to shopping list

Edit recipe Delete recipe

Store Consume

Figure B.6: Mock-up of View Recipe.



Home Choose user: Choose activity:

Shopping list

Create recipe

Name

Category (separate each category this a comma)

Description

Meal type:

Difficulty:

Preparation time: 0 days, 0 hours, 0 minuts

Number of servings: 1

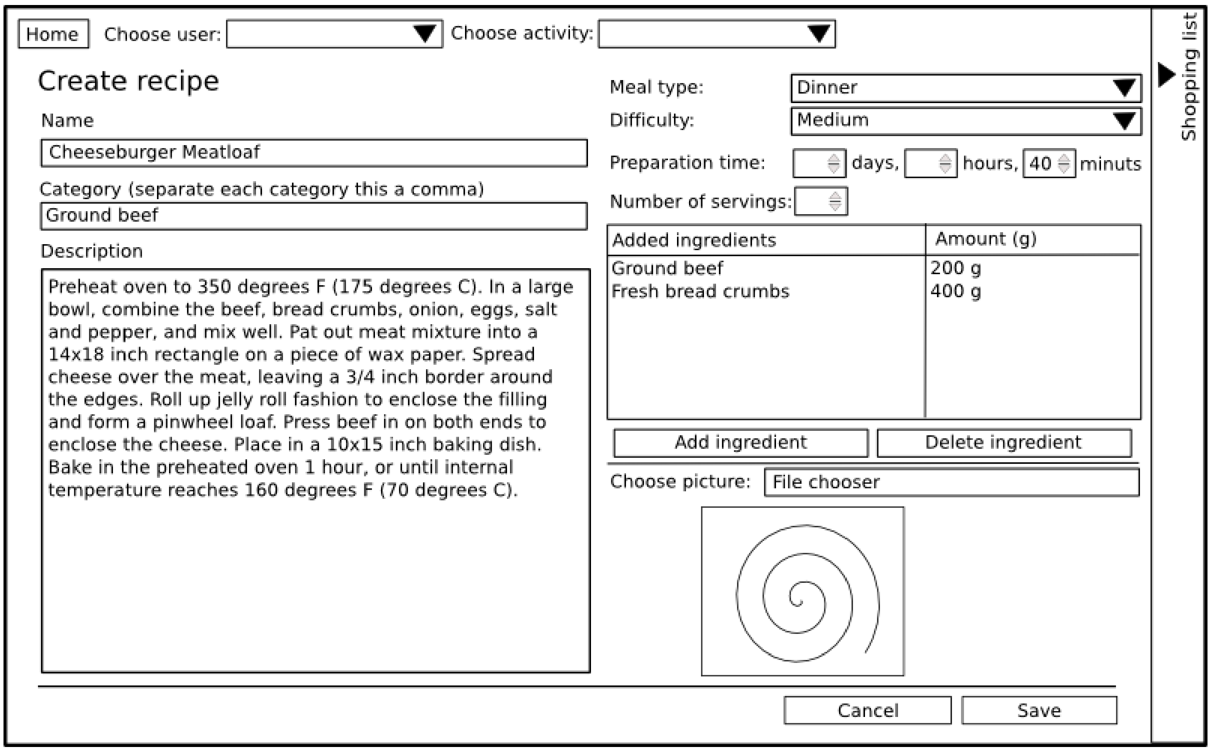
Added ingredients	Amount (g)

Add ingredient Delete ingredient

Choose picture: File chooser

Cancel Save

Figure B.7: Mock-up of Create Recipe.



Create recipe

Name: Cheeseburger Meatloaf

Category (separate each category this a comma): Ground beef

Description:

Preheat oven to 350 degrees F (175 degrees C). In a large bowl, combine the beef, bread crumbs, onion, eggs, salt and pepper, and mix well. Pat out meat mixture into a 14x18 inch rectangle on a piece of wax paper. Spread cheese over the meat, leaving a 3/4 inch border around the edges. Roll up jelly roll fashion to enclose the filling and form a pinwheel loaf. Press beef in on both ends to enclose the cheese. Place in a 10x15 inch baking dish. Bake in the preheated oven 1 hour, or until internal temperature reaches 160 degrees F (70 degrees C).

Meal type: Dinner

Difficulty: Medium

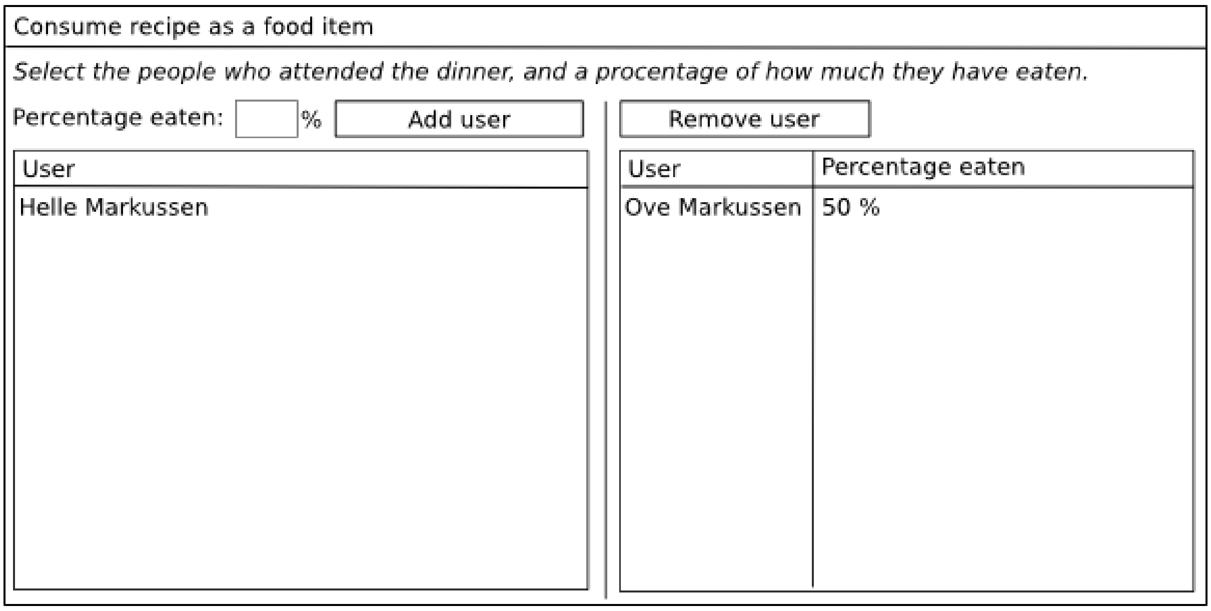
Preparation time: days, hours, 40 minuts

Number of servings:

Added ingredients	Amount (g)
Ground beef	200 g
Fresh bread crumbs	400 g

Choose picture: File chooser

Figure B.8: Mock-up of Edit Recipe.



Consume recipe as a food item

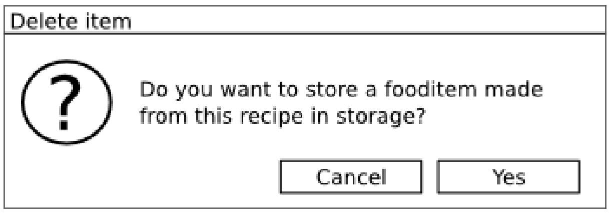
Select the people who attended the dinner, and a procentage of how much they have eaten.

Percentage eaten: %

User
Helle Markussen

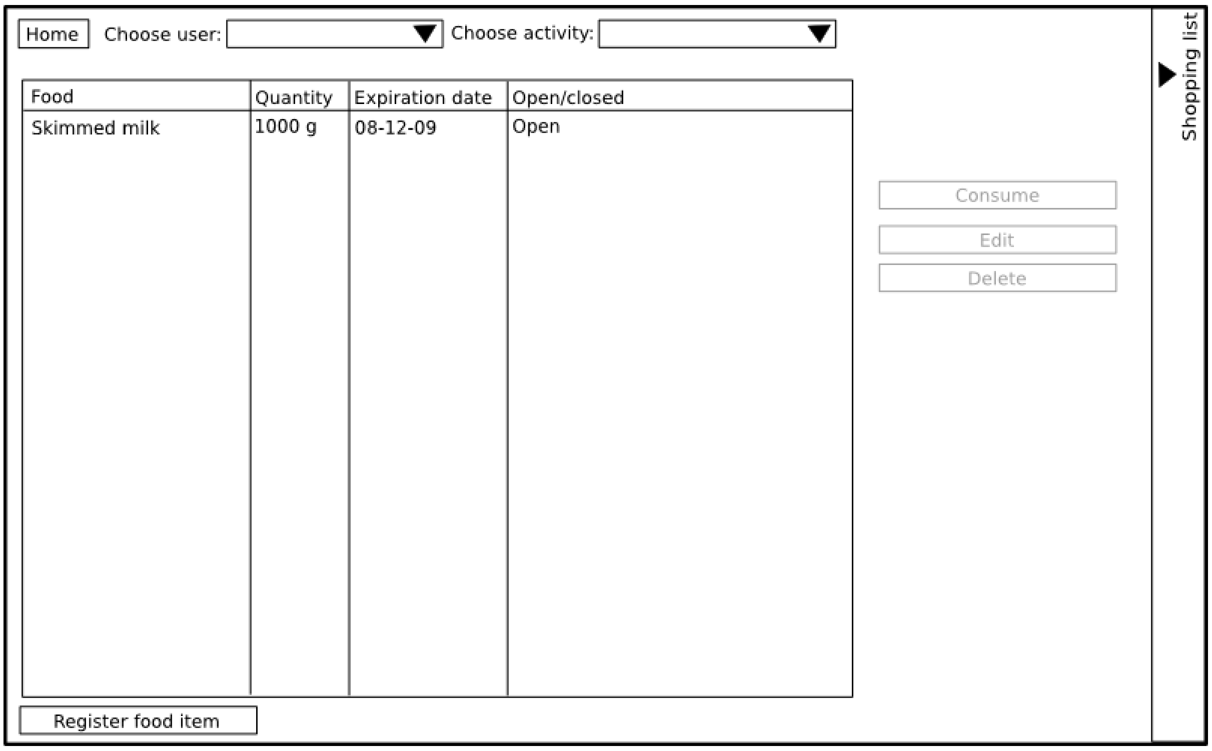
User	Percentage eaten
Ove Markussen	50 %

Figure B.9: Mock-up of consuming a FoodItem made from a Recipe.


Delete item

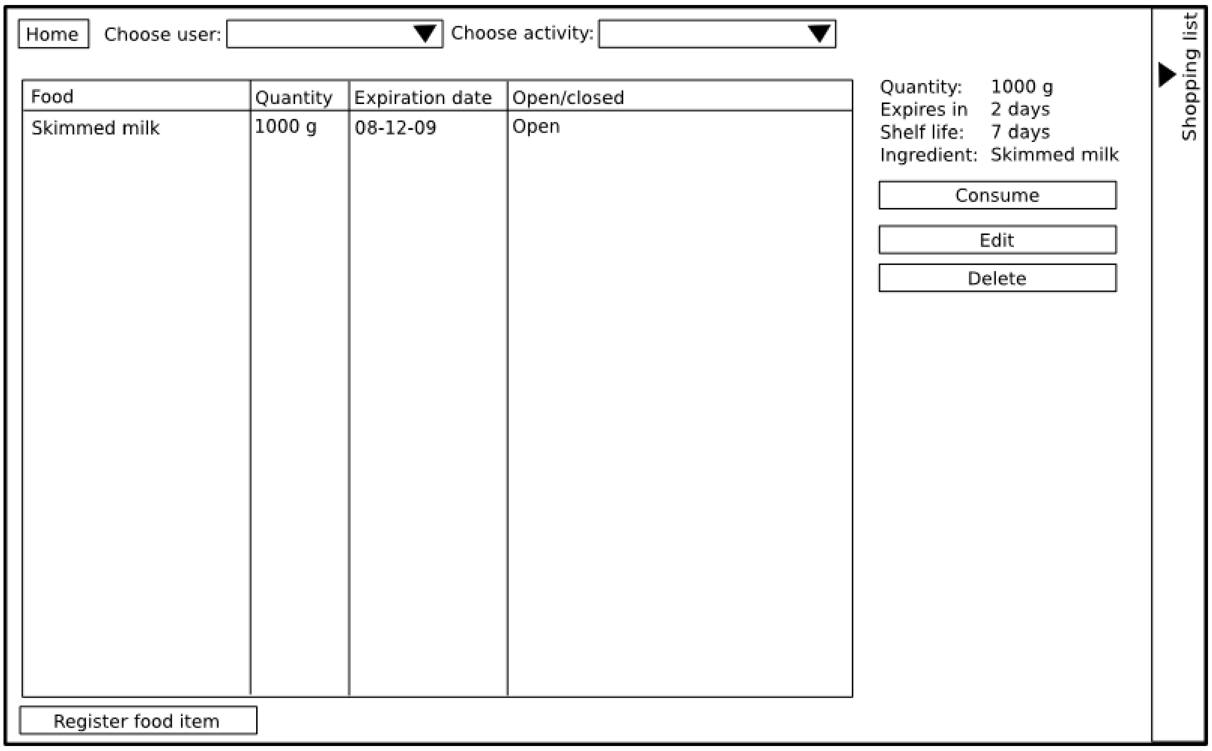
Do you want to store a fooditem made from this recipe in storage?

Figure B.10: Mock-up of storing a FoodItem made from a Recipe.



Food	Quantity	Expiration date	Open/closed
Skimmed milk	1000 g	08-12-09	Open

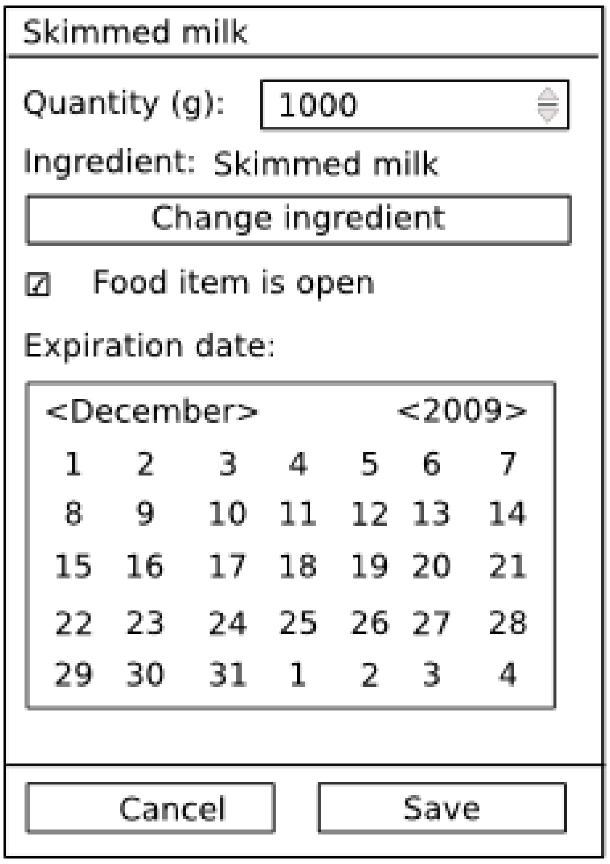
Figure B.11: Mock-up of Inventory.



Food	Quantity	Expiration date	Open/closed
Skimmed milk	1000 g	08-12-09	Open

Quantity:	1000 g
Expires in	2 days
Shelf life:	7 days
Ingredient:	Skimmed milk

Figure B.12: Mock-up of View Food Item.



Skimmed milk

Quantity (g): 1000

Ingredient: Skimmed milk

Change ingredient

☑ Food item is open

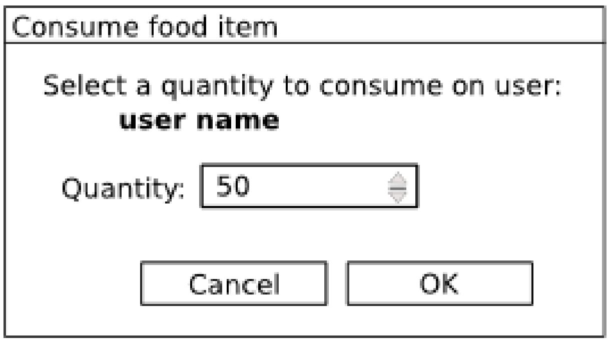
Expiration date:

<December> <2009>

1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29	30	31	1	2	3	4

Cancel Save

Figure B.13: Mock-up of Edit FoodItem.



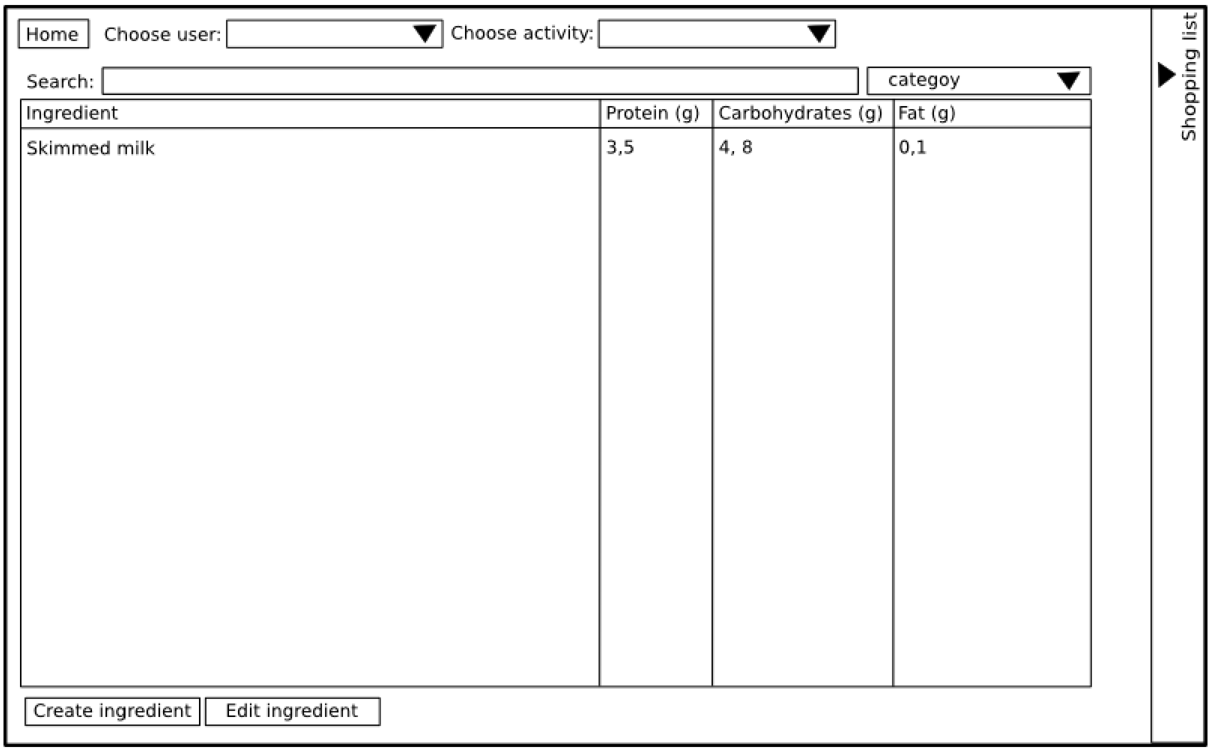
Consume food item

Select a quantity to consume on user:
user name

Quantity: 50

Cancel OK

Figure B.14: Mock-up of Consume FoodItem.

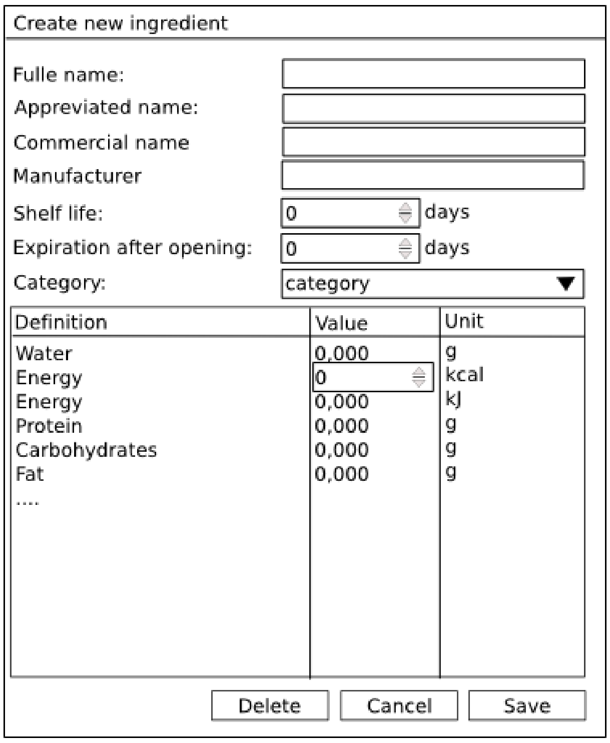


Home | Choose user: | Choose activity:

Search: | categoy

Ingredient	Protein (g)	Carbohydrates (g)	Fat (g)
Skimmed milk	3,5	4, 8	0,1

Figure B.15: Mock-up of View Ingredients.

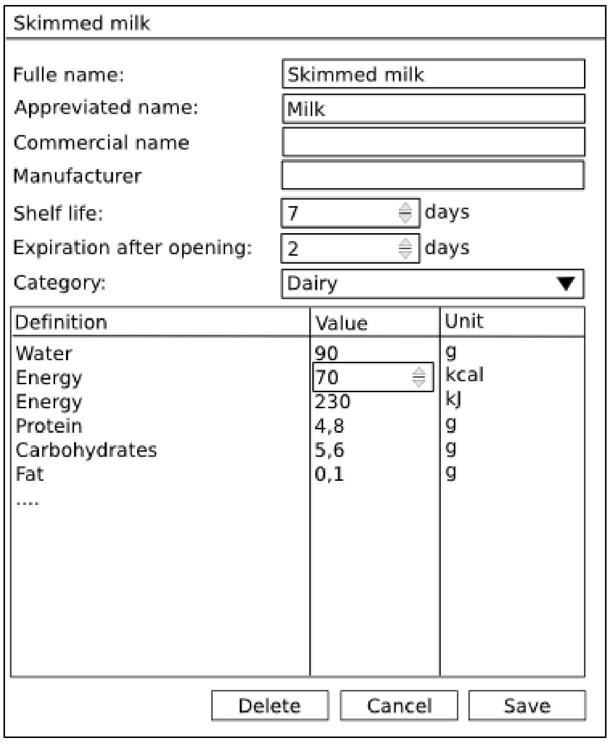


Create new ingredient

Fulle name:	
Appreviated name:	
Commercial name	
Manufacturer	
Shelf life:	0 days
Expiration after opening:	0 days
Category:	category

Definition	Value	Unit
Water	0,000	g
Energy	0	kcal
Energy	0,000	kJ
Protein	0,000	g
Carbohydrates	0,000	g
Fat	0,000	g
....		

Figure B.16: Mock-up of Create Ingredient.


Skimmed milk

Fulle name:	Skimmed milk
Appreviated name:	Milk
Commercial name	
Manufacturer	
Shelf life:	7 days
Expiration after opening:	2 days
Category:	Dairy

Definition	Value	Unit
Water	90	g
Energy	70	kcal
Energy	230	kJ
Protein	4,8	g
Carbohydrates	5,6	g
Fat	0,1	g
....		

Figure B.17: Mock-up of Edit Ingredient.

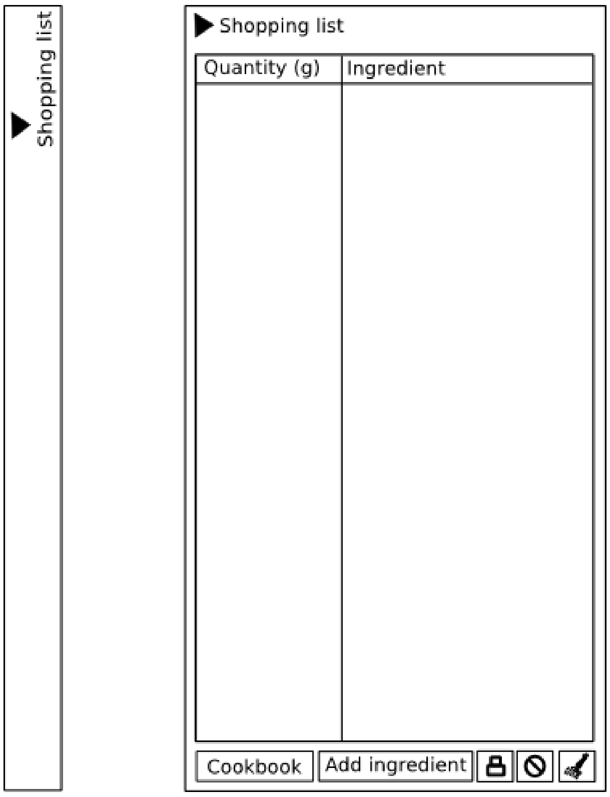
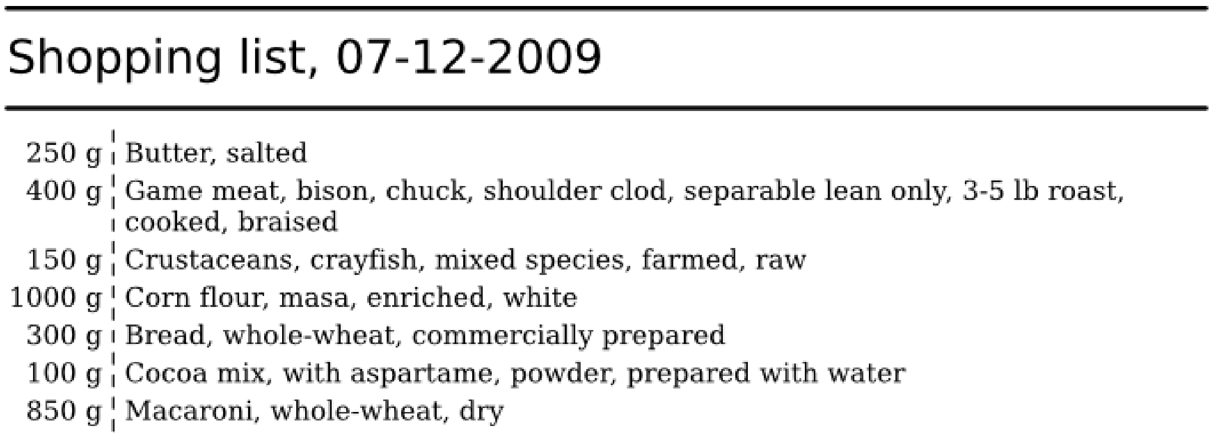


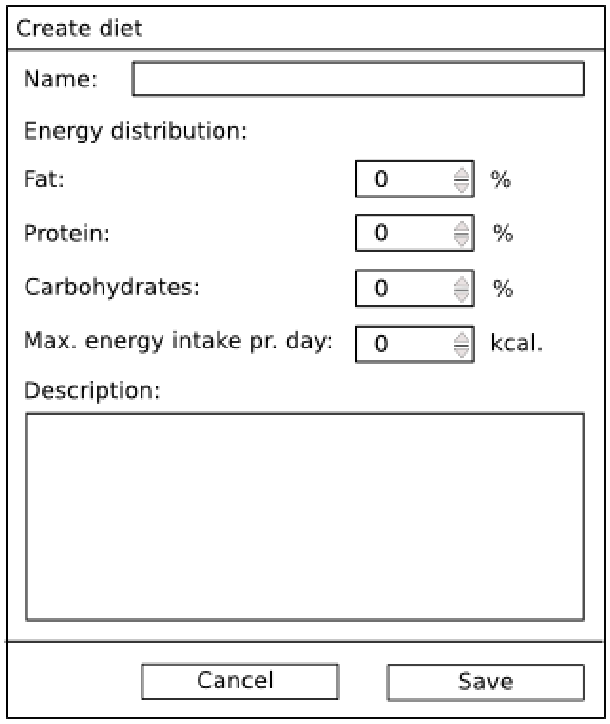
Figure B.18: Mock-up of ShoppingList.



Shopping list, 07-12-2009

250 g	Butter, salted
400 g	Game meat, bison, chuck, shoulder clod, separable lean only, 3-5 lb roast, cooked, braised
150 g	Crustaceans, crayfish, mixed species, farmed, raw
1000 g	Corn flour, masa, enriched, white
300 g	Bread, whole-wheat, commercially prepared
100 g	Cocoa mix, with aspartame, powder, prepared with water
850 g	Macaroni, whole-wheat, dry

Figure B.19: Mock-up of a print of ShoppingList.



Create diet

Name:

Energy distribution:

Fat: 0 %

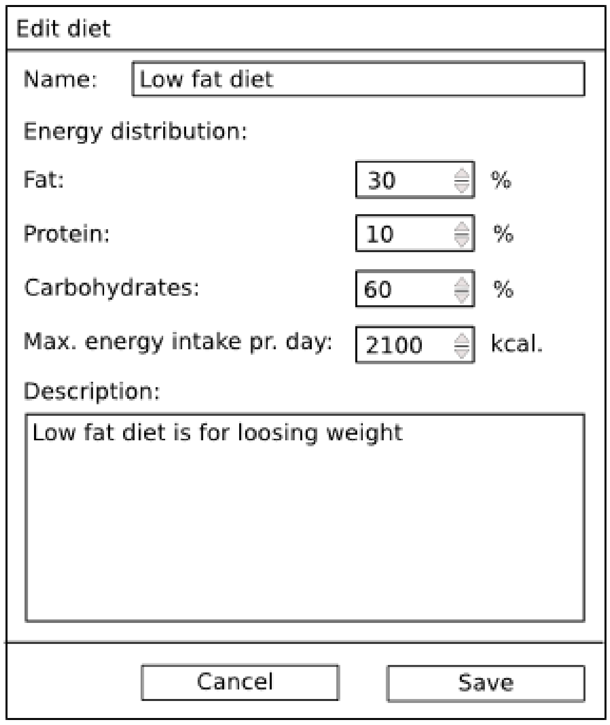
Protein: 0 %

Carbohydrates: 0 %

Max. energy intake pr. day: 0 kcal.

Description:

Figure B.20: Mock-up of Create Diet.



Edit diet

Name: Low fat diet

Energy distribution:

Fat: 30 %

Protein: 10 %

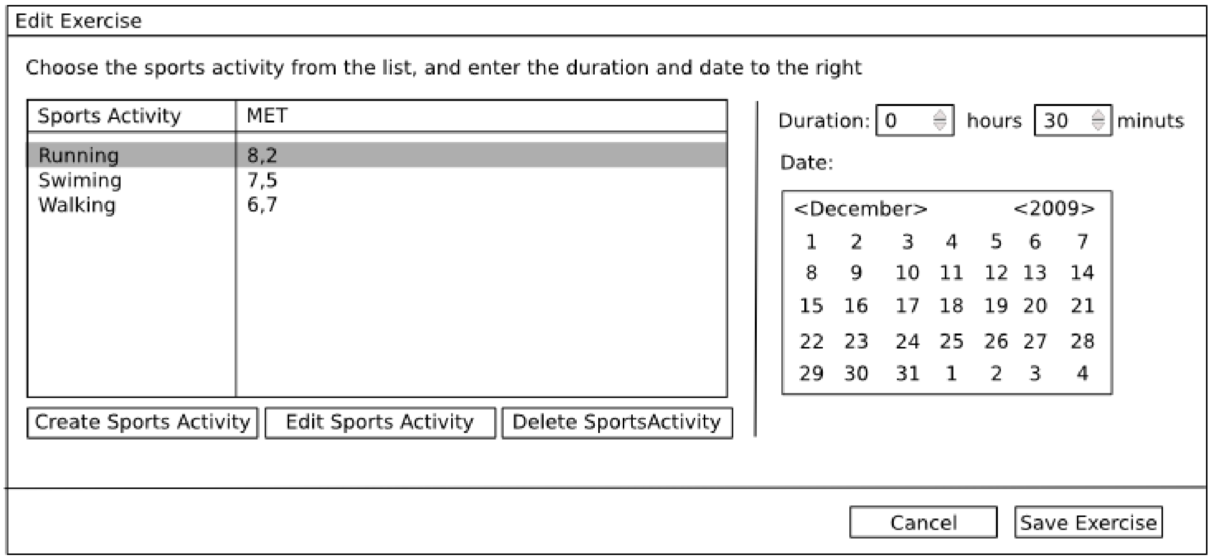
Carbohydrates: 60 %

Max. energy intake pr. day: 2100 kcal.

Description:

Low fat diet is for loosing weight

Figure B.21: Mock-up of editDiet.



Edit Exercise

Choose the sports activity from the list, and enter the duration and date to the right

Sports Activity	MET
Running	8,2
Swiming	7,5
Walking	6,7

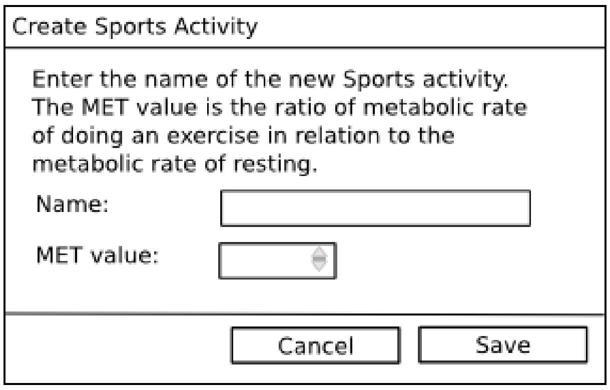
Duration: 0 hours 30 minuts

Date:

<December> <2009>

1	2	3	4	5	6	7
8	9	10	11	12	13	14
15	16	17	18	19	20	21
22	23	24	25	26	27	28
29	30	31	1	2	3	4

Figure B.22: Mock-up of Edit Exercise.



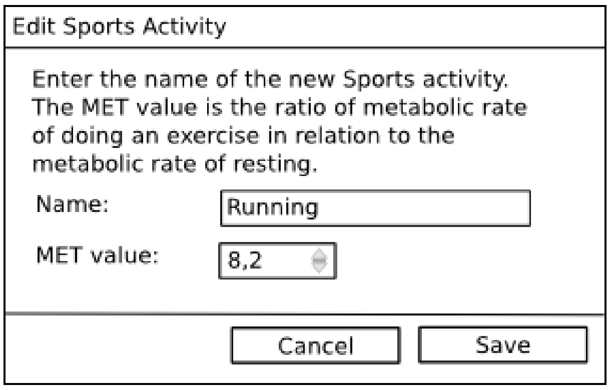
Create Sports Activity

Enter the name of the new Sports activity. The MET value is the ratio of metabolic rate of doing an exercise in relation to the metabolic rate of resting.

Name:

MET value:

Figure B.23: Mock-up of Create Sports Activity.



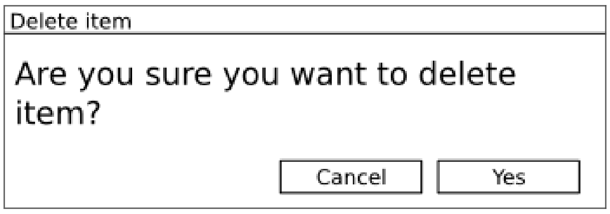
Edit Sports Activity

Enter the name of the new Sports activity. The MET value is the ratio of metabolic rate of doing an exercise in relation to the metabolic rate of resting.

Name: Running

MET value: 8,2

Figure B.24: Mock-up of Edit Sports Activity.



Delete item

Are you sure you want to delete item?

Figure B.25: Mock-up of Confirm Delete.

APPENDIX C

Usability Test, Log file 1 of 6

Test bruger A, Opgave 1 ***(Test user A, assignment 1)***

Time*(Time)*	Beskrivelse*(Description)*
9:04	Opgave udlevede *Assignment handed over.*
9:04	Brugeren forsøger at lave en ny bruger i ”Profile”. *(User attempts to create a new user via User Profile)*
9:05	Brugeren kan ikke finde et sted at input nye informationer.*(User can not find the right place to enter the new information.)*
9:06	Brugeren vil ændre den eksiterende bruger da han ikke kan finde en ”Ny bruger knap”.*(The test subject suggests to edit the al ready exiting as he can not find the “New User”-button.)*
9:07	Brugeren finder frem til ”Administration” fra hovedmenuen med hjælp fra tes-tleder.*(Test subject finds the “Administration” with help from test the monitor.”)*
9:08	Brugeren har svært ved at ”opdage” ny bruger knapperne i ”Administration”, men finder den til sidst.*(Test user has difficulties “discovering” new user button in “Adminis-tration”, but finds it at last.)*
9:09	”Bjarne” er gemt.*(“Bjarne is saved.”)*
9:10	Brugeren kan uden problemer ændre i ”Bjarnes” information*(Test user changes the details of “Bjarne” without any problems.)*
9:11	Brugeren sletter brugeren John Johnson, og erklærer Opgave 1 færdig*(Test user deletes John Johnson and declares assignment 1 finished.)*
	Kommentar til første opgave *(Comments from user on assignment 1)*: Det er svært at se når ”Create New User” knappen popper frem.*(It was difficult to se when the “Create New User”-button popped up.)*

Test bruger A, Opgave 2. ***(Test user A, assignment 2)***

Time*(Time)*	Beskrivelse*(Description)*
9:13	Brugeren skal skanne fooditems ind. *(Test user has to scan foods.)*
9:13	Brugeren sætter først diverse information ind om produktet, og forsøger derefter at skanne varen ind. *(Test user enters first the information of the food and then tries to scan the grocery.)*
9:14	Tilføjer varen, og opdager derefter at udløbsdatoen ikke helt stemmer, og forsøger derefter at rette fooditemen fra listen. *(Adds the grocery and discovers following that the expiration date is not quiet right, and then tries to correct the grocery from with in the list.)*
9:15	Han gør dette fra Inventory frem for den midlertidilig liste i "Register FoodItem". *(He does it from Inventory instead of from the temporary list in "Register FoodItem".)*
9:16 - 9:18	Første 5 genstande er indskannet uden problemer med stregkoden. *(The first 5 groceries are scanned with out any problems with the bar code)*
9:18	Brugeren skal indskanne en genstand uden stregkode. *(Test user must scan a grocery with out a pre registered bar code.)*
9:19	Brugeren finder uden problemer Bananer i ingredient listen, og tilføjer den til de andre. *(Test user finds Bananas in the Ingredient list and adds them to the other roceries.)*
9:20	Brugeren "Kom til at trykke på et eller andet", og kom ind i Register Fooditem? Uventet? *(Pushed some thing accidently and entered Register FoodItem? Unexpected?)*
9:20	Brugeren skal associer mælk med en ingredient. *(Test user associates milk with an ingredient.)*
9:21	Brugeren laver en ny ingredient til mælken. *(test user creates a new ingredient for the milk.)*
9:22	Begynder at indtaste Nutritional Information omkring mælken ind. *(Test user starts entering the nutritional information regarding the milk.)*
9:23	Brugeren gør opmærksom på at der ikke står nogensteder om det er per 100 g eller for hele produktet. *(Test user notes that there no where is stated if it is pr. 100 grams or the whole product.)*
9:23	Brugeren forstår ikke hvad "Adjusted Protein" er, Test leder forstår ham godt. *(Test user does not understand what "adjusted protein" means, test monitor agrees with him.)*
9:25	Brugeren har gemt Skummemælk til Ingredient databasen. *(Test user has saved Skimmed milk to the Ingredient database.)*
9:25	Brugeren skal nu slå lyden fra food registration. *(Test user now has to turn of the sound.)*
9:26	Brugeren kigger i "Administration" og "User Profile" efter indstillinger, snakkede om det måske kunne være i Register Fooditem. Da han kigger efter en ekstra gang, ser han den. *(Test user searches in the "Administration" and "User Profile" for settings, talked about he might find it in Register FoodItem. As he looks closer, he finds it.)*
9:27	Brugeren sletter en dåse tun fra databasen uden problemer. *(Test user deletes a can of tuna with out any problems.)*
9:27	Brugeren erklærer sig færdig med Opgave 2 *(Test user declares assignment 2 finished.)*

Test bruger A, Opgave 3. ***(Test user 1, assignment 3)***

Time*(Time)*	Beskrivelse*(Description)*
9:28	Brugeren skal lave mad til 6 personer. *(test user has to cook for 6 persons)*
9:28	Han kigger efter og finder løgsuppe og erklærer at han mangler alle tingene til løgsuppe. *(he looks for and find Onion soup and declares all ingredients for missing.)*
9:30	Brugeren skal rediger i opskriften og lave smør om til olie. *(Test user must edit the recipe and changes butter to oil.)*
9:30	Brugeren sletter smør fra recipe, og trykker add ingredient. *(Test user deletes butter from recipe and push add ingredient.)*
9:31	Brugeren ser der er 2 oliven olier, og vælger den ene ud fra beskrivelsen af olien. *(Test user notices two kinds of olive oil and chooses one of them from the description.)*
9:31	Brugeren gemmer opskriften, og noter at han ikke laver en ny. *(Test user saves the recipe, and comments that he does not make a new.)*
9:33	Der er en bug, hvor opskriften ikke bliver opdateret efter at blive redigeret i. *(A bug in the program is discovered - the recipe does not update after being editted!)*
9:34	Brugeren kigger på sine FoodItems efter udløbsdato. *(Test user view his groceries according to expiration date.)*
9:34	Brugeren skal lave en ny opskrift, og går ind i Create Recipe. *(Test user has to add a new recipe and enters the Create Recipe.)*
9:35	Brugeren vurdere sig frem til MealType, Difficulty og preparation time, og begynder at tilføje ingredienter. *(Test user)*
9:37	Brugeren har tilføjet ingredients og vurder at det er til en person, og begynder at skrive directions. *(Test user has added the ingredients and judges it for being for one person and starts to enter cooking directions)*
9:38	Testleder vurder bruger ikke behøver at skrive mer til directions. *(Test monitor estimates no need for test user to enter any further cooking directions.)*
9:38	Brugeren tilføjer opskriften, og mener at være færdig med Opgave 3. *(Test user adds the recipe and assumes assignment 3 to be accomplished.)*
	Kommentar fra brugeren: *(Comments from the test user:)* Det kan være svært at vælge ingredientser da man skal nogen gange kigge i flere katagorier, også fordi ingredienterne er på Dansk. Nævner også at søgefeltet skulle søge på samtlige katagorier og ikke kun den man er inden i for at hjælpe. *(It is hard to add ingredients as several categories must be looked through and because the ingredient database is in English and the groceries are Danish)*

Test bruger A, Opgave 4. ***(Test user 1, assignment 4)***

Time*(Time)*	Beskrivelse*(Description)*
9:41	Brugeren skal nu købe ind. *(Test user now has to shop)*
9:41	Brugeren finder shoppinglisten uden væsentlige problemer, og skal tilføje mørk chocolade. *(Test user finds the shopping list with out any note worthy problems and must add dark chocolate.)*
9:43	Brugeren skal drikke et glas mælk, og opdager at han glemte at tilføje den til Inventory, og blot lavede ingredienten. *(Test user has to drink a glas of milk and realises he forgot to add the milk to the Inventory, he only created the ingredient.)*
9:44	Brugeren tilføjer hurtigt mælken. *(Test user quickly add the milk.)*
9:45	Brugeren "Consumer" 200g mælk. *(Test user "consumes" 200 grams of milk.)*
9:45	Brugeren skal se på leg of lamb opskriften og tilføje ingredienter til shoppinglist hvis de ikke er der. *(Test user has to find the recipe for Leg ofLamb and add the missing ingredients)*
9:46	Brugeren finder en fejl - Man kan ikke ændre ingredient mængden ved at skrive et tal i servings, kun ved at klikke på pilene. *(Test user discovers an error - It is not possible to change the amount of serving by entering the number, one can only use the arrows.))*
9:47	Brugeren printer shoppinglisten. *(Test user prints the shopping list.)*
9:47	Brugeren er færdig med Opgave 4. *(Test user has finished assignment 4)*

Speed opgave(***Speed scanning assignment:)*** Brugeren kan skanne 10 Fooditems på 1 minut og 20 sekunder *(Test user scanned the 10 groceries in 1 minute and 20 seconds.)*

Bibliography

Composition of foods raw, processed, prepared usda national nutrient database for standard reference, 2008.

Determining daily calorie needs. http://www.freedieting.com/calorie_needs.html, 2009.

Brad Abrams and Krzysztof Cwalina. *Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries*. Addison-Wesley Professional, 2nd edition edition, 2008. ISBN 978-0321545619.

L'Iconoclaste Banal / Adam Hirschhorn Adam. reactionary. Retrieved 18. November 2009 from http://www.flickr.com/photos/adamhirschhorn/205440154/. Published under Creative Commons according to http://creativecommons.org/licenses/by-sa/2.0/deed.en., 2006.

Carly & Art. Amy. Retrieved 18. November 2009 from http://www.flickr.com/photos/wiredwitch/4108124042/. Published under Creative Commons according to http://creativecommons.org/licenses/by-sa/2.0/deed.en., 2009.

Jeff Brow. Zbar bar code reader library documentation. Retrieved 16. December 2009 from http://zbar.sourceforge.net/api/., 2009.

Thomas H. Cormen. *Introduction to Algorithms*. The MIT Pres, third edition edition, 2009. ISBN 978-0262533058.

Peter Dolog and Jan Stage. Adria: A method for abstract design of rich internet applications for the web 2.0. 2009.

FAO/WHO/UNU. Human energy requirements. Retrieved December 2009 from ftp://ftp.fao.org/docrep/fao/007/y5686e/y5686e00.pdf, 2001.

Martin Fowler, David Rice, Matthew Foemmel, Edward Hieatt, Robert Mee, and Randy Stafford. *Patterns of Enterprise Application Architecture*. Addison Wesley, 2002. ISBN 0-321-12742-0.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. 1998. ISBN 0-201-63361-2.

Ingorrr. Get out, cork guy. Retrieved 18. November 2009 from http://www.flickr.com/photos/ingorrr/361288215/. Published under Creative Commons according to http://creativecommons.org/licenses/by-sa/2.0/deed.en., 2007.

Jesper Kjeldskov, Mikael B. Skov, and Jan Stage. Instant data analysis: conducting usability evaluations in a day. In *NordiCHI '04: Proceedings of the third Nordic conference on Human-computer interaction*, pages 233–240, New York, NY, USA, 2004. ACM. ISBN 1-58113-857-1. doi: http://doi.acm.org/10.1145/1028014.1028050.

Lars Mathiassen, Andreas Munk-Madsen, Peter Axel Nielsen, and Jan Stage. *Object Oriented Analysis & Design*. Marko Publishing ApS, Aalborg, Denmark, 2009. ISBN 87-7751-150-6.

See ming Lee. Gay pride new york 2007 / sml. Retrieved 18. November 2009 from http://www.flickr.com/photos/seeminglee/694298595/. Published under Creative Commons according to http://creativecommons.org/licenses/by-sa/2.0/deed.en., 2007.

Jeffrey L. Roitman and Moira Kelsey. *Guidelines for Exercise Testing and Prescription*. American College of Sports Medicine, third edition edition, 1994. ISBN 0-683-00026-8.