Bachelor Project - DCRGraphs case study

Johan Arleth and Peter Øvergård Clausen

May 2016

Abstract

This study examines the pros and cons of designing a system around DCRGraphs. The focus is on the software developer's productivity, in terms of time saved or time spent, to design a system around DCRGraphs. We study this by developing a case system, designed to model some of the business processes in a fictional restaurant, where DCRGraphs are used to model business process logic. We found many positive influences on on our design and development of the system, but also found that tools and documentation around DCRGraphs are somewhat immature. Designing around DCRGraphs saved us a lot of time on iterative processes, such as updating system design, updating code and implementing new features. DCRGraphs is a valid option to use for modelling business processes, with the currently existing tools, but it can become much better and easier to work with if these tools are further developed.

Contents

Part I

Introduction and Problem Definition

1 Introduction

We live in a world surrounded by business processes. We encounter them everyday, for example when we go to a restaurant to eat. The food must be ordered by the customer, a waiter must receive the order, a chef must cook the order, and then a waiter must serve the order and the table must be made ready for the next customer. The payment can happen at any time during this process. Business processes such as these are not always happening in a linear fashion, and may be subject to change.

DCRGraphs (Dynamic Condition Response Graphs, henceforth referred to as DCRGraphs) have been created to model business processes such as these. DCRGraphs is trying to create a shared standard for modelling business processes, seeking to create a better alternative to earlier Business Process Model Notations (BPMN) such as swimlane diagrams and to create process-aware information systems [1, 9].

DCRGraphs is a relatively new tool to model business processes compared to swimlane diagrams. Implementing the DCRGraphs concepts and using DCRGraphs online modelling tools like `www.DCRGraphs.net` and `www.dcr.itu.dk` to develop a case system may lead to some benefits and/or disadvantages for the designers and implementers of the system.

Therefore, for the creators of DCRGraphs we hope this study will be beneficial for the further development of DCRGraphs process modelling language and online modelling tools. For developers interested in building a system using DCRGraphs, this project serves as argumentation on why one should consider using DCRGraphs and what one should be careful of when doing so. Also, inspiration can be drawn from our solution to build a system for another case, or expand on the system we have built in this project.

2 Problem definition

We aim to study the impact of using DCRGraphs and accompanying online tools to model business processes within a case system from the software developer's point of view. This leads us to the following problem statement:

How can DCRGraphs be implemented in a case system, and what are the pros and cons of using DCRGraphs as a Business Process Modelling tool?

To answer the problem statement above, we have divided the question into the following issues:

1. The eases and difficulties of incorporating DCRGraphs when designing and building a system.
2. The eases and difficulties of implementing new requirements or changes into a system built on DCRGraphs.

This project will not focus on the following issues:

- Comparering DCRGraphs with other Business Process Modelling Notations.

3 Methodology

We will perform a qualitative case study, reporting on our own experience as developers, as a basis for answering the above questions.

As a case, we have chosen to model the business processes within a fictitious restaurant that we define. We will then build a restaurant ordering system, that also supports the business processes within the restaurant as a process management system. We have chosen this case because it serves as a good and simple case that gets us around the major issues with DCRGraphs.

The restaurant ordering system will serve as a practical test for us to gather qualitative data about the issues that will occur related to DCRGraphs, through actual development.

We will document the development processes and its difficulties in daily work logs throughout the project. We will analyse these in the end of the project in order to address the issues and answer the problem statement above.

We are not making this prototype for any customers, as it was considered out of scope and too much work for the limited time given to the project. It also enables us to define the specific system requirements of the case system ourselves.

We will consider three main process variants:

1. A customer orders food at a table and has it served.
2. A customer orders food over the phone and picks it up at the restaurant.
3. A customer orders food on the website and has it delivered.

There can be variations and changes to these three main processes, such as a customer orders food over the Internet, but still arriving at the restaurant and eating. The system will also support the different variations and changes.

We will use the data that is given in the DCRGraphs online tools and build a restaurant ordering system around it. Making it possible for the developers to change the DCRGraph later in the development process and let the system change with it.

The main activities of this project will be:

- Design and develop a DCRGraphs based restaurant ordering case system including:
  - Web API and database - for storing and interacting with DCRGraphs and restaurant data.
  - User program - GUI application where restaurant workers can see and interact with orders.
  - Customer website - where customers can place an order.
- A qualitative and subjective study of the effect on developer productivity *in the initial development* of the restaurant system prototype.
- A qualitative and subjective study of the effect on developer productivity *in subsequent changes* to the restaurant system prototype.

In the *development of the Web API* we will study how DCRGraphs influences the development of a Web API, how DCRGraphs together with order data efficiently can be send and received over the web using HTTP requests. We will henceforth refer to the Web API as the DROM system Web API or just DROM system API.

In the *development of the database* we will study how DCRGraphs influences the design and implementation of a database for persistent storage of DCRGraphs together with order data, and efficiently interacting with the database. In this report we will henceforth refer to the database as the DcR Order Management system database (DROM system database), or just 'the database'.

In the *development of the user program* we will study how DCRGraphs influences the design and implementation of user applications together with sending and receiving data to the Web API. We will henceforth refer to our client program as the DcRgraph Order Management system client (DROM system client).

In the *initial development* of the restaurant system, eases and difficulties in design and implementation of the above system components will be studied, documented, analysed and discussed during this project. The whole system will henceforth be referred to as the 'DROM system'.

In the *subsequent changes* we will study the eases and difficulties of implementing changes to the system built on DCRGraphs. The subsequent changes can be changes in the DCRGraph, to see how the system reacts, and implementation of new requirements to the system.

The project may leverage existing implementation from the 2nd-year project; in this case repurposed code will be clearly marked out in the final product, both in report and in submitted code.

4 Structure of project

The overall structure of the project will start with a requirement elicitation phase where we will find which requirements we would like to build into the system. Next, we will have a design phase, where we design the restaurant ordering system. This will be followed by an implementation and testing phase of the initial system. After implementing the initial system we will then cooperate with our supervisor in finding subsequent changes that can be made to the system. We will then implement the subsequent changes.

We will be documenting the eases and difficulties we encounter during design, implementation of the initial system and subsequent changes during the project, to study our problem statement.

5 Structure of report

We have loosely based the project report on Object Oriented Software Engineering [6]. The book describes how to document the design and implementation of a programming project through a Requirement Analysis Document (RAD) and a System Design Document (SDD).

Making a full Requirement Analysis Document and a full System Design Document report is both time consuming and out of scope of this study, hence we will only use the parts that we find meaningful for this project, and focus on parts that have some relationship with DCRGraphs.

6 Scope

The scope is limited by the time we have for the project. The system we have made is a prototype, and it is not made for a customer. It is not meant to be a "production-ready" system, meaning it should succeed in the scenarios that we have defined, but it intentionally lacks features, stability, security and may not work as intended when using it in a large scale deployment.

7 Time constraints

Because of the limited time to do this study, there were features we did not have time to implement. These are mentioned here, along with the reasons why these features did not make it.

7.1 Expand DCRGraphs with new data

Initially we had planned to try and expand DCRGraphs by adding new data to them. However, we never encountered a situation that required us to do so. As we were under time pressure, we decided not to experiment with additional features requiring this.

7.2 Website

We planned to make this a three part project, a client, a web API and a website. This turned out to be too ambitious. We did not have the time to develop a website in addition to the client and web API.

During development we also found that it would not serve much purpose to make a website. Most, if not everything, of what could be discovered, related to our research goal, by making the website, we already discovered by developing the client and web API. This, in addition to lack of time, made us drop the notion of a website.

Part II

Background

8 The processes

In this section we will define the business processes of our fictitious restaurant.

A business process can be described as a linked series of activities, tasks or events that describes the delivery of a service or product to a client or customer.

There will be several activities and events that makes up the business process within our fictitious restaurant, which we will model with DCRGraphs. These are very closely linked with the actors described in chapter *11: Functional requirements*. The processes will be described here, but for a detailed explanation of exactly how they work, see chapter *11* sections: *Scenarios* and *Use case models*.

Making an order

The first step in any business process within our restaurant, is to make an order. This means that a customer contacts the restaurant to order some food. This can happen in various ways, such as talking to a waiter inside the restaurant, calling the restaurant by phone or by creating an order through the restaurant website.

Cooking food

After an order has been created, the chef must cook the food. The chef starts to cook the food, and finish cooking the food some time later.

After cooking the food, the chef must also make it ready for the next step which is delivery. He must either prepare it on a plate so that it can be served for a customer in the restaurant or otherwise put it in containers so it can picked up for takeaway or be delivered.

Serving

If an order is made for serving, and the food is cooked, it must be served to the customer sitting at a table. While the customer eats, he or she may order additional items that must be cooked and served for the customer.

When the customer is done eating, the table must be cleaned and made ready for the next customer. This is done by a waiter.

Takeaway

If an order is made for takeaway, and the food is cooked, the next step is simply to wait for the customer to arrive and give them the food when they do.

Delivery

If an order made for delivery, and the food is cooked, then an employee must take it and deliver it to the customers address.

Changing an order

Throughout the steps of cooking an order and giving it to the customer, changes can be made. This is the case where a customer decide they want something extra, want something different, or decide that they don't want some of the things they ordered.

To do this, the customer contacts an employee in the restaurant and asks them to change their order accordingly. In cases where the customer wants to change food that is already being cooked, or is done cooking, it is up to the employee to judge whether the change is reasonable.

Payment

Some time during the above processes a payment for the order must be done, in order for the restaurant to earn money.

This can either happen at the start of a process, when a customer places an order on the website.

It can also happen in the middle of a process, when the customer is done eating, but before the tables are cleaned.

It can also happen in the end of a process, when the customer picks up the food at the restaurant.

The important part is to ensure that the payment gets done at some point during the process, even though changes to the process might occur, such as changes to the delivery method, or additional items are added or removed from the order.

9 DCRGraphs

DCRGraphs is abbreviated from Dynamic Condition Response Graphs, which is a new way of modelling business processes, developed by researchers at the IT University of Copenhagen and employees at the company Exformatics A/S.

A DCRGraph is a directed graph [10], in which nodes represent events and edges represent a relation between two events.

Each event has three Boolean properties which can be either true or false (however the name of the value is different from ”true” and ”false” on each property). This results in 2^3 (two options for each of the three properties) which is equal to eight possible states that an event can be in. An overview of the eight different states can be found in *figure 1.*

9.1 Event states

The first property is ”Executed”, which denotes whether an event has been executed or not (if it has happened or not). An execution of a non executed event results in an executed event. An execution of an executed event results in an executed event. Creating an event using one of the online tools: `DCRGraphs.net` or `dcr.itu.dk` will be not executed by default. On execution, the event will change it's state and states of other events through it's relations which we will explain later.

The second property of an event is ”Included”, which denotes whether an event is included in the graph or not. If the event is not included, we call it ”excluded”. An event that is included can be executed, while an event that is excluded cannot be executed. When creating an event using one of the tools, the event will be included by default.

The third property of an event is ”Pending”, which denotes whether an event is pending to be executed or not pending to be executed. A simulation of a graph which has events in a pending and included state (event states s2 and s3 in *figure 1*), is not considered ”Accepting” (or allowed to terminate) before all pending events returns to an accepting state. When we are in an accepting state of a DCRGraph, we are allowed to terminate the simulation of the graph. If we are not in an accepting state, we have to keep executing events until all the events are in an accepting state. Note that if the remaining pending events are excluded (event states s6 and s7 in *figure 1*), all the DCRGraph's events are accepting. However if the event is included again while still being pending, then the simulation is not accepted again. A pending event becomes non pending when it is executed. A newly created event is not pending by default.

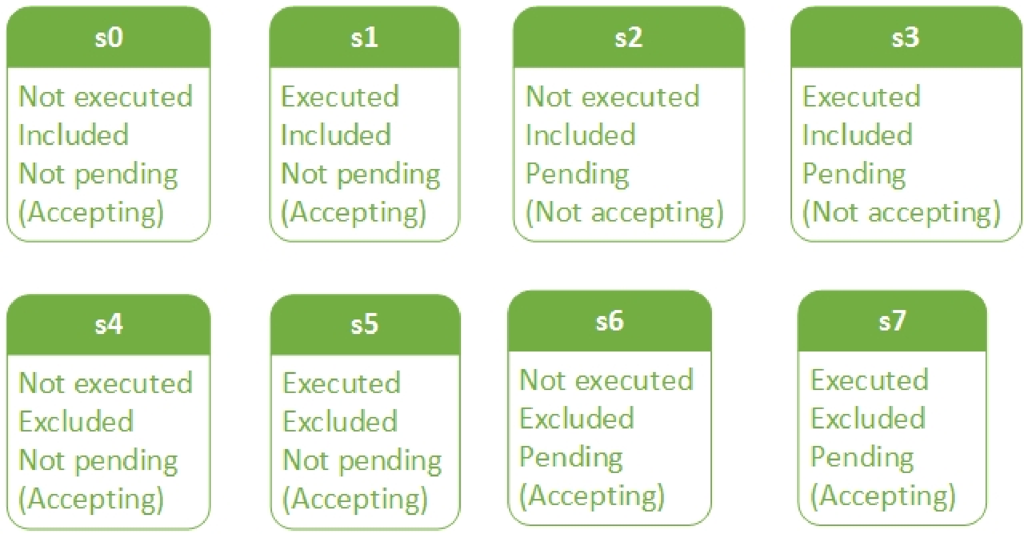


Figure 1: The eight different states of an event

The default state of an event upon creation is not executed, included and not pending (state s0 in *figure 1*). It is possible to change an event's starting state when using the editor on `dcrgraphs.net` or `dcr.itu.dk`, if one wants to have an event that starts in pending state for example.

9.2 Relations

A relation goes from one event to another event or itself. It can be interpreted as a directed edge between two nodes in a directed graph. The textual notation from `dcr.itu.dk` and the graphical notation from `dcrgraphs.net` can be seen in *figure 2*.

The first type of relation is the ”Include” relation. An include relation from event A to event B denotes that event A ”Includes” event B. Executing event A will change the Included property of event B to ”included”. If event B is excluded, it will become included. If event B is included already it will remain included.

The second type of relation is the ”Exclude” relation. An exclude relation from event A to event B denotes that event A ”Excludes” event B. Executing event A will change the Included property of event B to ”excluded”. If event B is included, it will become excluded. If event B is already excluded it will remain excluded. An Exclude relation can be interpreted as the opposite relation to the Include relation.

The third type of relation is the ”Response” relation. A response relation from event A to event B denotes that event A ”is making event B a response to event A”. After the execution of event A, event B will become pending. If event B is already pending it will remain pending. As we already stated, a simulation of a graph with pending events is not considered completed before all events are either included and not pending or excluded.

The fourth type of relation is the "Milestone" relation. A milestone relation from event A to event B denotes that "event A is a milestone for event B". If event A is pending and included, event B cannot be executed before event A becomes not pending or excluded.

The fifth type of relation is the "Condition" relation. A condition relation from event A to event B denotes that "event A is a condition for event B". Event B cannot be executed before event A is either executed or excluded.

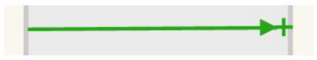
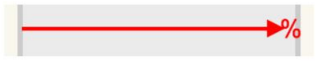
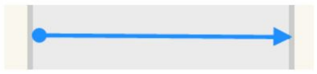
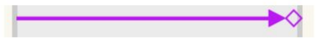
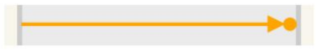
Name of relation:	Dcr.itu.dk notation:	Dcrgraph.net notation:
Include	-->+	
Exclude	-->%	
Response	*-->	
Milestone	--<>	
Condition	-->*	

Figure 2: The five different relations between events

9.3 Roles

A role denotes "who is able to execute this event". If an event is in a role, say 'Waiter', this event can only be executed by a user who has the role 'Waiter'. A user may have multiple roles. An event may have multiple roles. An event has no role by default and can be executed by every user, but when an event's role is specified only users with the specified role is able to execute it.

9.4 Groups

A user may choose to divide events into groups. An event can have multiple groups. Multiple events can be in the same group. Every user can execute events in every group, as long as the event's state and role allow it.

9.5 Nested events

Events can contain other events. The event that contains elements is called a "parent", and contained events are called "children". A relation from any event can go to the parent or children events. A relation on the parent element causes the effect of the relation on all children events. We have not used this functionality in this project, however we thought it was important to mention.

9.6 Other data fields of an event

9.6.1 Label

The 'Label' field in dcrgraphs.net is used to give an event a name. It is not unique, and as such two or more events can have the same 'Label'.

9.6.2 Id

The 'Id' field in dcrgraphs.net is generated for each event in a graph. It is a unique identifier used to differentiate between events.

9.6.3 Description

The 'description' field in dcrgraphs.net is used to describe an event with text.

9.7 Simulating a graph

By simulating (or running) a graph, a user may or may not be able to execute an event depending on it's state. By executing an event, a user is denoting "this event has happened". On execution, the event changes state and may change the state of other events and itself through its relations to other events or itself.

There can be multiple events which are able to be executed, and it is up to the user to choose which event to execute. The only way to execute an event in a non-executeable state, is to execute other events until the wanted event changes state to an executable state, and then executing it. While simulating a graph, it is not possible to edit the graph.

9.8 Modelling a business process with DCRGraphs

Using all of the above described rules and properties of DCRGraphs, it is possible to model complex business processes. We will now demonstrate a simple example of how a business process can be modelled with DCRGraphs: *figure 3*.

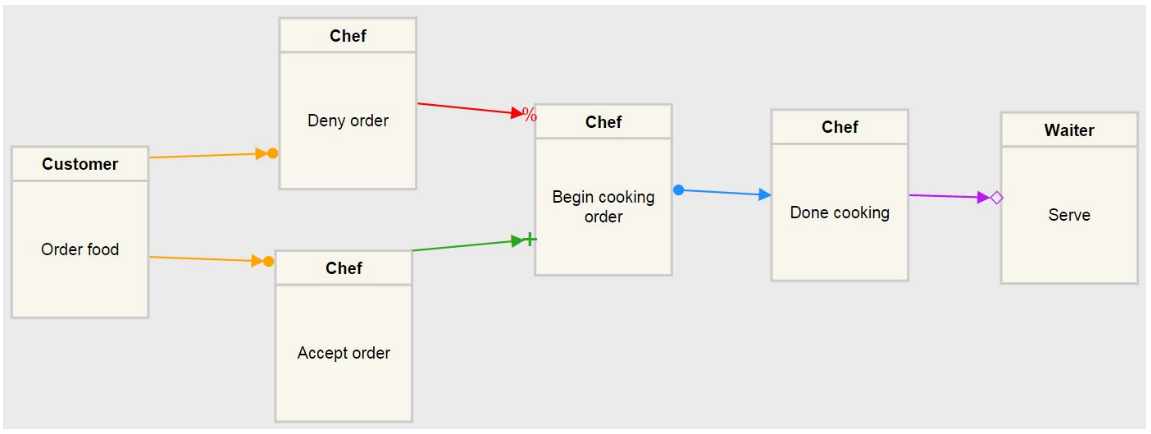


Figure 3: Example of a business process

Note that this example is very simplistic, far from realistic, and does not cover all possible execution paths. This example focuses on understandability, not completeness.

The business process we want to model:
In another fictitious restaurant, the business process for ordering, cooking and serving the food can be described as this.

The customer orders food.

The chef of the restaurant will then choose if he wants to Accept or Deny the order. (He could be too busy). Before the chef can Accept or Deny the order, the customer must have placed an order.

When accepting or denying an order, the chef will determine if he shall begin cooking the order.

If he accepts, he will begin cooking the order, if he denies, he will not begin cooking the order.

We expect him to finish cooking the order some time later.

Before the waiter can serve the order, the chef must be done cooking the order. Serving the order is the goal of this process.

Identifying events:
Events from this scenario can be found by looking for events that changes the process in some way, that we want to model.

The events have been specified as:

1. "Order food" done by customer.
2. "Accept order", "Deny order", done by chef.
3. "Begin cooking", done by chef.
4. "Done cooking", done by chef.
5. "Serve", done by chef.

Identifying relations:

1. "Before the chef can Accept or Deny the order, the customer must have placed an order", condition relation from "Order food" to "Deny order" and "Accept order" is added.
2. "If he accepts, he will begin cooking the order", an inclusion relation is added from "Accept order" to "Begin cooking order".
3. "if he denies, he will not begin cooking the order", an exclusion relation is added from "Deny order" to "Begin cooking order".
4. "We expect him to finish cooking the order some time later.", a response relation is added from "Begin cooking order" to "Done cooking".
5. "Before the waiter can serve the order, the chef must be done cooking the order. Serving the order is the goal of this process.", a milestone is added from "Done cooking" to "Serve".

What was modelled by this example
When simulating the graph. The user assigned to "Customer" is able to execute the "Order food" event.

Then the chef is able to accept or deny the order. If he denies it, he cannot execute "Begin cooking". If he accepts, he can. He can also choose to deny, then reconsider, and accept the order anyway.

The chef can then begin cooking the order. This will make "Done cooking" pending. The simulation is not allowed to terminate, as the chef must finish cooking what he started.

When the chef is done cooking, the waiter can serve the order. If the the chef has begun cooking the order, the waiter cannot serve the order, before the chef is done cooking the order.

What was not modelled by this example
There are almost no restrictions for the user.

The user is able to begin cooking even though the customer has not ordered yet.

The waiter can choose to serve before the customer has ordered yet.

Every event can happen multiple times, without any consequences.

Note that all these things were not modelled to keep the simplicity of the example.

It is very hard to make a simplistic example while also modelling every possible outcome. Hence DCRGraphs tends to get large and more complex as we will see throughout this project.

Part III

Development of the initial system

10 Overview

10.1 Purpose of the system

The system is meant as a business process management system to support the business processes in a restaurant. This means supporting and easing the work of the employees in the tasks that they have to undertake when handling customer orders.

10.2 Scope of the system

The system is meant to model the processes that employees go through from receiving an order in the restaurant, until the order has been delivered to the customer. In the case of customers eating in the restaurant, this means until the order has been served and eaten, and the customers have paid. For orders not being eaten in the restaurant, such as takeaway and delivery, it means until the order has been delivered to the customer and paid for.

The system will not support functionality for receiving payment from a customer. The system is a prototype and is developed for research and is not meant for deployment in a real business.

10.3 Objectives and success criteria of the project

The project aims to study the impact of developing a system around DCRGraphs, to model business processes within a restaurant, from the developers point of view. A successful development of the system will create a functioning prototype system featuring:

- A web API for sending and receiving data on the web.
- A database for persistent storage of orders and DCRGraphs.
- A client aplication to manage orders and business processes within the restaurant.
- A website for the customers to create orders.

11 Functional requirements

In the DROM system we will support the following five actors:

- **Customer** - A customer should be able to order food in three different ways; on the restaurants website, by calling the restaurant on the phone and by talking to an employee in person. The customer can choose to have the food delivered, pick it up at the restaurant, or eat it at the restaurant.
- **Chef** - A chef should be able to view orders which are waiting to be cooked, change the status from *waiting to be cooked* to *cooking* and finally to *cooked*.
- **Waiter** - A waiter should be able to accept and create new orders, view orders, edit orders and delete orders as well as see the current status of an order.
- **Delivery** - A delivery employee should be able to see which orders are *waiting to be delivered* and change the status to *being delivered* and *delivered*.
- **Manager** - A manager should be able to do everything the chef, waiter and delivery actors can.

12 Nonfunctional requirements

Usability

The system should be easy to use both by an experienced and non-experienced user. The interface should provide hints to what the user is intended to do, and provide helpful information if the user is doing something wrong.

Reliability

It is important to have data persistence, so that orders are not lost when the system is closed. This is to make sure that ongoing orders are not lost, in case of unexpected system shutdowns.

Performance

The DROM system API should strive to be RESTfull [17]. This is to create better support for concurrent usage, and expandability.

The DROM system client should be fast, so as not to slow down employees when they use the system. More than a second wait after an input is made is problematic.

Supportability

The DROM system should be easy to maintain, it shall be easy to fix possible defects or bugs, and it should be easy to expand with new functional requirements. Hence we will focus on loose coupling and high cohesion [8].

Implementation

Everything to do with the database should be done through Entity Framework [11]. This is to keep everything inside one framework.

Interface

Bringing the DROM system client around with you on the workplace is important, and as such, the DROM system client should have a responsive, scale-able and touch-friendly UI which can also work on mobile devices.

13 System models

13.1 Scenarios

Writing scenarios is part of requirements elicitation. Scenarios are short descriptions of common cases of use of the system. These focuses on understandability and not so much on completeness.

We ended up with a total of four scenarios that explains what the DROM system should be able to do. The four scenarios are as follows:

- A customer eats at the restaurant.
- A customer enters the restaurant to place an order for takeaway.
- A customer calls the restaurant on the phone to place an order for delivery.
- A customer orders on the website for delivery.

These scenarios will be described in detail below. There can be variations of the business processes, like changing items on the order or changing the delivery type, which are not described in the scenarios below. These variations and changes can be further examined in section *13.2: Use case models*.

A customer eats at the restaurant

A customer named Alice wants to eat at the the restaurant and goes inside. The waiter named Bob greets her and leads her to a free table and takes her order. Bob creates a new order using the DROM system client.

A chef named Charlie sees that Alice's order is in the DROM system client. He enters into the DROM system client that he is cooking her order and starts to cook it for serving. When he is done cooking her order, he enters that he is done cooking her order in the DROM system client.

The waiter named Bob sees that Alice's order is ready for serving. He serves the order to Alice, and enters that the order has been served in the DROM system client. When Alice is done eating, she asks Bob for the bill. Bob receives the payment and Alice leaves the restaurant. Bob enters in the DROM system client that Alice has paid.

Bob sees that Alice's table now need to be cleaned and made ready for the next customer, Bob cleans the table and enters that he is done cleaning the table in the DROM system client. Alice's order is now done, and Bob archives the order so that it is no longer visible in the DROM system client.

A customer enters the restaurant to place an order for takeaway

A customer named Alice wants to place an order for takeaway and eat it elsewhere. She enters the restaurant. A waiter named Bob greets her. Alice explains what she wants to order for takeaway. Bob enters Alice's order into the DROM system client and saves the order.

A chef named Charlie sees that Alice's order is in the DROM system. He enters that he begins to cook it for takeaway. When Charlie is done cooking he enters that he is done cooking the order in the DROM system client.

Bob sees that Alice's order is ready for pickup. Alice pays and receives her order and leaves. Bob enters that Alice has paid and that her order has been picked up. Bob sees that Alice's order is done, and archives it.

A customer calls the restaurant on the phone to place an order for delivery

A customer named Alice wants to place an order for delivery to her address. She calls the restaurant on the phone. A waiter named Bob answers the call. Alice explains that she want to order for delivery, what items she wants to order, and her address information. Bob enters these details into the DROM system client and saves the order. Bob and Alice hang up.

A chef named Charlie sees that Alice's order is in the system. He enters that he begins to cook the order for delivery. When he is done cooking, he enters that he is done cooking the order.

A delivery employee named Dan sees that Alice's order is ready for delivery. He enters that her order is being delivered, and takes the order to Alice's address. When Dan arrives, Alice pays for the order and Dan hands over her order. Dan enters that Alice has paid and that the order has been delivered into the DROM system client. Dan sees that the order is now done and archives it.

A customer orders on the website for delivery

A customer named Alice wants to place an order for delivery to her address. She enters the DROM system website. She enters items she would like, that she would like to have them delivered, and enters her address information. She then clicks 'Create order'.

In the restaurant, a waiter named Bob sees Alice's web order in the DROM system client. He checks that all information is correctly entered and that the order is something the restaurant can manage. He confirms Alice's web order. Alice receives an email, that says her order has been confirmed.

A chef named Charlie sees Alice's order in the DROM system client. He enters that he will begin to cook the order for delivery. When he is done cooking, he enters that he is done cooking in the DROM system client.

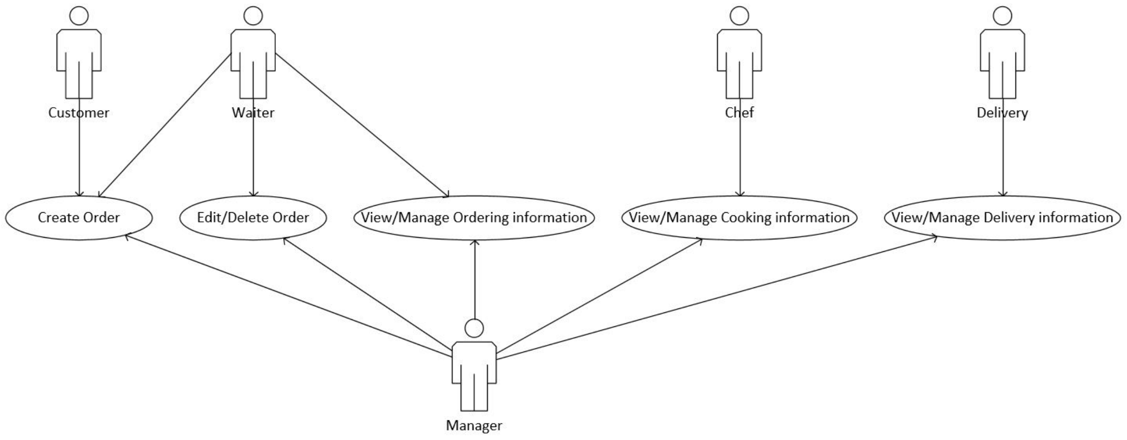
A delivery employee named Dan sees that Alice's order is ready for delivery. He enters that the order is being delivered. When Dan arrives at Alice's address, Alice pays for her order and Dan hands over her order. Dan enters that the order has been paid for and has been delivered. Dan sees that the order is now done and archives it.

13.2 Use case models

The following sections explains the use cases that each actor of the system must be able to do.

Overview

Figure 4 shows an overview of the use cases each actor must be able to do.



Customer
Waiter
Chef
Delivery
Create Order
Edit/Delete Order
View/Manage Ordering information
View/Manage Cooking information
View/Manage Delivery information
Manager

Figure 4: Use case model for the DROM system

Customer, *Waiter* and *Manager* must be able to create orders. The *Customer* shall be able to create orders using the DROM system website. This use case will be further described in use case *Customer creates order on website*. The *Waiter* shall be able to create orders using the DROM system client, as described in use case *Waiter creates order*. The *Manager* shall also be able to create orders as the *Waiter*, using the DROM system client, as described in use case *Waiter creates order*.

Waiter and *Manager* must be able to *Edit* and *Delete Order*s. The *Waiter* shall be able to edit the order, for example the items on the menu from a customer's request as described in use case *Waiter edits order*. The *Waiter* shall also be able to delete orders, as described in use case *Waiter deletes order*, using the DROM system client. The *Manager* shall be able to do the same use cases as the *Waiter*, by using the DROM system client as described in the use cases *Waiter edits order* and *Waiter deletes order*.

View/Manage Ordering information must be possible by the *Waiter* and the *Manager*. The *Waiter* shall be able to *View Ordering information* and get an overview of the orders relevant for the waiter, when managing orders, for example orders to be served to tables, or tables that needs cleaning. *Manage Ordering information* includes confirming incoming orders, as described in use case *Waiter confirms order*. Serve orders to tables, confirm payment and clean tables, as described in use case

Waiter serves order. The *Manager* shall be able to do the same use cases as the *Waiter*, as described in use cases *Waiter confirms order* and *Waiter serves order.*

View/Manage Cooking information must be possible by the *Chef* and the *Manager.* The *Chef* shall be able to get an overview of which items are be cooked on an order, which in *figure 4* is referred to as *Cooking information.* The *Chef* shall also be able to confirm that the items of an order is being prepared, and confirm when the order is ready to be served or ready for delivery as described in use case *Chef cooks order.* The *Manager* shall be able to do the same use cases as the *Chef*, as described in use case *Chef cooks order.*

View/Manage Delivery information must be possible by the *Delivery* and the *Manager.* The *Delivery* shall be able to view orders that are to be delivered, the address of the customer and which items the customer has ordered. The *Delivery* shall be able to signal that an order is being delivered, and that the order has been delivered as described in use case *Order delivery.* The *Manager* shall be able to do the same use cases as the *Delivery.*

Use cases

Use cases can be found in *26.1: Use cases.* We have omitted them here because they take to much space. The reader may read them in appendix if interested.

13.3 DCRGraph

13.3.1 Creating the DCRGraph

Part of the method described in the OOSE book [6] for requirement specification, is making activity and/or state diagrams from your use cases.

We saw a relation between use cases and DCRGraphs, and as such we decided to try and make a DCRGraph from the use cases instead of creating activity and state diagrams. DCRGraphs contain much of the same information as is portrayed in activity and state diagrams, and as such it should be possible to use them instead.

To successfully make a DCRGraph from use cases, it is important that the use cases have comprehensive coverage of the activities that can occur in the system. It also helps to try to design the use cases in a way that reflects events which you would put in a DCRGraph. This can be achieved by making the use cases small, and making good use of extension (having use cases point to each other instead of describing everything inside one use case). Once the use cases are well designed and they cover everything that can happen in the system, it is time to make the DCRGraph.

We found it useful to make this into a two step process.

- First we identify what events should be in the DCRGraphs. This is done by going through the use cases one by one, and checking for turning points. That is, events in the use cases where something is done with the system, where the order changes state.

This is also where you decide the scope of the DCRGraph. The scope is what to model and what not to model in the DCRGraph. An example of scope is figure *40: Customer creates order on website* where a customer makes an order over the DROM system website.

We decided that we would not model the website behavior through a DCRGraph, and as such set it as out of scope of the graph. Another example is figure *42: Waiter confirms order* which is what the previous example extends to. Here we see that a waiter needs to confirm an order after it has been placed on the DROM system website. We can see that there is a state change from unconfirmed order, to confirmed order.

This is something which happens in the DROM system client, which is the kind of functionality we want to model with DCRGraphs, as such, it is in scope of the DCRGraph. We now know that an event related to this needs to be created, in our case this became "Confirm web order".

- Once all the events have been identified, it is time to start working on relations. To do this, we once again start looking at the use cases. This time we look for anything that tells us whether something happens in a special order, requires certain things to happen, or otherwise specifies something about the flow of events. This is then made into relations between events.

When this has been done, you should hopefully end up with an initial DCRGraph that model the flow of events in your system.

If the use cases or requirements to the system are to be changed, it is necessary to repeat this process.

It is very hard to create a perfect DCRGraph the first time. Use cases only specify what needs to be modelled, but not exactly how to model it. As such the DCRGraph you have created from use cases only serves as the first draft. Use cases help you to make a more complete first draft, by serving as a guideline for what needs to be modelled and by helping you to better analyse exactly how events are related.

The final DCRGraph will be created from realising missing events and relations discovered from multiple iterations over the initial DCRGraph. This is however not something the developer should worry about, since iterating on the DCRGraph is a natural ongoing process. If designed correctly, the system will change according to the changes in the DCRGraph, as we will later discuss in section *21.5.2: Updating the design through DCRGraphs.* Hence creating the final DCRGraph is an iterative process.

13.3.2 The resulting DCRGraph

In our finished DCRGraph we have divided events into three types. The biggest part of the DCRGraph is the core logic, which describe the steps employees have to go through when doing the various tasks that come up, when working in the restaurant. This does not include events to do with editing an order or changing the flow of the DCRGraph which we will describe afterwards. The core logic consists of ten events:

1. **Confirm web order -** When this event becomes pending, it represents that a customer has placed an order through the DROM system website, and the order is waiting to be confirmed by an employee. Executing this event signifies that the order has been confirmed by an employee.
2. **Begin cooking order for delivery or pickup -** Once an order has been received, it needs to be cooked. When this event is executed, it represents that one or more chefs have begun cooking. This event is specific to delivery and takeaway orders, and when the chef executes it through the client, the chef will know that the food should be prepared for transportation.
3. **Begin cooking order to eat in restaurant -** This event represents the same as the above event, *Begin cooking order for delivery or pickup*, with the difference that here the food is prepared to be served directly in the restaurant.
4. **Done cooking -** This event represents that the cook is done preparing the food, and once it is executed, the DCRGraph will go into a state signaling that the order is waiting to be picked up, waiting to be delivered or waiting to be served.
5. **Picked up -** This event signifies that an order is ready and waiting to be picked up by a customer. Once it is executed, it indicates that the customer has picked up the order. This event requires that the order has been paid for, through a milestone from the event ”Pay”.
6. **Order is being delivered -** This event signifies that an order is ready and waiting to be be delivered to a customer. Once it is executed, it means that an employee has taken the food and is in the process of delivering it to the customer.
7. **Order has been delivered -** This event signifies that an order is in the process of being delivered. Once it is executed, it indicates that the customer has received the order at their address. This event requires that the order has been paid for, through a milestone from the event ”Pay”.
8. **Serve order to table -** This event signifies that an order is ready and waiting to be served to a customer. Once it is executed, it indicates that the food has been served to the customer at their table.
9. **Clean table -** This event signifies that the next step for this order would be to clean the table, a waiter uses this to know that customers are currently in the process of eating and should be checked up on regularly to determine if they have any further requests. Once it is executed, it means that the customers are done eating and have left their table, and that a waiter has cleaned the table. This event requires that the order has been paid for, through a milestone to the event ”Pay”.
10. **Pay -** This event indicates that an order has not yet been paid for. Once it is executed, it means that the customer has paid for their order. This event imposes restrictions on certain steps of the processes in the restaurant, since some steps would not make sense before an order has been paid for, such as picking up a takeaway order, which is modelled in the event ”Picked up”.

These events and their relations can be seen in *figure 5.*

There is a slight difference between how the events are used. Some events, such as ”Begin cooking order for delivery or pickup” and ”Begin cooking order to eat in restaurant” events, signal that an activity starts. Other events, such as ”Done cooking” signifies that a task is done. We used events this way because we had a need to model ongoing tasks, and not only immediate changes in state.

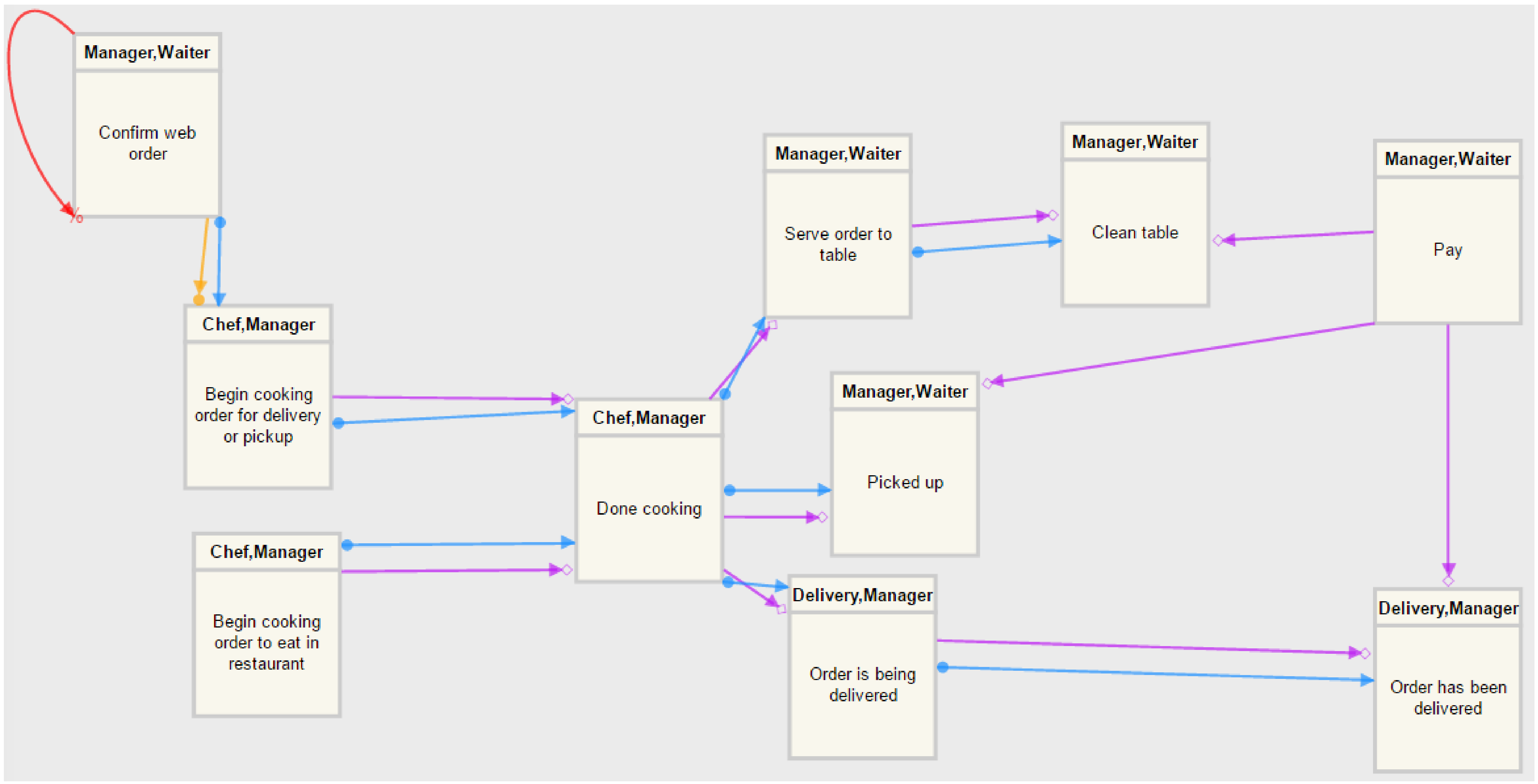


Figure 5: The core logic of the DCRGraph

The second type of events is "edit events". These events change the order in some way, such as changing from one delivery method to another, or adding new food items to an order. We have defined the following five edit events:

1. **For serving -** This event will exclude the events that are related to delivery and takeaway, and include events related to serving an order in the restaurant. As such, the event represents changing delivery type to serving.
2. **For delivery -** Same as the above, except this changes the delivery type to "delivery", which means delivering the food to a customers address.
3. **For takeaway -** Same as the two above, except this changes the delivery type to takeaway.
4. **Items have changed -** This event represents that food items have been either added or removed from an order. This event is special in the sense that it is not executed by the user, but is automatically executed by the system when an order is edited with added or removed food items.
5. **confirm changes -** This event is the followup to "Items have changed", in the sense that once items have changed, the chef should confirm that they have seen and acknowledged this change. This is done by executing this event.

These events and their relations can be seen in *figure 6*.

These events, except for "confirm changes", are not normally visible to the user, in the sense that they will only show up when the user enters the edit screen on the DROM system client. As such, these events are also exempt from the "pending only" rule which describes how events are shown in the DROM system client, described in *15: Order overview screen.*

Since these events are not necessarily something that has to be done, it would not make sense to have them pending, but at the same time they should still be available to the user. "confirm changes" is shown in the normal DROM system client view, and goes under the "pending only" rule.

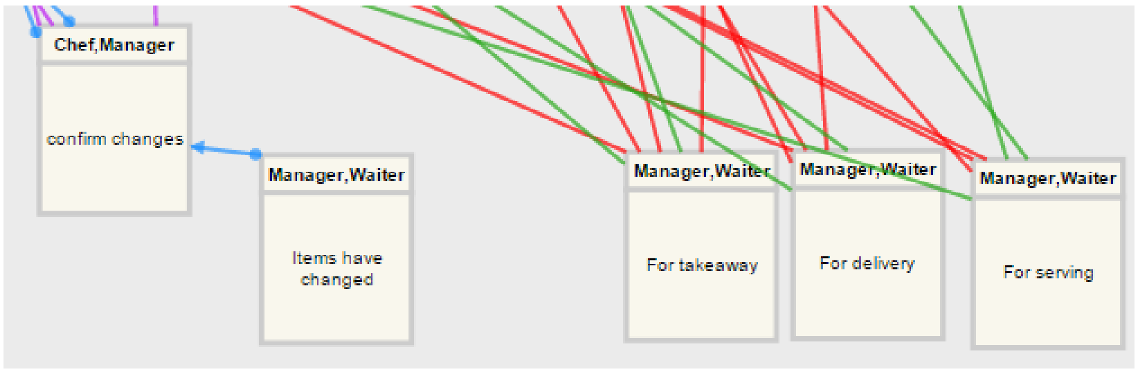


Figure 6: The edit logic of the DCRGraph

The last type of events, are setup events. These events are special events, whose purpose is to put the DCRGraph into a state where it will support a specific type of delivery. One, and only one, of these events are executed when a new order is created. The setup events are not executed by the user, but by the DROM system when a new order is created.

1. **Setup graph serving** - This event will exclude events related to other delivery types than serving. This means that this is the event that will be executed when a serving order is created in the DROM system client.
2. **Setup graph takeaway** - Same as above, except this is for takeaway.
3. **Setup graph delivery** - Same as above, except this is for delivery.
4. **Setup graph web takeaway** - Same as above, except this is for web takeaway. The "web" part means that this is the event to setup the graph when the order is created from the DROM system website.

   There are two main differences: One, we assume that web orders are paid when they are created, and as such "Pay" is not set pending. Two, web orders have to be accepted by the restaurant, which means that web orders use the event "Confirm web order".
5. **Setup graph web delivery** - This events is the same as "Setup graph web takeway", except it is for delivery instead.

These events and their relations can be seen in *figure 7*.

The setup events are different from normal events, in that they are executed by the DROM system API when new orders are created. The user will never see these events, and only one of them are ever executed for each instance of the DCRGraph.

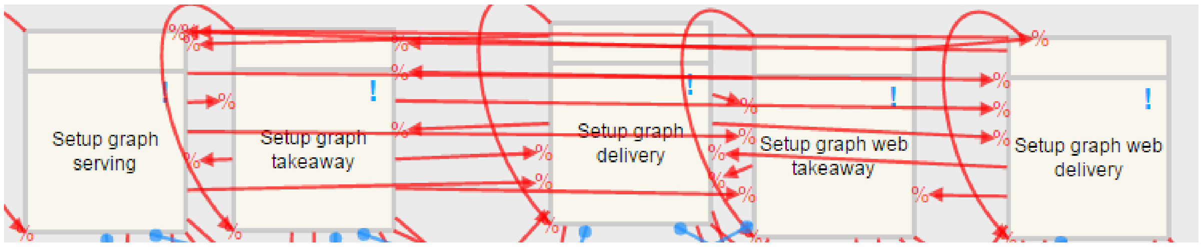


Figure 7: The setup logic of the DCRGraph

The full and final graph for the initial development of the system can be seen in *figure 8: The complete DCRGraph*.

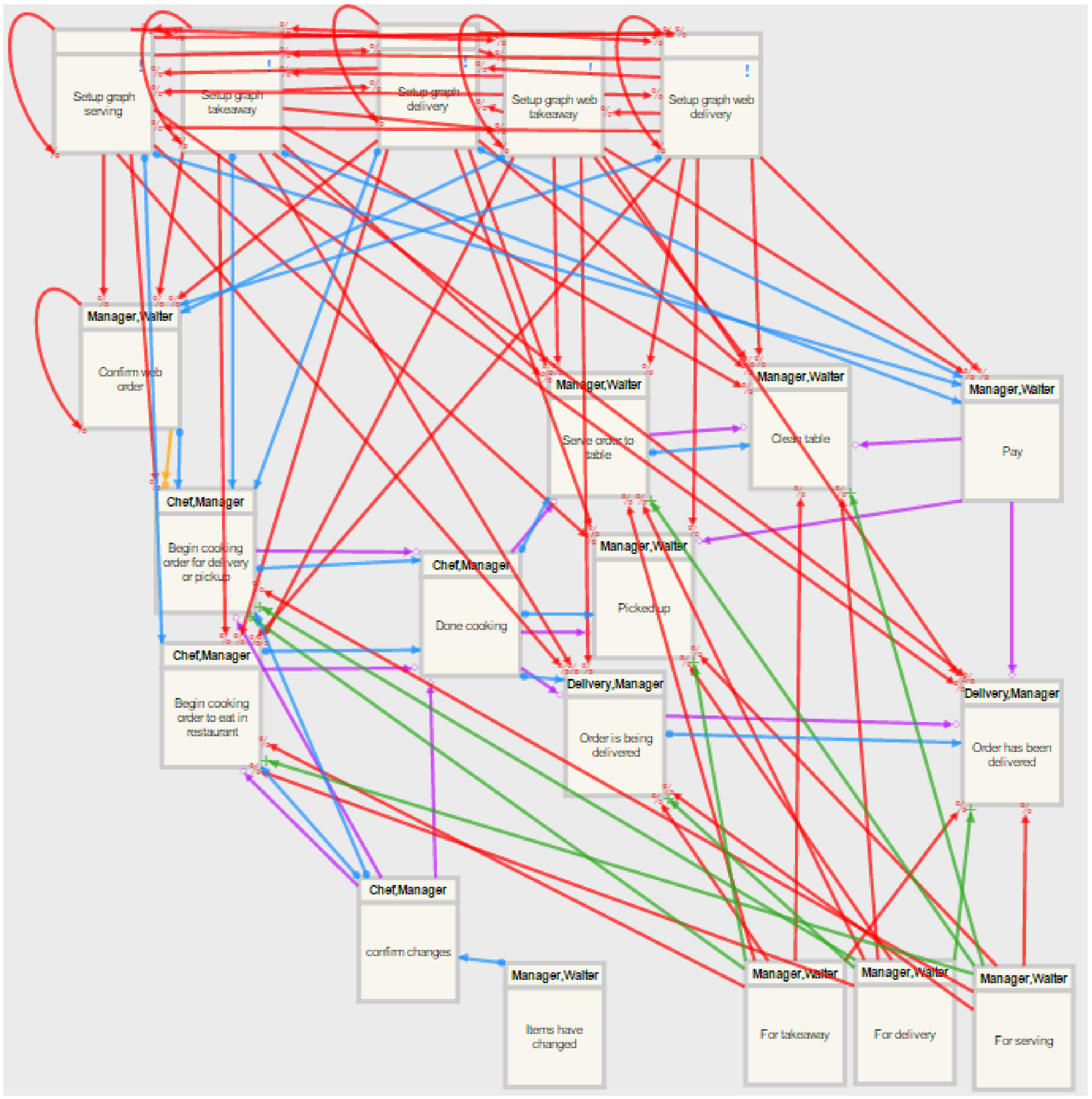


Figure 8: The complete DCRGraph

14 Software architecture

This section describes the design of the system that was developed to meet the requirements of the fictional restaurant.

14.1 Subsystem decomposition

There are four parts making up the DROM system:

1. An Application Programming Interface, referred to as Web API
2. A Database
3. A Client Application
4. A Website

14.1.1 Web API

The Web API is designed as a web service, meant to run on a server. The function of the Web API is to work as an interface for our database. The Web API is able to receive request over the Internet from a client, it will find the corresponding data in the database, and send back a request to the client.

It contains methods for creating new orders in the database, executing events on orders, update orders, archive orders and methods to get orders and order related data from the database. The specific available requests can be seen in the structure diagram in section *14.2.2: figure 10*.

The API also contains the logic to read DCRGraphs from xml files, exported from DCRGraphs.net, into the database. This is how a DCRGraph is created when new orders are made. The API is also responsible for the DCRGraph logic when executing events. This means that the API is checking whether event relations are in a correct state for an event to be executed, when it is asked to execute an event.

The API is developed on the ASP.NET 4.5.2 framework [4], utilizing the Web API template [5].

14.1.2 Database

In the DROM system we use a database for persistent data storage. This is because it would not make sense for ongoing orders to disappear, in the case that the system was forced to shut down. Furthermore, completed and cancelled orders are stored. This is to allow the possibility of analysing the order history, however we have not implemented tools to do this as it is out of scope of this project.

The database contains all data related to an order. This means storing order information about what is being bought, customer information, and the DCRGraph that is related to the order.

The database is built with Entity Framework version 6.1.3 [12], and all database interactions are handled through this framework. For the purpose of prototyping the system, we have been using a database file located together with the Web API project as our database, however the database can be deployed on a web service without any issues for the Web API to use it from there.

The database schema can be seen in section section 14.2.1, *figure 9: Database schema.*

The Entity Framework classes that make up the database design can be seen in the structure diagram for the Web API in section 14.2.2, *figure 10: Folder structure and classes of the DROM Web API project.*

14.1.3 Client Application

The DROM system client is the client application of the DROM system. It is the program that the employees in the restaurant will interact with. It is the employers tool to manipulate orders.

It contains classes that makes up the different screens of the application, classes for sending and receiving data to the Web API through HTTP requests, and classes for filtering and converting the data received to viewable data presented on the different screens.

Framework
Our client application is a Graphical User Interface (GUI) implemented in the Universal Windows Platform (UWP) application framework [16] for Microsoft Windows operating systems. UWP is the newest GUI development framework by Microsoft as of January 2016, when this project started. UWP enables easy development of native Windows applications, together with easy portability to PCs, tablets and phones running Windows 10.

We have chosen to use this framework because we are both Windows 10 users, we are familiar with the C# programming language, and because we already had some experience with the XAML visual markup language [15] developing WPF [20] applications in previous projects which are very similar to UWP. We thought it would be interesting for our case, to have the client application running on phones and tablets as well as PCs, however we did not have the equipment to test the application on other electronic devices than PC.

Our UWP application follows the recommended Model-View-ViewModel (MVVM) architectural pattern [14] for UWP applications. The pattern is great for separation of view logic (the View), data and business logic (the Model) and the controller that organizes the data between the two (the ViewModel). An important aspect of the pattern is that the View knows and binds to the ViewModel, but the ViewModel does not know about the View, also the ViewModel knows about the Model, but the Model does not know about the ViewModel. This pattern offers great testability and separation of concerns and goes great with the UWP framework.

14.2 Object models

14.2.1 Database schema

Due to the size of the database schema, it is shown on the next page in *figure 9*. This is the schema describing our database design. The notation used is called Chen's notation [7]. The blue rectangles are entity tables, the green circles are the attributes of the entity, and the purple diamonds are relationship tables.

A notable thing about the schema, in relation to Entity Framework, is that all the entity tables exists as classes in the DROM system API. They are the classes under DBObjects in *figure 10: Folder structure and classes of the DROM Web API project*.

The relationship table "OrderDetail" is an exception, as it also exists as a class. This is because it contains more information in the form of attributes than just the two foreign keys [13] from "Order" and "Item".

All entity tables and relationship tables are created by Entity Framework and exists as tables in the database.

Regarding the entity table "IntegerSpecifyingUIElements", we originally created this to allow events to be related to parts of the UI in the DROM system client. We still think it would a nice thing to have, such that events could contain information about elements of the UI, enabling features such as highlighting elements in the UI that are related to an event. However we never had the time to fully implement it.

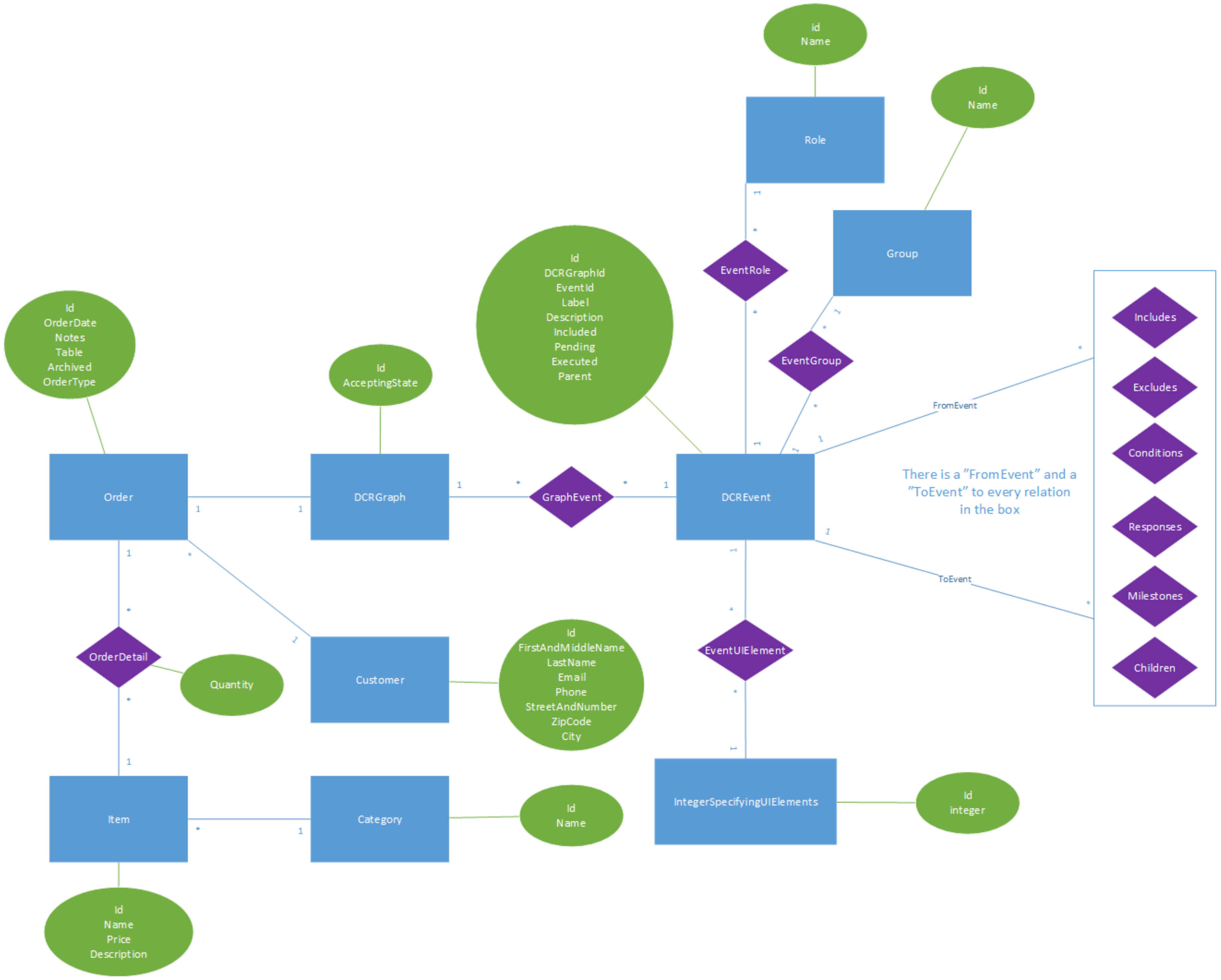


Figure 9: Database schema

14.2.2 Web API structure

In *figure 10* on the following page, we see the structure diagram for the DROM system API. The diagram is sectioned into folders, which contain different classes.

The "Controllers" folder contains the classes that handles Web requests, that can be made to the API, with methods such as *GetOrders* or *ArchiveOrder*.

The "DBObjects" folder contains the classes that make up the database in Entity Framework, this means that we can see all the tables in the database as classes here, except relationship tables with the exception of the "OrderDetail" relationship table.

The "XMLParser" folder contains classes to read DCRGraphs from xml files, exported from online modelling tool `www.DCRGraphs.net`.

There are also two classes which do not go under any subcategory: "DbInteractions" contains methods to interact with the DROM system database. "Mapper" is responsible for creating new orders by mapping a DCRGraph, parsed from an xml file, into Entity Framework database classes, and stores the order in the database.

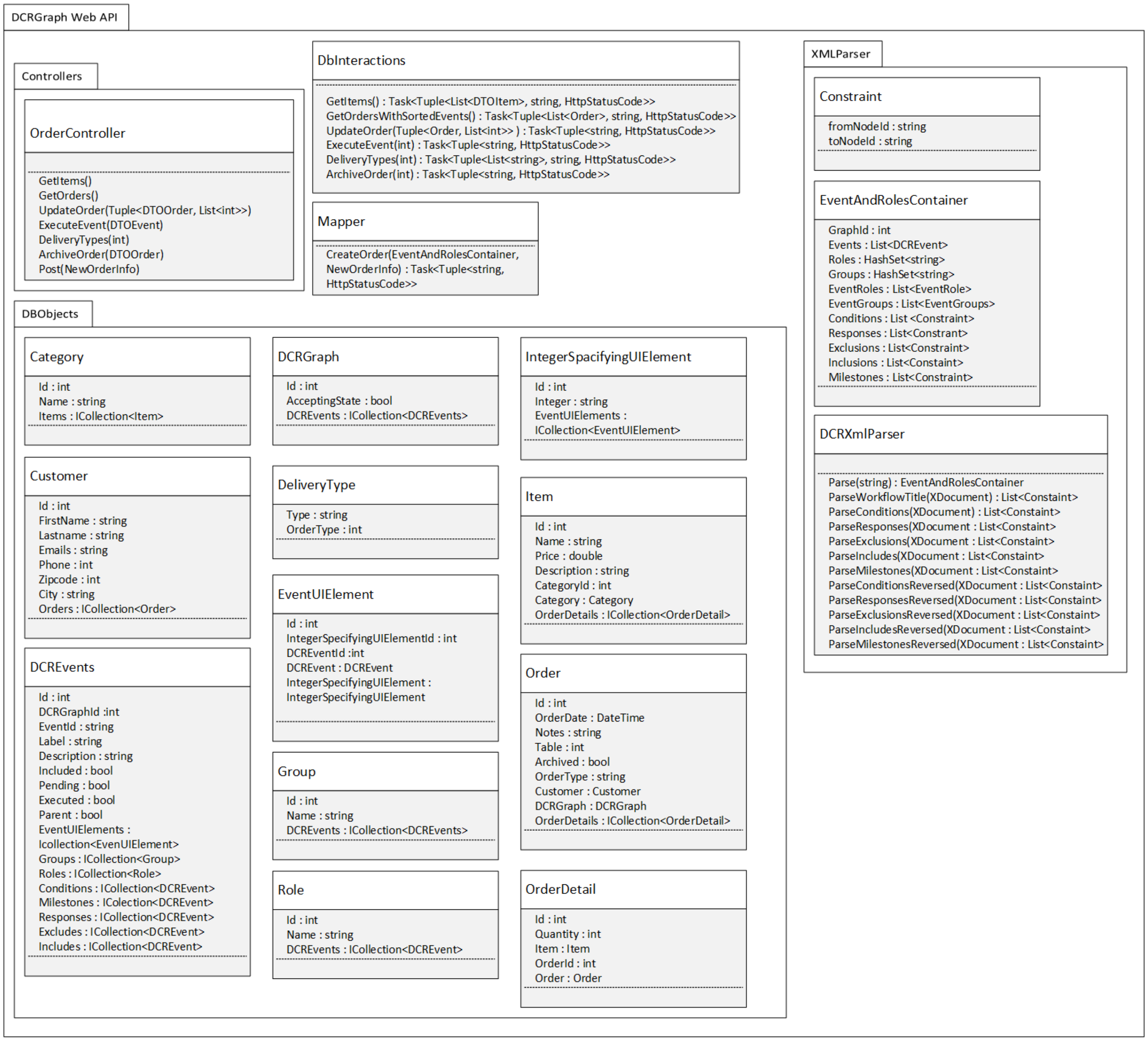

DCRGraph Web API
Controllers
OrderController
GetItems()
GetOrders()
UpdateOrder(Tuple<DTOOrder, List<int>>)
ExecuteEvent(DTOEvent)
DeliveryTypes(int)
ArchiveOrder(DTOOrder)
Post(NewOrderInfo)
DbInteractions
GetItems() : Task<Tuple<List<DTOItem>, string, HttpStatusCode>>
GetOrdersWithSortedEvents() : Task<Tuple<List<Order>, string, HttpStatusCode>>
UpdateOrder(Tuple<Order, List<int>>) : Task<Tuple<string, HttpStatusCode>>
ExecuteEvent(int) : Task<Tuple<string, HttpStatusCode>>
DeliveryTypes(int) : Task<Tuple<List<string>, string, HttpStatusCode>>
ArchiveOrder(int) : Task<Tuple<string, HttpStatusCode>>
Mapper
CreateOrder(EventAndRolesContainer, NewOrderInfo) : Task<Tuple<string, HttpStatusCode>>
XMLParser
Constraint
fromNodeId : string
toNodeId : string
EventAndRolesContainer
GraphId : int
Events : List<DCREvent>
Roles : HashSet<string>
Groups : HashSet<string>
EventRoles : List<EventRole>
EventGroups : List<EventGroups>
Conditions : List <Constraint>
Responses : List<Constrant>
Exclusions : List<Constraint>
Inclusions : List<Constaint>
Milestones : List<Constraint>
DCRXmlParser
Parse(string) : EventAndRolesContainer
ParseWorkflowTitle(XDocument) : List<Constaint>
ParseConditions(XDocument) : List<Constaint>
ParseResponses(XDocument : List<Constaint>
ParseExclusions(XDocument : List<Constaint>
ParseIncludes(XDocument : List<Constaint>
ParseMilestones(XDocument : List<Constaint>
ParseConditionsReversed(XDocument : List<Constaint>
ParseResponsesReversed(XDocument : List<Constaint>
ParseExclusionsReversed(XDocument : List<Constaint>
ParseIncludesReversed(XDocument : List<Constaint>
ParseMilestonesReversed(XDocument : List<Constaint>
DBObjects
Category
Id : int
Name : string
Items : ICollection<Item>
Customer
Id : int
FirstName : string
Lastname : string
Emails : string
Phone : int
Zipcode : int
City : string
Orders : ICollection<Order>
DCREvents
Id : int
DCRGraphId :int
EventId : string
Label : string
Description : string
Included : bool
Pending : bool
Executed : bool
Parent : bool
EventUIElements : Icollection<EvenUIElement>
Groups : ICollection<Group>
Roles : ICollection<Role>
Conditions : ICollection<DCREvent>
Milestones : ICollection<DCREvent>
Responses : ICollection<DCREvent>
Excludes : ICollection<DCREvent>
Includes : ICollection<DCREvent>
DCRGraph
Id : int
AcceptingState : bool
DCREvents : ICollection<DCREvents>
DeliveryType
Type : string
OrderType : int
EventUIElement
Id : int
IntegerSpecifyingUIElementId : int
DCREventId :int
DCREvent : DCREvent
IntegerSpecifyingUIElement : IntegerSpecifyingUIElement
Group
Id : int
Name : string
DCREvents : ICollection<DCREvents>
Role
Id : int
Name : string
DCREvents : ICollection<DCREvent>
IntegerSpacifyingUIElement
Id : int
Integer : string
EventUIElements : ICollection<EventUIElement>
Item
Id : int
Name : string
Price : double
Description : string
CategoryId : int
Category : Category
OrderDetails : ICollection<OrderDetail>
Order
Id : int
OrderDate : DateTime
Notes : string
Table : int
Archived : bool
OrderType : string
Customer : Customer
DCRGraph : DCRGraph
OrderDetails : ICollection<OrderDetail>
OrderDetail
Id : int
Quantity : int
Item : Item
OrderId : int
Order : Order

Figure 10: Folder structure and classes of the DROM Web API project

14.2.3 DROM Client structure

In *figure 11* below, an overview of folders in the DROM Client project is shown. One diagram with all contents would be too large. Hence we will go through each folder in the following sections.

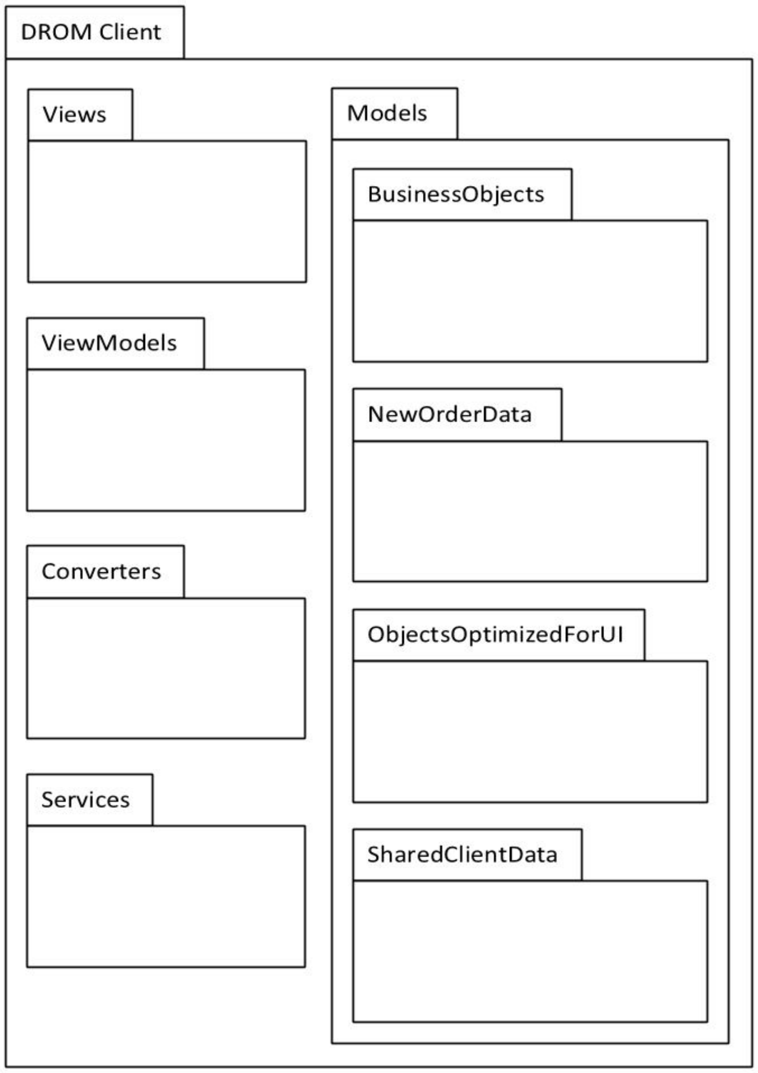


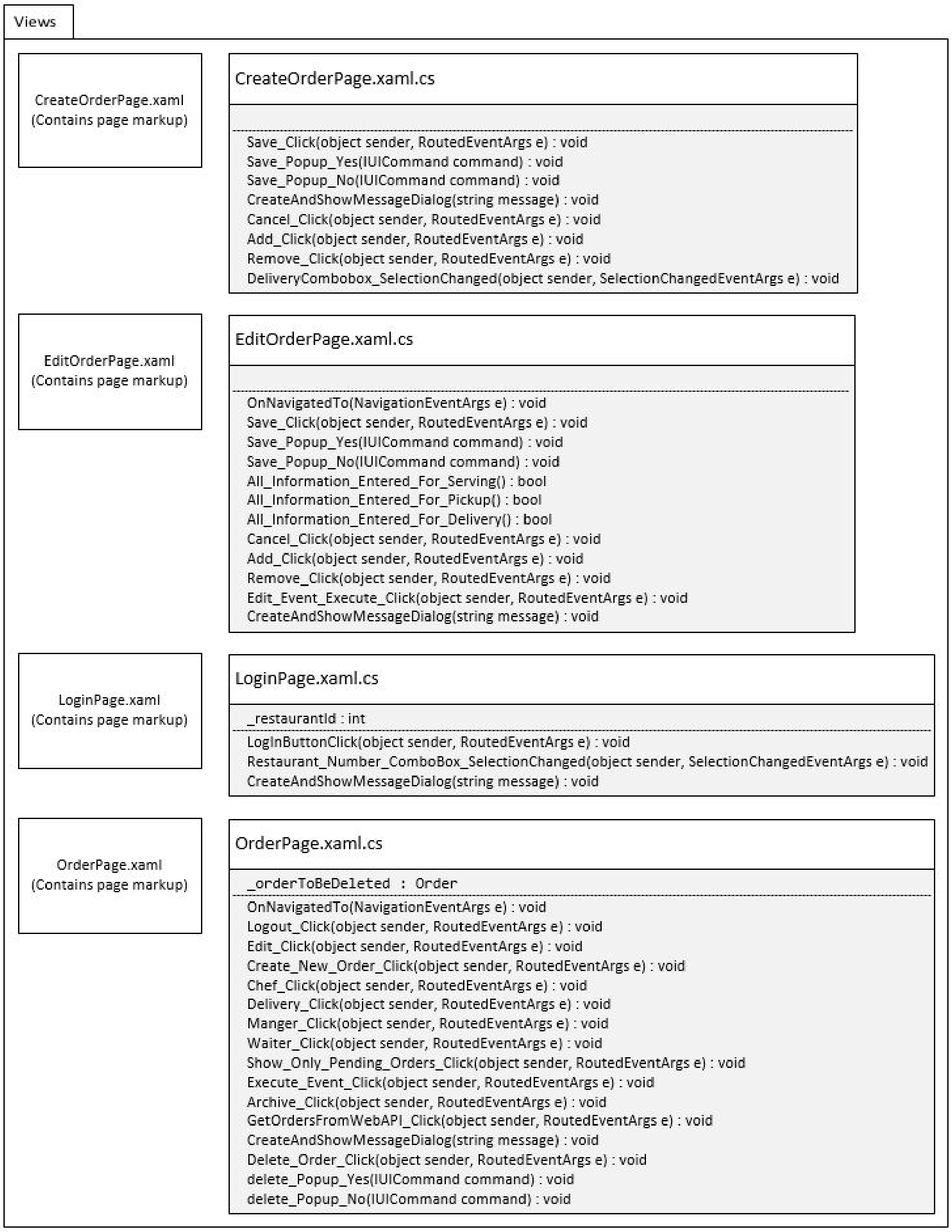
Figure 11: Folder overview of the DROM Client project

The DROM Client is build in the Universal Windows Platform (UWP) application framework [16] using the recommended MVVM pattern [14] for UWP applications. The framework and architectual pattern is described in more detail in section *14.1.3: Client Application.* This is the reason why the folders "Views", "ViewModels" and "Models" exists.

Views folder
"Views" contains the .XAML and .XAML.cs classes that make up the View logic of the DROM Client. An overview of the the View classes, their fields and methods can be seen in *figure 12: Views folder overview* on the following page.

The Views folder contains four Views (in eight classes) that makes up the four different screens you will see in our DROM Client application: A screen for creating new orders, a screen for editing orders, a screen for login to a restaurant, and the main order overview screen. The user interface will be shown and described later in section *15: User interface.*

They reason why two classes makes up one view, is because one .xaml file and one .xaml.cs class are compiled to one "page" which is run inside a "frame" that is the window of the application. The .xaml file contains the XML based visual markup of the page, while the .xaml.cs file contains page-specific logic called CodeBehind [21]. For example, a visual button on the page, it's position and how it looks is specified in the .xaml file. It's click-event-handler is specified in it's CodeBehind file. Each visual element can be bound to properties in a ViewModel, which we will go through shortly.



```
CreateOrderPage.xaml.cs
-----------------------
Save_Click(object sender, RoutedEventArgs e) : void
Save_Popup_Yes(IUICommand command) : void
Save_Popup_No(IUICommand command) : void
CreateAndShowMessageDialog(string message) : void
Cancel_Click(object sender, RoutedEventArgs e) : void
Add_Click(object sender, RoutedEventArgs e) : void
Remove_Click(object sender, RoutedEventArgs e) : void
DeliveryCombobox_SelectionChanged(object sender, SelectionChangedEventArgs e) : void
```

```
EditOrderPage.xaml.cs
-----------------------
OnNavigatedTo(NavigationEventArgs e) : void
Save_Click(object sender, RoutedEventArgs e) : void
Save_Popup_Yes(IUICommand command) : void
Save_Popup_No(IUICommand command) : void
All_Information_Entered_For_Serving() : bool
All_Information_Entered_For_Pickup() : bool
All_Information_Entered_For_Delivery() : bool
Cancel_Click(object sender, RoutedEventArgs e) : void
Add_Click(object sender, RoutedEventArgs e) : void
Remove_Click(object sender, RoutedEventArgs e) : void
Edit_Event_Execute_Click(object sender, RoutedEventArgs e) : void
CreateAndShowMessageDialog(string message) : void
```

```
LoginPage.xaml.cs
_restaurantId : int
-----------------------
LogInButtonClick(object sender, RoutedEventArgs e) : void
Restaurant_Number_ComboBox_SelectionChanged(object sender, SelectionChangedEventArgs e) : void
CreateAndShowMessageDialog(string message) : void
```

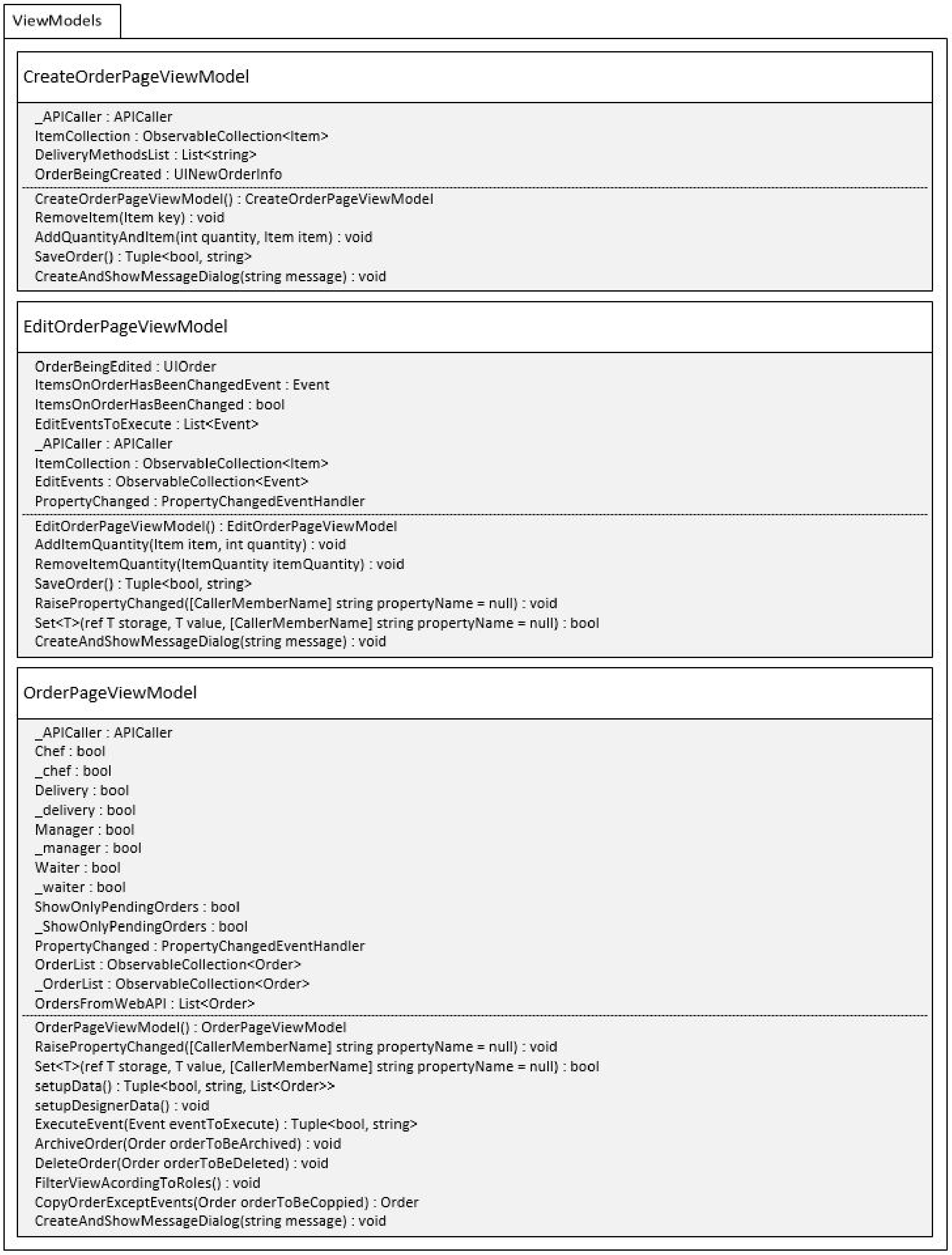
```
OrderPage.xaml.cs
_orderToBeDeleted : Order
-----------------------
OnNavigatedTo(NavigationEventArgs e) : void
Logout_Click(object sender, RoutedEventArgs e) : void
Edit_Click(object sender, RoutedEventArgs e) : void
Create_New_Order_Click(object sender, RoutedEventArgs e) : void
Chef_Click(object sender, RoutedEventArgs e) : void
Delivery_Click(object sender, RoutedEventArgs e) : void
Manger_Click(object sender, RoutedEventArgs e) : void
Waiter_Click(object sender, RoutedEventArgs e) : void
Show_Only_Pending_Orders_Click(object sender, RoutedEventArgs e) : void
Execute_Event_Click(object sender, RoutedEventArgs e) : void
Archive_Click(object sender, RoutedEventArgs e) : void
GetOrdersFromWebAPI_Click(object sender, RoutedEventArgs e) : void
CreateAndShowMessageDialog(string message) : void
Delete_Order_Click(object sender, RoutedEventArgs e) : void
delete_Popup_Yes(IUICommand command) : void
delete_Popup_No(IUICommand command) : void
```

Figure 12: Views folder overview

ViewModels folder
An overview of classes in the "ViewModels" folder can be seen in *figure 13: ViewModels folder overview* on the following page.

These classes makes up the ViewModels for the Views. In our project we have three ViewModels used by three of the four Views. Normally you would have one ViewModel for each View. The LoginPage View does not have a ViewModel because we did not find it necessary.

A ViewModel sends and receives data from Models and Services, and transforms the received data into properties that provides data to the View. One or more visual elements in a View can create a data binding to a property in a ViewModel, the visual element is then able to show the data from the ViewModel, and update whenever the bound data updates. A View knows it's ViewModel, however a ViewModel is not aware of it's View. This separates concerns and decouples the code between the View and the ViewModel.



```
ViewModels

CreateOrderPageViewModel
  _APICaller : APICaller
  ItemCollection : ObservableCollection<Item>
  DeliveryMethodsList : List<string>
  OrderBeingCreated : UINewOrderInfo
  ----------------------------------------
  CreateOrderPageViewModel() : CreateOrderPageViewModel
  RemoveItem(Item key) : void
  AddQuantityAndItem(int quantity, Item item) : void
  SaveOrder() : Tuple<bool, string>
  CreateAndShowMessageDialog(string message) : void

EditOrderPageViewModel
  OrderBeingEdited : UIOrder
  ItemsOnOrderHasBeenChangedEvent : Event
  ItemsOnOrderHasBeenChanged : bool
  EditEventsToExecute : List<Event>
  _APICaller : APICaller
  ItemCollection : ObservableCollection<Item>
  EditEvents : ObservableCollection<Event>
  PropertyChanged : PropertyChangedEventHandler
  ----------------------------------------
  EditOrderPageViewModel() : EditOrderPageViewModel
  AddItemQuantity(Item item, int quantity) : void
  RemoveItemQuantity(ItemQuantity itemQuantity) : void
  SaveOrder() : Tuple<bool, string>
  RaisePropertyChanged([CallerMemberName] string propertyName = null) : void
  Set<T>(ref T storage, T value, [CallerMemberName] string propertyName = null) : bool
  CreateAndShowMessageDialog(string message) : void

OrderPageViewModel
  _APICaller : APICaller
  Chef : bool
  _chef : bool
  Delivery : bool
  _delivery : bool
  Manager : bool
  _manager : bool
  Waiter : bool
  _waiter : bool
  ShowOnlyPendingOrders : bool
  _ShowOnlyPendingOrders : bool
  PropertyChanged : PropertyChangedEventHandler
  OrderList : ObservableCollection<Order>
  _OrderList : ObservableCollection<Order>
  OrdersFromWebAPI : List<Order>
  ----------------------------------------
  OrderPageViewModel() : OrderPageViewModel
  RaisePropertyChanged([CallerMemberName] string propertyName = null) : void
  Set<T>(ref T storage, T value, [CallerMemberName] string propertyName = null) : bool
  setupData() : Tuple<bool, string, List<Order>>
  setupDesignerData() : void
  ExecuteEvent(Event eventToExecute) : Tuple<bool, string>
  ArchiveOrder(Order orderToBeArchived) : void
  DeleteOrder(Order orderToBeDeleted) : void
  FilterViewAcordingToRoles() : void
  CopyOrderExceptEvents(Order orderToBeCoppied) : Order
  CreateAndShowMessageDialog(string message) : void
```

Figure 13: ViewModels folder overview

Models folder
The "Models" folder contains folders in which classes are used to carry data. An overview of the Models folder can be seen in *figure 14: Models folder overview* on the following page.

The "BusinessObjecs" folder contains classes that are used to represent parts of an order. These are data transfer classes, which are used to communicate with the DROM system API. These classes contains data that can be seen and manipulated in the client application.

The "NewOrderData" folder contains a single class, that is used when a new order is created.

The "ObjectsOptimizedForUI" folder contains classes which properties and methods are optimized for View-specific functionality. For example, a normal property does not notify the View when the value is updated. Hence a need to specify UI optimized classes, which properties does notify the View when the values updates, was created.

The "SharedClientData" folder contains a singleton class [19], that is used to carry data that can be accessed from any other class in the project. This was used instead of sending the same data between view models when navigating the UI.

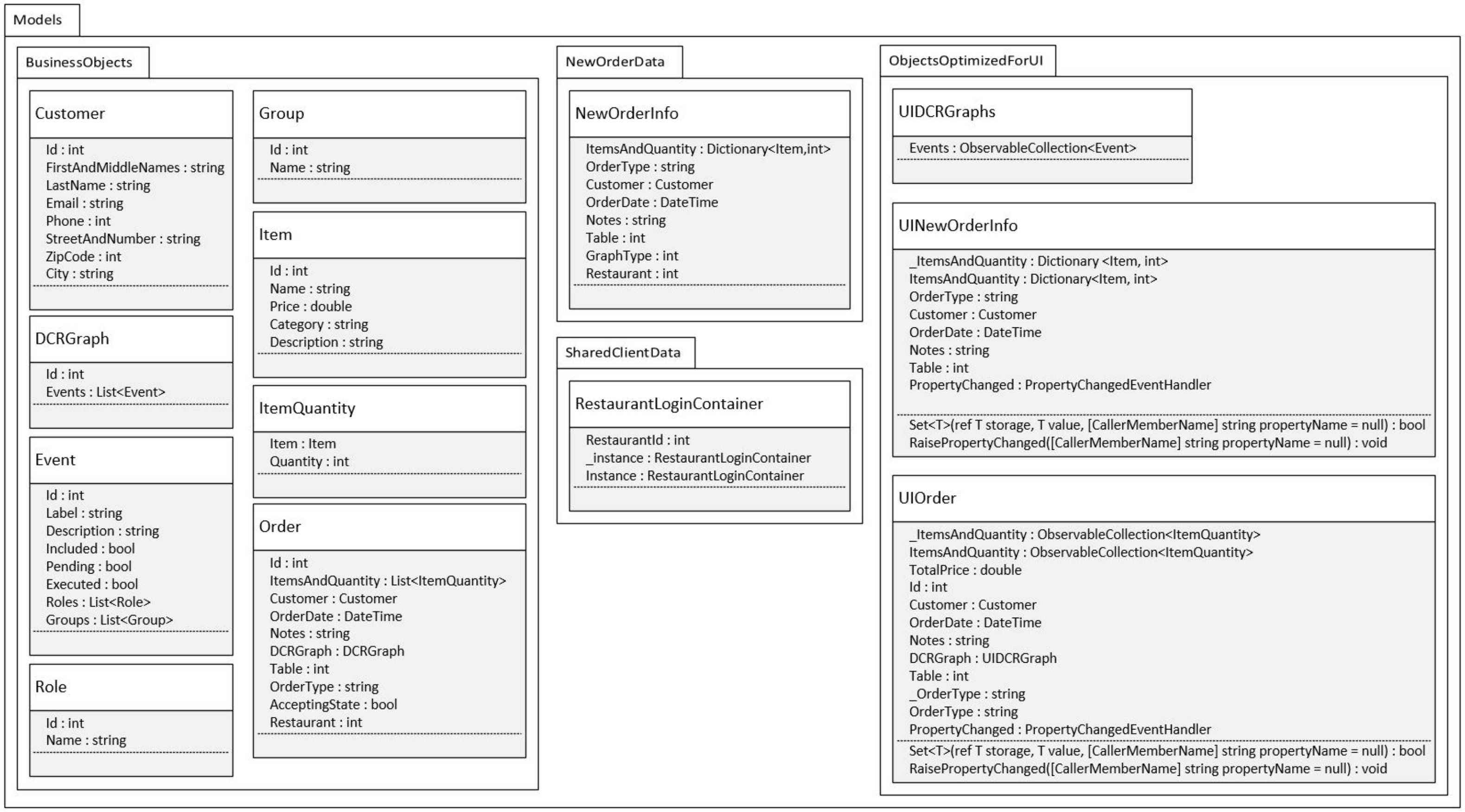
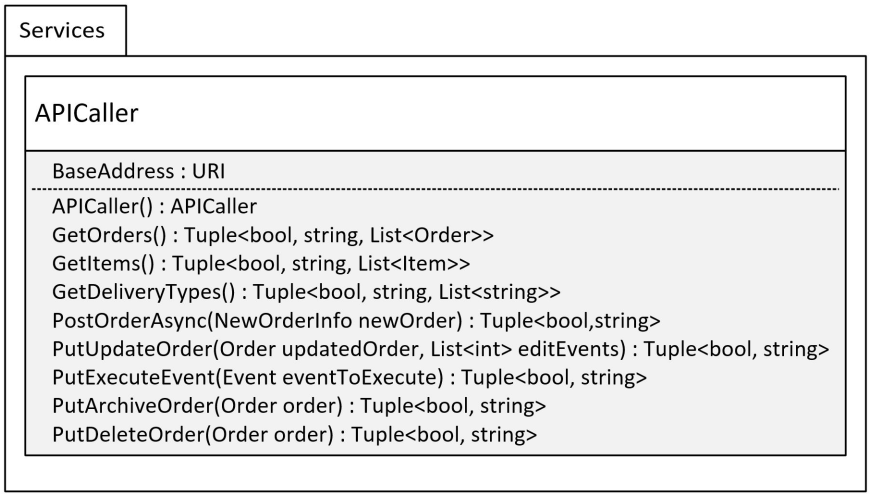


Figure 14: Models folder overview

Services folder
The "Services" folder contains classes that are used to contact online services such as a Web API. An overview of the "Services" folder can be found in *figure 15* below.

We only have one service to contact, hence only one class is needed. All communication with the DROM system API is handled through this class.



Services
APICaller
BaseAddress : URI
APICaller() : APICaller
GetOrders() : Tuple<bool, string, List<Order>>
GetItems() : Tuple<bool, string, List<Item>>
GetDeliveryTypes() : Tuple<bool, string, List<string>>
PostOrderAsync(NewOrderInfo newOrder) : Tuple<bool,string>
PutUpdateOrder(Order updatedOrder, List<int> editEvents) : Tuple<bool, string>
PutExecuteEvent(Event eventToExecute) : Tuple<bool, string>
PutArchiveOrder(Order order) : Tuple<bool, string>
PutDeleteOrder(Order order) : Tuple<bool, string>

Figure 15: Services folder overview

Converters folder
The "Converters" folder contains classes that are able to convert from one data type in the ViewModel to another more suitable for the View. This is useful when creating data bindings from a View to a ViewModel. The converters are used to decouple the View and ViewModel, as the ViewModel does not need necessarily to optimize the data types for the View. This is also done to avoid code duplications when changing from one data type to another on similar properties.

An overview of the "Converters" folder can be found in *figure 16* on the following page.

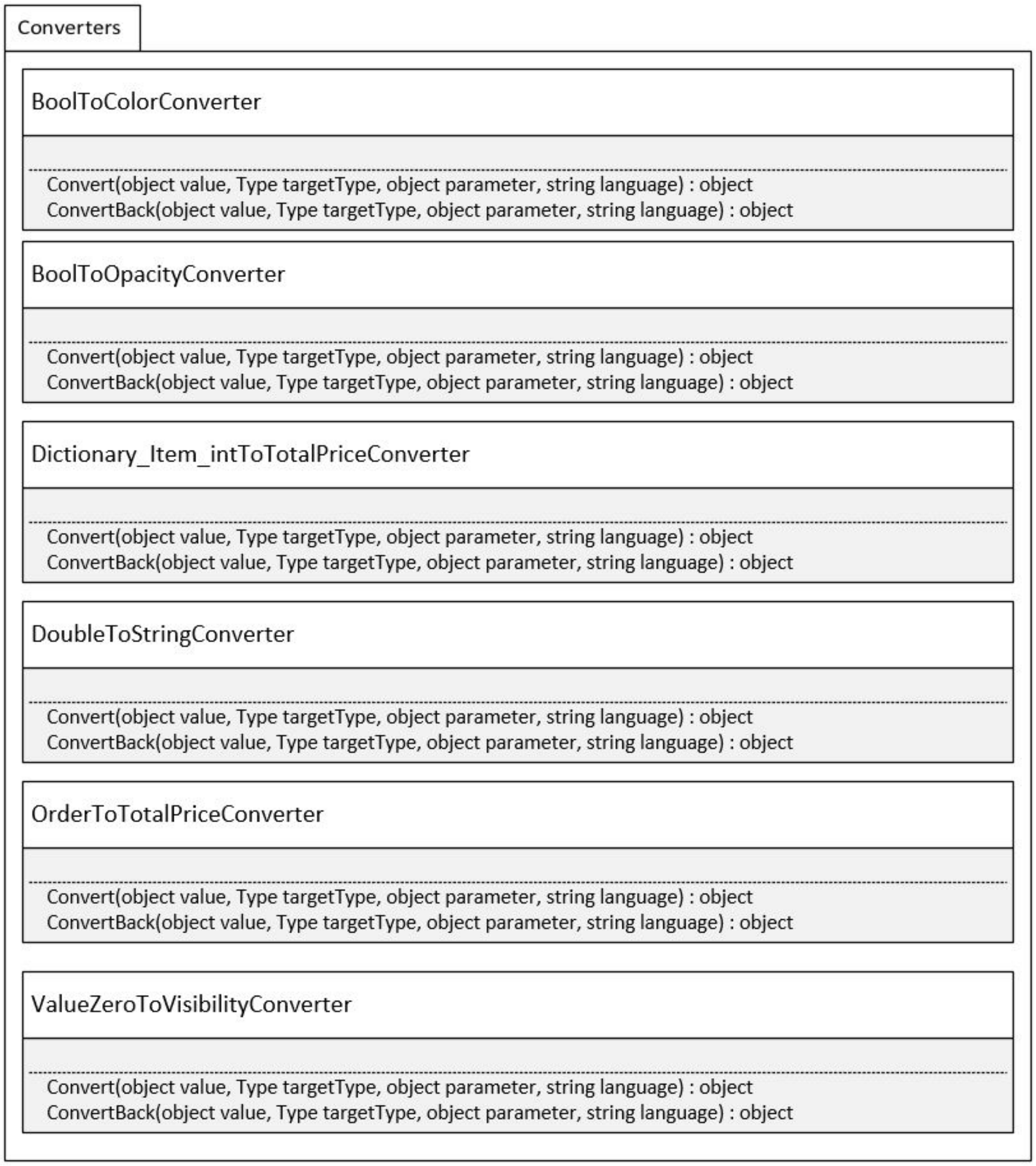

Converters
BoolToColorConverter
Convert(object value, Type targetType, object parameter, string language) : object
ConvertBack(object value, Type targetType, object parameter, string language) : object
BoolToOpacityConverter
Convert(object value, Type targetType, object parameter, string language) : object
ConvertBack(object value, Type targetType, object parameter, string language) : object
Dictionary_Item_intToTotalPriceConverter
Convert(object value, Type targetType, object parameter, string language) : object
ConvertBack(object value, Type targetType, object parameter, string language) : object
DoubleToStringConverter
Convert(object value, Type targetType, object parameter, string language) : object
ConvertBack(object value, Type targetType, object parameter, string language) : object
OrderToTotalPriceConverter
Convert(object value, Type targetType, object parameter, string language) : object
ConvertBack(object value, Type targetType, object parameter, string language) : object
ValueZeroToVisibilityConverter
Convert(object value, Type targetType, object parameter, string language) : object
ConvertBack(object value, Type targetType, object parameter, string language) : object

Figure 16: Converters folder overview

14.3 Access control and security

We have chosen not to implement access control in our project, nor have we focused on implementing security. This is because implementing these will not benefit us in our study of DCRGraphs. Since our project is just a prototype, there is no need for either access control or security.

What we do have, are different roles that the user can choose to be. These are described in the following section.

14.3.1 Roles

To make a flexible and scalable access scheme in the DROM system client, we have designed it with only one user, which can have one or more view options. The four different view options are:

1. Waiter
2. Chef
3. Delivery
4. Manager

The different view options corresponds with the actors described in section *11: Functional requirements.*

This means that a user has access to everything. The intention is that a number of physical client devices (tablets, phones or PCs running the DROM Client application) are set up in relevant positions, such as the kitchen and the reception. Portable solutions for the waiter and deliverer are also possible.

This will then work by having each client device set to the view option that suits its position. For example it would be practical to have the client device in the kitchen set to the "chef" view option, which would then have that client device showing orders which are relevant for the chef, such as orders waiting to be cooked.

The DROM system client supports that the user is able to choose the view options optimal for his or hers role. The user is able to enable multiple different view options at the same time, or quickly switching between different view options.

This makes it possible for a small restaurant, where the employee, who is the waiter and the chef of the restaurant, to manipulate orders both relevant for the waiter and chef roles, by enabling both of the different view options "Waiter" and "Chef".

This also makes it possible in a large restaurant for an employee, such as the chef, to quickly switch to the waiter view option, in case the waiter is busy and there are no more orders to be cooked.

This allows the chef to accept orders received from the website, so that the chef can quickly get new orders to cook.

In short, the view options are there to make it easy to access the relevant information for each employee. At the same time, it also allows the restaurant to utilize only a few client devices without penalties, due to the easy switching and mixing of the view options, or use many client devices to set up different independent view options.

The role based design does presume that employees are responsible enough, not to switch to another view option and use the functions which they are not supposed to use.

14.4 Global software control

14.4.1 Concurrency

To maintain concurrency in the database, we have decided on a simple lock based approach. The DCRGraph table in the database has three lock related properties. One property which is the lock itself, one for a lock id and one for the time it was locked.

The lock id is for identifying whether it is your own lock, or someone else has locked the DCRGraph. The idea is that whenever a request to the API reaches code that will change data in the database, a request to lock the DCRGraph is made. If the DCRGraph is already locked by someone else the request fails , otherwise the request locks the DCRGraph with a unique id and proceeds until database changes are done. At this point the DCRGraph is unlocked and free for the next data changing request to go through.

The time property is a security measure against locks that get left behind by critical system failures, such as power outages. When lock request are made, it is checked whether the lock is old. If the lock turns out to be old, then it is ignored and overwritten with a new lock. Old means more than a minute has passed since the lock was created.

15 User interface

The DROM Client application has four different screens. A 'Login screen', an 'Order overview screen', a 'Create order screen', and an 'Edit order screen'.

These will be shown and the design thoughts behind them will be explained.

Login screen

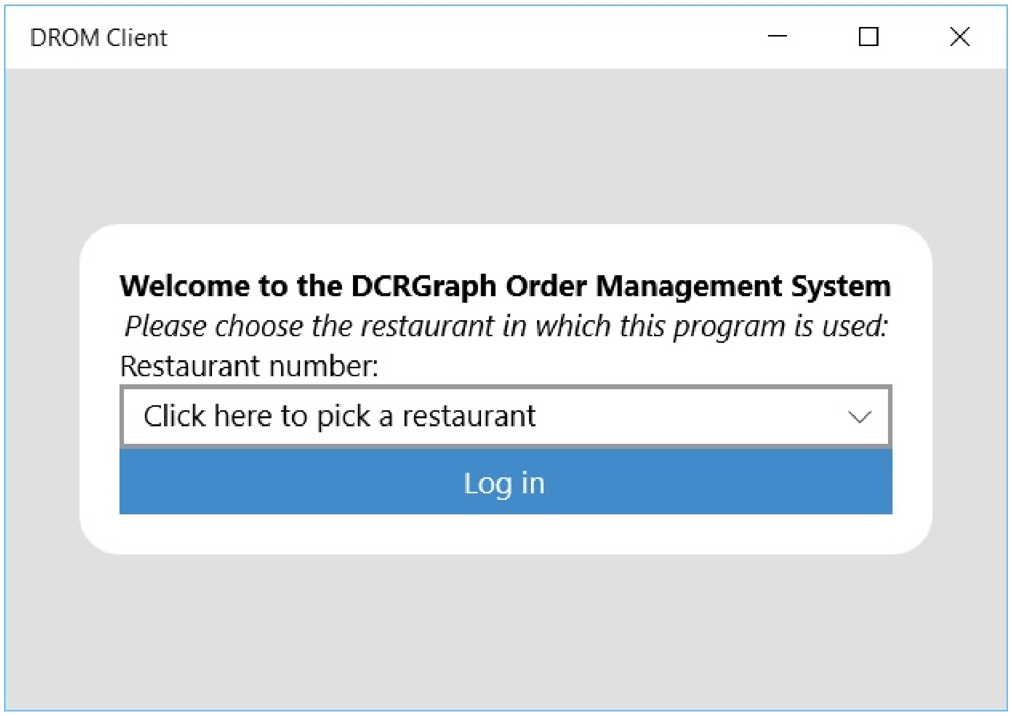

DROM Client
Welcome to the DCRGraph Order Management System
Please choose the restaurant in which this program is used:
Restaurant number:
Click here to pick a restaurant
Log in

Figure 17: Screenshot of the Login screen

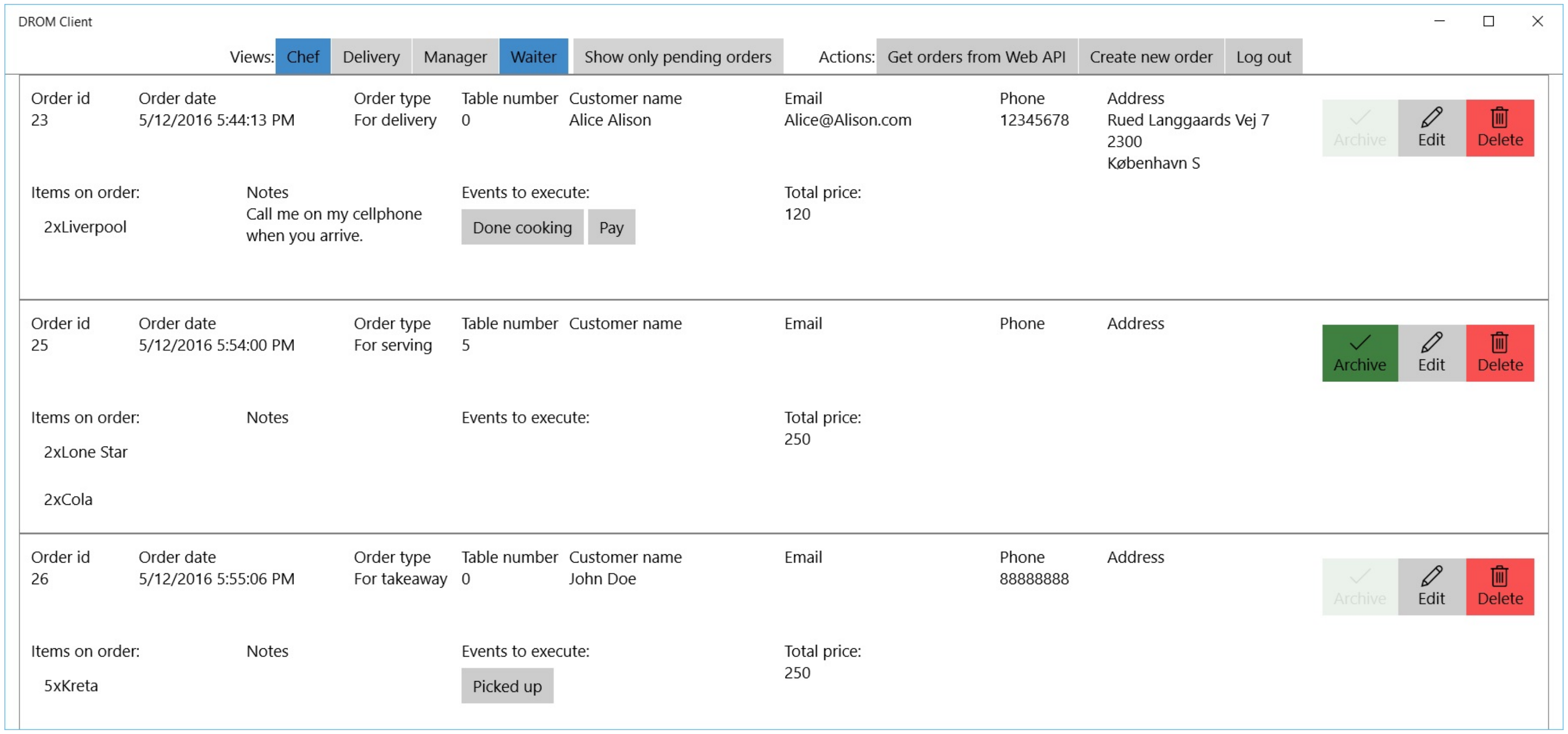
In figure 17 the Login screen of the DROM Client is shown. This is the first screen the user is presented with when running the DROM Client application.

We thought of a simple Login screen as seen on most web sites such as Facebook.

The ability to choose a restaurant was implemented as a subsequent change to the system, as described in section *17: Introducing more than one restaurant*. We choose to implement this as a Combobox, as it prevents the user from choosing a restaurant that does not exist in the system.

User authentication was not implemented in the DROM system, hence no password is required to log in.

Order overview screen


DROM Client

Views: Chef | Delivery | Manager | Waiter | Show only pending orders

Actions: Get orders from Web API | Create new order | Log out

Order id	Order date	Order type	Table number	Customer name	Email	Phone	Address
23	5/12/2016 5:44:13 PM	For delivery	0	Alice Alison	Alice@Alison.com	12345678	Rued Langgaards Vej 7 2300 København S

Items on order:	Notes	Events to execute:	Total price:
2xLiverpool	Call me on my cellphone when you arrive.	Done cooking, Pay	120

Order id	Order date	Order type	Table number	Customer name	Email	Phone	Address
25	5/12/2016 5:54:00 PM	For serving	5				

Items on order:	Notes	Events to execute:	Total price:
2xLone Star 2xCola			250

Order id	Order date	Order type	Table number	Customer name	Email	Phone	Address
26	5/12/2016 5:55:06 PM	For takeaway	0	John Doe		88888888	

Items on order:	Notes	Events to execute:	Total price:
5xKreta		Picked up	250

Figure 18: Screenshot of the Order overview screen

In *figure 18* the Order overview screen of the DROM Client is shown. This is the screen the user will be arriving at, after selecting a restaurant in the login screen.

The screen serves as an order overview page, from where the user may view and manipulate orders.

In the top of the screen, the user is able to choose different kinds of views, and do different actions such as creating a new order. Under the top bar orders are represented in rows.

In the top of the screen, the user may choose which employees in the restaurant will be using the screen. In *figure 18* the views: "Chef" and "Waiter" is toggled on and is marked with a blue color. The orders will only show events that can be executed by these actors.

Views which are toggled off are shown with a grey color. "Delivery" and "Manager" are toggled off in *figure 18.* A user may also choose to to toggle the "Show only pending orders", which will hide orders in which no events can be executed, by the currently selected actors. For example, if we were to click the "Show only pending orders" button in *figure 18*, the second order on the screen, the one with order id 25, would be hidden, as no events can be executed by the actors.

The screen can be changed to only show the orders relevant for the employees using it. This means that the DROM Client can be used on the same physical monitor by multiple different employees, or that a single user may have multiple roles in a restaurant. This also ensures that all actors are able to do all actions required of them.

The button "Get orders from Web API", as seen under Actions, is for manually fetching orders from the Web API. The DROM Client will also fetch orders automatically when navigating to this screen, or when an action on an order is performed by the user.

The "Create order" button navigates the user to the 'Create order screen', as we will describe in section *15: Create order screen.*

The "Log out" button navigates the user back to the 'Login screen' which is previously described in section *15: Login screen.*

Under the top bar, the orders are shown in rows. All order data is shown on each order. An expand and minimize function was not implemented due to time constraints. The user is able to view all the data available, including order information and customer information.

On the right side of an order, three buttons are shown;

A green button for "Archiving" an order. This button will turn green when the order is considered done (when its DCRGraph is in an accepting state). If an order is archived, it is not shown in the DROM Client. The DROM system API does not send archived orders to the DROM Client. If the order cannot be archived yet, the button will appear opaque as seen on the first and third order in *figure 18.*

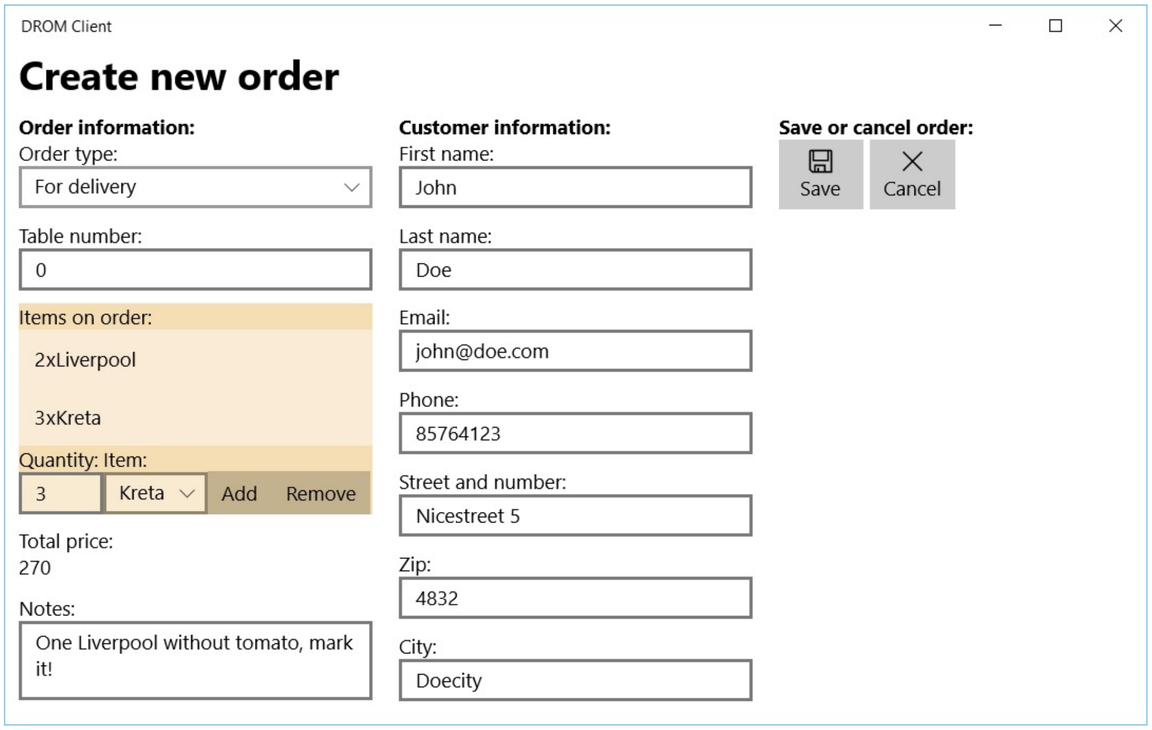
The "Edit" button will lead the user to the 'Edit order screen', as described in section *15: Edit order screen.*

The ”Delete” button will delete the order, if the user answers ”Yes” to a warning message. The order is actually not deleted from the database, but is only archived, and will not appear in the DROM Client after that.

The ”Events to execute” are DCRGraph events in the group ”only pending”, which are only showed on this screen when they are pending and if the roles of the events are the same as the selected views in the top of the screen.

We thought of implementing a feature so that the user could not ”Create new order” if the DROM Client was not used by a Manager or Waiter, however we thought it would only cause frustration and pain if the user was restricted in doing some functions on different roles. We kept the views to different roles that could execute different events, but do every action that other roles could do.

Create order screen


DROM Client

Create new order

Order information:

Order type: For delivery

Table number: 0

Items on order:

2xLiverpool

3xKreta

Quantity: 3 Item: Kreta

Total price: 270

Notes: One Liverpool without tomato, mark it!

Customer information:

First name: John

Last name: Doe

Email: john@doe.com

Phone: 85764123

Street and number: Nicestreet 5

Zip: 4832

City: Doecity

Save or cancel order:

Figure 19: Screenshot of the Create order screen

In *figure 19* the Create order screen is shown.

On this screen the user will fill in order information and customer information, and then create the order.

The information is divided into three columns, one named ”Order information”, one named ”Customer information”, and one named ”Save or cancel order”.

When we designed it, we thought of the first thing a user would do, which is entering the Order type, the items on the order and the notes. Next the user would enter the delivery address or the table number. Lastly the user would decide whether the order creation should be saved or cancelled.

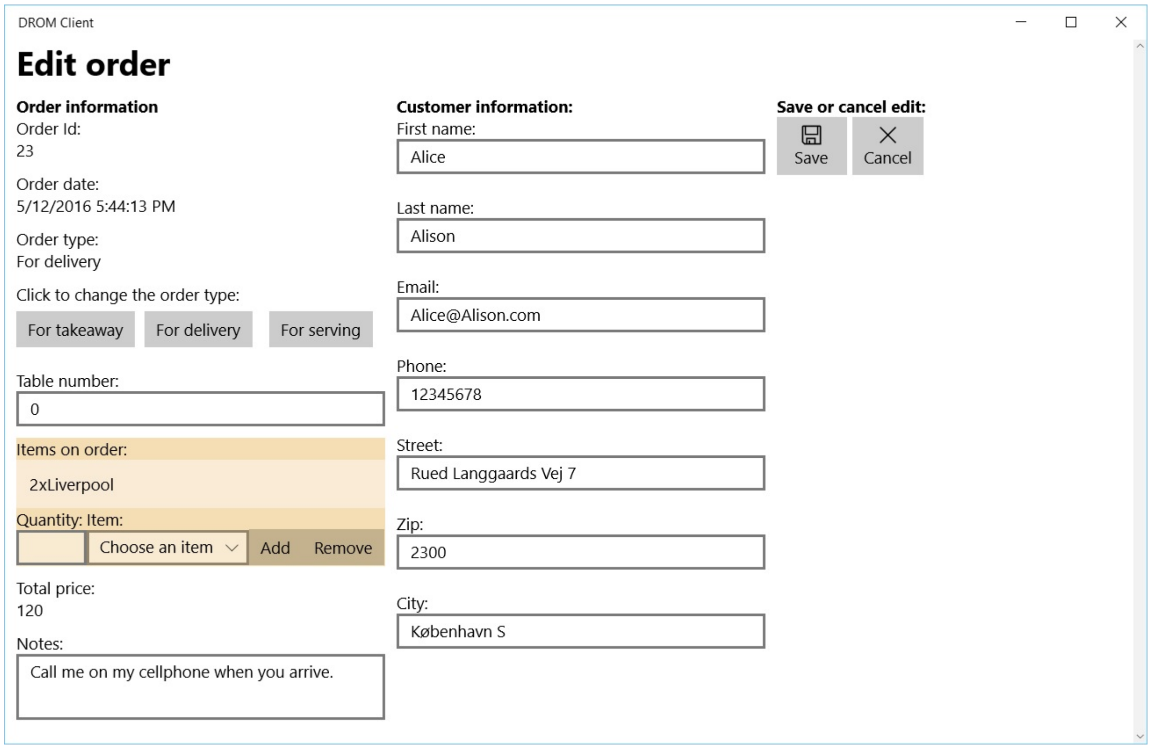
When the user clicks Save, a pop-up asks if the user is sure they want to save the order. If the user clicks yes, the order is saved and the screen navigates the user back to the Order overview screen. If the user clicks Cancel, it will lead the user back to the Order overview screen without creating the order.

We also thought of hiding the customer information column, when the Order type was ”for serving”, because the user is not required to enter the customer information with this order type, only the table number is required with this order type.

We also thought of hiding the table number, when the Order type was ”for delivery”, as it is unlikely the user would use a table number on the order type.

However we choose to let these input fields stay so the user may choose to use them if they wanted to.

Edit order screen


DROM Client

Edit order

Order information
Order Id:
23

Order date:
5/12/2016 5:44:13 PM

Order type:
For delivery

Click to change the order type:

Table number:
0

Items on order:
2xLiverpool

Quantity: Item:

Total price:
120

Notes:
Call me on my cellphone when you arrive.

Customer information:
First name:
Alice

Last name:
Alison

Email:
Alice@Alison.com

Phone:
12345678

Street:
Rued Langgaards Vej 7

Zip:
2300

City:
København S

Save or cancel edit:

Figure 20: Screenshot of the Edit order screen

In *figure 20* the Edit order screen is shown.

The Edit order screen looks similar to the Create order screen, however it contains a bit more information.

The Order Id and Order date is shown on this screen, however these cannot be changed by the user.

The Order type can be changed by clicking one of the three buttons ”For takeaway”, ”For delivery” or ”For serving”. It is not possible to change from the order type ”Bulk order” to any one of the former three, or likewise from the former three to ”Bulk order”. ”Bulk order” was a new order type implemented as a subsequent change described in section: *16: Introducing bulk orders.* This is a restriction in the DCRGraph, which is therefore reflected in the client.

The changed Order type will be saved in the system if the user choose to save the edited order. The buttons for changing the delivery type are events of the group "Edit events" in our DCRGraph, and are only showed on this screen to avoid cluttering the events that can be executed on the main order screen.

Changing an order from any Order type to another will still have all Customer information and Table number information, to make sure no information is lost. Changed order or customer information will be updated if the user chooses to save the edited order.

Adding or removing Items on an order will execute the event "Items have changed" in the DCR-Graph, when the edited order is saved.

We thought about hiding the Customer information when the Order type was "For serving", and hiding Table number when the Order type was any of the other three. However we did not implement this, as information could be lost, or the user could be unable to be edit this information, which is not preferable.

On the design in general

The user interface is designed so that it is "Touch friendly". This means that buttons appear large enough for the user to press them with the index finger or thumb on a touch enabled device. This is done so that the DROM Client is also optimized for mobile devices, such as a tablet or a smart phone.

Icons and text is used to guide the user, so that both a beginner and an expert is able to navigate the DROM Client with ease.

Error messages are shown when user input is not entered correctly, or when the DROM Client fails to send or receive data from the DROM system API.

Dialog messages are shown when the user is doing something that will make a large impact on the order, such as deleting it. The user has to confirm whether the action was intended. For example when an order is deleted, a dialog message shows up asking if the user is sure they want to delete the order. This is done so that the user is able to cancel a possible miss-click, and does not accidentally delete an order.

Every screen is designed so that it can scale to the width of the screen. This is also known as Responsive Design [18]. One can test this by dragging one vertical side of the window frame on a desktop computer. When the screen gets too narrow to fit all the visual elements in a row, the last or a group of visual elements will jump down to the next row. Screenshots taken, with the narrowest window frame setting, is shown in appendix section *26.2: Responsive UI screenshots.*

Part IV

Subsequent changes to the system

In this part we will study the eases and difficulties of implementing changes into the system, built on DCRGraphs.

We will see what parts of the system, if any, needs to be changed when updating the DCRGraph. We will also study if the DCRGraph needs to be changed when updating the system.

For the discussion of eases and difficulties we had implementing subsequent changes we refer to the section *22: Discussion on the development process of implementing subsequent changes.*

16 Introducing bulk orders

As our first change to the system, we decided to add a new delivery type to the DCRGraph. We decided on adding a catering delivery option, also referred to as bulk orders.

This change to the system represents the work that would be necessary for the developer, if a restaurant using our system, decided it wanted to support a new delivery type, in this case bulk orders.

16.1 Changes made to the graph

We already have three delivery types; serving (when people want to eat at a table inside the restaurant), takeaway and delivery. Want we want to achieve, is to add a new delivery type: "bulk orders". We do this by updating our DCRGraph with a series of new events. These new series of events represents the steps that employees will have to go through when receiving, cooking and delivering a bulk order.

To accommodate this change, we arrived at a relatively simple solution with only four new events, seen in *figure 21.*

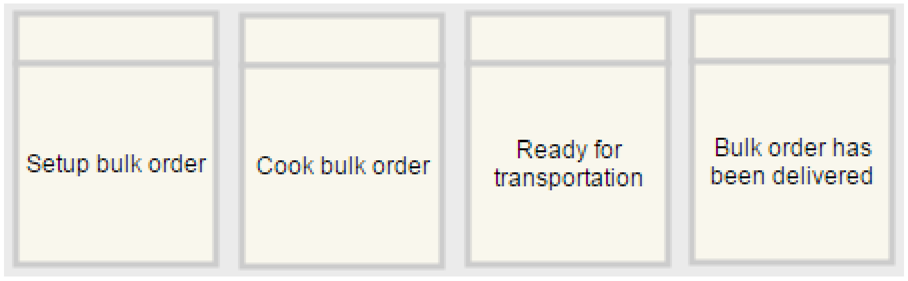


Figure 21: New events to enable bulk orders

The five steps we did to update the DCRGraph are as follows:

1. First, the "Setup bulk order" event is added. An event to set up the graph, that is executed on graph creation, and is not seen by the user.
2. Next, the "Begin cooking bulk order" event is added. This event represents that one or more chefs have begun cooking the food.
3. In the third step, we reuse the "Done cooking" event, which represents that the food of the order is now cooked and is ready for further processing.
4. "Ready for transportation" is the next step, which represents that the order has been prepared for transportation, this means that the food on the order has been loaded onto a catering van for delivery.
5. The final step is "Bulk order has been delivered". This represents that order has been delivered to the customer.

We decided that it would not make sense to allow switching delivery type on a bulk order, and as such, the event "Setup bulk order" excludes the events to switch delivery type.

In addition, we have put a milestone from "Pay" to "Cook bulk order", because with orders of this scale, we want to receive payment before we start cooking the food.

The resulting DCRGraph can be seen in *figure 22*. To aid readability, we have hidden exclude relations between setup events (the events in the top). We discuss the issues related to the readability of the DCRGraph in section *22.1.2: The negative.*

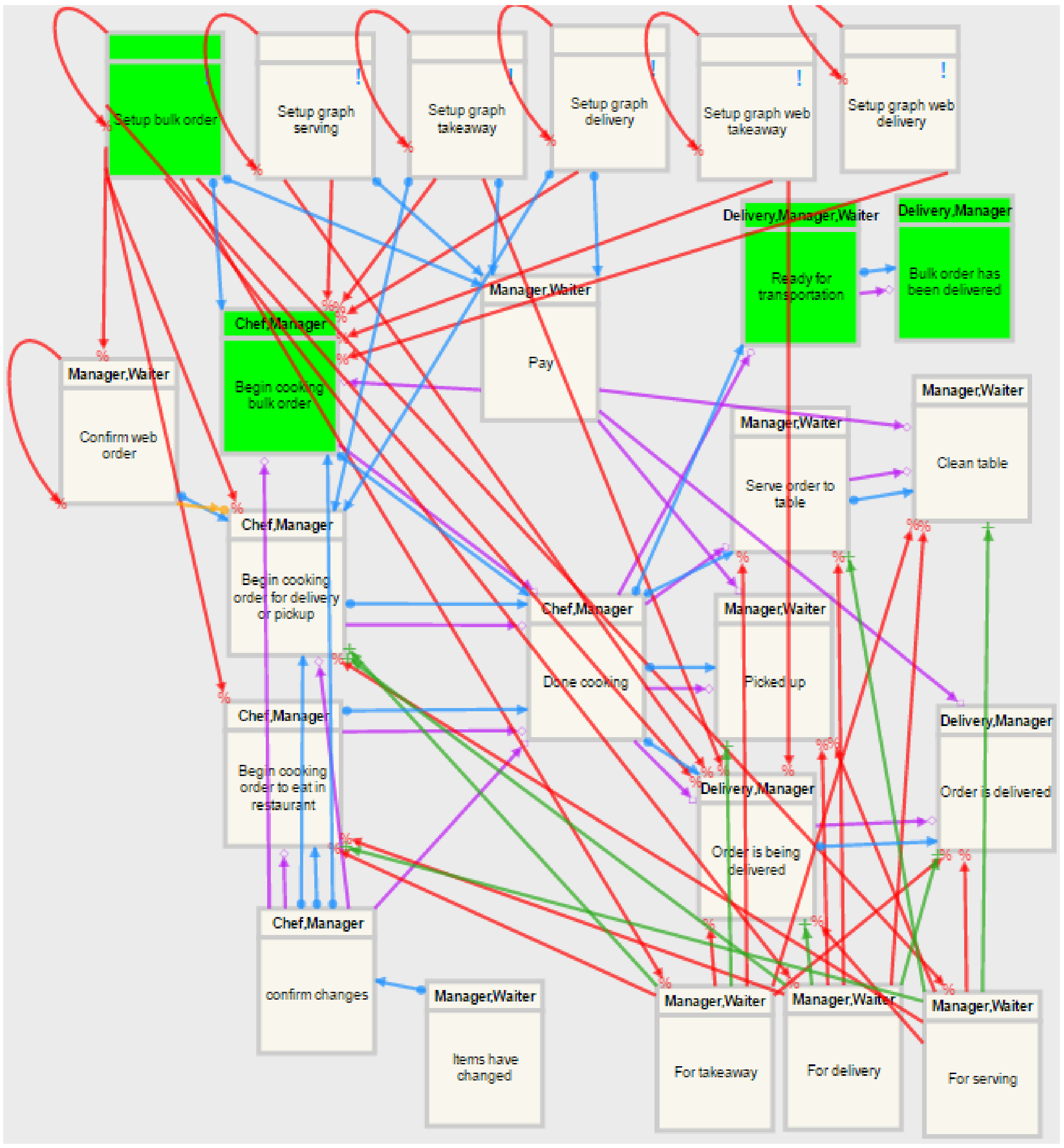


Figure 22: DCRGraph with bulk orders - new events in green

16.2 Changes made to the system

By adding bulk orders to the DCRGraph, we have touched upon two things that demands changes in the code.

New delivery type

First, we have added a new delivery type. Groups, Roles and Delivery types need to be added manually to the database, we are doing it with seed data when a new database is created. This was a very easy change and only required a very small change, as seen in *Listing 1: New code to support bulk orders in the seed method*. The code is simply creating an additional delivery type class, when seed data is generated for the database.

```
new DeliveryType()
{
    OrderType = 0,
    Type = "Bulk order"
}
```

Listing 1: New code to support bulk orders in the seed method

This change could be made unnecessary by changing the code that parses DCRGraph xml to dynamically add new Groups, Roles and Delivery types when they are discovered in the parsing process.

In our case, because the system is not deployed, and as such does not contain data that needs to be kept, it was fine to create a new delivery type by adding new seed data. If the database contained important data, this would instead be done by hitting the database with an SQL query, to create a new row in the "DeliveryTypes" table.

New setup event

Second, we have added a new setup event. Setup events are events which are run automatically by the system when orders are created, and as such they are hard-coded to be run. We have not found any better way to do this, and this means that to accommodate the new setup events, it is necessary to update the code with the new event. This was easy and only required a small change, as seen in *Listing 2: New code to support bulk orders in the mapper class.* Most of the code is error handling, and is exactly the same as for other setup events. As such it was simple to copy this and change the strings in line 1 and 3, of the code snippet, to the bulk order names.

```
case "Bulk order":
    dcrEvent = order.DCRGraph.DCREvents.FirstOrDefault(e =>
        e.Label.Contains("Setup bulk order"));
    if (dcrEvent != null)
    {
    await new DbInteractions().ExecuteEvent(
            dcrEvent.Id);
        break;
    }
    return new Tuple<string, HttpStatusCode>(
        "The DCRGraph does not contain the relvant setup event",
            HttpStatusCode.InternalServerError);
```

Listing 2: New code to support bulk orders in the mapper class

These changes were all in the API code, no changes were necessary in client code.

16.3 Conclusion

Once the new graph was put into the system, and the necessary changes were made, it was easy to test the new functionality by running the DROM system client and creating the new order type. In our first attempt, we discovered some missing relations in the new addition to the DCRGraph, but after adding them, everything worked as expected. In total, excluding the time spent designing the new DCRGraph, it took around 5-10 minutes to update the system to work with the new DCRGraph. We discuss this in section *22.1.1: The positive*.

17 Introducing more than one restaurant

Let's say that the DROM system has been used for a year, and the restaurant manager is opening a new restaurant in addition to the first. He wants to use the same system they are familiar with in the second restaurant and keeping the business processes unchanged.

How can we change the system, such that the DROM Client can be used in multiple restaurants, while still keeping the business processes of the second restaurant unchanged.

17.1 Changes to API

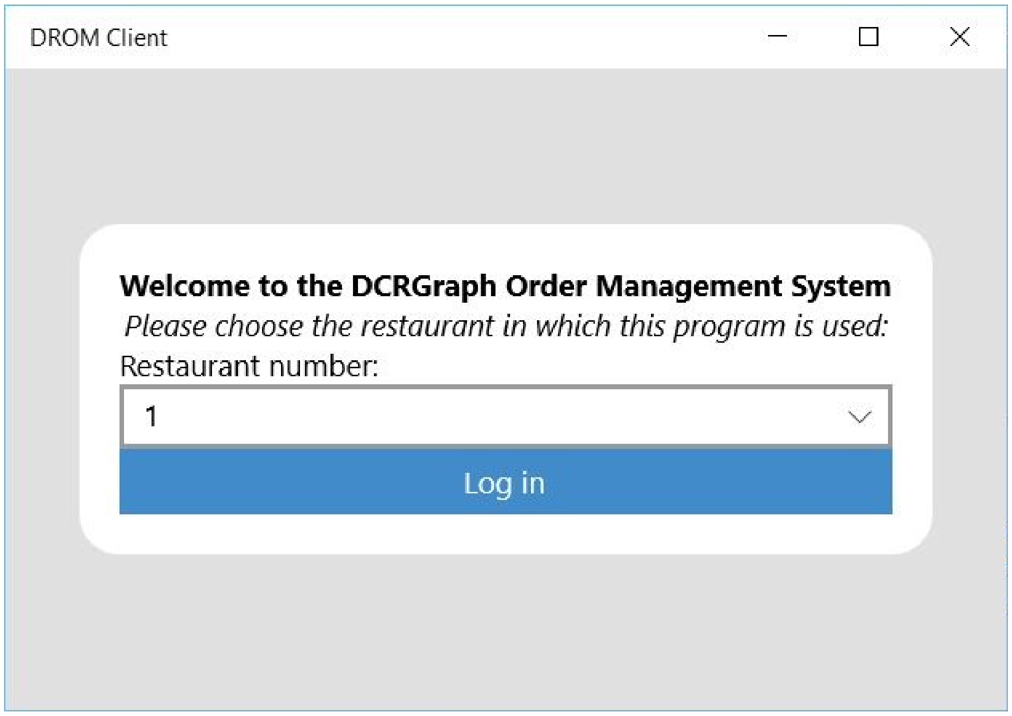
The changes necessary in the API were pretty simple and fast to make. First of we added a new field in the Entity Framework database class for orders, to identify restaurants. We did this with a simple integer.

The controllers were updated, so when creating or getting orders from the Web API, the Web API now expects an integer to identify restaurants with, when orders are requested or created. This is to ensure that changes only occur on the restaurant with the specified Id. As a result, it is now also necessary to include a restaurant id when getting orders or posting new orders to the API.

Other order related calls to the Web API did not need to be updated, as the Order id serves as an identifier when changing or archiving an order.

17.2 Changes to client

In order for the user to pick a restaurant, we had to change the DROM Client UI to make this possible. We added a Combobox menu in the login screen, in which the user may choose a restaurant before proceeding. A screenshot of the changed login screen can be seen in *figure 23*. To choose a restaurant we used integers: "1" and "2", however we could just as well have used strings showing addresses or names of the restaurants. For simplicity, we used "1" and "2". More restaurants can easily be added with more integers.



Welcome to the DCRGraph Order Management System

Please choose the restaurant in which this program is used:

Restaurant number:

1

Figure 23: Added a combobox for choosing a restaurant in log in screen

The API caller class in the DROM Client project was also updated in order to only get orders from the Web API relevant for the restaurant, and to post created orders to the Web API such that they get saved with relation to the restaurant.

The user is now able to choose which restaurant the system should be used with. The orders in the main view, and created orders will automatically be assigned to the restaurant chosen in the log in screen. It is also possible to log out, and choose another restaurant without restarting the application.

17.3 Conclusion

Only few changes was made to the DROM Web API and the DROM Client application to implement these changes. It took us under 2 hours to implement these changes, and the DCRGraph was untouched. It is fair to say that these changes were implemented easily. The developer is able to add non business processes functionality easily without having to change the DCRGraph.

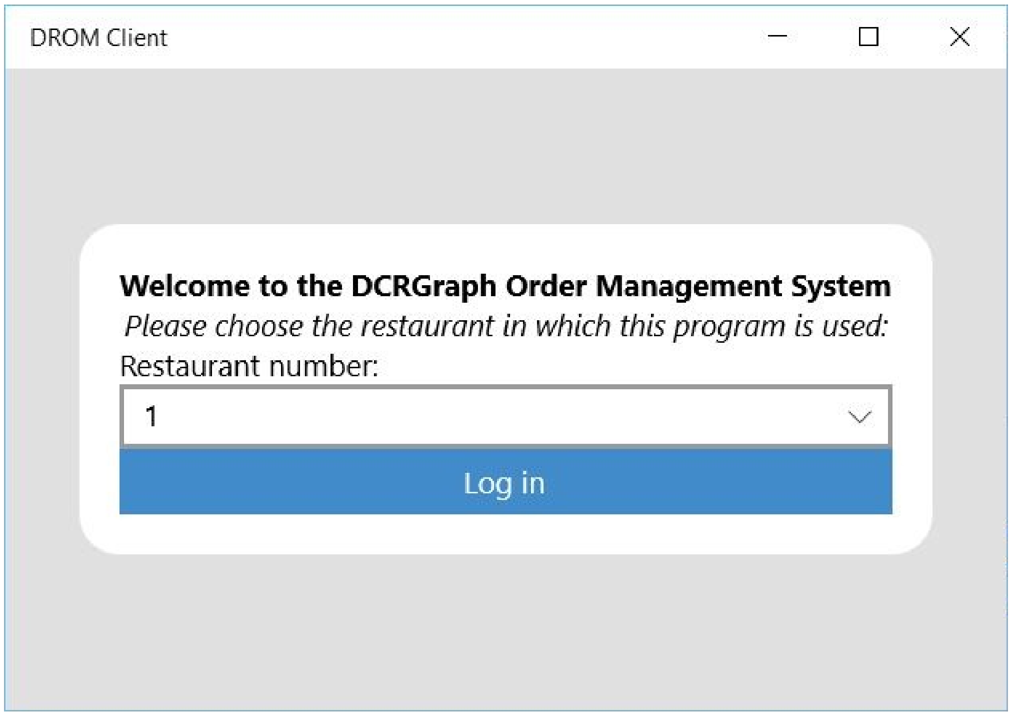
Part V

Final system

18 User manual

In this guide the DROM system client will be referred to simply as the ”client”.

The first thing that is encountered when the client is run, is the restaurant login screen.



Welcome to the DCRGraph Order Management System

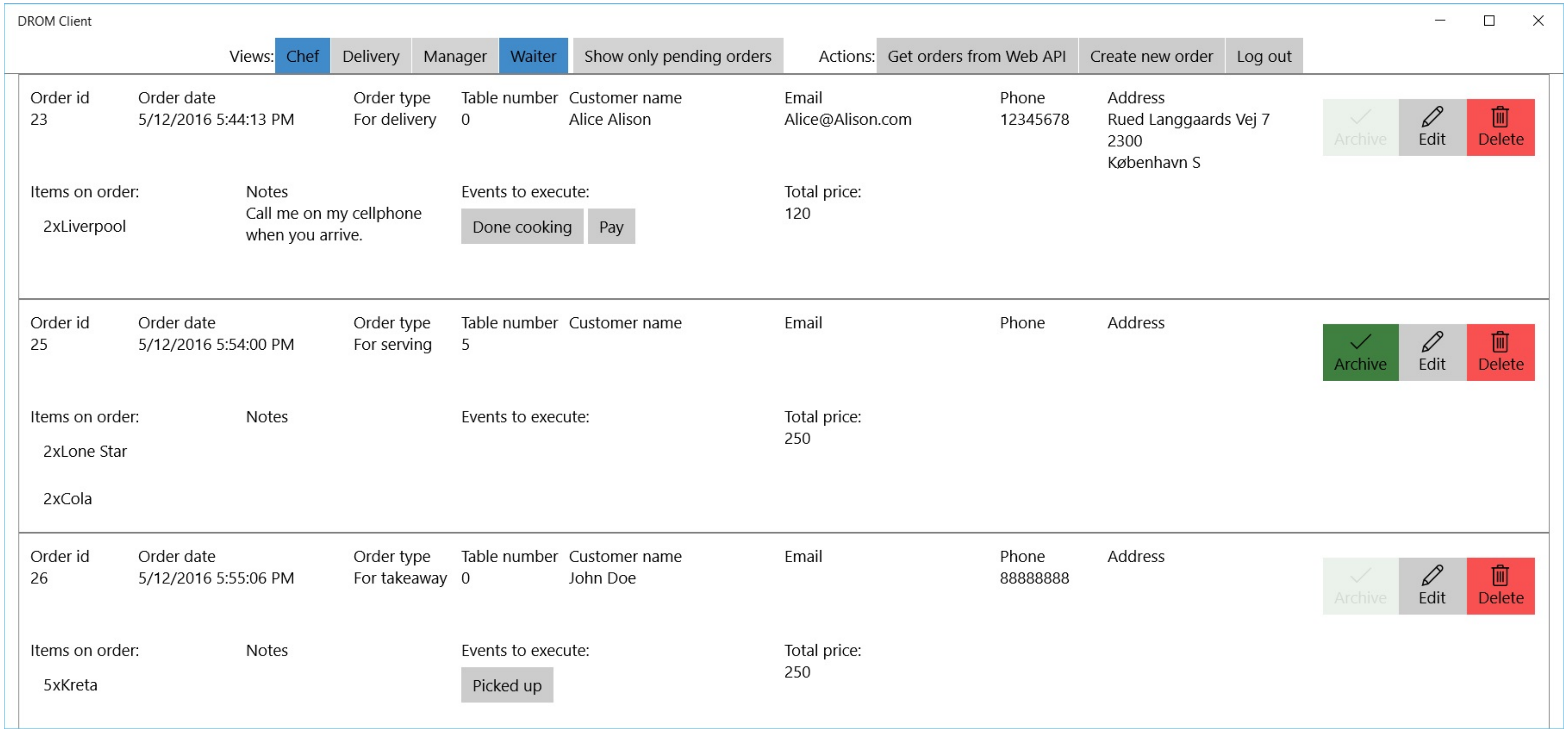
Please choose the restaurant in which this program is used:

Restaurant number: 1

Figure 24: Select restaurant

From this screen you can select a number, representing the restaurant you want to login to. Select a number and press ”Log in”.

This will bring you to the main order overview screen of the client, shown in *figure 25: Main window.*



Order id	Order date	Order type	Table number	Customer name	Email	Phone	Address
23	5/12/2016 5:44:13 PM	For delivery	0	Alice Alison	Alice@Alison.com	12345678	Rued Langgaards Vej 7 2300 København S

Items on order:	Notes	Events to execute:	Total price:
2xLiverpool	Call me on my cellphone when you arrive.	Done cooking, Pay	120

Order id	Order date	Order type	Table number	Customer name	Email	Phone	Address
25	5/12/2016 5:54:00 PM	For serving	5				

Items on order:	Notes	Events to execute:	Total price:
2xLone Star 2xCola			250

Order id	Order date	Order type	Table number	Customer name	Email	Phone	Address
26	5/12/2016 5:55:06 PM	For takeaway	0	John Doe		88888888	

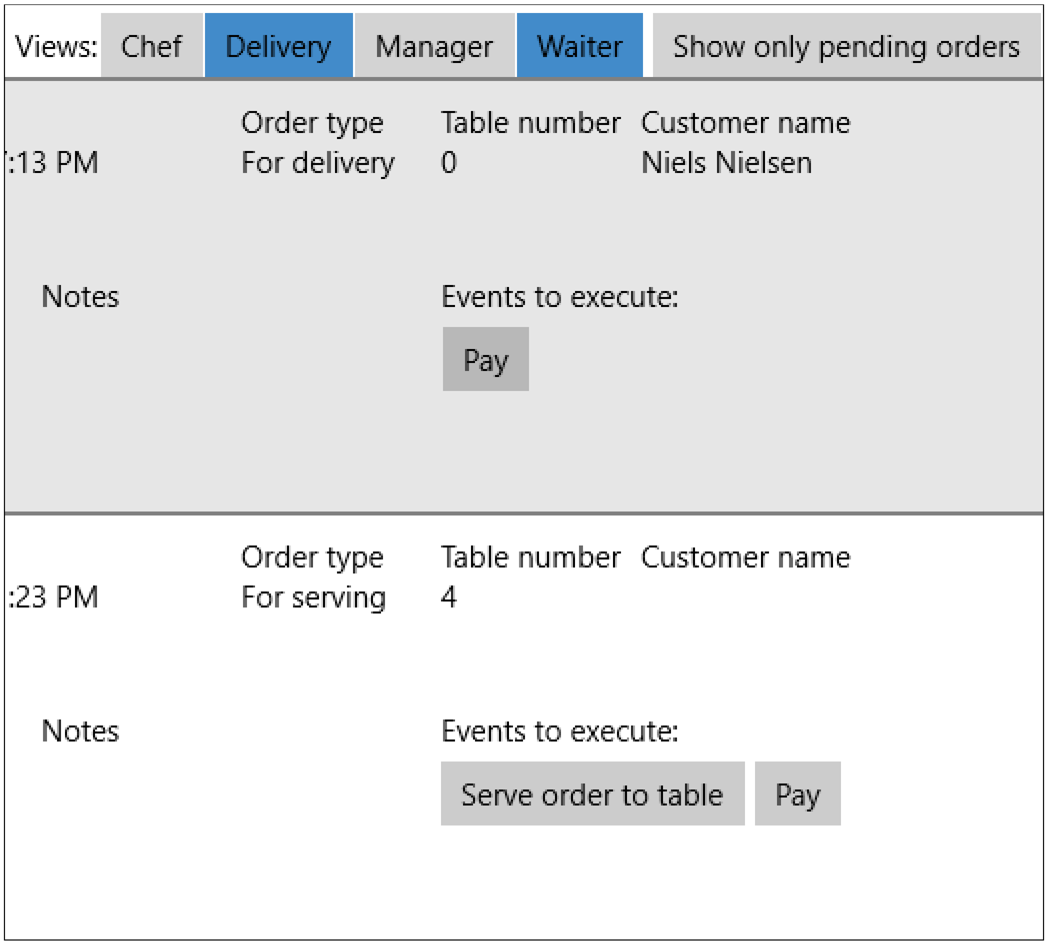
Items on order:	Notes	Events to execute:	Total price:
5xKreta		Picked up	250

Figure 25: Main window

The main order overview screen of the client is where you will spend most of your time. The biggest part of the screen, is a list of all active orders for the restaurant you selected in the previous screen.

In the top of the screen, there are various buttons. The buttons under ”Views” are for filtering the available events on orders. One or more of these views can be toggled on by clicking on it, in which case that type of view is selected. For example you could click on ”Waiter” and ”Delivery” to see all the events that are relevant for the waiter and delivery roles.

Once views are selected, the events will show up as buttons on the currently active orders.



	Order type	Table number	Customer name
:13 PM	For delivery	0	Niels Nielsen

Notes

Events to execute:

	Order type	Table number	Customer name
:23 PM	For serving	4	

Notes

Events to execute:

Figure 26: Waiter and delivery views

Selecting the ”Manager” view will show all events available, no matter which role it belongs to.

The button ”Show only pending orders” will hide orders that do not currently have any events available to the selected views.

In the top of the screen, there are a group of buttons under ”Actions”. These are shown in *figure 27: Actions*.

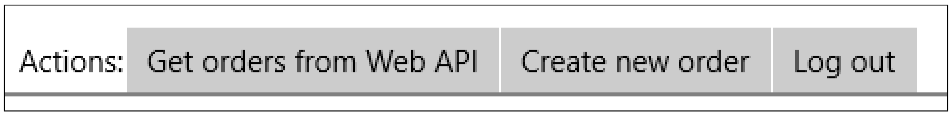


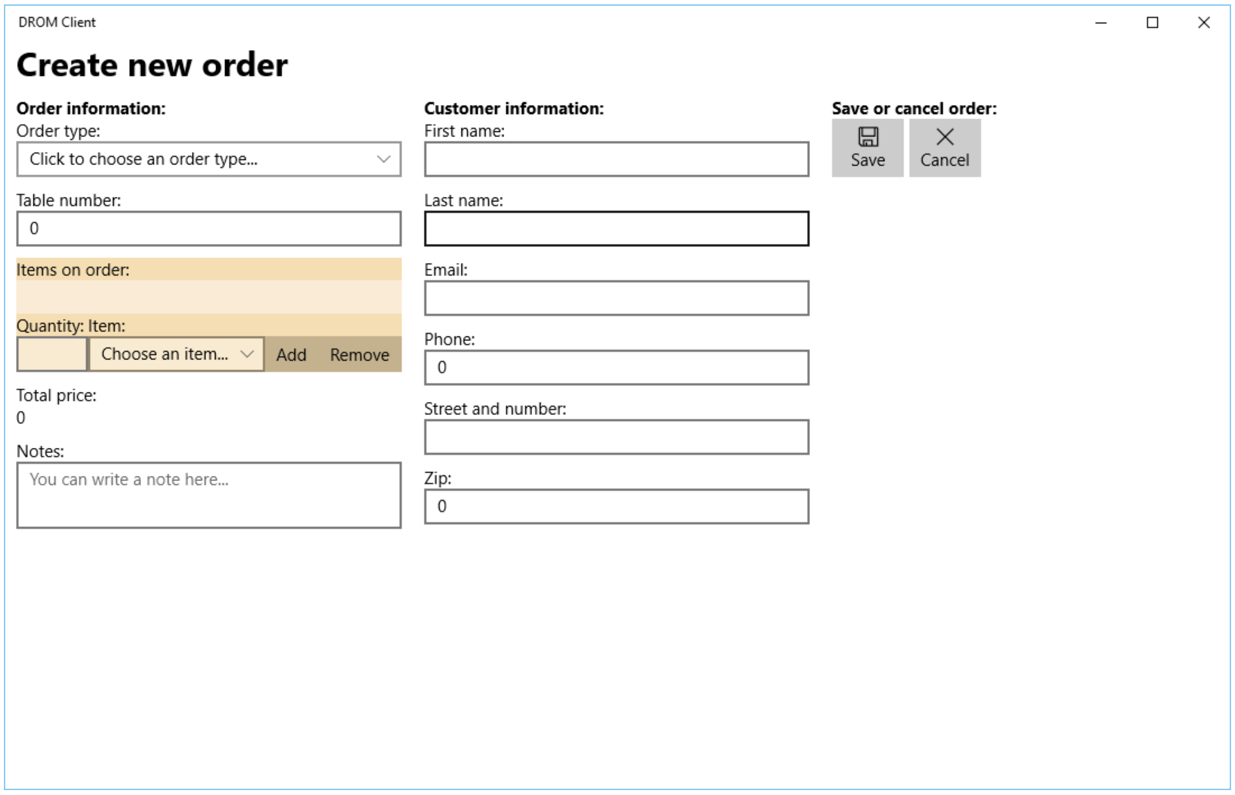
Figure 27: Actions

”Get orders from Web API” is a manual refresh button. By clicking this, the client will fetch orders from the Web API. Note that this also happens automatically when navigating to this screen or when performing an action on an order.

”Log out” will bring you back to the login screen where you can select a restaurant to login to.

”Create new order” is for creating new orders. Pressing this button will bring you to a new window where information about the order can be filled in.

The Create new order screen can be seen in *figure 28: Create new order*.


DROM Client

Create new order

Order information:
Order type: Click to choose an order type...
Table number: 0
Items on order:
Quantity: Item: Choose an item... Add Remove
Total price: 0
Notes: You can write a note here...

Customer information:
First name:
Last name:
Email:
Phone: 0
Street and number:
Zip: 0

Save or cancel order:
Save Cancel

Figure 28: Create new order

The first action to do in this screen is to select an order type. This is the top left field under **Order Information** called "Order type". All orders, except serving, requires that customer information is filled out. Serving orders only require the rows on the left, under **Order Information**. You can still fill out customer information on serving orders, but it is not necessary.

Table number - Serving orders require a table number. For other order types not needing it, it can be left as 0.

Adding items to the order - To add an item to the order, fill in a number in the quantity field, select the item where it says "Choose an item..." and then click "Add".

To remove an item, simply click the item on the list of added items and click "Remove":

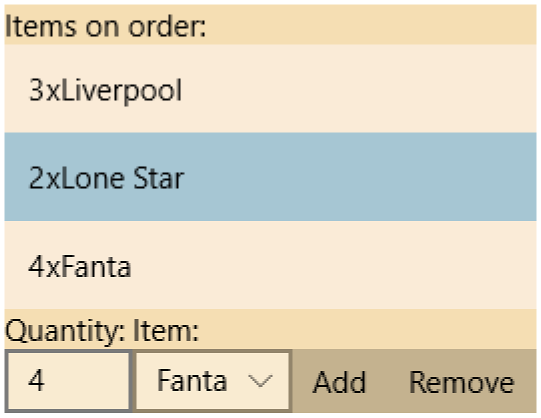


Figure 29: Item selected on added item list

Note than an order must contain items in order to be saved. An error message will show if this is not done correctly.

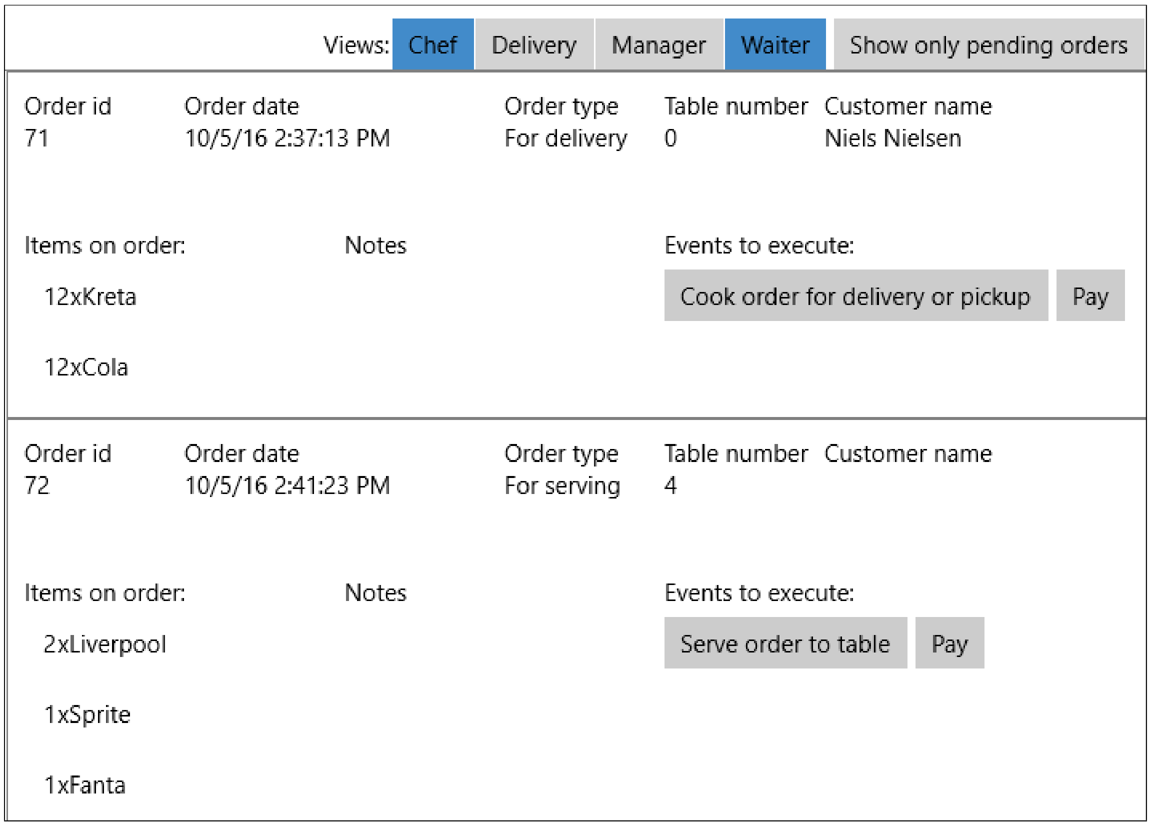
Finishing the order - Once all the information has been filled out, it is time to press "Save". This will save the order and bring you back to the main screen. If at any time during the process you decide that the order should not be created, simply press "Cancel" to return to the main screen without creating the order.

If you have not filled out the information needed by the order type, an error message will show, explaining what you need to fill out in order to be able to save the order.

Working with orders

Now that you know what the buttons do, and how to create orders, it is time to use the system to track what has been done and what needs to be done to orders.

Under "Events to execute" you see what is waiting to be done on the order.



Order id	Order date	Order type	Table number	Customer name
71	10/5/16 2:37:13 PM	For delivery	0	Niels Nielsen

Items on order:

12xKreta

12xCola

Notes

Events to execute:

Order id	Order date	Order type	Table number	Customer name
72	10/5/16 2:41:23 PM	For serving	4	

Items on order:

2xLiverpool

1xSprite

1xFanta

Notes

Events to execute:

Figure 30: Events waiting to be done

Clicking on these will advance the order to the next step, continuously showing what needs to be done to progress. Once the order reaches a point where it is done, the ”Archive order” button will become active.

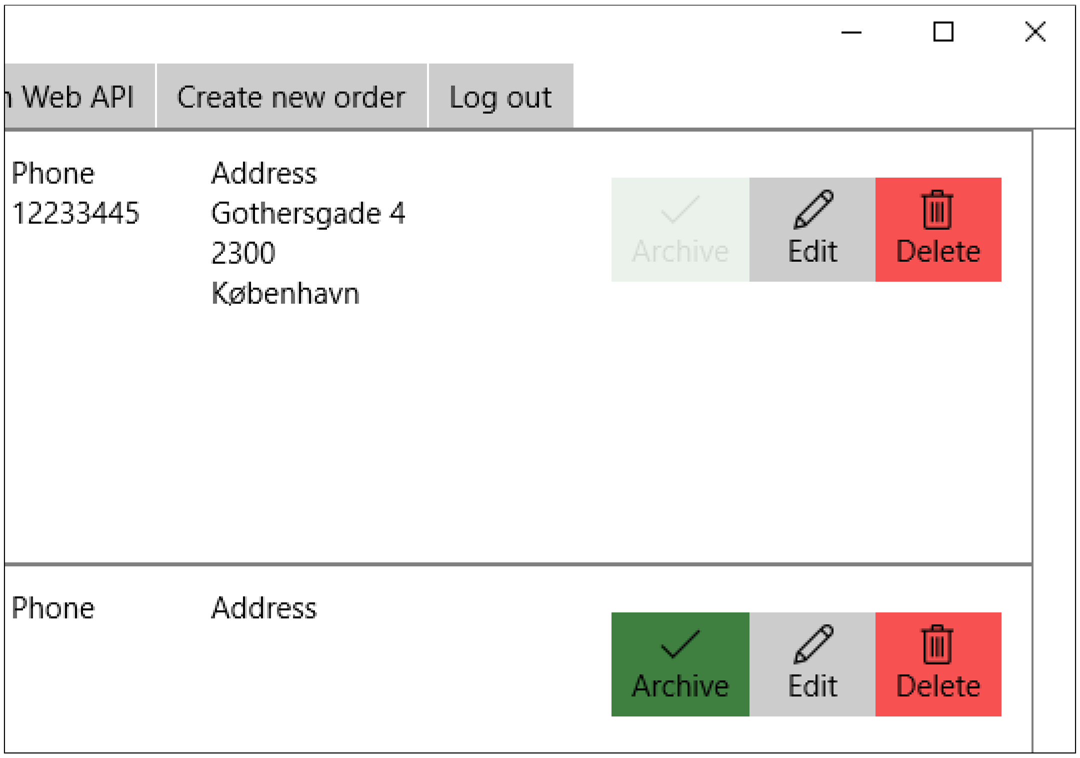


Figure 31: Active archive button

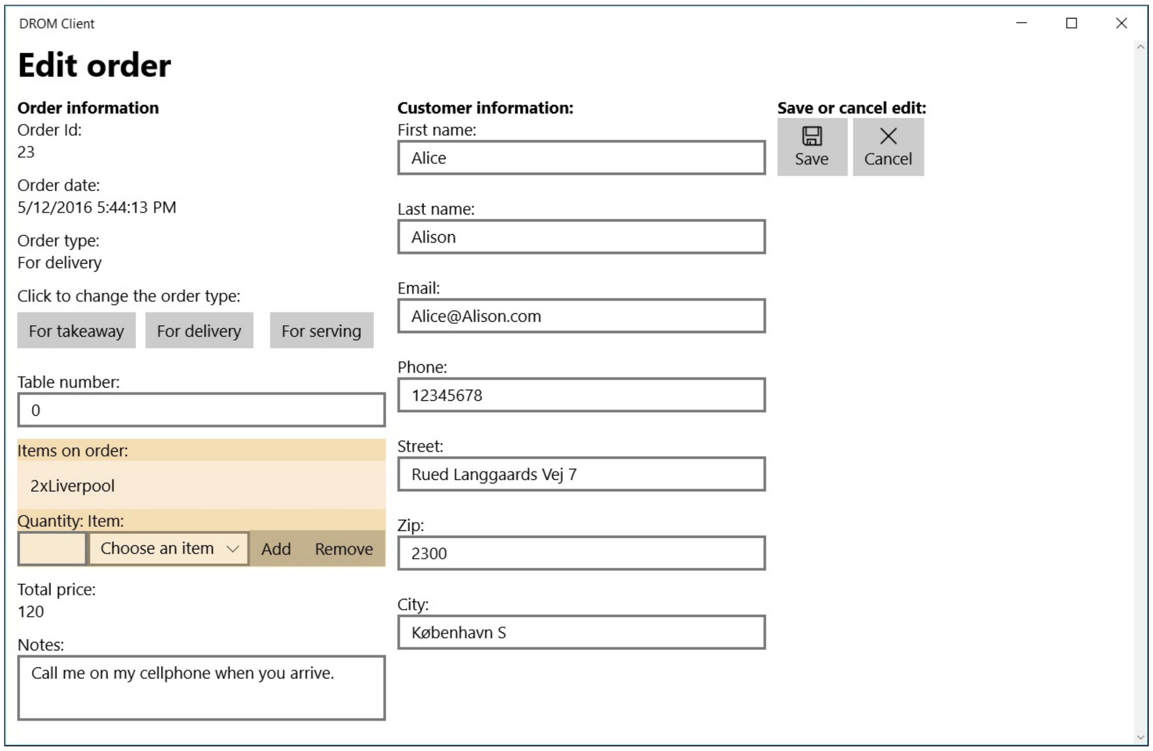
Once this button is active, shown by its green color, it means that the order is done. Clicking on the green ”Archive order” button will remove it from the list of active orders and store it as a completed order.

Edit and delete

At any point, between creating and archiving an order, it is also possible to click the ”Edit” and ”Delete” buttons.

The ”Delete” button will delete the order. The order will disappear from the list of active orders and no further interaction with the order is possible.

”Edit” will bring you to the edit order screen shown in *figure 32: Edit order screen.*


DROM Client

Edit order

Order information

Order Id:
23

Order date:
5/12/2016 5:44:13 PM

Order type:
For delivery

Click to change the order type:

Table number:
0

Items on order:
2xLiverpool

Quantity: Item:

Total price:
120

Notes:
Call me on my cellphone when you arrive.

Customer information:

First name:
Alice

Last name:
Alison

Email:
Alice@Alison.com

Phone:
12345678

Street:
Rued Langgaards Vej 7

Zip:
2300

City:
København S

Save or cancel edit:

Figure 32: Edit order screen

From here you can change the the order type by clicking one of the three buttons ”For takeaway”, ”For delivery” or ”For serving”.

You can also change the table number, items on order, the notes of an order or the customer information.

When you are done making your changes, press ”Save” to save the order. Press ”Cancel” to cancel the edit.

19 Test strategy

We did not have enough time to make automated tests throughout the project. Instead we tested the system manually whenever new functionality was developed.

This was mainly done by running the DROM system API and DROM system client, and utilizing the functionality in the client, to see if the results were as expected. In this way we both see whether the logic in the DROM system client was working, as well as whether the DROM system API was handling requests and database interactions correctly.

When something was not as expected, we used the visual studio debug tool to analyse the data send and received from the Web API. It was then easy to find the component which caused the mistake.

Part VI

Discussion, conclusion and reflection

In this part of the report the development process in relation to DCRGraphs will be discussed and reflected upon. Lastly, we will answer the problem statement.

20 From use cases to DCRGraph - a possible methodology

During the design phase of the DROM system, we decided to replace activity and state diagrams with a DCRGraph. This is described in section *13.3.1: Creating the DCRGraph.*

20.1 Positives

- We found this method of building the DCRGraph to be helpful when trying to make sure everything was taken into account, and as such made the creation of the graph easier. You could say that the use cases provided a form of checklist, for what should be in the graph.
- If you are so lucky that your buyer (assuming you're developing the system on the request of a buyer), understands DCRGraphs, then this also works as a good way to work the design of the business process for the system. The buyer could even make a DCRGraph and present it to you, as a way of describing what is wanted from the system you are to develop.
- Even if you have to show your DCRGraph to someone who doesn't know DCRGraphs, they have a major advantage over many other tools of representation. This is because there are tools to simulate execution paths through DCRGraphs, such as the visualiser on `www.dcr.itu.dk` or the simulator on `www.dcrgraphs.net` (Further details about `www.dcrgraphs.net` and the simulator can be found in Web-Based Modelling and Collaborative Simulation of Declarative Processes [3]).

  This means that people can easily get hands on experience with what the DCRGraph is modelling, simply by trying out what paths are allowed, without having to worry about understanding the notation.
- It also has to be considered, that this saved us all the work it would have been to create activity and state diagrams. The DCRGraph had to be made no matter what, but if it can also serve the same purpose as activity and state diagrams, then this can be a good time saver in the design phase.

  DCRGraph also contain a lot of details, through relations, which means that they can potentially describe things more in-depth than would be achieved with for example state diagrams. The issue here is that this is pretty closely linked with the first negative we describe below, about presentation issues with DCRGraphs *20.2: Negatives.*

- With this approach, we also now have a huge part of the design in one single diagram. Usually, at least in our experience, design is communicated in several diagrams that show different parts of the design.

  With a DCRGraph, we have much more information in a single diagram. While DCRGraphs do have some visualisations issues, as discussed in *22.1.2: The negative* and *22.1.3: Suggestion - new view mode*, it is also a way of representing huge amounts of design information in a single diagram.

  It is still necessary to communicate design not related to the DCRGraph in another way, but a DCRgraph can take the place of many other diagrams, such as activity and state diagrams. This is much easier maintain in the long run, both because it is very fast to update a DCRGraph, but also simply because it is much harder to keep track of what needs to be done when there are more than one diagram to keep updated.

  This also encourages making design changes more. Our experience from previous projects tells us that the mere thought of updating ten or more diagrams, will make you very reluctant to change anything.

20.2 Negatives

- The original purpose of creating activity and state diagrams, is to analyse, showcase and agree on what processes there are, and how they look in the system to be developed. DCRGraphs contain all this information, but the presentation can be somewhat lacking.

  In simple DCRGraphs, information is very clear and easy to read, but when a DCRGraph becomes big, which is often nececary to model complex flows, it suffers from becoming cluttered and hard to read. This is not so much an issue for the developers, but more an issue of presentation if DCRGraphs are used to showcase workflows to people that do not have prior knowledge of the system.

  There are some options available to alleviate such issues, such as hiding parts of a DCRGraph to only show the relevant part, which makes them much more accessible, even so, we found that they are still less intuitive than activity and state diagrams. Further alleviating this problem is the option to show DCRgraphs in an interactive form, as described in the positives above *20.1: Positives*. These issues have also been investigated in Hybrid process technologies in the financial sector [2].

- Sometimes it can be hard to make DCRGraphs reflect the flow of events that are described in use cases. This is greatly alleviated with experience with modelling DCRGraphs, but sometimes there are limits to DCRGraphs, which can make it a frustrating tool to model certain aspects of work flows. We talk about such cases here: *21.1.1: Familiarity* and *21.4: Working with system constraints on DCRGraphs.*

21 Discussion on the development process of implementing the initial system

21.1 Difficulties in designing the DCRGraph

Initially it was very hard for us to design a DCRGraph that could satisfy the demands of the restaurant system. We began designing the DCRGraph on the 17th of February, but we didn't get a satisfactory DCRGraph before the 14th of March. We did not only spend time on the DCRGraph in this period, but around two workdays a week were spent mainly on designing the DCRGraph. There are several reasons it took that long.

21.1.1 Familiarity

Probably the biggest reason it took us so long to design the DCRGraph, was our unfamiliarity with the more advanced functions of DCRGraphs. We had previous experience with DCRGraphs, but during this project we realised that our previous understanding of DCRGraphs was very limited.

Documentation available for DCRGraphs
There is documentation available, mainly in the form of a wiki [9], but the explanations there are somewhat lacking in some cases, such as milestones. Our lack of understanding lead us to design several graphs that were utilizing very ugly logic (workarounds with excessive use of exclude and include for example) to achieve things such as iteration in the DCRGraph.

Without a good understanding of DCRGraphs it was very hard to design a iterative graph in a meaningful way. The iterated behavior had to do with our wish to have edit functionality reflected in the graph, thus making it necessary to be able to jump back in the flow of the DCRGraph.

21.1.2 Understanding the asymmetrical nature of DCRGraphs

While working with DCRGraphs we found it frustrating and confusing that we sometimes could not go back to a previous state of a DCREvent. For example, when a DCREvent is executed, it can never go back to a state where it has not been executed, and so one has to think carefully of when the Condition relation is applicable. The Condition relation may be applicable in a linear graph, but might not be applicable in a graph with a cycle.

A good modelling or programming language prevents the user from making mistakes, and it can be argued that the notation of DCRGraph may be tempting the inexperienced user to do something wrong. An example of this will be given in the next section *21.1.3: Wrong use of relations.*

To truly understand the possible states of a DCREvent, we made a state diagram of possible states a DCREvent could obtain, and the transitions between the different states. This state diagram can be seen in *figure 33.* The asymmetry can be explained in that we can freely move between the different states s0, s2, s3 and s4 of the DCREvent by using the Exclude, Include and Response relations, but once we execute from s0 or s3 to s1, we can never go back to the states s0, s2, s3 or s4. We can however freely move between states s1, s5, s6 and s7 by using the Exclude, Include and Response relations.

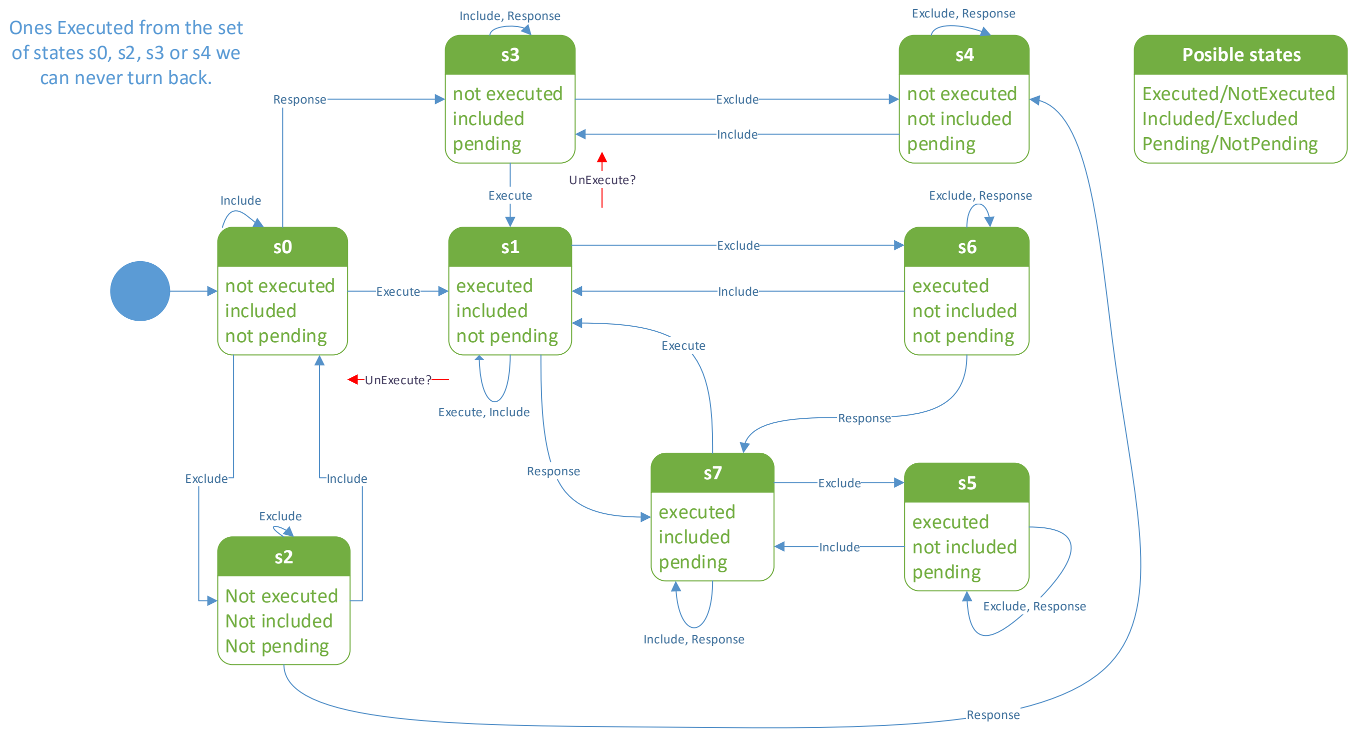


Figure 33: DCREvent state diagram

This may result in complications in special scenarios when building DCRGraphs, when we want to go back to the states s0, s2, s3 or s4 because of how we want the graph to work, but we cannot do so. For example, once we have transitioned from the aforementioned states, the Condition relation can never apply again unless we restart the simulation of the graph, and so one has be be careful of when to use the Condition relation.

21.1.3 Wrong use of relations

An example of the temptation where the user might be doing it wrong but does not know about it, is when we want to iterate through the same events multiple times, but we still want constraints between some of the events such that one has to happen before the next in every iteration. As an example, we have made *figure 34: Right and wrong way of building DCRGraphs*, where we have event A, B and C. To illustrate this example, we can say that A is "Begin cooking", B is "Finnish cooking", and C is "Order more".

The scenario we want to obtain is that A has to be executed before B. If C happens we have to do both A and B. Notice that we cannot do B before we have done A, and that C can happen at any time.

Step 1: We want to execute A before we execute B: The right way is to add a condition relation from A to B (step 1 in the right way in *figure 34*). Note that we might be tempted to have B excluded, and then let A include B (step 1 in the wrong way in *figure 34*). The behavior between the two is the same so far when simulating the graph: A has to be executed before B.

Step 2: We want A and B to be executed whenever C happens: The right way is to add a response relation from C to A and from C to B (step 2 in the right way). Note that we might be tempted to have A and B excluded, and then including them with C. This could be obtained by A excluding itself, and B excluding itself (step 2 in the wrong way). The behavior here starts to differ. In the right way, C makes A and B pending, such that the graph cannot be terminated before A and B has been executed. In the wrong way, C does not require A and B to be executed before terminating, however when simulating the graph, the user is able to execute C at any time, and cannot execute B before A, so the behavior still seems right for the user, and the user might not know about pending and accepting state.

Step 3: When we have executed A then B then C, and now want A to be executed before B, the condition does not apply because A is already executed in the previous iteration. The right way to fix this is to add a milestone from A to B (step 3 in the right way). An inexperienced DCRGraphs user may try to use the exclude relation from C to B and let the include relation of A include B when executed (step 3 in the wrong way). The behavior is now very different on the two graphs: In the right way we have achieved what we want: A has to be executed before B, when C happens we have to execute both A and B. We cannot execute B before we have executed A, and C can happen at any time. In the wrong way A has to be executed before B, we cannot execute B before we have executed A, C can happen at any time, BUT we are missing that when C happens we also need to execute both A and B.

If we compare the two graphs we have 4 relations in the right way and 6 relations in the wrong way, and we are still not achieving the wanted behavior in the wrong way.

As demonstrated, the DCRGraphs modelling language does not prevent or warn the user from heading the wrong way. The approach might seem right when starting out, but exponentially more relations must be added to keep modelling what one wants. This can be fixed by adhering to best practices as we will later describe in section *21.1.5: Conclusion*.

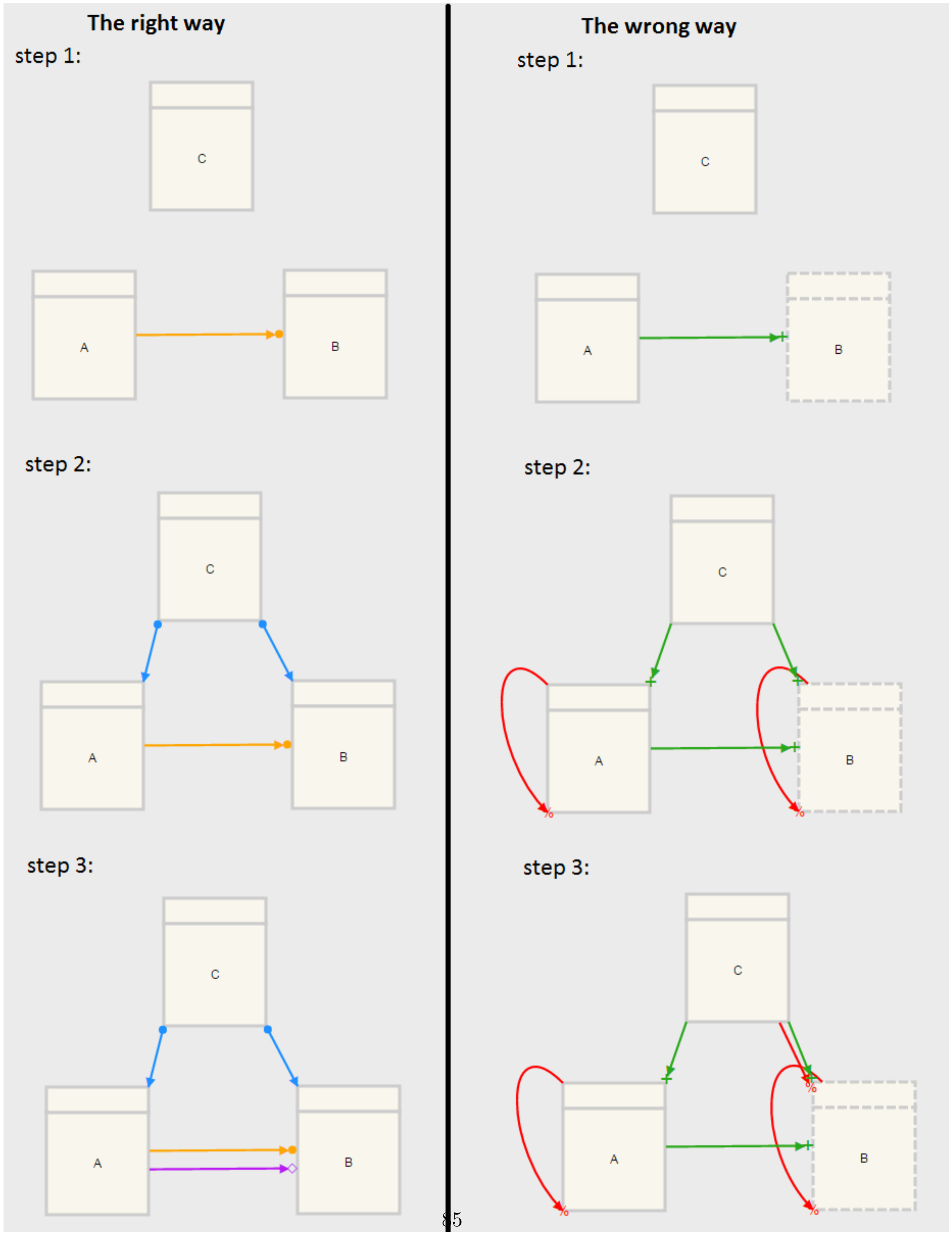


Figure 34: Right and wrong way of building DCRGraphs

21.1.4 Tools

In the following discussion on the DCRGraphs editors. We will refer to `http://dcr.itu.dk` simply as dcr.itu.dk, and `http://dcrgraphs.net` simply as dcrgraphs.net.

Why we started out using dcr.itu.dk
Initially, we used dcr.itu.dk to design the DCRGraph. We were aware of two tools to do this, the second being dcrgraphs.net. The reason we started out with dcr.itu.dk was that we liked that each line we wrote represented a new rule in the graph, a rule being a new event or relation between two or more events or groups. We were also able to add comments to each line and structure the rules, that generated the graph, according to the type of relation we added, and argue why one or a set of rules was necessary for the graph.

We also found that working with this tool was a faster and more modular way of developing DCRGraphs compared to dcrgraphs.net. Working with text using dcr.itu.dk made it easier to put new rules in, to cut a set of rules out and to try a different set of rules to see how the DCRGraph was influenced by the changes, or to run a subset of the graph and save previous sub versions of the DCRGraph.

Why we switched to dcrgraphs.net
When we developed a large graph, we ran into issues using dcr.itu.dk. Apparently there were some stability problems when working with large graphs and/or some parts of DCRGraph logic was not fully working, resulting in inability to execute some DCREvents in the DCRGraph for no apparent reason. Making it impossible for us to continue building the graph using dcr.itu.dk. This was very frustrating, and in the end we decided to use dcrgraphs.net instead, even though we feel that the design of this tool is inferior.

Dcrgraphs.net is however much richer in features, having features such as view levels (can be very valuable in large graphs), sorting the view according to groups and roles, with the additional feature of storaging the graphs online, as well as ways to export graphs into different file formats such as XML. In general, it was very clear that dcrgraphs.net is a more mature tool, though we feel that dcr.itu.dk is potentially better to work with, once it has matured more.

Working with dcrgraphs.net
Using dcrgraphs.net also had complications. With large and complex DCRGraphs, we found it gradually harder to comprehend each relation and it's significance to the DCRGraph compared to dcr.itu.dk, where each line of text represented a relationship. Analysing the DCRGraph for missing relations was easier to do line by line in text using dcr.itu.dk than systematically check each visual relation in a clutter of relations in dcrgraphs.net.

The automatic layout feature of events and relations in the dcr.itu.dk editor made it easy to create and run small graphs, however with big and complex graphs the graph simply becomes too large and complex to comprehend with the automatic layout function. The manual layout of events and colors of relations in dcrgraphs.net made it easier to place events just right to keep an overview of the graph, however this solution is also more time consuming. A feature to automatically layout events in DCRGraph.net and a feature to manually layout events using dcr.itu.dk would be nice.

21.1.5 Conclusion

An inexperienced user of DCRGraphs may think he or she is on the right track when building the DCRGraph because during simulation, it seems like the graph is correct. However introducing wrong relations leads to more wrong relations and workarounds, causing the graph to become more complex, incomprehensible and exponentially difficult to expand because of the workarounds.

When building large DCRGraphs it is difficult to foresee the full scenario one wants to obtain, and so even an experienced DCRGraph user may also run into troubles. This is especially applicable in large and complex graphs with multiple iterations of the same events.

This is the reason why we only show pending events in the DROM Client UI in the main view (after log in); It is easier to make events pending and then execute or exclude the pending event to make it not pending.

The user is not able to do such things when using the modelling tools, he is able to see events that are pending and choose to execute them, but he can also execute events that are already executed and not pending, which may lead to mistakes.

It is visibly unclear when the graph is in an accepting state using the DCRGraphs.net tool, which may lead to the user not knowing what the response relation is for and then never learn what the milestone relation is for.

A guide to "Best practices when building DCRGraphs" would come in handy for a new user of DCRGraphs.

We have created a short guide through this project on DCRGraphs based on our experiences:

1. Learn all relations and their effects before building a big and complex graph.
2. Consider all condition relations when iterating multiple times on the same events.
3. Use the response and milestone relations to specify constraints when iterating through events.
4. Do not exclude and include events unless it is absolutely necessary, the inclusion and exclusion of events may spread across the graph when trying to fix what is already done wrong.
5. Systematically check that every relation in the graph has a reason for being there.
6. Do not add relations unless you have a unique reason for it alone being there. Think twice before adding it.
7. If you get the feeling that you need to make a workarround with multiple relations, then your graph may already have flaws that makes you do so.

The asymmetrical nature of DCRGraphs is not something we have tried to change or experiment with in any way, but it is important to be aware of it when building DCRGraphs.

21.2 Developing a Web API for DCRGraphs

21.2.1 DCRGraphs to objects

Our chosen approach to getting DCRGraphs into a program is to parse xml files with DCRGraphs in them. This was based on our use of `http://www.dcrgraphs.net` to create DCRGraphs, where it is possible to export the DCRGraphs as xml files.

To do this, we created parsing logic in the DROM system API, to pull DCRGraph data out of xml files and into classes which could be worked with. This was a time consuming process. A better way of doing this would make working with DCRGraphs much more accessible. This could for example be an option to get DCRGraphs directly from a service where you create your DCRGraphs (in this case, dcrgraphs.net). Having a recommended database schema available would also have been beneficial.

21.3 Developing a front end application for DCRGraphs

21.3.1 Specifying where to put events in the GUI in DCRGraph

When designing the DROM Client UI, we thought that it would be useful and user friendly to only show events that were important for what the user was doing in the view. For example, when editing an order, it would add visual clutter if we showed all events possible for execution, the user may not be interested in events that are not related to the edit order screen. The events necessary for editing would be hidden among other events, hence the clutter.

To show different events on different screens, and not have our system dependant on single events, we would have to be able to specify how the event should be shown using the DCRGraph editor. We solved this by using groups:

- Events in the "Edit events" group will only be showed in the Edit order screen. Events in this group is: "For takeaway", "For delivery" and "For serving".
- Events in the "Hidden edit events" will not be shown in the client at all, but will be executed by the client program automatically. Events in this group is: "Items have changed".
- Events in the "only pending" group will only be showed in the main order overview screen. All other events not specified in the previous two groups are in this group.

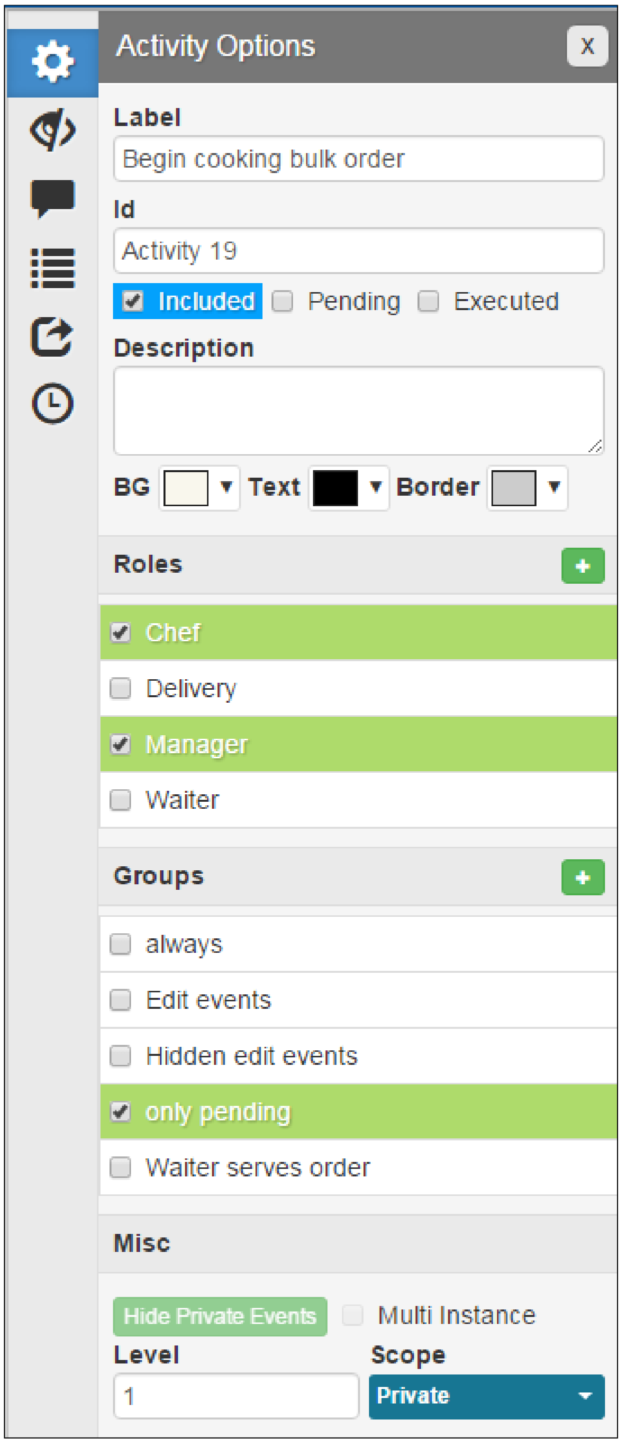
Other group names could be used to specify specific functionality in the client application for a group of events. Otherwise the name of the event can be used. However, when doing this, the developer has to be cautious to add groups to new events in the DCRGraph when adding events, and to know which events in the DCRGraph, are also in the DROM system code.

One could argue this is bad design and couples the system with the DCRGraph, however we have not found better way of specifying specific behavior on events using the DCRGraph.net tool.

21.3.2 How we used the DCRGraphs.net tool

To model and edit DCRGraphs, we have used the online DCRGraph editors dcr.itu.dk and DCRGraphs.net. In order to be able to change the DCRGraph, and make the system reflect the changes made in the DCRGraph. We have tried to preserve the logic and data input fields given in DCRGraphs.net. We came a long way before running into difficulties when we wanted to design something different that what could be specified in the DCRGraph. We will now explain how we used the DCRGraphs.net tool, what was difficult, and which data input fields we would like the developers of DCRGraphs.net to think about.

Using DCRGraphs fields to our advantage:
An example of how we used `http://www.DCRGraphs.net` can be seen in figure *35: Input fields for an event using DCRGraphs.net tool.*



Activity Options

Label: Begin cooking bulk order

Id: Activity 19

[✓ Included] Pending Executed

Description:

BG Text Border

Roles

[✓ Chef]
Delivery
[✓ Manager]
Waiter

Groups

always
Edit events
Hidden edit events
[✓ only pending]
Waiter serves order

Misc

Hide Private Events
Multi Instance

Level: 1

Scope: Private

Figure 35: Input fields for an event using DCRGraphs.net tool

We used the Label input field to specify the name of a DCRGraph event (in this section referred to as DCREvent).

We didn't use the Id field to input data, as the DCREvent would receive a new Id in the database. This is necessary because DCREvent ids are only unique to one instance of a DCRGraph, so if there are more than one instance of the same DCRGraph, then the DCREvent id's are no longer unique.

We used the Included, Pending and Executed check boxes to enter the starting state of DCREvents.

We thought the Description field could be used as the pop-up tool tip in the DROM Client UI, to display a message when a user hovers over the field with the mouse pointer. The message would contain information about when to execute the DCREvent. We did not have time to actually implement this.

We used the Roles field to specify which actors in the restaurant could execute a given DCREvent, for example if a DCREvent could only be done by the Waiter or by the Chef. We used the role 'Manager' to be able to execute all DCREvents.

We used the Groups field to specify how the DCRGraph should be shown in the DROM Client, as specified earlier in section *21.3.1: Specifying where to put events in the GUI in DCRGraph.*

Unavoidable dependencies between the DCRGraph and the system
We wanted to build a restaurant ordering system, in which processes are dependant on a DCRGraph, which could be updated and inserted into the system, and then let the system reflect the changes made in the DCRGraph. Therefore, we wanted a minimum amount of dependencies between the system and the DCRGraph, so that a minimum amount of changes would have to be made in the system, to run an updated DCRGraph.

In order to be able to build DCRGraphs using DCRGraphs.net, and exporting it and using it in our system, as described in section: *21.2.1: DCRGraphs to objects*, and also be able to continue changing the graph with ease and letting the system change with it, we had to keep ourselves in the boundaries of what could be made with the tool when designing the user interface.

If we were to add DCRGraph related data to the DCRGraph outside the tool, we would have to do it in code when parsing the DCRGraph. This would be hardcoded for each DCRGraph or DCREvent we wanted to add data to, which is bad for the extensibility of the DCRGraph. Adding data to the graph or an event using code tied to our program would result in a DCRGraph and a system that would be very hard to change, with many lines of code that would have to be changed with each update in the DCRGraph. This is not preferable.

However sticking to the data input fields given in the DCRGraphs.net tool limits what can be done when designing a user interface. It is hard to specify how things should be shown in the UI when the only way to specify it, is in the DCRGraph itself through the input fields of an event.

When sticking to a UI design, that must be modelled in the DCRGraphs, it creates challenges that can't be solved easily or in a good way without creating dependencies between the system and the DCRGraph.

We had to be creative in the making of the DCRGraph in order to manage how the DCREvent should be shown in our DROM Client UI. Specifying groups on events that tells the system how it should show the DCREvent was one solution to this challenge. Another was to specify roles in the system on each DCREvent to let the system sort out uneccesary events for the user. But sometimes it was just impossible to avoid using the label name of an event to handle specific behavior in the UI for a specific DCRGraph event. This creates requirements to the DCRGraph, in groups, roles and sometimes event labels that must be specified in the DCRGraph, otherwise the system may not work as intended.

It would be nice if the creators of DCRGraphs.net would consider adding custom fields to a DCR-Graph that could be used as an input field on events. A custom field would make it possible to add data that is not necessarily related to an event in DCRGraphs like check lists, pictures for icons, Boolean values, numbers etc. This would make it easier for the developer to add more data to the graphs on specific events, that could be used in the program, while not creating dependencies between the DCRGraph and the system. A bit like a content management system for a Website, but for DCRGraphs. However we do not know how this would work with the XML export.

All in all we ended up with a system, where a minimum amount of changes would have to be made in the code, in order to run on a new DCRGraph. The DCRGraph must also contain certain events and data, otherwise the system will not run as intended.

21.4 Working with system constraints on DCRGraphs

21.4.1 Constraints from the DROM system client

In our client, as part of the functionality to edit orders, we have programmed it to automatically execute an event with the group "hidden edit events" when the edit contains changes to the items that need to be cooked. This was a design that we arrived at, because we found it problematic if the user had to manually execute this event, when the user edits the part of an order it is related to.

The event needs to be executed whenever food items are added or removed from an order, and if the user had to execute it manually, it could easily be forgotten and create cases where the DCRgraph would no longer reflect what had happened to an order. To avoid this, we decided that this event must be executed by the system, when items are added or removed from an order. This makes a constraint on the DCRGraph that it needs to contain one event, and only one event, which has the group "hidden edit events", because the client is expecting there to be such an event it can execute.

In order to be able to sort events depending on who is using the DROM Client application, events in the DCRGraph must have a role specified to be shown in the DROM Client UI. This creates another dependency from the system on the DCRGraph.

These dependencies from the system to DCRGraphs are of course unwanted by the developers designing the system and should be kept to a minimum, in order to keep the extensibility of the system. However, these dependencies are unavoidable when designing the system as previously described in section *21.3.2: How we used the DCRGraphs.net tool.*

21.4.2 Constraints from the DROM system API

We have created events that setup the DCRGraph to be in a state appropriate to different order types. As with the "hidden edit events" described above, we decided that these events do not make much sense to the user and as such should be executed automatically without user interaction. This is done by having the API check the order type when new order creation request are received, and execute the relevant setup event.

21.4.3 Conclusion

We found it to be reasonable to establish rules that a DCRGraphs has to follow, before it can work with our system. This seems to be necessary to use DCRgraphs when modelling cases such as this one, where you have events which you do not want the user to execute manually. However, doing this makes it extremely important to be absolutely sure that anything to do with events, which are hardcoded to work with the system, is final, and does not need to be changed later on.

Normally it is very easy to change the DCRGraph to reflect new designs and to remove mistakes. This is no longer quite as easy with events that are bound to the code, because touching anything that has to do with them, now also needs to be changed in the code.

Some things are still pretty easy to handle, such as new setup events, as can be seen in our subsequent changes with bulk order: *16.3: Conclusion*, but if it is decided to completely change how things work in the DCRGraph, then it will most likely also require completely new code to follow along with it.

While this can be seen as a negative thing, it also has to be considered that pretty much any system, whether it is using DCRGraphs or not, is likely to require changes in the code to accommodate major changes to functionality. If anything, the ease with which we were able to include the bulk order addition to the DCRGraph in *16: Introducing bulk orders* suggest that DCRGraphs helped to ease the work required to implement new changes.

21.5 What was good about DCRGraphs in the initial system

21.5.1 Usecases to DCRGraphs

In *13.3.1: Creating the DCRGraph* and *20: From use cases to DCRGraph - a possible methodology* we talk about how we replaced activity and state diagrams with DCRGraphs. This felt like a very nice way to do it, and at the same time we saved a lot of time on making diagrams which can otherwise be a very time consuming task for what can often feel like very little return value. Activity and state diagrams do serve a very vital role, but if DCRGraphs can serve in the same role, then we will happily skip making them in advantage of spending more time on making a DCRGraph.

21.5.2 Updating the design through DCRGraphs

During development it is close to inevitable that mistakes in the design will be discovered. In our case this mainly meant discovering logic mistakes in our DCRGraph. The nice thing is that it is very easy to update a DCRGraph and then simply putting the new updated DCRGraph into the system, at least with our approach of storing them as xml files that are read by the system.

Compared to redesigning code repeatedly, every time a mistake is found, or a revision of the design is made, we found it to be much easier and faster to redesign the DCRGraph instead. There is of course still things that exists outside of the DCRGraph, such as the DROM system client UI design, but most of the core design exists inside the DCRGraph where it is easily changeable.

22 Discussion on the development process of implementing subsequent changes

22.1 Introducing bulk orders in the DCRGraph

In *16: Introducing bulk orders* we updated our DCRGraph with new functionality, specifically the ability to have bulk orders. There were some positive experiences with this, but it also revealed some issues.

22.1.1 The positive

- Initially we were quite negative about constraints imposed on DCRGraphs by the DROM system, discussed in *21.4: Working with system constraints on DCRGraphs* and **??**: **??**, but we were positively surprised when we actually tried to change a part of the graph that were connected to such a constraint.

  In *16.3: Conclusion* we mention the results of putting in a new DCRGraph with bulk order functionality, and one of the things we discovered, was that even though we had to edit code for the new DCRGraph work, it was actually only required two very small changes, and it did not take more than 10 minutes to make the changes.

  In this change, we added a new delivery type, which can be said to be a pretty major new functionality in the system, and it only took around teen minutes to make the necessary changes in the code. Some of the reason that it went that fast was of course that we had been working on the code regularly in the previous weeks, and as such were very familiar with it.

  Even so, changing what amounts to about two lines of code is never going to be a very time consuming process, and it honestly felt pretty amazing that after we had created an updated

DCRGraph, we got it working so fast and with no bugs. This is really showcases the potential ease of working with systems designed around DCRGraphs.

- Modelling new functionality in a DCRGraph is much faster than if everything had to be modelled through code. What we mean here, is that we have essentially offloaded a lot of what would otherwise have been code in our system, into DCRGraphs. Updating the DCRGraph then updates the system, which we feel is generally much faster than it would have been to update the code.

22.1.2 The negative

- When we came to the point in our project where it was time to implement a change to the DCRGraph, it had been a few weeks since we had made any changes to the DCRGraph that we created for the initial system. It is very reasonable to expect people to spend a bit of time to re-familiarize themselves with a DCRGraph made earlier, but we found issues with the ease of reading the DCRGraph.

  Basically, there are arrows everywhere (arrows representing relations), and they clutter the view, making it very hard to see what events actually has which relations. Since we are the ones who created the DCRGraph, it was of course not a major issue, but if we, as the creators of the DCRGraph, found it annoying, then someone which has not seen it before will definitely have problems. We suggest a solution to help with this issue here: *22.1.3: Suggestion - new view mode.*

22.1.3 Suggestion - new view mode

In *22.1.2: The negative*, we discuss an issue with the readability of a DCRGraph as it is represented in DCRGraphs.net. The same issue is also present in the other tool at dcr.itu.dk.

If we take a look at our DCRGraph as it looks without hiding anything, *figure 36*, we can see that the arrows representing relations are pretty much all over the place, overlapping with events and each other. This makes it very tough to see which arrows belong to which event. If we look at an early version of our DCRGraph in dcr.itu.dk, as shown in *figure 37*, there are similar issues with arrows representing relations making the DCRGraph hard to read.

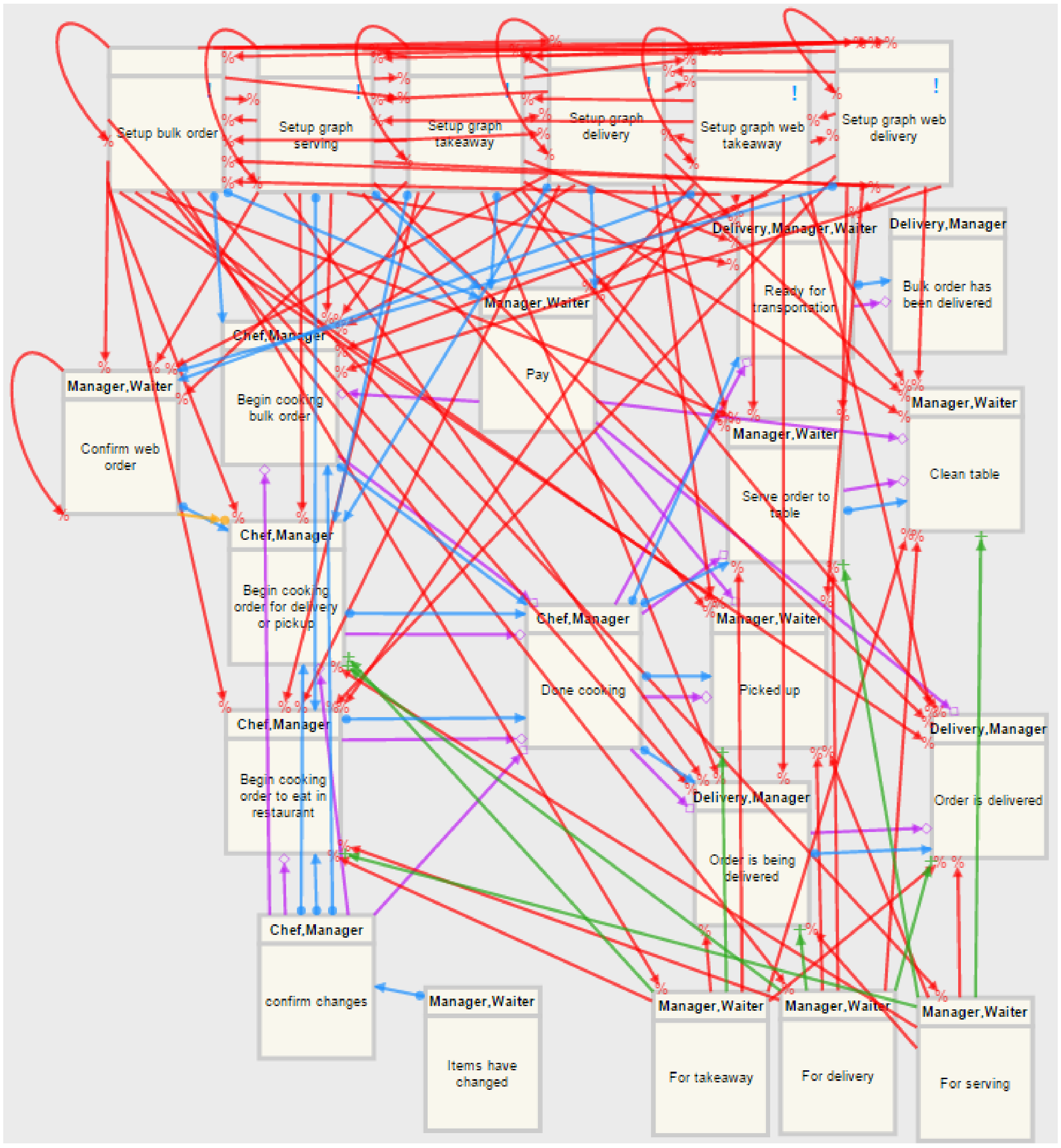


Figure 36: The complete DCRGraph after adding bulk orders

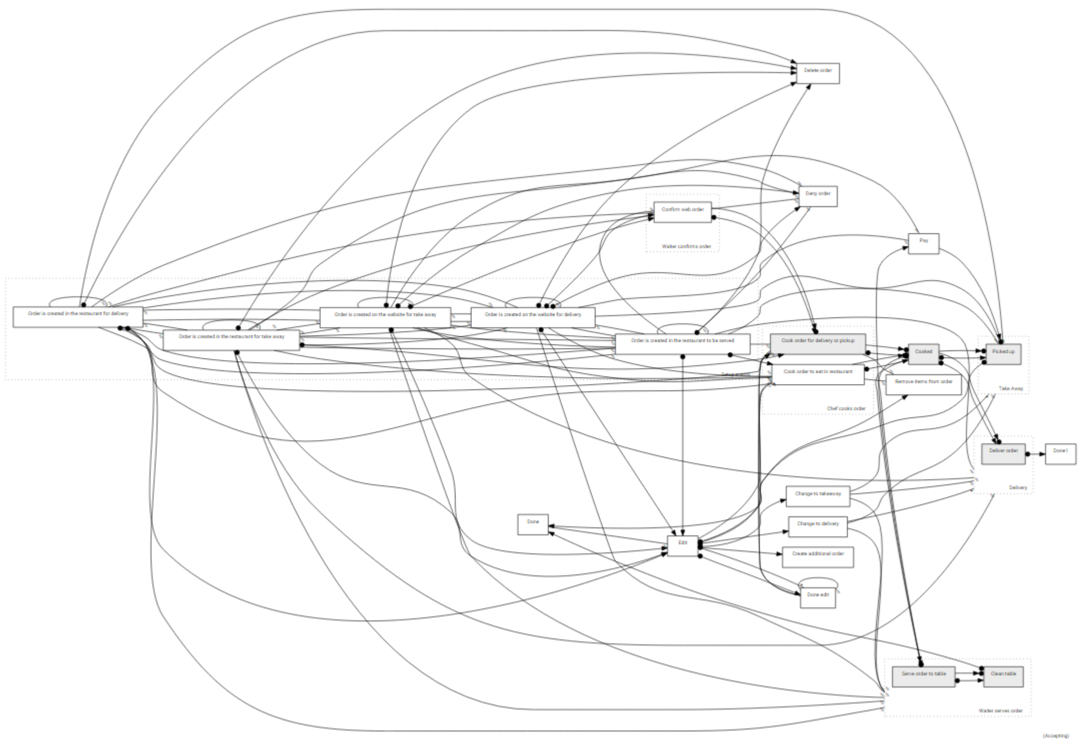
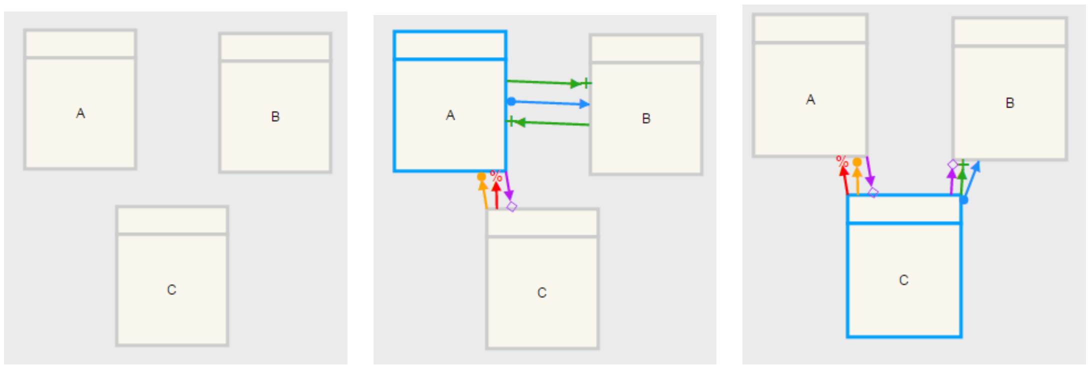

Delete order
Deny order
Confirm web order
Waiter confirms order
Pay
Order is created in the restaurant for delivery
Order is created in the restaurant for take away
Order is created on the website for take away
Order is created on the website for delivery
Order is created in the restaurant to be served
Cook order for delivery or pickup
Cooked
Picked up
Cook order to eat in restaurant
Remove items from order
Take Away
Chef cooks order
Deliver order
Done I
Delivery
Change to takeaway
Change to delivery
Done
Edit
Create additional order
Done edit
Serve order to table
Clean table
Waiter serves order
(Accepting)

Figure 37: Early version of our DCRGraph, on dcr.itu.dk

To help with this, we suggest implementing a new view mode in the tools to develop/show DCR-Graphs. This new view mode should only show relations on selected events. This means that when you are looking at a DCRGraph, while not selecting anything, you will only see events with no relations. When you want to see relations on an event, you select it and then all the relation to and from the selected event will be shown. It should also be possible to select multiple events, to show all the relations on more than one event at a time.

If we look at *figure 38a*, this is how a DCRGraph would look in our suggested view mode, if no event is selected.

This is a very clean view, where we only see the events of the DCRGraph and no relations. This is of course not very useful just by itself, since relations are very relevant for a DCRGraph. From this view, you can then select one or more events, to see their relations, while still hiding the relations of events that are not selected. See *figure 38b, figure 38c* and *figure 39* for examples.



(a) No selected events (b) Relations of selected event (c) Select another event

Figure 38: Examples of new view mode

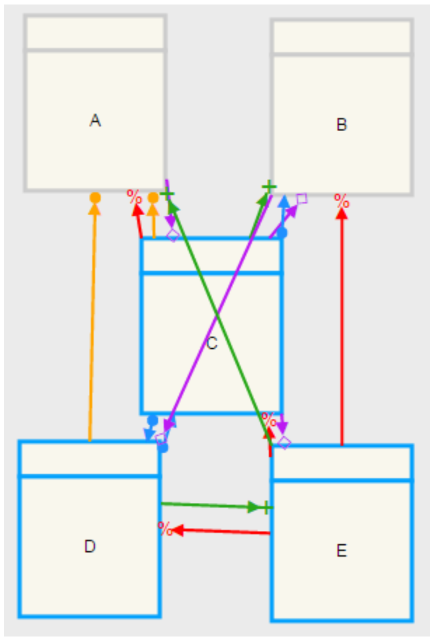


Figure 39: Select multiple events to show their relations

With a view like this, it is very easy to pick an event and see exactly the information that is related to that event, while hiding all the information that you are not interested in at the moment. This would make it easier to figure out what is going on in big DCRGraphs, that are otherwise very cluttered and hard to read.

22.2 Introducing more restaurants

In *17: Introducing more than one restaurant* we add the functionality to use the DROM system with more than one restaurant. We discuss our experiences with adding this functionality to the system here.

22.2.1 DCRGraphs

It was possible to make all the required changes to the system, without touching upon logic directly connected to DCRGraphs. It was nice to see that our system has succeeded in creating high decoupling between different kinds of logic.

It could be feared that DCRGraphs would sink into every part of the design of the system, creating strong dependencies on the system functionality, which would be hard to expand, but it is obviously possible to decouple DCRGraphs from the system, making it possible to implement new features without updating the DCRGraph.

23 Overall conclusion

During this project we have implemented a restaurant ordering system with DCRGraphs.

In regards to the first issue of our problem definition, "**The eases and difficulties of incorporating DCRGraphs when designing and building a system**":

We found that DCRGraphs are a potentially powerful tool which can help developers in creating systems with good support for iterative changes and expandability. However, to achieve this potential it is required to spend the time necessary to deeply understand DCRGraphs. Otherwise you will inevitably end up having to remake your DCRGraph countless times due to bad design. DCRGraphs are also held back by the tools to model them with, which are lacking in features and stability.

While the tools to model DCRGraphs are somewhat lacking in some areas, we still found it be much faster to model and update a DCRGraph through them, than it would be to do the equivalent in creating and updating diagrams and code that would cover the same information.

We found that DCRGraphs can save time in the design phase of a system. DCRGraphs incorporate very well with the activities that take place during system design, and proved to be suited to replacing some of the time consuming design activities. This worked out very well as developers, but we do not believe the tools, to visualize DCRGraphs, are good enough to use DCRGraphs when interacting with people that do not have experience with DCRGraphs. Small DCRGraphs are not an issue, and can easily be understood, but DCRGraphs very quickly become too complex to be understood with the currently available visualization options.

Designing a system to incorporate DCRGraphs was not a seamless process. The information contained inside DCRGraphs does not intuitively translate into elements in a user interface. It was a challenge to create a meaningful user interface which was based on information obtained from a DCRGraph. However, incorporating DCRGraphs into parts of the system which were not related to a user interface, was not an issue.

In regards to the second issue of our problem definition, "**The eases and difficulties of implementing new requirements or changes into a system builton DCRGraphs**":

While developing the initial system was a bumpy process, when we began to implement new functionality on the existing system, DCRGraphs really started to shine.

Introducing new functionality to the system through a DCRGraph is extremely fast. It was so fast and easy to implement a new delivery type in the system, that it didn't even feel like we had begun before it was already done.

Introducing functionality which was not related to DCRGraphs was a slightly longer process, but was still very easy.

In general, we could feel that DCRGraphs had pushed us into designing a system which is easy to expand upon.

Working with DCRGraphs in general
If we were to create a new system, where a lot of business processes are to be modelled in the system, and the business processes were likely to change, we would definitely consider using DCRGraphs again.

Overall, DCRGraphs had a positive influence on our case system, and is definitely something to look into when designing systems such as these. We liked working with DCRGraphs.

24 Reflection

When planning this project, we had a large system in mind. However when planning and developing the system, we realised we did not have time to develop all the things we wanted.

We feel that we have studied our research question well with the system that we built, which was the main goal of the project.

A lot of nice-to-have features were not implemented because we simply did not have time to develop them. These features would have made the system more complete, and more suited for use in an actual business, but ultimately would not have contributed much towards the actual research goal.

Part VII

Sources and appendix

25 Sources

References

[1] T. Hildebrandt, R. R. Mukkamala, and T. Slaats. "Designing a Cross-Organizational Case Management System Using Dynamic Condition Response Graphs". In: *Enterprise Distributed Object Computing Conference (EDOC), 2011 15th IEEE International.* Aug. 2011, pp. 161–170. DOI: 10.1109/EDOC.2011.35. URL: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=6037570.

[2] Søren Debois et al. "Hybrid process technologies in the financial sector". In: *Proceedings of the Industry Track at the 13th International Conference on Business Process Management 2015 co-located with 13th International Conference on Business Process Management (BPM 2015), Innsbruck, Austria, September 2015.* 2015, pp. 107–119. URL: http://ceur-ws.org/Vol-1439/paper9.pdf.

[3] Morten Marquard, Muhammad Shahzad, and Tijs Slaats. "Web-Based Modelling and Collaborative Simulation of Declarative Processes". In: *Business Process Management - 13th International Conference, BPM 2015, Innsbruck, Austria, August 31 - September 3, 2015, Proceedings.* 2015, pp. 209–225. DOI: 10.1007/978-3-319-23063-4_15. URL: http://dx.doi.org/10.1007/978-3-319-23063-4_15.

[4] *ASP.NET 4.5.2.* URL: http://wiki.dcrgraphs.net.

[5] *ASP.NET web API template.* URL: http://www.asp.net/web-api.

[6] Bernd Bruegge and Allen H. Dutoit. *Object Oriented Software Engineering.* 3rd edition. Pearson Education, Inc. ISBN: ISBN 10: 0-13-606125-7 ISBN 13: 978-0-13-606125-0.

[7] *Chen's notation explained.* URL: https://en.wikipedia.org/wiki/Entity%E2%80%93relationship_model.

[8] *Cohesion explained.* URL: https://en.wikipedia.org/wiki/Cohesion_(computer_science).

[9] *DCRGraphs Wiki.* URL: https://msdn.microsoft.com/en-us/library/ms171868(v=vs.110)#v452.

[10] *Directed graph explained.* URL: https://en.wikipedia.org/wiki/Directed_graph.

[11] *Entity Framework.* URL: https://msdn.microsoft.com/en-us/data/ef.aspx.

[12] *Entity Framework 6.1.3.* URL: https://www.nuget.org/packages/EntityFramework/6.1.3.

[13] *Foreign key explained.* URL: http://www.w3schools.com/sql/sql_foreignkey.asp.

[14] *Microsoft article - The MVVM Pattern.* URL: https://msdn.microsoft.com/en-us/library/hh848246.aspx.

[15] *Microsoft article - What is XAML?* URL: https://msdn.microsoft.com/en-us/library/cc295302.aspx.

[16] *Microsoft article - What's a Universal Windows Platform (UWP) app?* URL: https://msdn.microsoft.com/en-us/windows/uwp/get-started/whats-a-uwp.

[17] *Representational state transfer.* URL: https://en.wikipedia.org/wiki/Representational_state_transfer.

[18] *Responsive design explained.* URL: https://en.wikipedia.org/wiki/Responsive_web_design.

[19] *Singleton pattern explained.* URL: https://en.wikipedia.org/wiki/Singleton_pattern.

[20] *WPF explained.* URL: https://en.wikipedia.org/wiki/Windows_Presentation_Foundation.

[21] *XAML and Codebehind explained.* URL: https://msdn.microsoft.com/en-us/library/cc295302.aspx#Anchor_3.

26 Appendix

26.1 Use cases

Use case	Customer creates order on website
Actors	Customer named Alice
Initial assumptions	None
Flow of events	1. Alice enters the DROM system website. 2. Alice looks at and picks the items she would like to order. 3. Alice enters checkout. She has the option to create an account. *Extends in figure 41: Customer creates accout on website.* She can also proceed without an account. 4. Alice proceeds without an account and fills in her delivery information and goes to payment. 5. Alice fills in her payment information and completes her order. 6. *Extends in figure 42: Waiter confirms order.*
State of completion	The order is now waiting to be confirmed.
Divergences	• In step 4 and 5, Alice would have had the option to use an address and payment information saved on her account, if she was logged in and had added such to her account.

Figure 40: Customer creates order on website

.

Use case	Customer creates account on website
Actors	Customer named Alice
Initial assumptions	None
Flow of events	1. Alice has entered the account creation page on the DROM system website. 2. Alice fills in her email and password and confirms that she wants to create her account. 3. Alice receives a verification email that she has to confirm her account with. Alice confirms her account through the email. 4. Alice doesn't want to write her address every time she orders, and as such she wants to add an address to her account. She enters the add address page for her account and fills in her address. 5. Alice doesn't want to write her payment information every time she orders, and as such she wants to add a payment option to her account. She enters the add payment options page for her account and fills in her payment details.
State of completion	Alice now has an account with a pre-filled address and payment option.
Divergences	None

Figure 41: Customer creates accout on website

Use case	Waiter confirms order
Actors	Waiter named Bob
Initial assumptions	An order has been created through the DROM system website.
Flow of events	1. Bob looks in the DROM system client, and sees that an order is waiting to be confirmed. 2. Bob checks the order and judges it to be acceptable. 3. Bob confirms the order. This sends an email to the customer who placed the order, confirming the order has been accepted, and also gives an estimated delivery time, if Bob filled one in when he confirmed the order. 4. *Extends in figure 46: Chef cooks order.*
State of completion	The order is now waiting to be cooked.
Divergences	• In step 2, Bob can decide that the order is not acceptable and is able to deny it, if the restaurant is too busy for a large order. This sends an email to the customer who placed the order, with a description Bob fills in, describing why the restaurant could not take the order.

Figure 42: Waiter confirms order

Use case	Waiter creates order in restaurant
Actors	Waiter named Bob, Customer named Alice
Initial assumptions	Bob is logged into the DROM system client. Alice is in the restaurant at a table.
Flow of events	1. Bob asks Alice for her order. She expresses her desire to order some items from the menu. 2. Bob enters into the new order creation form on the DROM system client. 3. Alice tells Bob which items she would like, and that she wants to eat in the restaurant. Bob enters the items and which table she is sitting at, into the order form. 4. After Bob has noted down all of the details of Alice's order, Bob saves the order on the DROM system client. 5. *Extends in figure 46: Chef cooks order*
State of completion	The order is now pending for a chef to make the food.
Divergences	• Alice can decide to change her order later. This case extends in figure 44: Waiter edits order • Alice can decide to cancel her order at some point. This extends in figure 45: Waiter deletes order

Figure 43: Waiter creates order

Use case	Waiter edits order
Actors	Waiter named Bob, Customer named Alice
Initial assumptions	Alice has placed an order in the restaurant, it is in a pending state, waiting for a chef to start making the food. Bob is logged into the DROM system client.
Flow of events	1. Alice has decided that she would like to change her order. She contacts the waiter Bob and makes her request. 2. Bob finds Alice's order in the DROM system client and enters the order edit function. 3. Alice would like to change one of the items she has ordered. Bob looks at the order and sees that the food is not being cooked yet, and as such it is still acceptable for her to change an item. 4. Bob changes the item in question and finishes the edit. The chef will be notified of the change.
State of completion	The order has been updated.
Divergences	• In step 3, if the order is already being cooked, it is up to the waiter to make a decision on whether it is reasonable to change the order. • It is also possible to change the way the order is delivered, as part of the edit. For example changing a serving order into a takeaway order.

Figure 44: Waiter edits order

Use case	Waiter deletes order
Actors	Customer named Alice, Waiter named Bob
Initial assumptions	Alice has created an order for delivery on the DROM system website. She has decided that she does not want it after all. Bob is logged into the DROM system client.
Flow of events	1. Alice calls the restaurant with the phone number from the website. 2. Bob picks up the phone. 3. Alice says she does not want the order after all. 4. Bob looks up her order in the DROM system, and sees that it is not being cooked yet. 5. Bob deletes it. 6. Bob and Alice hangs up.
State of completion	The order is deleted in the DROM system.
Divergences	• In step 4, Bob could discover that the order is being cooked, has been cooked, or is being delivered. Bob might need to inform the chef or the delivery guy of the cancellation or inform Alice that it is to late to cancel. • The waiter is able to delete any type of order, not just web delivery orders.

Figure 45: Waiter deletes order

Use case	Chef cooks order
Actors	Chef named Charlie
Initial assumptions	An order has been created, either through the DROM system website or by a waiter. The order has been confirmed and is waiting to be cooked. Charlie is logged into the DROM system client.
Flow of events	1. Charlie looks in the DROM system client for orders waiting to be cooked. 2. Charlie looks at the oldest order, its order type and its items, and enters that he starts cooking the order. 3. Charlie prepares the food for delivery, and places the cooked order where other deliveries are. 4. Charlie enters that he is done cooking the order. 5. The flow of events branches out into different delivery methods, described in the following use cases: use case *47: Waiter serves order*, use case *48: Order delivery* or use case *49: Order pickup.*
State of completion	Charlie has cooked the items on the order. The order is ready for delivery, and Charlie may choose a new order.
Divergences	• In step 2, Charlie could discover that the order type is "for serving". If so, he will cook and prepare the order for serving. If the delivery type is "for delivery", or "for pickup" he will cook and prepare the order for delivery or pickup.

Figure 46: Chef cooks order

Use case	Waiter serves order
Actors	Customer named Alice, waiter named Bob
Initial assumptions	An order has been cooked and is waiting to be served to a table. Bob is logged into the DROM system client.
Flow of events	1. Bob looks in the DROM system client and sees that there is an order which has been cooked and needs to be served to a table. 2. Bob picks up the food and goes to the table where Alice is waiting for her food. 3. Alice receives the food from Bob and starts eating 4. Bob goes back to the DROM system client, and confirms that he has served the order. 5. After Alice is done eating, she pays her bill to Bob. Bob find Alice's order in the DROM system client and confirms the payment. 6. Alice leaves the restaurant. Bob sees that Alice has paid for her order, and sees that her table needs to be cleaned and made ready for the next customer. Bob cleans the table and enters in the DROM system client that the table has been cleaned. 7. Bob sees that Alice's order is done and archives it.
State of completion	Alice's order is now done and has been archived.
Divergences	None

Figure 47: Waiter serves order

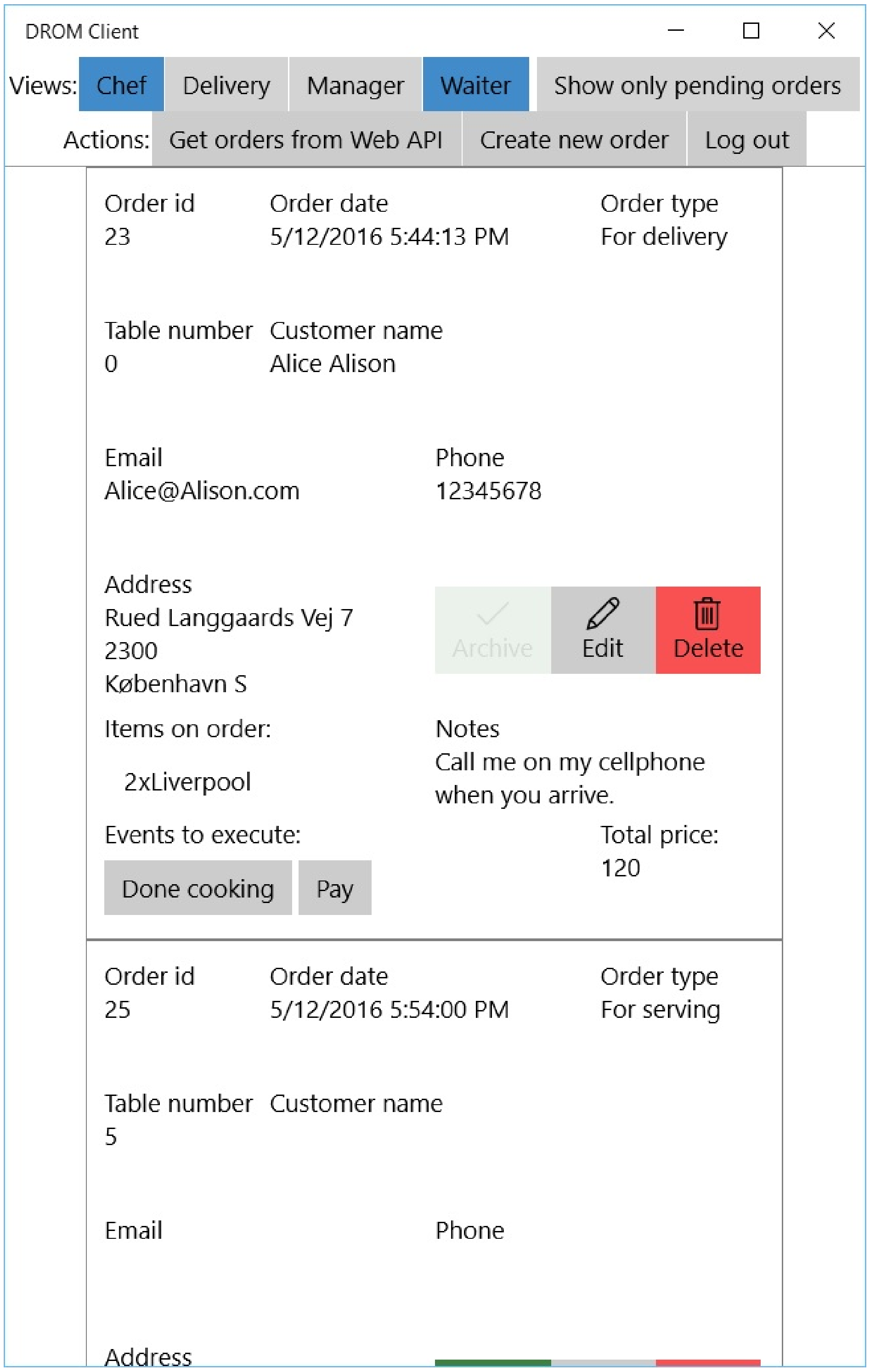
Use case	Order delivery
Actors	Delivery guy named Dan, Customer named Alice
Initial assumptions	Alice has made an order earlier, which is now ready to be delivered to her address. Alice has not yet paid for the order. Dan is logged into the DROM system client.
Flow of events	1. Dan looks in the DROM system client, for orders waiting to be delivered. Dan sees Alice's order. 2. Dan changes Alice's order status to "Being delivered". 3. Dan picks up the food for the order and drives to Alice's address. 4. Dan arrives to Alice's delivery address. Dan checks if Alice's order has been paid. If it has not been paid then Dan receives the payment from Alice. 5. Dan now hands over Alice's order. 6. Dan confirms that Alice's order has been delivered in the DROM system client. 7. Dan sees that the order is done and archives it.
State of completion	Alice's order is finished and has been archived.

Figure 48: Order delivery

Use case	Order pickup
Actors	Customer named Alice, Waiter named Bob
Initial assumptions	Alice has made an order for pickup earlier, which has now been cooked. Alice has not yet paid for the order. Bob is logged into the DROM system client.
Flow of events	1. Alice walks into the restaurant and asks Bob for her order. 2. Bob checks that her order is done in the DROM system client. 3. Bob sees that Alice has not yet paid for the order. 4. Alice pays for the order. Bob enters that Alice has now paid for her order. 5. Bob now hands over Alice's order. Bob enters that her order has been picked up. 6. Bob sees that the order is done and archives it.
State of completion	Alice's order is finished and has been archived.

Figure 49: Order pickup

26.2 Responsive UI screenshots


DROM Client
Views: Chef Delivery Manager Waiter Show only pending orders
Actions: Get orders from Web API Create new order Log out

Order id	Order date	Order type
23	5/12/2016 5:44:13 PM	For delivery

Table number	Customer name
0	Alice Alison

Email	Phone
Alice@Alison.com	12345678

Address
Rued Langgaards Vej 7
2300
København S

Items on order:

2xLiverpool

Notes
Call me on my cellphone when you arrive.

Events to execute:

Total price:
120

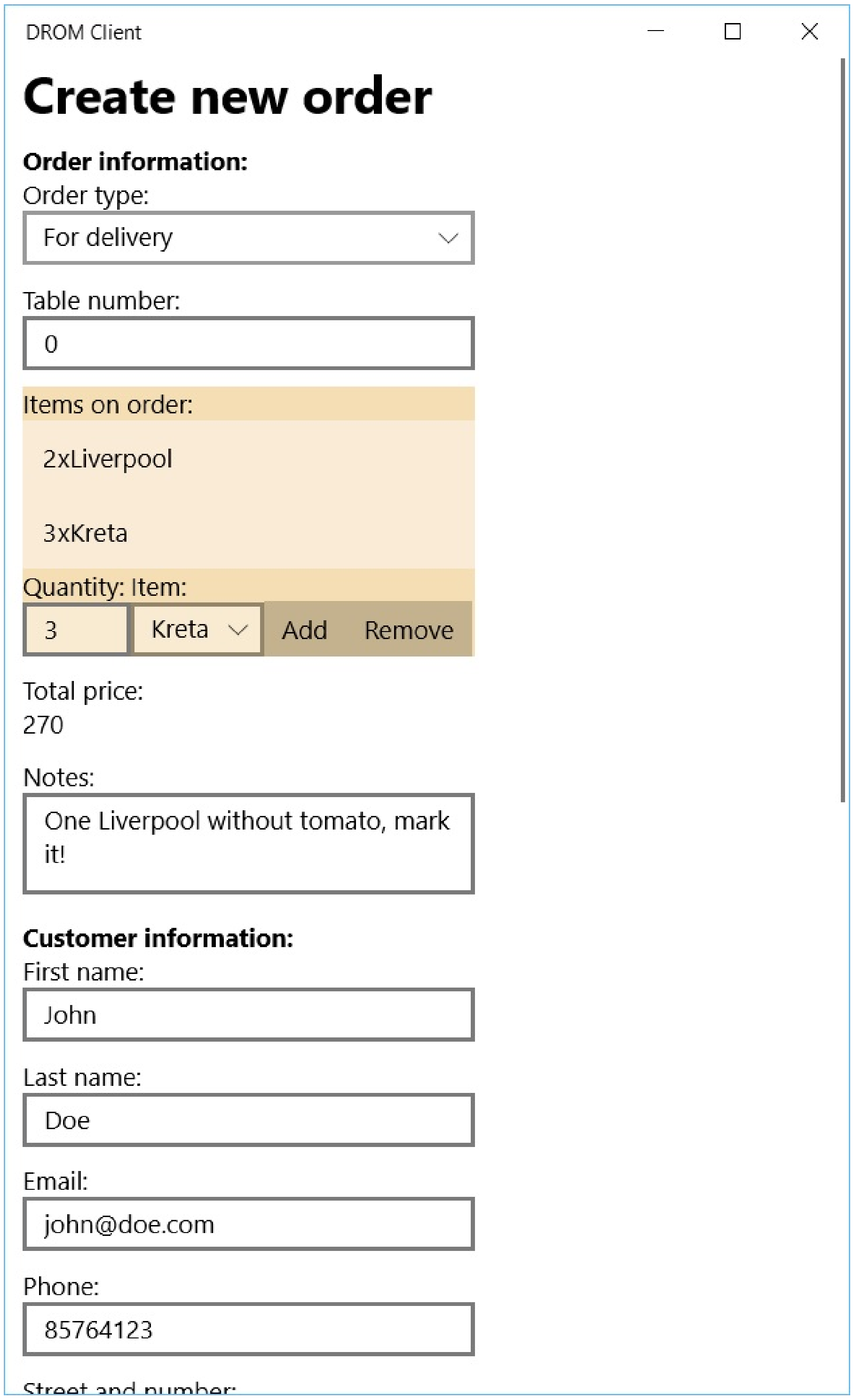
Order id	Order date	Order type
25	5/12/2016 5:54:00 PM	For serving

Table number	Customer name
5	

Email	Phone

Address

Figure 50: Screenshot of the Order overview screen with minimum width



DROM Client

Create new order

Order information:

Order type: For delivery

Table number: 0

Items on order:

2xLiverpool

3xKreta

Quantity: 3 Item: Kreta

Total price:
270

Notes: One Liverpool without tomato, mark it!

Customer information:

First name: John

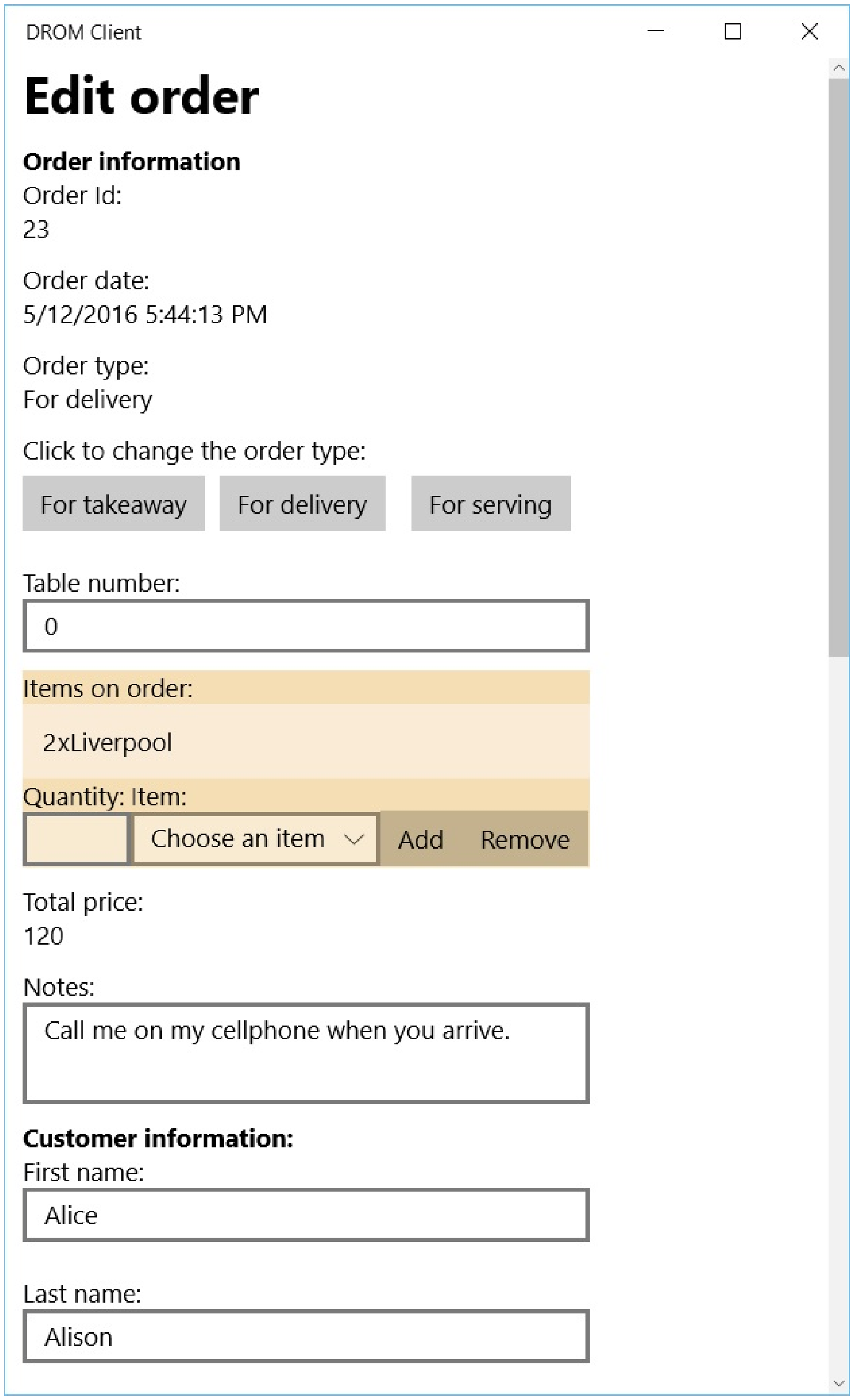
Last name: Doe

Email: john@doe.com

Phone: 85764123

Street and number:

Figure 51: Screenshot of the Create order screen with minimum width



DROM Client

Edit order

Order information

Order Id:
23

Order date:
5/12/2016 5:44:13 PM

Order type:
For delivery

Click to change the order type:

Table number:
0

Items on order:
2xLiverpool

Quantity: Item:

Total price:
120

Notes:
Call me on my cellphone when you arrive.

Customer information:

First name:
Alice

Last name:
Alison

Figure 52: Screenshot of the Edit order screen with minimum width