Resource planning

Joachim Kirkegaard Friis

Kongens Lyngby 2012
IMM-BSc-2012-18

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk IMM-BSc-2012-18

Summary (English)

The goal of the thesis is to implement a proof of concept for teachers on DTU to get information about course allocations and the ability to get their course allocated. The allocations should work on a resource planning model. This model should try to achieve to satisfy given soft constraints and should still not contain any conflicts with the hard constraints. The soft constraints is that the distance between a given allocated lecture and exercise rooms should be as short as possible.

The implementation was done in the .NET framework using the ModelViewControl pattern, which enables the easy use of F# for the model implementation. The view and control was implemented in XAML and C# within a usual WPF application environment.

The sheer size of data caused difficult debugging and testing which had to be resolved by applying smaller basic test samples, which still showed the functional integrity.

The end product was a simple GUI containing only the most important functions, and serves as a prove of concept that an implementation would benefit the teachers, lecturers or any administrative people involved in the business of allocating over 200 courses per semester.

Summary (Danish)

Målet med specialet er at gennemføre en proof of concept hvor lærere på DTU kan få oplysninger om kursustildelinger og evnen til at få deres kursus tildelt. Tildelingen bør virke på baggrund af en ressource planlægning model. Denne model bør forsøge at opnå at stilfredsstille givet bløde constraints og bør stadig ikke indeholde eventuelle konflikter med de hårde contraints. De bløde contraints er, at afstanden mellem en given tildelt forelæsningslokale og øvelsesrum skal være så kort som muligt.

Implementeringen blev udført i. NET Framework ved hjælp af ModelViewControl design pattern, som gør det muligt at nemt bruge F# for model implementering. Viewet og control blev gennemført i XAML og C# inden for en sædvanlig WPF applikation.

Alene størrelsen af datamængden forårsaget vanskelig debugging og test, der skulle løses ved at anvende mindre grundlæggende test smaples, der stadig viste den funktionelle integritet.

Slutproduktet var et simpel GUI, der kun indeholder de vigtigste funktioner, og fungerer som et proof of concept, at en egentlig implementering ville gavne de lærere, forelæsere eller enhver administrativ person inblandet i at afsætte over 200 kurser pr semester.

Preface

This thesis was prepared at the department of Informatics and Mathematical Modelling at the Technical University of Denmark in fulfilment of the requirements for acquiring an M.Sc. in Informatics.

The thesis deals with resource planning in context to the current solution of the one DTU's uses to allocate lecture rooms and exercise rooms. The current solution doesn't enable teachers to view current allocation. This motivates us to create a visualization solution and long term storage in terms of an SQL database.

The thesis consists of description of the problem, a description of the database, a description and analysis of the some resource planning models, a description of the implementation and tests.

Lyngby, 25-June-2012

[illegible]

Joachim Kirkegaard Friis

Acknowledgements

I would like to thank my supervisor, Michael Reichhardt Hansen, for his constructive advice, patience and answers for my many questions.
Also for supporting me with lots of general information and guidance in terms of ideas for implementation and different resource planning models.
And also for motivating me to explore more technologically advanced areas of general software technology, such using F# as a strong tool for managing the resource planner, SQL server and other general functions for the end product. This has opened new doors for me in context to a very Java dominated world of software technology studies on DTU.

I would also like to thank my family and friends for understanding my priorities during the work of this project.

Contents

Chapter 1

Introduction

1.1 Short introduction to Resource Planning

The problem when talking about resource planning, is to allocate your resources effectivly and in a reasonable manner. A bad resource allocation can result in great losses in money and time. A good resource planning tries to achieve and utilialize some sort of optimization. This motivates us to investigate on how we can implement a solution and monitor and analyze its performance.

The Contraint Resource Planning problem is seen in many applications such as: tele communication, memory allocation and time scheduling in general. As a result, substantial research has been done on this matter because of it's broad relevans in many technological applications.[1]. For this reason it serves as a motivation to strive towards an implementation and experimentation on one of the known solutions to a problem, which now leads us to discribe the problem.

The problem in this case is to optimize on the current method of allocating lecture and exercise rooms used by the administration of DTU[2]. Some courses e.g. has lectures in one end of campus and exercises in another, which leads to

[1]source 1 - 1 Introduction, 1.1 related works

[2]Danmarks Tekniske Universitet, *Technical University of Denmark*

up to a 15 minutes walk between lecture and exercises - which is time lost and wasting money in terms of the payof an assistant teacher.

1.1.1 The environment of the problem

The enviroment in which the problem resides is basicly the campus of DTU. A course has a lecture in a given time table, and then exercises generaly in the same. A day consists of two timetables. i.e. monday morning from 8 am to 12 am is "1A" and 1 pm to 5 pm is "2A". The lecture room and exercise room(s) are to different locations, which causes the long distance probability.

The possibility of an optimazation stands clear, since the current scheme used, can also cause conflicts on different departments of the university. Meaning that there is a chance that one course might have to share an exercise room or lab if it's under two different deparments. This has a simple solution, but there exsist another problem in which the solution will not be as simple. The problem at hand is about optimizing on the distance between a lecture room and an exercise room or lab. Because as mentioned - many students and lecturors waste time on transport from a given lecture room to the assigned exercise room or lab.

1.1.2 Goal

So when assigning a room for a course, you have to take several variables in consideration. As mentioned the most straight forward and clear constraint is to avoid conflicts in the allocation of rooms - meaning that two courses cannot have the same room assigned at the same time table. This is a hard constraint, which means it's a constraint we have to follow strickly and that a solution cannot conflict with it. The optimization on the distance between lecture room and labs is a soft constraint, meaning that a solution should strive towards it.

The implementation should also contain a GUI[3], which should serve as a simple overview with basic functions. The implementation we should end up with shouldn't serve as representation of an end product, but more as a proof of a concept. Meaning that the GUI should be simple and more or less serve as a visualization ofa tool and the end solution of an implemented model. It should work on ModelViewControl as the design pattern - in which the View is implemented in XAML[4], Controller in C# and the model in F#.

[3]Grahpical User Interface
[4]eXtensible Application Markup Language

Furthermore we want to store the information about courses, rooms and allocationg in an SQL database. This should serve as a simple way to handle and create sources and a proof of concept since we want to implement it in F#. These should show to possibility of implementing a tool in an unusal enviornment like F# .

With this said - our goal is to achieve a solution that implements an optimized model and provide teachers a tool for viewing information about allocations for their course or other. This should happen through research and evaluation of different solutions and a steppwise optimization on the chosen solutions.

1.2 Requirements Specification

This sections is going to describe the requirements for the end product. The requirements will be listed and then be discussed in a detailed manner.

The overall idea is to implement a solution, which can allocate courses one lecture room and the needed amount of exercise rooms. This implementation should work on data from an SQL database, to show the concept of an F# implementation coexisting with SQL.

The phrase ”end product” doesn’t emphasize that it’s going to be an actual end product but rather a proof of concept. The requirements are as listed below:

- An optimization on how to allocate resources, by implementing a solution which can allocate resources computationally.
- A simple GUI implementation capable of showing the user the relevant information.
- The GUI should contain search ultilities for rooms and courses.
- Options for different solution allocation models.
- Create ultility for a course.
- An SQL database implemented implemented with F# .
- The database should be of such quality that is satisfies the first four normal forms.

The allocation model should work on the following hard constraints:

- A course must have one lecture room and the necessary amount of exercise rooms allocated.
- Two courses cannot have the same room allocated at the same timestamp.
- The number of students allocated for a course must not exceed the given room capacity.

The soft constraint:

- An allocation should offer an optimized suggestion in terms of distance between lecture room and exercise room/labs.
- A course with larger efficientcy[5] should be prefered over a course that is less efficient.

The GUI should be simple and straightforward to use. Basic functions such as: Adding a course, searching for a course or a room. Searching for current allocations and testing the models and their performance.

The SQL database should work in the background with the model and should be optimized in terms of normal forms.

The design pattern of the GUI implementation should follow the ModelView-Control pattern, in which the View is implemented in XAML, the controller in C# and the model in F#.

[5]This denotes how much room a given course utilizes

Chapter 2

Database design

2.1 Introduction

In this chapter we will discuss and illustrate the solution, in which the database is constructed. After this we must analyze the problem at hand. The illustration should not stand alone as a description of the solution. The discussion of normal forms must be applied to the solution, which will be done as well.

If we take a look back at the requirements specification, the database must fulfill the following points:

- An SQL database implemented implemented with F# .
- The database should be of such quality that is satisfies the first four normal forms.

The overall implementation, hence the first point states that is must be implemented with F#, this will be discussed in the later chapter 4.

A summary will conclude the chapter by describing given discoveries and achievements in relation to the requirements.

2.2 Description of the tables

Since the problem description holds the overall picture of the problem, we could do a setup. The database should hold all the information regarding courses, lecture rooms and exercise room. This information should be defined in a reasonable way, such that updates shouldn't cause any problems and no conflicts should arise. Applying this ability will be discussed in the section about normal forms.

2.2.1 Courses

The first entity at hand is a course. A course should have a unique identifier, which could be a course number or a course name. Seeing that courses at DTU all have a unique course number, it is clear that the most simple solution is to inherit this property. Another property of a course is the amount of students it has assigned. This should be a simple integer value. The last property would be to give the course a name, so it's easily identifiable. And with this we can denote the following for a course:

- Course - Id number, number of students and a name.

At first glance it looks as if this structure doesn't hold any information regarding it's allocated rooms or resources. This is a deliberate choice, which will be discussed later in the section about the analysis.

2.2.2 Rooms

The second entity will be the rooms. Rooms should contain two main properties. That is weather if it's a lecture or exercise room. This should be presented by a string, applying either "Auditorium" or "Holdlokale"[1] . A room at DTU has a room number of 1 to 999. This though cannot be used as a unique identifier, since there might exist rooms by the same number but in different buildings. This means we should introduce a building number as a property. And an id number should then serve as the unique identifier. The size of the rooms, which should work in terms of comparing it with the number of students of a given

[1]other description of an exercise room is used due to the scheme of the data provided. Please refer to Appendix A

course, is going to be presented by an integer value. With this we can denote following of a room:

- Rooms - Room Id, Room number, building number and size.

This leaves out a very important property - the position of the room. The importance lies with optimization model in which we use the positions. This brings us to the next entity - Locations.

2.2.3 Locations

The position of a room can be presented in different ways. The most reasonable here is to use coordinates. These coordinates should be evaluated as though we put a grid over the map of DTU. Since the campus of DTU has a square-like shape, implementing this should be straight forward. And as we have these coordinates, calculating the distance between two rooms is simple, by applying triangular distance calculations.

Now, evaluating where a room is precisely in practice is another matter. The official map of campus of DTU doesn't state where particular rooms of a building are at. So we have to settle with the position of the building in which the room resides. So the location entity should be as following:

- Location - Building number, coordinate x, coordinate y.

2.2.4 Allocations

The last entity is allocations of a room. When we talk about an allocation, we simply want to know which room that is allocated. For this we need the associated unique room number, as described earlier. The course of which the room is allocated to will also be identified with its number.
This leaves us with a new property, which is a the schema table. A schema table, as described in the problem description, on DTU denotes the time of the day - monday through friday. A day in divided into two parts, morning and afternoon. So in total there is ten different schema tables for one room. So the following should describe an allocation:

- Allocation - Allocation Id, room Id, course number and schema location.

2.3 Entity-relationship diagrams

To further illustrate the solution, E-R diagrams will be illustrated to show the overall relations between the data entities.

This first diagram 2.1 will center it's focus around an allocation.

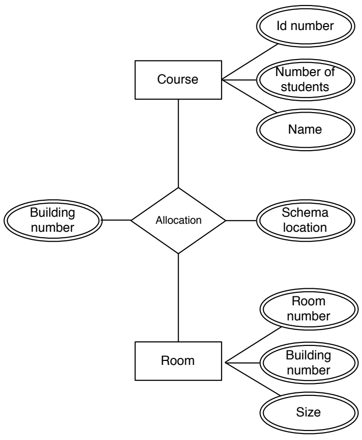


Figure 2.1: E-R diagram 1

This last diagram 2.2 shows the relation between a location and its associated entity:

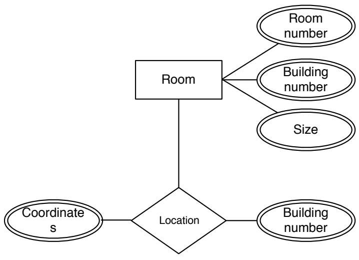


Figure 2.2: E-R diagram 2

2.4 Analysis

This section will discuss the given solution in relation to normal forms and an estimate on how it will perform.

2.4.1 Normal forms

Normal forms, when talking about database design, serve as a useful tool to determine the overall quality of a given design. One typically wants to achieve the highest type of normal form as possible, which ranks from 1 to 4[2] The normal forms generally focus on dividing entities into more detailed ones. These divisions will optimize on the general operations done on a database, such as: delete, insert, modify and update. The optimization lies with the amount of updates you will have to do upon calling these operations and the level of consistency.

Below are the normal forms described in short:

- The first normal form accepts a table or database for this matter, which presents the data in a faithful manner with no repetitions. [3]
- The second normal form accepts a database, which is foremost in first normal form. Furthermore properties or attributes, which are not contributing as unique identifier, should appear as dependent on a unique identifier as a whole. This is typically avoided by dividing a table into two new table, which now has that non-prime attribute as a unique identifier. [4]
- The third normal form accepts a database, which is in second normal for. Much like the second normal form, we're talking about the non-prime attributes. These must be dependent on the unique identifier, but nothing more. [5]
- The fourth normal form concerns on multivalued dependencies, where as the first three are concerned on functional dependencies. Multivalued dependencies can be described as different data sets, which on their own don't describe an attribute, but together describe it. The fourth normal

[2]higher normal forms can be achieved, but in this case we will only discuss these four.
[3]defined by E.F. Codd, 1970
[4]defined by E.F. Codd, 1971
[5]defined by E.F. Codd, 1971

form accepts a database containing multivalued dependencies, which are contains a key. [6]

2.4.2 Verifying the design

Now that the normal forms are in place, it's time to look back at the database design. The database will be now be compared and analyses to each of the described normal form, starting with the the first normal form.

The unique identifier for each table in the database, should serve as a tool to satisfy the first normal form. Though the data it self will serve as a variable to the first normal form, which is concerned about repetitious entities.

Taking the second normal form at hand, we see that the tables reside independently. No attributes are contained in one table and then in another. If we took the following entity solution as on example: Allocation: courseId, courseStudents, roomNr, building,... basically with all the attributes we could think of. We then see that updating the database would cause that we would also have to update other tables than just the allocations table. This would cause inconsistencies on updates, creating new and deleting existing entities. This is the reason for dividing the tables into reasonable separate tables for each information of interest: courses, rooms, allocations and locations.

The database design contains non prime attributes which are only dependent on unique identifier. Lets take a course as an example. Here we have the entity was a unique identifier, course name and number of students. We see that this entity has a unique identifier, which contains two non prime attributes: name and number of students. These are only present in the course entity and therefore satisfy the third normal form.

The last normal form, the fourth, as mentioned concerns about multivalued dependencies. If we had decided to describe an allocation by the coursename as an identifier, we would have had to take multivalued decencies into concern. But since an allocation has a unique id number, the allocation is then not dependent on the course number, and then has no multivalued depencies.

[6] defined by Ronald Fagin, 1977

2.5 Summary

In this chapter we described the database design system, by discussing the problem environment and illustrating the solution. The design was then put though a quality check by comparing it with the four normal forms described.

Chapter 3

Resource Planning Models

3.1 Introduction

Now that the database is described, it is time to discuss the the problem it self. Different solutions will then be analyzed and compared to the problem, in which a solution is going to reveal it self. The end solutions will then be described.

3.2 Analysis

3.2.1 The problem it self

The problem is to allocate courses a lecture room and then exercise rooms, either one or more if necessary. Referring to the requirements specification, we see that the allocation should satisfy both the soft and hard constraints. Here we of course want to prioritize the hard constraints over the soft.

So when allocating a course a room, we then first and foremost want to find a room, which satisfies the course in terms of the amount of students and is available, i.e. no other course has the room already allocated. When this is sorted out, the course still need exercise rooms to be allocate, which is where the soft constraints should come to work.

The exercise rooms should be as close as possible to the given lecture room. So determining which course is most suitable for a room is in the interest of the problem. This reveals a complex puzzle with allot of pieces. And since there are about 250 courses per semester and just as many rooms to accommodate, it would be a rather waste to allocated this by hand, i.e. without any computational effort.

So how do we solve this problem? This will now be discussed in the next section about general solutions for the resource planning problem.

3.2.2 Different solutions to the Resource Planning Problem

As mentioned in the introductory section, the resource planning problem is a well discussed subject and has many solutions. The reason for the many possible solutions is that is a NP-complete problem.

3.2.3 Brute forcing and the n queen problem

The n-Queen is a problem which cannot be solve in polynomial time, and also NP-complete, and therefore has solutions which are quite naive in a sense. The problem it self consist of a $n \times n$ board with n queens which have to be placed on the board such that no queen is able to attack another queen. The illustration 3.1 below should clarify. The solution to this problem relies on backtracking.

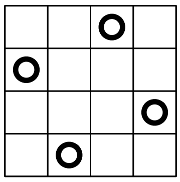


Figure 3.1: Illustration of n queen problem, n = 4

It starts by placing a queen at a start point in the first row, and then recursively places the next queen in the next row. It then backtracks if there's no available places in the given row. And continues until is yields a result.

The worst case running time for this algorithm is pretty practical in small problems. The worst case relies on the number of different solution the algorithm can potentially yield. So for a 8×8 board, there would be $8^8 = 16,777,216$ possibilities. And for larger problems the time complexity grows exponentially.

This problem relates to the resource planning problem, since it concerns about allocating resources such there are no conflicts to a given constraint. The relevance to the allocation problem will be discussed in a later section in this chapter.

3.2.4 Simulated annealing

Simulated annealing is a model which works on finding or at least trying to find a global optimum in wide range of different problems. It's name originates from annealing in metallurgy i.e. a specific heating method of a piece of metal in which the atoms gain randomly states of higher or lower amounts of energy. The simulated annealing then replaces the current solution of atoms with random energy with the heuristic of trying to get a more evenly distributed energy sample. It does so by selecting a random *state* and checks it's neighboring states and determines if there's another global minimum/maximum. The pseudo code looks as following **??**:

Algorithm 1 General application of simulated annealing

```
1: $s = s0, e = E(s)$
2: $sbest = s, ebest = e, k = 0$
3: while $k < kmax \wedge e > emax$ do
4:   $T = temperature(k/kmax)$
5:   $snew = neighbour(s)$
6:   $enew = E(snew)$
7:   if $P(e, enew, T) > random()$ then
8:     $s = snew, e = enew$
9:   end if
10:  if $e < ebest$ then
11:    $sbest = snew, ebest = enew$
12:  end if
13:  $k = k + 1$
14: end while
15: return $sbest$
```

This version terminates when either there's no more time left or it has done the max calculations. All these parameters are set before the calculations. And the longer it runs the closer it should get to a global maximum/minimum, depending on the heuristics.

3.2.5 Direct heuristics and the SCHOLA system

Another method to the resource planning, or in another sense time tabling problem, would be to use the SCHOLA system. Which is a slight abriviation to the bruteforce method. The difference though is that is works on a strategy, which removes the naivety of the n queens solution.

The SCHOLA system uses simple rules and direct heuristics to allocate resources by comparing most *urgent* lecture, determined by how it fits according to the constraints. Allocating a lecture is done by the three following steps:

A Assign the most urgent lecture to the most favorable period for that lecture.

B When a period can be used only for one lecture, assign the period to that lecture.

C Move an already-scheduled lecture to a free period so as to leave the period for the lecture that we are currently trying to schedule.[1]

This method will at most work on the first two steps. But once it cannot satisfy them, it will then start to swap allocations around to reach a solution. Step C would serve as a sort of backtracking tool, to where step A might have done some bad decisions.

This method needs some modifications though, if we want it to fit our constraints, which will be discussed in the next section.

3.3 The solution

Now that the chosen model solutions have been described, it's time to put them in relation to the allocation problem. It will be done in the same order as in the previous section.

3.3.1 n Queens

Comparing the n Queens problem to our problem, we see some similarities. The problem is in essence to allocate resources by avoiding conflicts in previous allocations. This relates to our hard constraints, such as not placing two courses in the same time table in the same room, but also not to put a course in a room that's too small. If we design a solution which places courses working on these constraints, we should end with a rather naive and bruteforcy but valid design. So how should we so this?

If we look back at the boards in the n queens problem with columns and rows, we can then redesign it. If we say that each row represents a room and each column a time stamp, we then get the following 4.1:

[1]Schaerf - 1999 - A Survey of Automated Timetabling, page 95 to 96

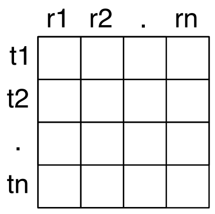


Figure 3.2: Illustration of n queen in relation the allocation of courses

We can then place a course on each coordinate once, if it's for lecture rooms. And if we want to allocate exercise rooms for a course, we then place that course on a coordinate until it's amount of students is *used up*, e.g. a course might have to get more than one exercise room allocated, since exercise room are generally smaller than lecture rooms.

The worst case running time for this particular solution is rather different from the standard n queen problem, because of the constraints. Let say we place a course on (r_1, t_1). This will still enable us to place the next course on the next table of the same room i.e. (r_1, t_2). This gives us the worst case running time of $O((r \cdot t)!)$. But by applying simple heuristics to this solution, we can achieve more desirable computation times. One could be to sort the courses by the number of students and the rooms by their capacity, this would yield as few conflicts as possible and therefore have a complexity of $o((r \cdot t)!)$. Another could be to allocate course a room with it's capacity just enough to satisfy the course.

This bruteforce algorithm will then terminate once every course is placed on the board, while the hard constraints are met.

The limitation to this solution, is that it doesn't optimize it's allocations following the soft constraints. But we are sure, that it will yield a solution, in which all of the courses are allocated.

3.3.2 Simulated annealing

To follow up on our soft constraints, i.e. the distance constraint. We will then use simulated annealing, modified to fit the allocation problem. To optimized on the distance between a lecture and exercise rooms of a course, we must compare eventually free rooms and already allocated rooms with rooms of a given course.

The reason for this, is that the bruteforce method only cares about allocating course into available and capable (in term of capacity) rooms. This is where the naivety of the solution comes into play and motivates us to optimize on the distance between lecture and exercise rooms.

This is done as following 2

Algorithm 2 Simulated annealing applied to the allocation problem

```
1: $k = 0, t = 0)$
2: while $k < kmax \wedge t > tmax$ do
3:     $c1 = randomcourse, c2 = randomcourse$
4:     $canAllocateCloser(c1), canAllocateCloser(c2)$
5:     if $notTheSame(c1, c2) \wedge isMoreEfficient(c1, c2)$ then
6:         $swapAllocations(c1, c2)$
7:     end if
8:     $k = k + 1$
9:     $t = takeTime$
10: end while
```

Here we choose two random courses. We then check if there free room for either course closer to their lecture room[2] . If not, the two courses are then compared to their efficiency. The efficiency of a course should be distance/person. This means we want to achieve the shortest distances for the biggest courses.

The simulated annealing will terminate when either time has run up, or if x amount of comparisons have been done.

By running simulated or any other solution which works on the soft constraints, we want to achieve an overall increased effectivity compared to the normal bruteforce algorithm and maybe even compared to the allocations of DTU at a given semester. The results will reveal themselves in the later chapter about tests.

3.3.3 SCHOLA

After the description of the two previous solutions. It stands clear to research and evaluate a solution, which which follows both the hard and soft constraints and the same computation, i.e. it's all done simultaneously. This is where the

[2]The function canAllocateCloser checks if there's closer available rooms and reallocates them if there are.

SCHOLA system comes into play. In this solution we hope to achieve a solution, which satisfies all of the constraints by applying simple heuristics.

So modifying the SCHOLA system to fit our problem will take some work. First we have to setup some heuristics, which will satisfy our constraints. Not only the hard constraints but also the soft constraint. Let's start by listing the following solution. The steps are described in a more detailed manner compared to the earlier described system in the section about the generic solution.

A Allocate a course X to the largest lecture room available.

B If there are no available rooms in the chosen time table, compare X with the currently allocated courses. Use step A on the least suitable course.

C Find the closest building to the one where the lecture resides (can be the same building)

D If there are no available exercise rooms in the given building, compare the current allocations. Delete the current allocation of which if the X is more suitable and reallocate the deleted allocation Y.

E If there are no available exercise room at the given table, reallocate the lecture room in another available time table.

In step A, we have a current course C_1 which needs a lecture room and the necessary amount of exercise rooms. If there is no lecture rooms available (step B) in the building B, it will then start to compare C_1 with the currently allocated courses.
If C_1 is more suitable than a course C_2 already allocate, the allocation of C_2 is deleted and C_1 is then allocated. If not, we then jump to another building than B and perform step A for course C_1

After step B, a course C_3 is now allocated exercise rooms. Step C finds the closest building and then checks if there are available rooms. If not, it then step D will compare the currently allocated courses much like in step B.

This algorithm will give us a solution, but it doesn't insure us that all courses are allocated. This is because of the somewhat naive heuristics. If we take the following illustration at hand.

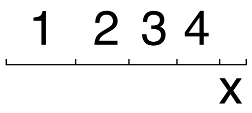


Figure 3.3: Illustration of allocation problem with SCHOLA

We see that if a building has X amount of space in a given timetable and Y courses allocated. There is then going to be some place left unused, if there is no other course which can occupy the space left due to capacity constraints. This sums up to be a problem about finding a sum of numbers and get closest to a given sum. Lets say we have a set S of numbers $2, 3, 4, 1, 2, 5$, how do we find the use of these number (we can only use each number once), which combination gets closest to a given number? This is a fairly complex problem, which only contains bruteforce solution, where you would take the numbers by random and compare them with other combinations.

This will not be taken into account in the end solution since, we are just interested in getting results in terms of the soft constraints in this solution.

3.4 Summary

Before evaluating the solution, we determined the constraints and the overall complexity of the problem. We then started by describing the general solutions relevant to this problem. Then we analyzed them to then modify them to the problem it self. Both the bruteforce and SCHOLA methods had it's pros and cons. While the bruteforce found a solution, it is very inefficient and with a exponential worst case running time. The SCHOLA method on the other hand follows all of the constraints but doesn't insure all courses get allocated.

Simulated annealing would be a good choice if we want a solution, which meets all the hard constraints and tries to satisfy the soft.

Chapter 4

Implementation

4.1 Short introduction

We now have the solutions at hand, which we are now going to implement using the database previously described in the database design chapter 2. It is first and foremost important to evaluate the choice of technologies for the database, the model and the GUI. The implementation it self will then be described and discussed in terms of limitations and restrictions which were taken. In relation the requirements specification the implementation should would meet the following:

- A simple GUI implementation capable of showing the user the relevant information.
- Search utilities for rooms and courses.
- Allocation options for different solution models.
- Create utility for a course.

The given requirements should serve as guide for the this chapter. The first point describes a GUI, which should be capable of different operations. The third point says we have to implement the different solution, which is going to be the bruteforce solution and SCHOLA described in the previous chapter 3.

4.2 Choice of technologies

4.2.1 Design pattern

To start of, it's most reasonable to discuss the design pattern. The most convenient design pattern for this solution is the ModelViewControl design pattern. The reason for this, that we can solely separate each language for each part of the design. It will work well in collation with the model implementation, since it will be coded in F#. This gives us the ability to create nice and concrete expressions of the functions, so they can be evaluated out of context to the rest of the program.

4.2.2 Xaml, C# and F#

The solution will be implemented on the currently newest version of the .NET 4.0 platform. This gives us a variety of tools to implement the different parts of the program. As mentioned the model will be implemented using F#.

The reason for this is to explore the advantages of F# for designing the model for any given software implementation. An F# implementation will give us nice and clear function declaration in a first class order, compared to other different languages, such as OO[1] languages. For further improvements on the end solution to the project, the F# implementation would motivates us to design a multi threaded system, since we have avoided mutable values as much as possible. Another advantage of F# is its multi paradigm language property, meaning that it can exist as a declarative functional language and still express the same object oriented structures as we would in e.g. C#.

The control of the visual part will be coded in C# for purposes of keeping the model implementation separated from the the controller.

The visual part will be coded in XAML for simplicity and general flexibility of the GUI declaration.

[1]Object Oriented

4.2.3 Linq to SQL

So why do we want a database and not just work with simple unique objects, as we do in modern programming? Objects, when talking about programming in general, are unique and exists independently to other objects, and would just for that sole reason by an attractive choice of data representation. But in contrast to objects a database would provide us with longterm and reliable data storage - which relates to the problem at hand. Although working with a database as a program designer does raise some problems. A sort of missing link would exist between code and a representation of ones queries. This brings us to the goal of this section.

Communicating with the SQL database will require tools for the program to execute queries and should still be a reasonable way to code and design the solution. The Linq to SQL library will provide us with tools to construct such queries. In relations to other means of doing this, e.g. different API's in which you would declare queries using a string formats, Linq will allow us to easily describe the queries and provide us with syntax checking in the Visual Studio coding environment.

The properties of Linq seemed very attractive and was therefore the chosen tool for this purpose. And by this together with F# we want to achieve nice and concise declaration of the relevant queries.

The typeprovider used, which will be further described later on, will allows us to connect to an SQL database. And all we will need to do, is to open the relevant library and declare a type, in this case *schema*. This will be our SQL connection containing a connectionstring, e.g.:
"Data Source=MY-PC \DatabaseName;Initial Catalog=Database;Integrated Security=True".
The security tag declares to use the integrated Windows authentications utility.

```
open System.Data.Linq.SqlClient ;;

type schema = SqlDataConnection<"CONNECTIONSTRING">;;
```

Once we have achieved the connection to the server, the use of DataContext will allow us to access tables within the database:

```
let db = schema.GetDataContext()
```

In this case *db* will contain all the relevant information, in which we can get all of our tables in F#.

The ability to access tables will now allow us to delete and add entities within a given table. For this we need to get the type of an entity such as a room. This is done as following:

```
type room = schema.ServiceTypes.Rooms
```

The most important and visible advantage of Linq to SQL is revealed as we want to construct our queries. A query in F# is declared as a function containing a sequence-like structure. The function will then return a lazy *IQuerable* sequence, meaning that the query is not materialized until needed. An example of a query would look as following:

```
let getRoom id = query { for room in db.Rooms do
                         where (room.Id = id)
                         select(room)
                         exactlyOne
                         }
```

[2]

The Linq.QueryBuilder contains a comprehensive documentation in which the usual key components of an SQL query are described.

Taking the examples illustrated above shows us that Linq to SQL does provide us with queries that are easily constructed and easy to understand.

4.2.4 The TypeProvider

A typeprovider is an essential part of programming in F#. Type providers will eliminate the barrier in terms of type inconsistency between the program implementation and exterior sources like a database. The types of an SQL database and F# aren't necessarily the same. Take an F# boolean value, this is denoted as a *bit* in the SQL database. The integrated F# data TypeProvider will make sure that we maintain type consistency.

The TypeProvide of F# also works well together with Linq, which is obviously very desirable. The typeprovide in this case is the sole reason we can do this and connect to the database as described before.

[2]Further explanation of the query it self will be in the later section about queries

4.3 Overall design

The overall design of the model will consist of different libraries for different purposes. The first library will be the database library `ResourceDatabase.dll` that will contain the functions, in which we create our database and queries. The second will be `ResourceAllocator.dll`. This will contain the implementations of the resource planning models. The third will be `DataIO.dll` containing the text processing function in which we go from text to adding entities to the database. The last library will be the `ResourceStatistics.dll` containing function to obviously calculate the statistics for test.

The idea is to get these libraries to work together and also contain function for which the GUI shall use. The functions are mainly going to be information in the database in a string format.

4.4 Database

Now that we have the technologies and the overall design described sorted out. It is time to implement the database using the tools described earlier.

4.4.1 Setup and creating the database

Although the tools provided are very versatile and useful, they have their limits. You are e.g. not allowed to setup the SQL server it self within the code designing phase. This has to be done in the SQL setup environment. Though when this is done, we can access the database much like described in the section about Linq to Sql section:

```
let connString = @"CONNECTIONSTRING"

let conn = new SqlConnection(connString)

conn.Open()
```

Once the connection is established the next step is construct a query, which handles the connection and a query string, which should contain a formal SQL query. If we then take a look back at chapter 2, we can take a room as an example. Here we want: unique identifier integer, building number string, room number string, property string and a size capacity integer. The following query would look as following:

```
execNonQuery conn "CREATE TABLE Rooms (
Id int NOT NULL,
BuildNr varchar(50) NOT NULL,
RoomNr varchar(50) NOT NULL,
Prop varchar(50) NOT NULL,
Size int NOT NULL,
PRIMARY KEY (Id))"
```

Node that types are declared as SQL type and not necessarily F# types. This has to be taking into consideration, especially for the strings - since they have a limited length. But for this example i will not raise any problem, though for a course name we might want to consider a longer string, since we're not sure how long a name of a course is. This means we would declare the name as a string in the form of : "*varchar(MAX)*".

The *execNonQuery* is a function which receives an SQL connection and the query string and returns the type unit. The function is declared as following:

```
let execNonQuery conn s =
        let comm = new SqlCommand(s, conn, CommandTimeout = 10)
        comm.ExecuteNonQuery() |> ignore
```

This issues an SQL command using the ExecuteNonQuery operation supported by SQL. This query is often used when you want alter a given database but you don't necessarily want to receive any information in return.

4.4.2 Queries

To talk about queries more in a formally fashion than we did earlier in the section about Linq to SQL, we can start by describing the overall structure of a query using Linq to SQL. The following illustration should clarify:

$$query = \{\ queryexpression\}^{3}.$$

With this said, we will now take a few implementations of queries as examples, and explain their structure and functionality.

[3] FPBook, 11.8.1 Query Expressions

The first one isn't exactly a query, but a very essential part of the database. Adding entities to the database is implemented the following way:

```
let addCourse courseId courseName students =
        let c = new course(CourseId = courseId,
        CourseName = courseName, Students = students)
        db.Courses.InsertOnSubmit(c)
        db.DataContext.SubmitChanges()
```

This function is of the following type : `int -> string -> int -> unit`. It creates a course provided by the type course. It then adds it to the database and tells the datacontext that changes have been made. The consistently of the unique identifier here relies on the input data. Though if we wanted to add a room, we would have to implement our own rather naive solution. This would be a function which counts the number of rows in the *Rooms* table and applies that as a key. This function would look as following:

```
let nextRoomId() = query { for room in db.Rooms do
                        count }
```

This does though raise inconsistencies in terms of update and deleting entities in a table. Say we would want to delete one allocation and substitute it with another. This will cause us to rely on another method than the one for the rooms. So having a mutable integer value, which is incremented every time we add an allocation will serve as a solution to the problem, although it has it's limits in terms of the boundary to the integer type, in this case the *Int32*. This will cover at most 2,147,483,647 allocations, which will suffice.

So now that we can add data, lets explore the query for getting an entity by it's unique identifier:

```
let getRoom id = query { for room in db.Rooms do
        where (room.Id = id)
        select(room.Id, room.BuildNr, room.RoomNr, room.Prop,
        room.Size)
        exactlyOne
        }
```

[4]

This function is of type: `int -> IQuerable<int*string*string*string*int>`. It iterates through the Rooms table and selects the row where the desirable id's

[4]The example doesn't represent the end implementation. In the real implementation the indentation is important. The QueryBuilder objects must be in the same indentation within the expression itself

equal. The query object *exactlyOne* insures that we don't have multiple identical entities, which would violate the first normal form as described in chapter 2.
So now that we can add and get entities, we would now be able to delete. This should be done by using the previous techniques used in the add function and get function. Once you have and entity's id, you can delete it and submit changes.

The next relevant query would be one, which takes advantage of the join query object. The newest syntax for the time[5] of writing this report for a join is:

join *variable* in *table* on (*permission*)

This will enable us to compare a course with it's allocations. In following query we find the building in which a course has it's allocated lecture room:

```
let getBuildingOfCourse c =
        query { for allocation in db.Allocations do
                join (for room in db.Rooms ->
                        allocation.RoomId = room.Id)
                where (room.Prop.Equals "Auditorium"
                       && allocation.CourseId = c)
                select (room.BuildNr)
                exactlyOne }
```

The function is of type: `int -> IQuerable<string>`. It iterates over Allocation and joins it with the room of which satisfy the permission - in this case the once with matching room ID's. It then checks the property.

The last query which we will be describing is one of the more advanced ones. If we want all exercise rooms combined with all possible time tables we would have to do a combination where we query for all exercise rooms and then construct a `int*string list`.

```
let getExerciseRooms =
  let qa = query { for room in db.Rooms do
    where (room.Prop.Contains "Holdomraade" ||
    room.Prop.Contains "Holdlokale" ||
    room.Prop.Contains "LilleGruppe")
    sortBy (room.Size)
    select (room.Id) }
  let mutable ls = []
  for room in qa do
```

[5]this is mentioned, since the syntax actually changed during the implementation stage

```
    for table in schematables do
      ls <- (room,table)::ls
  ls
```

One could just declare the query within the for loop, but this method results in a much for desirable form and more easy to understand and read.

The end implementation will contain several queries for different purposes. They are all contained in same library - `ResourceDatabase.dll - ResourceQuery`. All of these follow the same techniques and designs described above - so it would be a bit redundant to discuss them all, but every query has a friendly and well declared name.

4.5 Input data and text processing

Before we can implement the different resource planning models, we need some data. The data we need is rooms, courses, DTU's own allocation for a given semester and the location for buildings.
All the data is taken from DTU's own official database from the website[6]. The spring semester of 2011 is to be taken as example for this project. The data unfortunately come is different formats such as pdf's containing tables. This means we must resort to a text processing scheme.

By using the provided `System.Text.RegularExpressions` library we can easily construct regular expressions and line by line iterate of txt files containing the relevant input. Lets take courses as an example. Here we have the a line as the following:

```
"02101 Indledende programmering - 154"
```

The corresponding regular expression will then look as following:

```
let regCourse =
  Regex @"\s*([\d]+)(?:\s+([^\s]+))*\s*[\055][\055]\s+([\d]+)
```

By using this, we can iterate over a file containing the information about courses. The same goes for the rest of the data we want.[7]

[6]www.portalen.dtu.dk

[7]For further detail please refer to Appendix A and C for a coordinate system over a map a DTU

4.6 Functions for allocations

The constructed queries now enable us to implement a resource planner. The overall idea is to have an empty Allocations table, in which it is going to get allocations progressively added through a solution.

4.6.1 Bruteforce

As mentioned in chapter 3, the bruteforce solution is going to allocated all courses one lecture room and the necessary amount of exercise rooms. This is done by trial and error - meaning that it naively places on course on the *board* and then recursively tries the same for the next course. This now leads us to describe the overall solution containing a discussion of data structure and algorithmic implementation.

As mentioned, the bruteforce method is a naive algorithm which in worst case tries all combinations, of which you would be able to allocated all courses. So the most reasonable thing would be to help it on its way. Imagine if we sorted all lecture rooms courses in descending size. By doing this we would avoid allot of cases, where it would have to backtrack. This leads us to the talk about data structures.

4.6.1.1 Data structures

Due to backtracking we can't rely on the database to carry the allocation information during the procedure. The only information we can rely on from the database is the static information and attributes such as the number of students of a course or the room capacity of a room. This means we must construct our own static data structures.

To start with we need a list of course numbers - `int list`. This way we can query the number of students of the given course. We also need a list of rooms. Each room should have a time table - so we'll end with a `(room Id, table) list` i.e. `int*string list`. With this we also need a list of allocation containing the combination of a room, course and time table giving the following structure `int*int*string list`. When the algorithm terminates this list will then contain a solution containing successful allocations that satisfy the hard contraints. Which then could be added to the database if needed.

In context to the bruteforcing of lecture rooms, the bruteforcing of exercise room will take in a list of lecture room allocation and through that iterate over courses, since there's only one course pr. element in the list and we get the time table of which the exercise rooms must be available in. And it will also take in a list of just room numbers, since we already have the current time table.

4.6.1.2 Implementation and backtracking

So what we want to do is: we have a course, we then find the next room that is available and big enough. If this has occurred we then add the allocation to the allocations list. Then we jump to the next course in the list of courses recursively. And when we/if we reach the end of the list of courses and we didn't *use* all the rooms, we have succeeded.

The bruteforcing will be split into two parts. First we allocate all courses lecture rooms and then exercise rooms. This is done for the purpose of easier debugging. The end implementation looks as following:

```
let rec bruteforceLectures cs a fl =
  match cs with
   |[]        -> Some a
   |c::cs' ->
     match fl with
      |[]          -> None
      |(r,t)::fl' ->
        if fits c r && isAvailable r t a then
          let a' = (r,c,t)::a
          match bruteForceLectures cs' a' roomList with
          | Some result -> Some result
          | None -> bruteForceLectures cs a fl'
        else bruteForceLectures cs a fl'
```

The function is of type: `int list -> int*int*string list -> int*string list -> int*int*string list`. There are two functions which need explaining: `fits : int -> int -> bool` and `isAvailable : int -> string -> int*int*string list -> bool`. The first function containing a query and a boolean calculation determining if the given course can fit in the given room. The second checks if the given combination of room and time table exists in the current allocations list[8].

The backtraking is done by taking advantage of the utilities of the `Option` type. This type enables us to save the current state of which a course was added an allocation which did not cause any problems. The backtracking will always take one step back and try to allocate the course another room. If it then exhaust the room list, it jumps another step back to the course before that and allocates it another room. The function terminates when the end of the course list has been reached.

Bruteforcing allocation for exercise rooms does motivates us to apply the same method of solution. Although this is possible we have to make some modifications. The main difference between allocating a lecture and exercise rooms, is obviously that we have to allocate at least one or more rooms. This isn't the only difference, we also have to make sure that all the rooms allocated are in the same building.

The solution to this would be to incorporate some values that maintain information in the recursive calls. The values will be the number of students we have allocate already and which building we're in. The end implementation looks as following:

[8]Please refer to the Appendix A for further detail.

```
let rec bruteForceExercises cs a fl i (b:string) =
  match cs with
   |[] ->  Some a
   |(_,c,t)::cs' ->
     match fl with
      |[]          ->  None
      |r::fl'      ->
        let studentsLeft = (getSizeOfCourse c) - i
        let b2 = getBuilding r
        if studentsLeft > 0 then
          if isAvailable r t a && (b2.Equals b || b.Equals "") then
            let a' = (r,c,t)::a
            let alloRoom = getSizeOfRoom r
            match bruteForceExercises cs a' roomList (i + alloRoom)
            b2 with
            | Some result -> Some result
            | None -> bruteForceExercises cs a fl' 0 ""
         else bruteForceExercises cs a fl' i b
       else bruteForceExercises cs' a basicExerciseList 0 ""
```

The function is of type: `int*int*string list -> int*int*string list -> int list -> int -> string -> int*int*string list`. Here we see that the overall design of the implementation looks allot like the implementation of the allocation of the lecture rooms. The functions `getSizeOfCourse`, `getBuilding` and `getSizeOfRoom` are all queries[9].
For a course we get a time table and a course number. We then calculate how many students there are left to be allocated a room, if it's below 0 - we're done with the current course and jump to the next. If not, we check if the current room is in the same building[10]. If so, we check for availability and instead of checking if it fits we simply allocate as many students as possible in the given room. The idea of backtracking is identical to the one in the lecture room allocation.

4.6.1.3 Short summary

The overall look and feel of the implementation described above motivates to discuss the choice of components for achieving backtracking.
Taking the `bruteforceLectures` as an example, we see that by choosing the `Option` type as means of backtracking, we get nice, concise and easily under-

[9]Please refer to Appendix A for further detail about the queries.
[10]That's if we have already allocated a room for the course

standable code. This also provided powerful debugging compared to the use of F#'s sequence expressions. This brings us to the alternatives to implementing backtracking in F# - sequence expressions in context to the SQL queries denote what we want instead of declaring a computation. This would also provide us with a simple looking and concise implementation, but the `Option` type seemed as a more attractive tool because of it's sequential computations and therefore much easier to debugging.

4.6.2 SCHOLA

An implementation which satisfies the hard contraints is now in place and it's now time to implement the SCHOLA algorithm as described in chapter 3. This implementation should serve as a tool to get as close as possible to satisfy the soft constraints and still conceive a solution which allocates as many courses as possible. The problems with the algorithm were discussed in chapter 3. The implementation would also raise some limitations. The described step in which we want to swap to allocation, if one is more suitable than the other, also restrict us to a limited form of solution exploring.

Unlike the bruteforce implementation, we want to allocate courses step-wise, i.e. first lecture room for a given course and then exercise rooms for the same course.[11]

4.6.2.1 Allocating a lecture room

Much like the brutefroce method it seems most reasonable thing to do is sort the lecture rooms and courses in descending size. This is done by the Querybuilders `sortByDescending` method. This now leads us to the start of step A. We want to get the available table for the largest room.

```
let getAvailableTable r (k:string list) =
  if Seq.length (q1 r) < 10 then
    if List.length k < 10 then
      let t = Set.difference (Set.ofList schematables)
        (Set.ofSeq (q1 r))
      if List.isEmpty k then
        Some (List.nth (Set.toList t) 0)
      else
```

[11]For further detail about the algorithm please refer to chapter 3. For further detail about the implementation please refer to Appendix A.

```
        let mutable t2 = Set.toList t
        for k1 in k do
          t2 <- List.filter (fun t' -> not(k1.Equals t')) t2
        if List.length t2 > 0 then
          Some (List.nth t2 0)
        else None
      else None
    else None
```

This is done by the function `getAvaiableTable : int -> string list -> string option`. It takes a room number and list of strings, which is used for backtracking[12] purposes, which will be described later on. It then returns the next available table, if there are any, as an `Option` type string. This is done for easier matching purposes and nicer code. This type is used many places in the code for the same reason. Getting the available table is done by comparing the already used tables with all the possible tables using the high order function `Set.difference`. Using this function should serve as a save and clear way of implementation.

If there is an available table it adds an allocation to the database. If not, it then compares the current allocations for the given room in all tables. Does is done by the function `isMoreSuitable : int -> int -> int -> string -> int`.

```
let rec isMoreSuitableThan c r n k =
  if (n + (List.length backTrackTables)) < 10 then
    let t = List.nth schematables n
    if not (List.exists (fun t2 -> t.Equals t2 ) k) then
      let c2 = getCourseInRoom r t
      if getSizeOfCourse c2 < getSizeOfCourse c then c2
      else isMoreSuitableThan c r (n+1) k
    else isMoreSuitableThan c r (n+1) k
  else c
```

This function compares the sizes of two courses and returns the course of which the initial course is bigger[13] then. It is then checked, if the given course returned is a new one or the same. If it is the same, it then recursively jumps back and tries to allocate the course for the second largest room and so forth. If it happens that the course is more suitable than one of those already allocated, it then deletes the relevant allocation and adds a new.

[12]Note that backtracking in the context is to try other possible tables for a given course
[13]"bigger" denotes the number of students

4.6.2.2 Allocating exercise rooms

Once a course has it's lecture room allocated, it's time to allocate the course the necessary amount of exercise rooms. Much like the bruteforce method, the overall scheme between allocating a lecture and exercise room is about the same, but with some modifications to the function of allocating exercise rooms.

The first thing to take into account, is the soft constraint about the distance between lectures and exercises. This means we must find the closest building with available rooms or rooms of which the current course would put into better use i.e. more suitable. So finding the closest building is done by using the function `findSuitableBuilding : int -> string -> int -> string option`. The function looks as following:

```
let rec findSuitableBuilding size b n =
  let building = findClosestBuilding b n
  match building with
  | Some building ->
    let buildNr = fst3 building
    if potentialSizeOfBuilding buildNr > size then
      Some buildNr
    else findSuitableBuilding size b (n+1)
  | None           -> None
```

The function `findClosestBuilding : string -> int -> string option` goes through the database and applies a distance function using triangular distance calculations and returns the closest one. The potential size of a building denotes the summed size of exercise rooms in buildings. This is done by querying the database for exercise rooms for the given building and materializing the results to a list and applying the high order function `List.fold` - this is a general pattern of analyzing and treating data from the queries. So this description also serves as a general description of the code design.

After finding a suitable building, we compare the current course with courses that already have allocated rooms. And swap the allocations if it's more suitable. If this though doesn't yield a solution, i.e. there no available rooms or possible swaps of allocations, we should backtrack. The backtracking consists of avoiding time table already tried. This enables us to explain the extra variables in the functions `getAvailableTable` and `isMoreSuitableThan`[14] , these also check if the already used tables are used. The backtracking is done by using a mutable `usedTables : string list`.

[14]For lecture allocations

This served as a description of the model part of the overall implementation. So what's left is the view and control.

4.7 GUI

Taking a look back at the requirements specification we see that the GUI should contain the following functionalities:

- Search utilities
- Creation of a course
- Allocation option for different allocation models

Not only should it contain the listed functionalities, but it should also serve as a simple and user friendly visualization of a tool, that teachers can use to get information about their course and rooms allocated. This brings us to the description of the view.

4.7.1 View

As mentioned in the section about the choice of technologies, the view should be implemented using XAML. This will allow us construct the GUI in a declarative manner, which serves as an easy way to then implement the control, since XAML and C# are key components in a WPF Application.

The functionalities should be presented by buttons and input fields. Whilst the results should be printed out in a textblock. And a statusbar should serve as a tool for giving the user constructive feedback about the process of the asynchronous worker on queries.

The end idea of the view looks as following:

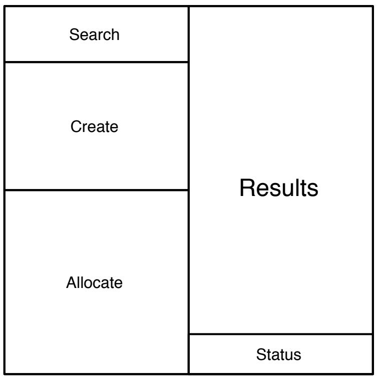


Figure 4.1: Illustration of the gui and segmentation of functions

4.7.2 Control

Once we have the View constructed, we are to link the buttons their action listeners. This is done through the Visual Studio environment, which decreases the workload design-time. This allows us to quickly divide work onto the buttons and link them to the results panel and statusbar.

4.7.3 Calling functions asynchronously

Asynchronous calls will serve as a part of the user friendly GUI. Calling the queries and large data sources will normally take some time to return a result. Because of this, these calls will cause the GUI to freeze, since calling functions in a normal fashion uses the main GUI-thread. This motivates us to take advantages of the ability to easily call functions in another thread within the .NET platform.

One way of doing this is construct a background worker using the `BackgroundWoker` component within the `System.ComponentModel` library. And once we initialize the WPF application we add different delegate even handlers to the background worker, such as when it has to do work and when it's completed. The delegates are then declared. This event handler should then give the user a visual feedback, such as disabling the a search button when a search queries are issued.

4.8 Summary

So with the implementation done, both problems and successful implementations were part of the experience.

4.8.1 Encountered problems

One might wonder why the implementation for simulated annealing isn't described. This is simply because it wasn't implemented, due to time constraints. It would though we very interesting and a reasonable step to take in terms of model implementation.

Technological problems also occurred. The most profound problem was the initial setup of an SQL server. This caused problems such as determining the correct connectionstring or troubleshooting general connectivity related problems.

Since there is about 200 courses and over 100 rooms on the campus of DTU, the sheer size of the data and time complexity of the solutions caused debugging to be rather difficult and tedious. This motived smaller and constructed cases of data for debugging, which will be further discussed in the chapter 5 about tests.

4.8.2 Possible optimizations

As mentioned simulated annealing was not implement, which would be the first model to implement, to further optimize on the overall performance of the product. The performance would be better, since simulated annealing takes a solution which doesn't conflict with the hard constraints and works on the soft. This is though done by the SCHOLA, but as discussed in the analysis chapter 3, it does involve a success rate.

There is also room for improvement in the current model implementation, such as further improving the heuristics of the SCHOLA method. This could be to implement binary search function when finding a suitable building, which should cause less conflicts. One could also try to solve the problem about finding the right combination of courses, which will put the given building to best use.

Another improvement would be to implements the models to work on multi threaded environments. Giving the time complexity and amount of data, this improvement would stand clear as one that should be highly prioritized.

Chapter 5

Tests

5.1 Short introduction

With the implementation sorted out it's time to test the product. The testing will consist of three different parts. A black box test, a white box test and a statistics test. As these tests are done a short summary for each is to conclude in context to the goal of the project.

5.2 Black Box Testing

Black box testing takes the product and tests its essential functionalities at hand. This test should serve as a clarification of the end product's quality in terms of the requirements specifications. The tester should be unaware of the overall struture of the program, but should be able to go the through given cases, which should show that the program works as it should.

5.2.1 Goal

So the goal for this test is to construct cases for the test. The cases are then to be evaluated and then compared with the expected outcome.

5.2.2 Test cases

Test case 1

Description	Seacrhing for a room
Actors	User/teacher
Preconditions	A connection to the SQL server is established
Events	- Program is started - A room number is typed into the relevant textbox - The "Search" button for a room is clicked - The button gets disabled - The results are displayed in the output textblock - The button gets enabled again [1]
Alternative events	A room number is not typed in the relevant textbox - All rooms are displayed in the output textblock

Test case 2

Description	Creating a course
Actors	User/teacher
Preconditions	A connection to the SQL server is established
Events	- Program is started - A course number, name and number of students is typed into the relevant textbox - The "Add" button is clicked - The button gets disabled - The results are displayed in the output textblock
Alternative events	One of the input boxes contain invalid input - An error message in the statusbar is displayed

Test case 3

[1] The same schema would be the same for searching for a course and would be redundant to evaulate

Description	Calling the bruteforce allocation.
Actors	User/teacher
Preconditions	A connection to the SQL server is established
Events	- Program is started - The button for bruteforce allocation is clicked - The button gets disabled - The results are displayed in the output textblock

5.2.3 Results

Case	Result for event
1	The room was displayed as predicted
2	The course was added successfuly
3	The button was clicked and the results were displayed

Case	Result for alternative event
1	All rooms for displayed
2	-
3	-

A screenshot of the GUI and a short user manual is provided in Appendix B.

5.2.4 Summary

The black box text didn't reveal any problems and then confirms the vailidity of the GUI it self. This dosn't contain much substance since the GUI didn't evolve much doing the process and it wasn't the goal either. The test does reveal that there's room for improvement in terms of exception handling. If one would want to search for a room with an invalid key, the program will stop end raise an execption.

Much more comprehensive testing is to be done in the following white box testing.

5.3 White Box Testing

Compared with the black box testing, the white box testing takes the program at hand and tests its underlying functions and structures. The tester is aware of the enviornment and the overall solution, which provides the test cases to be relevant and effective.

White box testing contains different techniques such as control flow testing, data flow testing and bracnhing testing[2]. In this case we will focus on the bracning testing.

The overall idea is to choose a function or goal of which we want to achieve. The tester then starts to test the branching functions. E.g. if we wanted to test the search for a room, we tester would then start to test the most low level function. This would be a query, that takes in required input set by the tester. The query would then give some output which should be interpreted and evaluated. The next step would then be to test the GUI call function by the same procedure.

5.3.1 Goal

The goal for this testing is to test the implementation by selecting some functionalities. Not all the functionalities will be tested, but more as a proof of concept - a selected sample will be tested.

5.3.2 Test description

The chosen functions we want to test are:

- Searching for a course
- An iteration of scholar
- Bruteforce on small case

The test will be done by describing the function it self and predict the output. After that, we list the results of all the relevant functions that are called and used.

[2] $http://en.wikipedia.org/wiki/White-box_testing$

5.3.3 Tests and Results

Searching for course 2405. This test only uncludes one function, which is `getCourse : int -> int * string * int`. The following results are:

```
> ResourceQuery.Queries.getCourse 2405;;
val it : int * string * int = (2405, " Sandsynlighedsregning ", 120)
```

Here we see that the function returns the expected 3-tuple containing the course number, name and number of students.

Next is to test an iteration of the scholar implementation. The first iteration is done in the `runScholar : unit` function, which contains a loop which calls the function `scholar : int -> unit` for each course in the database. It's start out by calling the `allocateLectureRoom : int -> int -> string option`. In this the first branched out function is `sortedLectureRooms : System.Linq.IQueryablie<int>`. This yields our first test:

```
> Seq.nth 0 ResourceQuery.Queries.sortedLectureRooms;;
val it : int = 4.
```

This is gives us room 4 for the lecture that is going to be checked for availability. This brings us to check for an available table using the room and the `usedTables : string list`, which is used for our backtracking described in the imlementation chapter 4.

```
> let u = ResourceAllocator.Allocator.usedTables;;
val u : string list = []
> let t = ResourceQuery.Queries.getAvailableTable 4 u;;
val t : string option = Some "3A"
```

This gives us table 3A, which will yield an allocation by calling the `addBasicAllocation : int*int*string -> unit`. This brings us back the the `scholar` function, which checks if the previous function yielded an allocation. Since it did it, ir calls the function that allocates exercise rooms. In the `AllocateExerciseRoom : int -> string -> string -> int -> bool`, the first function of interest is `findSuitableBuilding : int -> string -> int -> string option`. The test looks as following:

```
> let size = ResourceQuery.Queries.getSizeOfCourse 2405;;
val size : int = 120
> let b = ResourceAllocator.Allocator.findSuitableBuilding size "116"
0;;
```

```
val b : string option = Some "116".
```

The next step in the function is to check if we got a building or not. Since we did, the function will then call the function `potentialSizeOfBuilding : string -> int`.

```
> ResourceQuery.Queries.potentialSizeOfBuilding "116";; val it : int
= 540
```

The space left in the building is then calculated, but since there are no courses yet allocated the space left in building would be equal to the potential size of the building. And since this is larger than the size og the course, it then starts to allocate exercise rooms in the building by substracting each roomsize to the size of the course for each allocation added.

The last test is to try the bruteforce model in a basic case. The basic case will consist of the following data:

```
Courses : [26471;12104;10033;11375;2815;2101;11563;10020]

Lecture rooms : [(47,"1A");(48,"1A");(48,"2A");(7,"1A");(73,"1A")
;(71,"1A");(71,"2A");(76,"1A")]

Exercise rooms : [24;25;26;27;28;30;31;32;33;34;35;59;60
;61;62;77;78;79;88;89;67;68]
```

This case is structured to provoke backtracking, to show that the backtracking works. The bruteforcing of lecture rooms yields the following results:

```
> ResourceAllocator.Allocator.basicLectureAllocations;;
val it : (int * int * string) list = [(47, 10020, "1A");
(48, 11563, "2A"); (48, 2101, "1A"); (73, 2815, "1A");
(7, 11375, "1A"); (71, 10033, "2A"); (71, 12104, "1A"); (76, 26471,
"1A")]
```

The last part of the bruteforce model is to allocate exercise rooms. This yiels the following results:

```
> ResourceAllocator.Allocator.basicExerciseAllocations;;
val it : (int * int * string) list = [(68, 26471, "1A");
(67, 26471, "1A"); (89, 12104, "1A"); (88, 12104, "1A");
(32, 10033, "2A"); (31, 10033, "2A"); (30, 10033, "2A");
(79, 11375, "1A"); (78, 11375, "1A"); (77, 11375, "1A");
(62, 2815, "1A"); (61, 2815, "1A"); (60, 2815, "1A");
(59, 2815, "1A"); (35, 2101, "1A"); (34, 2101, "1A");
(33, 2101, "1A"); (32, 2101, "1A"); (31, 2101, "1A");
(30, 2101, "1A"); (27, 11563, "2A"); (26, 11563, "2A");
(25, 11563, "2A"); (24, 11563, "2A"); (28, 10020, "1A");
(27, 10020, "1A"); (26, 10020, "1A"); (25, 10020, "1A");
(24, 10020, "1A")]
```

Now we have been shown that a simple query as finding a course is consistant, a simple iteration of the scholar and the bruteforce model on a basic case works. But the reason for doing this project is also to evalaute the performance in relation to DTU's own allocations. This brings us to the next section.

5.4 Statistics

An interesting test would also be to evaluate an allocation solution itself, by analysing the data. This is done so we can put some numbers to an allocation solution, and say if it's a good or bad one.

So what is a good allocation? Looking at the soft contraints we might evaluate a solution on it's mean distance between a lecture and exercice rooms of a course. But the standard deviation, should also serve as a tool to do some interesting evaluation. Since a solution might not be good if there exists a group course of which the distance is quite far.
A last statistic we want to look at is the mean distance/person. This should take the case of the larger courses having a short distance and maybe smaller once having larger. [3]

Solution	Mean distance	Standard deviation	distance/person
SCHOLA	47.69	28.07	0.52
DTU	8.73	25.83	0.12

Because of the time compexity of the bruteforce solution, the tests for that solution were not run, since one test would take over a day.

It should be mentioned that the scholar method only achieved a successrate of 31.5% - allocationg 77 courses out of 245. This is causes by the limitations of the model discussed in both chapter 3 and 4. So a naive comparison of the these statistics would be ambisious, but it would still stand as a point of discussion.

[3]Please refer to Appendix C for a map of DTU. The scale of the results is 1:10

We see that because of the sorted course and lecture rooms, we ran into a low successrate and therefore also have a high meandistance, since all exercise building are used for so few courses. Although the standard deviations are close, shows us that this solution might be viable with some modifications and optimizations to the heuristics.

The main reason that DTU's own allocations show these relative good results, is that it is done by hand and is therefore pretty close to an optimal solution. One of the main reasons that some courses get long distances, is if it's a new course and doesn't at first get an optimal allocation. This is generaly then first taken care of the semester after by the administration.

5.5 Conclusion of the tests

The blackbox testing mainly tested the functionality of the GUI, which returned the expected output. But revealed possible optimizations in terms exception handling.

The white box testing shows the functionalitites behind the GUI and the main solution. The white box testing could contain more different test case and thorough branching in function. But the described test in this project are mainly to show the concept of a whitebox test.

The statistic revealed to us, that the schola solution does yield a solution that tries to statisfy the soft constraints, although with a low successrate. Compared with the allocations of DTU, the schola solution does have a chance to compete. Looking closer at the DTU allocations[4], we see that there are some overbooking, meaning that large courses get rooms with a too low capacity allocated. This tells us that there might another means of optimization to the schola solution. This optimzation could be done by implementing a *buffer*, which would alow courses to exceed the limits by a given amount.

[4]Please refer to Appendix C

Chapter 6

Summary

6.1 Future improvements

Lots of improvements stand clear as discussed in chapter 4 about the implementation. To sum up, the first most important improvement would be to improve the heuristics of the schola solution. One would be to optimized on the problem described in chapter 3 about achieving the optimal combinations, but this would require a brute force solution, which would further increase the overall time complexity.

Another obvious improvement would to take advantage of the limited amount of mutable values in the solution and explore possible multi threaded solutions.

Since the simulated annealing wasn't implemented, it would be interesting to see the resulting statistics of such a solution, which would work on an allocation given by one of the current to models. The most obvious one would be to run it on the yielded solution of the bruteforce solution, since we know optimizations would take place in context to the soft constraints. And that the bruteforce has a 100% success rate.

Current exception handling in the GUI is non existing. If we e.g. search for a room with an invalid search key, the implemented F# database library will raise a `InvalidOperation` exception.

6.2 Did we reach our goal?

Looking back the requirements specification, taking the first point:

- An optimization on how to allocate resources, by implementing a solution which can allocate resources computationally.

The optimization ended with a product, which contained a GUI with basic functions. The functions will surface as a simple example on how teachers would manage their course in terms of room allocations. The product contained to different solution - a brute force and a schola solution. The brute force solution yielded a solution on basic cases, but the time complexity did hinder a real world test. The schola solution tried to satisfy the soft constraints, by using direct heuristics. The chosen heuristic, preparation of data, and limitation of the solution it self introduced a success rate, i.e. not all courses were allocated.

The next requirements were:

- A simple GUI implementation capable of showing the user the relevant information.
- The GUI should contain search utilities for rooms and courses.
- Options for different solution allocation models.
- Create utility for a course.

The blackbox testing confirmed that these operations were implemented. The GUI contained textboxes and buttons to initiate a command. The output was then displayed in a textblock, showing the data in a string manner.

And the last requirements:

- An SQL database implemented implemented with F# .
- The database should be of such quality that is satisfies the first four normal forms.

Implementing a good database solution initially contains lots of possible solutions. At start the first database design used was inconsistent and did not even satisfy the second normal form. This was because mainly because the solution restricted it self to only two different tables. Butt he end solution did follow all four normal forms and later on in the model implementation stage, turned out to be nice and consistent to work with.

So in all - alot of improvement should be done before an end product would be taken in to consideration. But as a proof of concept we showed that computational allocation coexisting with a nice and user friendly interface is achievable.

Appendix A

Source code

A.1 ResourceCreate.fs

```
namespace ResourceCreate

open System.Configuration
open System.Data
open System.Data.SqlClient

module Create =

    let connString = @"Data␣Source=
␣␣␣␣JOACHIM-PC\RESOURCEDATABASE;
␣␣␣␣Initial␣Catalog=Resources;
␣␣␣␣Integrated␣Security=True"

    let conn = new SqlConnection(connString)

    conn.Open()

    let execNonQuery conn s =
        let comm = new SqlCommand(s, conn,
        CommandTimeout = 10)
        comm.ExecuteNonQuery() |> ignore

    execNonQuery conn "CREATE␣TABLE␣Courses␣(
␣␣␣␣␣␣␣␣␣␣␣␣CourseId␣int␣NOT␣NULL,
␣␣␣␣␣␣␣␣␣␣␣␣CourseName␣varchar(50)␣NOT␣NULL,
␣␣␣␣␣␣␣␣␣␣␣␣Students␣int␣NOT␣NULL,
␣␣␣␣␣␣␣␣␣␣␣␣PRIMARY␣KEY␣(CourseId))"

    execNonQuery conn "CREATE␣TABLE␣Rooms␣(
␣␣␣␣␣␣␣␣␣␣␣␣Id␣int␣NOT␣NULL,
␣␣␣␣␣␣␣␣␣␣␣␣BuildNr␣varchar(50)␣NOT␣NULL,
␣␣␣␣␣␣␣␣␣␣␣␣RoomNr␣varchar(50)␣NOT␣NULL,
␣␣␣␣␣␣␣␣␣␣␣␣Prop␣varchar(50)␣NOT␣NULL,
␣␣␣␣␣␣␣␣␣␣␣␣Size␣int␣NOT␣NULL,
␣␣␣␣␣␣␣␣␣␣␣␣PRIMARY␣KEY␣(Id))"

    execNonQuery conn "CREATE␣TABLE␣Allocations␣(
␣␣␣␣␣␣␣␣␣␣␣␣AllocNr␣int␣NOT␣NULL,
␣␣␣␣␣␣␣␣␣␣␣␣RoomId␣int␣NOT␣NULL,
␣␣␣␣␣␣␣␣␣␣␣␣CourseId␣int␣NOT␣NULL,
␣␣␣␣␣␣␣␣␣␣␣␣TableSection␣varchar(50)␣NOT␣NULL,
␣␣␣␣␣␣␣␣␣␣␣␣PRIMARY␣KEY␣(AllocNr))"

    execNonQuery conn "CREATE␣TABLE␣Locations␣(
␣␣␣␣␣␣␣␣␣␣␣␣BuildNr␣varchar(50)␣NOT␣NULL,
␣␣␣␣␣␣␣␣␣␣␣␣CoordinateX␣float␣NOT␣NULL,
␣␣␣␣␣␣␣␣␣␣␣␣CoordinateY␣float␣NOT␣NULL,
␣␣␣␣␣␣␣␣␣␣␣␣PRIMARY␣KEY␣(BuildNr))"
```

A.2 ResourceQuery.fs

```
namespace ResourceQuery

    open System ;;
    open System.Data ;;
    open Microsoft.FSharp.Data.TypeProviders ;;
    open System.Linq ;;
    open System.Data.Linq.SqlClient ;;

    [<Generate>]
    type schema = SqlDataConnection<"Data␣Source=
␣␣␣␣␣␣JOACHIM−PC\RESOURCEDATABASE;
␣␣␣␣␣␣Initial␣Catalog=Resources;
```

```
␣␣␣␣␣␣Integrated␣Security=True">;;

    module Queries =

        let private db = schema.GetDataContext()
        let rooms = db.Rooms

        //type declarations

        type room = schema.ServiceTypes.Rooms
        type course = schema.ServiceTypes.Courses
        type allocation = schema.ServiceTypes.Allocations
        type location = schema.ServiceTypes.Locations

        //basic queries

        let private nextRoomId() =
          query { for room in db.Rooms do
                     count }

        let private amountOfCourses() =
          query { for course in db.Courses do
                     count }

        let mutable aId = 0

        let private incrementAllocationId() = aId <- aId + 1

        let addRoom (building, e, room, prop, size)
                =     let id = nextRoomId()
                      let r = new room(Id = id, BuildNr = building,
                        RoomNr = room, Prop = prop, Size = size)
                     db.Rooms.InsertOnSubmit(r)
                     db.DataContext.SubmitChanges()

        let addCourse (courseId, courseName, students)
                =     let c = new course(CourseId = courseId,
                   CourseName = courseName, Students = students)
                     db.Courses.InsertOnSubmit(c)
                     db.DataContext.SubmitChanges()

        let addBasicAllocation (room, course, table)
                =    let id = aId
                    incrementAllocationId()
                    let a = new allocation(AllocNr = id,
                      RoomId = room, CourseId = course,
                      TableSection = table)
                    db.Allocations.InsertOnSubmit(a)
                    db.DataContext.SubmitChanges()

    let addLocation (building, x, y)
            =    let l = new location(BuildNr = building,
                  CoordinateX = x, CoordinateY = y)
                    db.Locations.InsertOnSubmit(l)
                    db.DataContext.SubmitChanges()

    let getRoom id = query { for room in db.Rooms do
                             where (room.Id = id)
                             select(room.Id, room.BuildNr,
                               room.RoomNr, room.Prop, room.Size)
                             exactlyOne
                             }

    let getRoomId r b = query { for room in db.Rooms do
                                where (room.RoomNr.Contains r
                                  && room.BuildNr.Contains b)
                                select(room.Id)
                                headOrDefault
                                }

    let getCourse id = query { for course in db.Courses do
                               where (course.CourseId = id)
                               select (course.CourseId,
                                 course.CourseName, course.Students)
                               exactlyOne
                               }

    let getAllocation id =
      query { for allocation in db.Allocations do
                where (allocation.AllocNr = id)
                select (allocation.AllocNr, allocation.RoomId,
                  allocation.CourseId,allocation.TableSection)
                exactlyOne }

    let getLectureAllocationId r c t =
      query { for allocation in db.Allocations do
                join room in db.Rooms on
                  (allocation.RoomId = room.Id)
                  where (allocation.RoomId = r &&
                    allocation.CourseId = c &&
                    allocation.TableSection.Contains t &&
                    room.Prop.Contains "Auditorium")
                  select (allocation.AllocNr)
                  exactlyOne }

let getExerciseAllocationIds c b t =
  query { for allocation in db.Allocations do
    join room in db.Rooms on
      (allocation.RoomId = room.Id)
    where (room.BuildNr.Contains b &&
      allocation.CourseId = c &&
      allocation.TableSection.Contains t &&
      (room.Prop.Contains "Holdomraade" ||
      room.Prop.Contains "Holdlokale" ||
      room.Prop.Contains "LilleGruppe"))
    select (allocation.AllocNr) }

let getLocation id =
  query { for location  in db.Locations do
    where (location.BuildNr.Contains id)
    select (location.CoordinateX, location.CoordinateY)
    exactlyOne }

let getBuilding id =
  query { for room in db.Rooms do
    where (room.Id = id)
    select(room.BuildNr)
    exactlyOne }

let getAllRooms =
  query { for room in db.Rooms do
    select(room.Id, room.BuildNr, room.RoomNr,
      room.Prop, room.Size) }

let getAllCourses =
  query { for course in db.Courses do
    select(course.CourseId, course.CourseName,
      course.Students) }

let getAllCourseIds =
  query { for course in db.Courses do
    sortByDescending(course.Students)
    select(course.CourseId)}

let getAllocatedCourses =
  query { for allocation in db.Allocations do
    join course in db.Courses on
      (allocation.CourseId = course.CourseId)
    select(course.CourseId,course.CourseName,course.Students)
    distinct }

let getAllAllocations =
  query { for allocation in db.Allocations do
    select(allocation.AllocNr, allocation.RoomId,
      allocation.CourseId, allocation.TableSection) }

let getAllocationsForCourse c =
  query {for allocation in db.Allocations do
    where (allocation.CourseId = c)
    select(allocation.AllocNr, allocation.RoomId,
      allocation.CourseId, allocation.TableSection) }

let getAllocationsForRoom r =
  query {for allocation in db.Allocations do
    where (allocation.RoomId = r)
    select(allocation.AllocNr, allocation.RoomId,
      allocation.CourseId, allocation.TableSection) }

let getAllLocations =
  query { for location in db.Locations do
    select (location.BuildNr, location.CoordinateX,
      location.CoordinateX) }

let getBuildings = query { for room in db.Rooms do
                           select (room.BuildNr)
                           distinct}

//converting sequences to lists for simplicity

let allRooms = Seq.toList getAllRooms

let allCourses = Seq.toList getAllCourses

let allAllocations = Seq.toList getAllAllocations

let allLocations = Seq.toList getAllLocations
let allBuildings = Seq.toList getBuildings

let allAllocatedCourses = Seq.toList getAllocatedCourses

//allocation queries

let sortedLectureRooms =
  query { for room in db.Rooms do
    sortByDescending(room.Size)
    where (room.Prop.Contains "Auditorium")
    select (room.Id) }

//gets the relevant allocated tablesections for a room
let  q1 r = query { for room in db.Rooms do
  join allocation in db.Allocations on
    (room.Id = allocation.RoomId)
  where (room.Id = r)
  select(allocation.TableSection) }

let getTableForLecture c =
  query {for allocation in db.Allocations do
    join room in db.Rooms on (allocation.RoomId = room.Id)
    where (allocation.CourseId = c &&
      room.Prop.Contains "Auditorium")
    select (allocation.TableSection)
    exactlyOne }

let getCoursesInBuilding b t =
  query { for allocation in db.Allocations do
    join room in db.Rooms on (allocation.RoomId = room.Id)
    where (room.BuildNr.Contains b &&
      allocation.TableSection.Contains t)
    select(allocation.CourseId) }

let getCourseInRoom r t =
  query { for allocation in db.Allocations do
    where (allocation.RoomId = r &&
      allocation.TableSection.Contains t)
    select (allocation.CourseId)
    exactlyOne }

let getBuildingOfCourse c =
  query { for allocation in db.Allocations do
```

```
            join room in db.Rooms on (allocation.RoomId = room.Id)
            where (room.Prop.Contains "Auditorium" &&
              allocation.CourseId = c)
              select (room.BuildNr)
              exactlyOne }

        let getSizeOfCourse c =
          query { for course in db.Courses do
            where (course.CourseId = c)
            select (course.Students)
            exactlyOne }

        let getSizeOfRoom r =
          query { for room in db.Rooms do
            where (room.Id = r)
            select (room.Size)
            exactlyOne }

        let schematables = ["1A";"2A";"3A";"4A";"5A";"1B";"
␣␣␣␣␣␣␣␣␣␣␣2B";"3B";"4B";"5B"]

        //gets all lecture rooms and their tables
        let getLectureRooms =
            let qa = query { for room in db.Rooms do
                             where (room.Prop.Contains "Auditorium")
                             sortBy (room.Size)
                             select (room.Id) }
            let mutable ls = []
            for room in qa do
                for table in schematables do
                    ls <- List.append ls [(room,table)]
            ls

        let getExerciseRooms =
            let qa = query { for room in db.Rooms do
                             where (room.Prop.Contains "Holdomraade" ||
                                    room.Prop.Contains "Holdlokale" ||
                                    room.Prop.Contains "LilleGruppe")
                             sortBy (room.Size)
                             select (room.Id) }
            let mutable ls = []
            for room in qa do
                for table in schematables do
                    ls <- (room,table)::ls
        ls

    let getExerciseRoomsInBuilding b =
      query { for room in db.Rooms do
      where (room.BuildNr.Contains b &&
        (room.Prop.Contains "Holdomraade" ||
        room.Prop.Contains "Holdlokale" ||
        room.Prop.Contains "LilleGruppe"))
      select (room.Id, room.BuildNr, room.RoomNr,
        room.Prop, room.Size) }

    let getLectureRoomForCourse c t =
      query { for allocation in db.Allocations do
      join room in db.Rooms on
        (allocation.RoomId = room.Id)
      where (room.Prop.Contains "Auditorium" &&
        allocation.TableSection.Contains t &&
        allocation.CourseId = c)
      select (room.Id)
      exactlyOne }

    //gets the sizes of exercise rooms
    let getExerciseRoomsForCourse c b t =
      query { for allocation in db.Allocations do
        join room in db.Rooms on
          (allocation.RoomId = room.Id)
        where (room.BuildNr.Contains b &&
          (room.Prop.Contains "Holdomraade" ||
          room.Prop.Contains "Holdlokale" ||
          room.Prop.Contains "LilleGruppe") &&
          allocation.TableSection.Contains t &&
          allocation.CourseId = c)
        select (room.Size) }

    let getAllocatedExerciseRooms b1 t =
      query { for allocation in db.Allocations do
        join room in db.Rooms on (allocation.RoomId = room.Id)
        where (room.BuildNr.Contains b1 &&
          (room.Prop.Contains "Holdomraade" ||
          room.Prop.Contains "Holdlokale" ||
          room.Prop.Contains "LilleGruppe") &&
          allocation.TableSection.Contains t)
        select (room.Id, room.BuildNr, room.RoomNr,
          room.Prop, room.Size) }
```

```
let getAllLectureAllocations =
  query {  for allocation in db.Allocations do
    join room in db.Rooms on (allocation.RoomId = room.Id)
    where (room.Prop.Contains "Auditorium")
    select (allocation.AllocNr)}

let hasAllocatedLecture c =
    let qa = query { for allocation in db.Allocations do
      join room in db.Rooms on (allocation.RoomId = room.Id)
      where (allocation.CourseId = c &&
        room.Prop.Contains "Auditorium")
      select (allocation.AllocNr) }
    not (qa = null)

let fits c r =
    let qa = query { for room in db.Rooms do
                     where (room.Id = r)
                     select (room.Size)
                     exactlyOne }
    getSizeOfCourse c < qa+1

let deleteLectureAllocation c =
    let qa = query { for allocation in db.Allocations do
                     join room in db.Rooms on
                       (allocation.RoomId = room.Id)
                     where (room.Prop.Contains "Auditorium" &&
                       allocation.CourseId = c)
                     select (allocation) }
    for allocation in qa do
    db.Allocations.DeleteOnSubmit(allocation)
    db.DataContext.SubmitChanges()

let deleteExerciseAllocations b c t =
    let qa = query { for allocation in db.Allocations do
                     join room in db.Rooms on
                       (allocation.RoomId = room.Id)
                     where (room.BuildNr.Contains b &&
                            allocation.CourseId = c &&
                            allocation.TableSection.Contains t
                           &&
                            (room.Prop.Contains "Holdomraade" ||
                             room.Prop.Contains "Holdlokale" ||
                             room.Prop.Contains "LilleGruppe"))
                     select(allocation) }
    (for allocation in qa do
        db.Allocations.DeleteOnSubmit(allocation)
        db.DataContext.SubmitChanges())

let getAvailableTable r (k:string list) =
    if Seq.length (q1 r) < 10 then
        if List.length k < 10 then
            let t = Set.difference
              (Set.ofList schematables) (Set.ofSeq (q1 r))
            if List.isEmpty k then
                Some (List.nth (Set.toList t) 0)
            else
                let mutable t2 = Set.toList t
                for k1 in k do
                    t2 <- List.filter
                      (fun t' -> not(k1.Equals t')) t2
                if List.length t2 > 0 then
                    Some (List.nth t2 0)
                else None
        else None
    else None

let getAvailableExerciseRooms b t =
  Set.toList (Set.difference (
    Set.ofSeq (getExerciseRoomsInBuilding b))
      (Set.ofSeq (getAllocatedExerciseRooms b t)))

let distance (l1:(float*float),l2:(float*float))
    = System.Math.Sqrt(System.Math.Pow(System.Math.Abs
      ((fst l2)-(fst l1)),2.0)+System.Math.Pow(
      System.Math.Abs((snd l2)-(snd l1)),2.0))

let findDistance = function
    |(null,_)    -> 0.0
    |(_,null)    -> 0.0
    |(b1,b2)     -> distance (getLocation b1,getLocation b2)

let findClosestBuilding b1 n =
    let sortedBuildings = List.sortBy
      (fun (b2,x,y) -> findDistance(b1,b2)) allLocations
    if n < List.length sortedBuildings then
        Some (List.nth sortedBuildings n)
    else None
```

```
let potentialSizeOfBuilding b1 =
  List.fold (fun count (_,_,_,_,s) -> count + s) 0
    (Seq.toList (getExerciseRoomsInBuilding b1))

let allocatedRoomCount b1 t =
  List.fold (fun count (_,_,_,_,s) -> count + s) 0
    (Seq.toList (getAllocatedExerciseRooms b1 t))

let allocatedSpaceForCourse b t c =
  List.fold (fun count s -> count + s ) 0
    (Seq.toList (getExerciseRoomsForCourse c b t))

let sortedCoursesInBuilding b t n =
  List.nth (List.sortBy (fun c -> getSizeOfCourse c)
    (Seq.toList (getCoursesInBuilding b t)) ) n

//functions for statistics

let findLectureRoom c =
  query { for allocation in db.Allocations do
    join room in db.Rooms on
      (allocation.RoomId = room.Id)
    where (allocation.CourseId = c
     && room.Prop.Contains "Auditorium")
    select(room.BuildNr)
    headOrDefault }

let findExerciseRoom c =
  query { for allocation in db.Allocations do
    join room in db.Rooms on
      (allocation.RoomId = room.Id)
    where (allocation.CourseId = c &&
      (room.Prop.Contains "Holdomraade" ||
     room.Prop.Contains "Holdlokale" ||
     room.Prop.Contains "LilleGruppe"))
    select(room.BuildNr)
    headOrDefault }

//auxiliary functions for GUI calls

let courseToString(courseId:int,courseName,students:int) =
  (string courseId) + "␣␣␣" + courseName + "␣␣␣"
    + (string students)
```

```
let roomToString(roomId:int,buildNr,roomNr,prop,size:int) =
  (string roomId) + "␣␣␣" + buildNr + "␣␣␣" + roomNr + "␣␣␣"
    + prop + "␣␣␣" + (string size)

let allocationToString(alId:int,roomId:int,courseId:int,table)
  (string alId) + "␣␣␣" + (string roomId) + "␣␣␣"
    + (string courseId) + "␣␣␣" + table

let allRoomsToString =
  List.fold (fun s (id,bnr,rnr,p,st) ->
    s + roomToString (id,bnr,rnr,p,st) + "\n")
    "" allRooms

let allCoursesToString =
  List.fold (fun s (id,name,st) ->
    s + courseToString (id,name,st) + "\n")
    "" allCourses

let allAllocationsToString =
  List.fold (fun s (alId,room,course,table) ->
    s + allocationToString (alId,room,course,table) + "\n")
    "" allAllocations


//setup

open TextIO.TextProcessor

let private addRoomFromData s = addRoom (extractRoomData s)

let private addCourseFromData s = addCourse
  (extractCourseData s)

let private addAllocationFromData s = addBasicAllocation
  (extractAllocationData s getRoomId)

let private addLocationFromData s = addLocation
  (extractLocationData s)

let setupRooms = fileIter (addRoomFromData) p

let setupCourses = fileIter (addCourseFromData) p2
```

```
        let setupAllocations = fileIter (addAllocationFromData) p3

        let setupLocations = fileIter (addLocationFromData) p4
```

A.3 ResourceAllocator.fs

```
namespace ResourceAllocator

open ResourceQuery

    module Allocator =

        open ResourceQuery.Queries

        let schematables = ["1A";"2A";"3A";"4A";"5A";"1B";"2B";
          "3B";"4B";"5B"]

        //auxiliary for getting elements of 3-tuples

        let fst3 (a,_,_) = a
        let snd3 (_,b,_) = b
        let thd3 (_,_,c) = c

        //auxiliary for getting elements of 4-tuples

        let fst4 (a,_,_,_) = a
        let snd4 (_,b,_,_) = b
        let thd4 (_,_,c,_) = c
        let fth4 (_,_,_,d) = d

        //auxiliary for getting elements of 5-tuples

        let fst5 (a,_,_,_,_) = a
        let snd5 (_,b,_,_,_) = b
        let thd5 (_,_,c,_,_) = c
        let fth5 (_,_,_,d,_) = d
        let fif5 (_,_,_,_,e) = e

        //Bruteforce

        let lectureRoomsList = getLectureRooms

        let sortedLectureRoomsList =
      List.sortBy (fun (r,t) -> getSizeOfRoom r)
        lectureRoomsList

  let exerciseRoomsList = getExerciseRooms

  let basicLectureList = [(47,"1A");(48,"1A");(48,"2A");
      (7,"1A");(73,"1A");(71,"1A");(71,"2A");(76,"1A")]

  let basicExerciseList =
      [24;25;26;27;28;30;31;32;33;34;35;
        59;60;61;62;77;78;79;88;89;67;68]

  let basicCourseList = [26471;12104;10033;
      11375;2815;2101;11563;10020]

  let isAvailable r (t:string) al =
      not (Seq.exists (fun (r2,_,t2) -> r = r2 && t.Contains t2) al

  let rec bruteForceLectures cs a fl =
      match cs with
      |[] -> Some a
      |c::cs' -> match fl with
        |[]           ->    None
        |(r,t)::fl' ->    if fits c r &&
                                isAvailable r t a then
                              let a' = (r,c,t)::a
                              match bruteForceLectures cs'
                                a' basicLectureList with
                              | Some result   ->  Some result
                              | None          ->
                                bruteForceLectures cs a fl'
                              else bruteForceLectures cs a fl'

  let lectureAllocations =
      Option.get (bruteForceLectures
        (Seq.toList getAllCourseIds) [] sortedLectureRoomsList)

  let basicLectureAllocations =
      Option.get (bruteForceLectures basicCourseList []
        basicLectureList)

  let rec bruteForceExercises cs a fl i (b:string) =
      match cs with
      |[]  ->  Some a
```

```
        |(_,c,t)::cs'   ->
          match fl with
          |[]           ->  None
          |r::fl'       ->  let studentsLeft =
            (getSizeOfCourse c) - i
            let b2 = getBuilding r
            if studentsLeft > 0 then
              if isAvailable r t a &&
                (b2.Equals b || b.Equals "") then
                let a' = (r,c,t)::a
                let alloRoom = getSizeOfRoom r
                match bruteForceExercises cs a'
                  basicExerciseList (i + alloRoom) b2 with
                | Some result   ->  Some result
                | None          ->  bruteForceExercises cs a fl' 0 ""
              else bruteForceExercises cs a fl' i b
            else bruteForceExercises cs' a
              basicExerciseList 0 ""

let basicExerciseAllocations =
  Option.get (bruteForceExercises
    basicLectureAllocations [] basicExerciseList 0 "")

//SCOLA

let mutable (usedTables:string list) = []

let rec isMoreSuitableThan c r n k =
  if (n + (List.length usedTables)) < 10 then
    let t = List.nth schematables n
    if not (List.exists (fun t2 -> t.Contains t2 ) k) then
      let c2 = getCourseInRoom r t
      if getSizeOfCourse c2 < getSizeOfCourse c then c2
      else isMoreSuitableThan c r (n+1) k
    else isMoreSuitableThan c r (n+1) k
  else c

let rec isMoreSuitableThan2 c b t n =
  if n < (List.length (Seq.toList
    (getExerciseRoomsInBuilding b))) then
    let c2 = List.nth (Seq.toList (getCoursesInBuilding b t)) n
    if getSizeOfCourse c2 < getSizeOfCourse c then c2
      else isMoreSuitableThan2 c b t (n+1)
    else c
```

```
let rec allocateLectureRoom c n =
  if n < (Seq.length sortedLectureRooms) then
    if (List.length usedTables) < 10 then
      let r = Seq.nth n sortedLectureRooms
      let t = getAvailableTable r usedTables
      match t  with
      | Some t2 -> addBasicAllocation  (r, c, t2)
                               Some (getBuildingOfCourse c)
      | None -> let c2 = isMoreSuitableThan c r 0 usedTables
                        if not (c2 = c) then
                          let t = getTableForLecture c2
                          let mutable id = 0
                          deleteLectureAllocation c2
                          addBasicAllocation (r, c, t)
                          allocateLectureRoom c2 n
                        else
                          allocateLectureRoom c (n+1)
    else None
  else None

let rec findSuitableBuilding size b n =
  let building = findClosestBuilding b n
  match building with
  | Some building -> let buildNr = fst3 building
    if potentialSizeOfBuilding buildNr
      > size then Some buildNr
    else findSuitableBuilding size b (n+1)
  | None              ->   None

let rec allocateExerciseRoom c b t n =
    let size = getSizeOfCourse c
    let b2 = findSuitableBuilding size b n
    match b2 with
    | None -> false
    | Some b2 ->
      let spaceLeft = potentialSizeOfBuilding b2 -
        allocatedRoomCount b2 t
      if (spaceLeft > (getSizeOfCourse c)) then
      //if the available space is greater than size of course
        let arooms = getAvailableExerciseRooms b2 t
        let count = getSizeOfCourse c
        for room in arooms do
          if count > 0 then
```

```
                    let count = count -
                      (getSizeOfRoom (fst5 room))
                    addBasicAllocation ((fst5 room), c, t)
                    true
                  else
                    let c2 = isMoreSuitableThan2 c b2 t 0
                    if (not (c2 = c) &&
                      (spaceLeft - getSizeOfCourse c2) >
                        getSizeOfCourse c) then
                        //if the course is more suitable then
                        //one already allocated in the building
                        let ids = Seq.toList (getExerciseAllocationIds c2
                        deleteExerciseAllocations b2 c2 t
                        let arooms = getAvailableExerciseRooms b2 t
                        let mutable count = getSizeOfCourse c2
                        for room in arooms do
                          if count > 0 then
                            let count = count -
                              (getSizeOfRoom (fst5 room))
                              addBasicAllocation ((fst5 room), c2, t)
                              true
                            else
                              allocateExerciseRoom c b2 t (n+1)
                              //go to next building

        let rec scholar c =
          let b = allocateLectureRoom c 0
          match b with
          | Some b   -> let t = getTableForLecture c
            let a = allocateExerciseRoom c b t 0
            if not a then
              let room = getLectureRoomForCourse c t
              usedTables <- t::usedTables
              deleteLectureAllocation c
              scholar c
            else
              usedTables <- []
           | None      ->  usedTables <- []

        let runScholar =
            for course in getAllCourseIds do
                scholar course

        let runBasicScholar =
              for course in basicCourseList do
                  scholar course

//          GUI-calls

          let basicBLectures = List.fold
          (fun s (room,course,table) -> s + (string room) +
            "  " + (string course) + "  " + table + "\n") ""
            basicLectureAllocations

          let basicBExercises = List.fold
            (fun s (room,course,table) -> s + (string room) +
              "  " + (string course) + "  " + table + "\n")
              "" basicExerciseAllocations

          let basicBruteForce = basicBLectures + basicBExercises

          let basicSchola =
              runBasicScholar
              allAllocationsToString
```

A.4 ResourceStatistics.fs

```
namespace ResourceStatistics

    module Statistics =

        open System

        open ResourceQuery.Queries

        let distance id = findDistance
          ((findExerciseRoom id),(findLectureRoom id))

        let distancePerPerson id =
          distance id/Convert.ToDouble (getSizeOfCourse id)

        let sumDistance = List.fold
          (fun d (id,_,_) -> d + distance id)
          0.0 allAllocatedCourses

        let sumDistancePerPerson = List.fold
          (fun d (id,_,_) -> d + distancePerPerson id)
          0.0 allAllocatedCourses

        let meanDistance =
          sumDistance /
            Convert.ToDouble allAllocatedCourses.Length

        let meanDistancePerPerson =
          sumDistancePerPerson /
            Convert.ToDouble allAllocatedCourses.Length

        llet variance =
          (List.fold (fun s (id,_,_) ->
            s + Math.Pow(((distance id) - meanDistance),2.0))
            0.0 allAllocatedCourses)
            / Convert.ToDouble allAllocatedCourses.Length

        let stdDeviation = sqrt variance
```

A.5 DataIO.fs

```
namespace TextIO

open System.Text.RegularExpressions
open System
open System.IO

    module TextProcessor =

            let private schematables = ["1A";"2A";"3A";"4A";"
␣␣␣␣␣␣␣␣␣␣␣␣␣␣5A";"5B";"2B";"1B";"4B";"3B"]

            let p = "C:\Users\Joachim\Desktop\Skole\Bachelor\
␣␣␣␣␣␣␣␣␣␣␣␣␣␣lokaler.txt"

            let p2 = "C:\Users\Joachim\Desktop\Skole\Bachelor\
␣␣␣␣␣␣␣␣␣␣␣␣␣␣optkurser.txt"

            let p3 = "C:\Users\Joachim\Desktop\Skole\Bachelor\
␣␣␣␣␣␣␣␣␣␣␣␣␣␣kursusallokeringer.txt"

            let p4 = "C:\Users\Joachim\Desktop\Skole\Bachelor\
␣␣␣␣␣␣␣␣␣␣␣␣␣␣lokationer.txt"
```

```
    let regRoom = Regex @"\s*([^\s]+)\s+([^\s]+)\s+
␣␣␣␣␣␣␣␣␣␣␣␣␣␣([^\s]+)\s+([^\s]+)\s+([\d]+)\s*"

    let regCourse = Regex @"\s*([\d]+)(?:\s+([^\s]+))
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣*\s*[\055][\055]\s+([\d]+)\s*"

    let regAllocation = Regex @"\s*([\d]+)\s+([^\s]+)\s+
␣␣␣␣␣␣␣␣␣␣␣␣␣␣([^\s]+)\s+([\d]+)\s+([^\s]+)\s+([\d]+)\s*"

    let regLocation = Regex @"\s*([^\s]+)\s+([\d]+)
␣␣␣␣␣␣␣␣␣␣␣␣␣␣\s+([\d]+)\s*"

    let private toTable = function
        | "mandag␣0800"     -> List.nth schematables 0
        | "mandag␣1000"     -> List.nth schematables 0
        | "mandag␣1300"     -> List.nth schematables 1
        | "mandag␣1500"     -> List.nth schematables 1
        | "tirsdag␣0800"    -> List.nth schematables 2
        | "tirsdag␣1000"    -> List.nth schematables 2
        | "tirsdag␣1300"    -> List.nth schematables 3
        | "tirsdag␣1500"    -> List.nth schematables 3
        | "onsdag␣0800"     -> List.nth schematables 4
        | "onsdag␣1000"     -> List.nth schematables 4
        | "onsdag␣1300"     -> List.nth schematables 5
        | "onsdag␣1500"     -> List.nth schematables 5
        | "torsdag␣0800"    -> List.nth schematables 6
        | "torsdag␣1000"    -> List.nth schematables 6
        | "torsdag␣1300"    -> List.nth schematables 7
        | "torsdag␣1500"    -> List.nth schematables 7
        | "fredag␣0800"     -> List.nth schematables 8
        | "fredag␣1000"     -> List.nth schematables 8
        | "fredag␣1300"     -> List.nth schematables 9
        | "fredag␣1500"     -> List.nth schematables 9
        | _                 -> ""

    let private captureSingle (ma:Match) (n:int) =
      ma.Groups.[n].Captures.[0].Value

    let private captureList (ma:Match) (n:int) =
        let capt = ma.Groups.[n].Captures
        let m = capt.Count - 1
        [for i in 0..m -> capt.[i].Value]

    let extractRoomData s =
    let m = regRoom.Match s
    (captureSingle m 1, captureSingle m 2,
      captureSingle m 3, captureSingle m 4,
      Int32.Parse(captureSingle m 5))

let extractCourseData s =
    let m = regCourse.Match s
    (Int32.Parse(captureSingle m 1),
      List.fold (fun t1 t2 -> t1 + "␣" + t2 + "␣") ""
      (captureList m 2), Int32.Parse(captureSingle m 3))

let extractAllocationData s g =
    let m = regAllocation.Match s
    (g (captureSingle m 6) (captureSingle m 5),
      Int32.Parse(captureSingle m 1),
      toTable(captureSingle m 2 + "␣" + captureSingle m 3))

let extractLocationData s =
    let m = regLocation.Match s
    (captureSingle m 1,
    Convert.ToDouble(Int32.Parse(captureSingle m 2)),
    Convert.ToDouble(Int32.Parse(captureSingle m 3)))

let private fileXiter g path =
  use s = File.OpenText path
  while not(s.EndOfStream)
    do g s
  s.Close()

let fileIter g s =
  fileXiter (fun s -> g (s.ReadLine())) s
```

A.6 MainWindow.xaml

```
<Window x:Class="ResourceGUI.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="ResourcePlanner" Height="411" Width="763"
          ResizeMode="NoResize">
    <Grid>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="400"/>
            <ColumnDefinition Width="400"/>
```

```
</Grid.ColumnDefinitions>
<Grid.RowDefinitions>
    <RowDefinition Height="0"/>
    <RowDefinition Height="0"/>
    <RowDefinition/>
    <RowDefinition Height="20"/>
</Grid.RowDefinitions>
<ScrollViewer Grid.Row="2" Margin="5,5,49,5"
  CanContentScroll="True" VerticalScrollBarVisibility="Auto"
  Grid.Column="1">
    <TextBlock Name="DataOutput"  HorizontalAlignment="Left"
      Grid.Row="2" TextWrapping="Wrap" VerticalAlignment="Top"
      Height="Auto" Width="Auto"/>
</ScrollViewer>
<Rectangle Name="TopBackGround" Grid.Row="0" Grid.RowSpan="2"
  Grid.Column="1" Fill="LightGray" Margin="0,0,41,0"/>
<Rectangle Name="SideBackGround" Grid.Column="0"
  Grid.Row="0" Fill="LightGray" Grid.RowSpan="4"/>

<GroupBox Header="Search" HorizontalAlignment="Left"
  VerticalAlignment="Top" Height="84" Width="380"
  Margin="10,14,0,0" Grid.Row="2"/>
<GroupBox Header="Create" HorizontalAlignment="Left"
  Width="382" Margin="8,144,0,143" Grid.Row="2"/>
<GroupBox Header="Allocate" HorizontalAlignment="Right"
  Grid.Row="2" VerticalAlignment="Top" Height="148"
  Grid.RowSpan="2" Width="382" Margin="0,220,8,0"/>

<TextBox Name="courseBox" HorizontalAlignment="Left"
  Margin="76,35,0,0" TextWrapping="Wrap" Text=""
  VerticalAlignment="Top" Width="143" Height="22"
  Grid.Row="2"/>
<TextBox Name="roomBox" HorizontalAlignment="Left"
  Margin="76,62,0,0" TextWrapping="Wrap" Text=""
  VerticalAlignment="Top" Width="143" Grid.Row="2"
  Height="22"/>
<TextBox Name="courseNumberBox" HorizontalAlignment="Left"
  Margin="66,174,0,0" Text="Number" TextWrapping="Wrap"
  VerticalAlignment="Top" Width="68" Height="22"
  Grid.Row="2"/>
<TextBox Name="courseNameBox" HorizontalAlignment="Left"
  Margin="139,174,0,0" TextWrapping="Wrap" Text="Name"
  VerticalAlignment="Top" Width="97" Grid.Row="2"/>
<TextBox Name="courseStudentsBox" HorizontalAlignment="Left"
```

```
    Margin="241,174,0,0" TextWrapping="Wrap" Text="Students"
    VerticalAlignment="Top" Grid.Row="2"/>
<TextBox Name="alloCourseBox" HorizontalAlignment="Left"
    Margin="67,241,0,0" Text="Number" TextWrapping="Wrap"
    VerticalAlignment="Top" Width="152" Height="22"
    Grid.Row="2"/>
<TextBox Name="alloRoomsBox" HorizontalAlignment="Left"
    Margin="67,268,0,0" Text="Number" TextWrapping="Wrap"
    VerticalAlignment="Top" Width="152" Height="22"
    Grid.Row="2"/>

<Button Content="Search" Name="SearchCourseButton"
    HorizontalAlignment="Left" Margin="227,35,0,0"
    VerticalAlignment="Top" Width="75" Grid.Row="2"
    Click="searchForCourse" Height="22"
    RenderTransformOrigin="-0.12,0.636"/>
<Button Content="Search" Name="SearchRoomButton"
    HorizontalAlignment="Left" Margin="227,62,0,0"
    VerticalAlignment="Top" Width="75" Grid.Row="2"
    Click="searchForRoom" Height="22"/>
<Button Content="Cancel" Name="SearchCourseButtonCancel"
    HorizontalAlignment="Left" Margin="307,35,0,0"
    VerticalAlignment="Top" Width="75" Grid.Row="2"
    Click="Cancel" Height="22"/>
<Button Content="Cancel" Name="SearchRoomButtonCancel"
    HorizontalAlignment="Left" Margin="307,62,0,0"
    VerticalAlignment="Top" Width="75" Grid.Row="2"
    Click="Cancel" Height="22"
    RenderTransformOrigin="-0.12,0.636"/>
<Button Content="Add" Name="AddCourseButton"
    HorizontalAlignment="Left" Margin="303,174,0,0"
    VerticalAlignment="Top" Width="78" Grid.Row="2"
    Click="addCourse"/>
<Button Content="Search" Name="SearchAlloCourseButton"
    HorizontalAlignment="Left" Margin="224,241,0,0"
    VerticalAlignment="Top" Width="75" Grid.Row="2"
    Click="searchForAlloCourse" Height="22"
    RenderTransformOrigin="-0.12,0.636"/>
<Button Content="Cancel" Name="SearchAlloCourseButtonCancel"
    HorizontalAlignment="Left" Margin="304,241,0,0"
    VerticalAlignment="Top" Width="78" Grid.Row="2"
    Click="Cancel" Height="22"/>
<Button Content="BruteForce" Name="BruteForceButton"
    HorizontalAlignment="Left" Margin="23,341,0,0"
```

```
        VerticalAlignment="Top" Width="100" Grid.Row="2"
        Click="bruteForce" Height="22"
        RenderTransformOrigin="-0.12,0.636"/>
      <Button Content="Cancel" Name="RoomAlloButtonCancel"
        HorizontalAlignment="Left" Margin="304,268,0,0"
        VerticalAlignment="Top" Width="78" Grid.Row="2"
        Click="Cancel" Height="22"/>
      <Button Content="SCHOLA" Name="ScholaButton"
        HorizontalAlignment="Left" Margin="154,341,0,0"
        VerticalAlignment="Top" Width="100" Grid.Row="2"
        Click="schola" Height="22"
        RenderTransformOrigin="-0.12,0.636"/>
      <Button Content="SA" Name="saButton" IsEnabled="False"
        HorizontalAlignment="Left" Margin="282,341,0,0"
        VerticalAlignment="Top" Width="100" Grid.Row="2"
        Height="22" RenderTransformOrigin="-0.12,0.636"/>
      <Button Content="Search" Name="SearchAlloRoomButton"
        HorizontalAlignment="Left" Margin="224,268,0,0"
        VerticalAlignment="Top" Width="75" Grid.Row="2"
        Click="searchForAlloRoom" Height="22"
        RenderTransformOrigin="-0.12,0.636"/>

      <TextBlock Name ="Statusbar"  HorizontalAlignment="Left"
        Grid.Row="3" Grid.Column="1" VerticalAlignment="Top"
        Background="LightGray" Width="439" Height="20"
        Margin="0,0,-80,0"/>
      <TextBlock HorizontalAlignment="Left" Margin="27,35,0,0"
        TextWrapping="Wrap" Text="Course:␣"
        VerticalAlignment="Top" Height="22" Width="44"
        TextAlignment="Center"
        RenderTransformOrigin="0.636,0.545" Grid.Row="2" />
      <TextBlock HorizontalAlignment="Left" Margin="27,62,0,0"
        TextWrapping="Wrap" Text="Room:␣" VerticalAlignment="Top"
        Height="22" Width="44" Grid.Row="2" />
      <TextBlock HorizontalAlignment="Left" Margin="22,174,0,0"
        TextWrapping="Wrap" Text="Course:"
        VerticalAlignment="Top" Height="22" Grid.Row="2"/>
      <TextBlock HorizontalAlignment="Left" Margin="23,244,0,0"
        TextWrapping="Wrap" Text="Course:" VerticalAlignment="Top"
        Height="22" Grid.Row="2"/>
      <TextBlock HorizontalAlignment="Left" Margin="22,268,0,0"
        TextWrapping="Wrap" Text="Room:␣" VerticalAlignment="Top"
        Height="22" Width="40" Grid.Row="2" />
  </Grid>
```

```
</Window>
```

A.7 MainWindow.xaml.cs

```
using System;
using System.Windows;
using ResourceQuery;
using ResourceAllocator;
using System.ComponentModel;

namespace ResourceGUI
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {

        private System.ComponentModel.BackgroundWorker
          backgroundWorker =
          new System.ComponentModel.BackgroundWorker();

        public MainWindow()
        {
            InitializeComponent();

            this.backgroundWorker.DoWork += new
              System.ComponentModel.DoWorkEventHandler
              (this.backgroundWorker_DoWork);
            this.backgroundWorker.RunWorkerCompleted += new
              System.ComponentModel.RunWorkerCompletedEventHandler(
              this.backgroundWorker_RunWorkerCompleted);
            this.backgroundWorker.WorkerSupportsCancellation = true;
            this.backgroundWorker.WorkerReportsProgress = true;
        }

        private void disableButtons()
        {
            SearchCourseButton.IsEnabled = false;
            SearchRoomButton.IsEnabled = false;
            SearchAlloRoomButton.IsEnabled = false;
            SearchAlloCourseButton.IsEnabled = false;
            BruteForceButton.IsEnabled = false;
            saButton.IsEnabled = false;
            ScholaButton.IsEnabled = false;
        }

        private void enableButtons()
        {
            SearchCourseButton.IsEnabled = true;
            SearchRoomButton.IsEnabled = true;
            SearchAlloRoomButton.IsEnabled = true;
            SearchAlloCourseButton.IsEnabled = true;
            BruteForceButton.IsEnabled = true;
            saButton.IsEnabled = true;
            ScholaButton.IsEnabled = true;
        }

        private void disableCancelButtons()
        {
            SearchCourseButtonCancel.IsEnabled = false;
            SearchRoomButtonCancel.IsEnabled = false;
            SearchAlloCourseButtonCancel.IsEnabled = false;
            RoomAlloButtonCancel.IsEnabled = false;
        }

        private void enableCancelButtons()
        {
            SearchCourseButtonCancel.IsEnabled = true;
            SearchRoomButtonCancel.IsEnabled = true;
            SearchAlloCourseButtonCancel.IsEnabled = true;
            RoomAlloButtonCancel.IsEnabled = true;
        }

        bool _allcourses = false;
        bool _allrooms = false;
        bool _allallocourses = false;
        bool _allallorooms = false;
        bool _schola = false;
        bool _bruteforce = false;

        private void backgroundWorker_DoWork(object sender,
          DoWorkEventArgs e)
        {
            if (_allcourses == true) e.Result = allCourses();
            if (_allrooms == true) e.Result = allRooms();
            if (_allallocourses == true) e.Result = allAllocations();
            if (_allallorooms == true) e.Result = allAllocations();
            if (_schola == true) e.Result = scholaResult();
            if (_bruteforce == true) e.Result = bruteforceResult();

            _allcourses = false;
            _allrooms = false;
            _allallocourses = false;
            _allallorooms = false;
            _schola = false;
            _bruteforce = false;

            if (this.backgroundWorker.CancellationPending)
            {
                e.Cancel = true;
                return;
            }
        }

        private void backgroundWorker_RunWorkerCompleted
          (object sender, RunWorkerCompletedEventArgs e)
        {
            if (e.Cancelled)
            {
                Statusbar.Text = "Process Cancelled.";
            }
            else
            {
                Statusbar.Text = "Processing completed. ";
                DataOutput.Text = (string)e.Result;
            }
            enableButtons();
        }

        private void Cancel(object sender, RoutedEventArgs r)
        {
            this.backgroundWorker.CancelAsync();
            enableButtons();
            disableCancelButtons();
        }

        private void searchForCourse(object sender, RoutedEventArgs e)
        {
            DataOutput.Text = "";
            dynamic coursedata;
        if (courseBox.Text != "")
        {
            coursedata = Queries.getCourse(
              Int32.Parse(courseBox.Text));
            DataOutput.Text = Queries.courseToString(
              coursedata.Item1, coursedata.Item2,
              coursedata.Item3);
        }
        else
        {
            disableButtons();
            _allcourses = true;
            this.backgroundWorker.RunWorkerAsync();
        }
    }

    private void searchForRoom(object sender, RoutedEventArgs e)
    {
        DataOutput.Text = "";
        dynamic roomdata;
        if (roomBox.Text != "")
        {
            roomdata = Queries.getRoom(
              Int32.Parse(roomBox.Text));
            DataOutput.Text = Queries.roomToString(
              roomdata.Item1, roomdata.Item2, roomdata.Item3,
              roomdata.Item4, roomdata.Item5);
        }
        else
        {
            disableButtons();
            _allrooms = true;
            this.backgroundWorker.RunWorkerAsync();
        }
    }

    private void searchForAlloCourse(object sender,
      RoutedEventArgs e)
    {
        DataOutput.Text = "";
        dynamic coursedata;
        if (alloCourseBox.Text != "")
        {
            coursedata = Queries.getAllocationsForCourse(
              Int32.Parse(alloCourseBox.Text));
            DataOutput.Text = Queries.allocationToString (
              coursedata.Item1, coursedata.Item2, coursedata.Item3,
              coursedata.Item4);
        }
        else
        {
            disableButtons();
            _allallocourses = true;
            this.backgroundWorker.RunWorkerAsync();
        }
    }

    private void searchForAlloRoom(object sender, R
      outedEventArgs e)
    {
        DataOutput.Text = "";
        dynamic roomdata;
        if (alloRoomsBox.Text != "")
        {
            roomdata = Queries.getAllocationsForRoom(
              Int32.Parse(alloRoomsBox.Text));
            DataOutput.Text = Queries.allocationToString(
              roomdata.Item1, roomdata.Item2, roomdata.Item3,
              roomdata.Item4);
        }
        else
        {
            disableButtons();
            _allallorooms = true;
            this.backgroundWorker.RunWorkerAsync();
        }
    }

    private void addCourse(object sender, RoutedEventArgs e)
    {
        int cId = Int32.Parse(courseNumberBox.Text);
        string cName = courseNameBox.Text;
        int students = Int32.Parse(courseStudentsBox.Text);
        Queries.addCourse(cId, cName, students);
        courseNumberBox.Text = "";
        courseNameBox.Text = "";
        courseStudentsBox.Text = "";
        }

        private void bruteForce(object sender, RoutedEventArgs e)
        {
            _bruteforce = true;
            this.backgroundWorker.RunWorkerAsync();
        }

        private void schola(object sender, RoutedEventArgs e)
        {
            _schola = true;
            this.backgroundWorker.RunWorkerAsync();
        }

        private string allCourses()
        {
            return Queries.allCoursesToString;
        }

        private string allRooms()
        {
            return Queries.allRoomsToString;
        }

        private string allAllocations()
        {
            return Queries.allAllocationsToString;
        }

        private string bruteforceResult()
        {
            return Allocator.basicBruteForce;
        }

        private string scholaResult()
        {
            return Allocator.basicSchola;
        }

    }
}
```

Appendix B

Test results

B.1 User manual

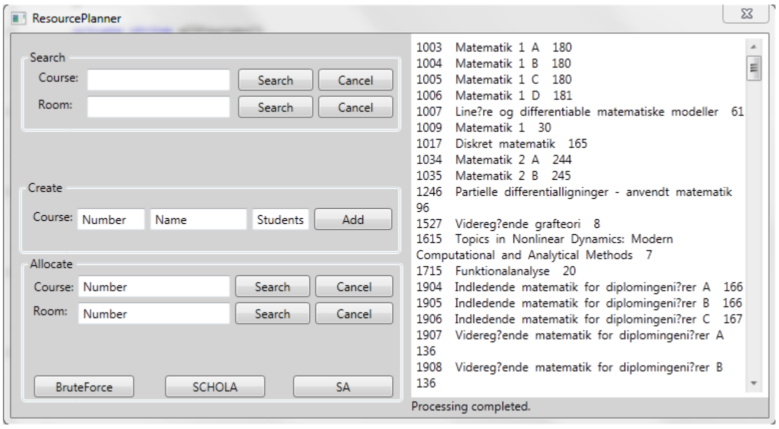
The window contains functionalities represented by textfields and buttons.

Searching for a course or room - Enter the given course number or room number which you want to view (the number must be valid) and press the "Search" button next to it. If you want to view all courses or room, simply press the button leaving either textfields empty.

Creating a course - Enter a number, name and number of students in the correct fields and press the "Add" button.

Searching for allocations of a course or room - Enter the course number or room number in the correct fields and press the "Search" button.

Running the brute force solution or schola on basic cases - Press either the "BruteForce" or "SCHOLA" button.



```
1003   Matematik 1 A   180
1004   Matematik 1 B   180
1005   Matematik 1 C   180
1006   Matematik 1 D   181
1007   Line?re og differentiable matematiske modeller   61
1009   Matematik 1   30
1017   Diskret matematik   165
1034   Matematik 2 A   244
1035   Matematik 2 B   245
1246   Partielle differentialligninger - anvendt matematik
96
1527   Videreg?ende grafteori   8
1615   Topics in Nonlinear Dynamics: Modern
Computational and Analytical Methods   7
1715   Funktionalanalyse   20
1904   Indledende matematik for diplomingeni?rer A   166
1905   Indledende matematik for diplomingeni?rer B   166
1906   Indledende matematik for diplomingeni?rer C   167
1907   Videreg?ende matematik for diplomingeni?rer A
136
1908   Videreg?ende matematik for diplomingeni?rer B
136
Processing completed.
```

Figure B.1: Assigned coordinates over DTU campus

Appendix C

References

- Schaerf - 1999 - A Survey of Automated Timetabling
- Gu et al. - Unknown - Building University Timetables Using Constraint Logic Programming.
- Burke et al. - Unknown - Examination Timetabling in British Universities A Survey 2 The Timetabling Problem
- The system programming series - An Introduction to Database Systems, snd Edition C. J. DATE.
- Introduction to programming using SML
- Functional Programming using F#, Michael R. Hansen and Hans Rischel, Version 0.8.9 (March 8, 2012).

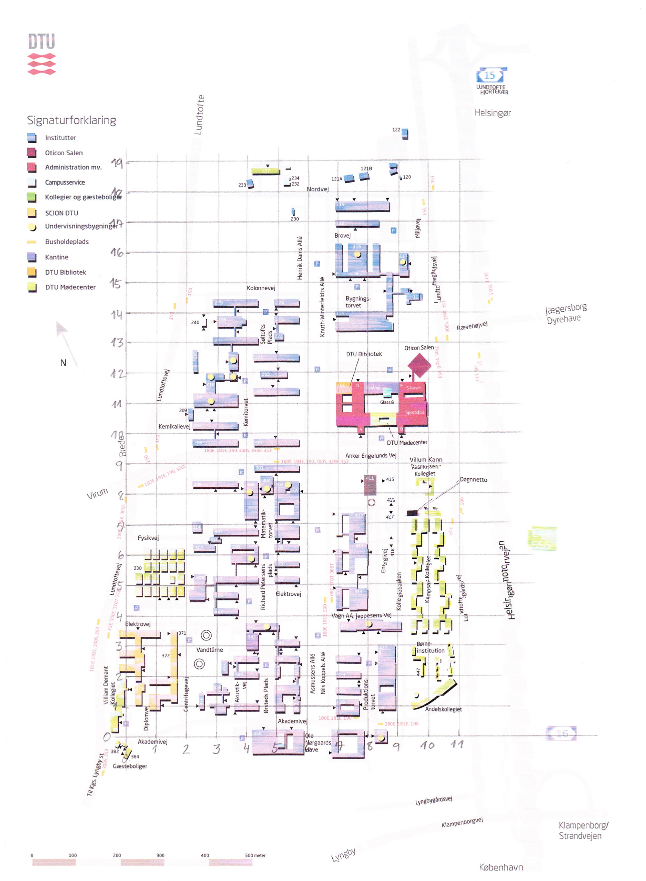


Figure C.1: Assigned coordinates over DTU campus