PlanIt

DAT1, FALL SEMESTER 2008

GROUP D102A

DEPARTMENT OF COMPUTER SCIENCE

AALBORG UNIVERSITY

DECEMBER 19TH 2008

AALBORG UNIVERSITET

Department of Computer Science
University of Aalborg
Selma Lagerlöfs Vej 300
9220 Aalborg Ø
Telefon +45 99409940
Fax +45 99409798
http://cs.aau.dk

Title: PlanIt
Theme: Program Development
Project period: Dat1, fall semester 2008
Project group: d102a
Group members:

Lasse B. Carstensen

Steffen Dalbro Eriksen

Kenneth S. Jacobsen

Kasper K. Kastaniegaard

Jan Larsen

Jeppe Thaarup

Supervisors: Bjørn Haagensen

Abstract:

Once upon a time in the valley of computer science was a small village called Cassiopeia. Once a year the inhabitants would have to sacrifice a young maiden due to the terrifying curse of the infinite schedule planning. The maiden would be locked in a tower, and not allowed food nor water untill the teaching schedule of the local school was done. This was not an easy task for such a young maiden, and they often succumbed to the curse before the work was done. However one day a noble knight of information technology rode through the valley and heard of this terribly curse from the villagers. He quickly conjured the magical artifact known as "PlanIt", which would forever help the school with all their scheduling needs, thus saving the maidens from the terrible curse and eternal damnation.

Copies: 8
Number of pages: 110
Appendices: 6 pages + CD containing sourcecode, Doxygen documentation and the compiled system
Completion date: December 18, 2008

Preface

This report deals with the development of a scheduling system. The problem of planning a schedule is that it very quickly gets complex with a larger amount of data (e.g. a large number of people, courses, teachers and rooms when creating a schedule e.g. for a school). Development of an automated process is therefore sought after in order to minimize the amount of work hours required to create a schedule.

This report will be divided into the following chapters:

Chapter one; containing the analysis document which is an analysis of the scheduling problem, along with a detailed walkthrough of the problem- and application domain.
Chapter two; containing the design document which focuses on discussing and finding criteria in order to make a working system along with laying the ground for the implementation and user interface.
Chapter three; containing sourcecode of the most important methods and classes, as well as unit testing of these.
Chapter four; containing the usability evaluation report carried out for the system.
Chapter five; containing the user manual for the system.
Chapter six; containing a conclusion which summarizes the previous chapters.
Chapter seven; containing the student report which describes the process of developing the system and writing this report.

Contents

1 Analysis Document

1.1 The Task

1.1.1 Predefined Terms

The first thing that needs to be done is to give the reader a knowledge base about some of the terms which are used in this report. These terms will be used throughout the entire report and also in the system associated with it. The basic terms are: *Tutor*, which, as the name implies, represents the tutors or lecturers that teach courses. *Student set*, which describe groups of students attending exactly the same combination of courses. A *course* will be taught by a tutor and attended by students. To tie everything together, the term *room* represents where the different courses will be held.

Definition of a Schedule

A schedule consists of a number of time periods for different purposes, in this case courses will take place in these time periods. The courses have tutors and student sets attached, and needs a room to accommodate these. Generally, there will be some days in a schedule where the time periods can not be placed because of weekends or holidays. And sometimes tutors have special periods they will, or can not teach. There are some general rules for a schedule, these are as follows:

- Courses can only be held on days which are not during weekends or holidays.
- The schedule starts at a specific time in the morning and ends at a specific time in the afternoon.
- A teacher can not teach two different courses in the same time period.
- A student set can not attend two different courses in the same time period.
- A room can not accommodate two courses in the same time period.
- A room can only be used for a course if it has the capacity for the attached students.

1.1.2 Purpose

The system described in this document is a tool for handling and creating schedules at a given organization where scheduling is needed. The problem with schedule planning has always been that it is a very time consuming task to generate optimal schedules. In many cases scheduling is done manually and is often considered a very time-consuming and complex process. A system that automates this process, given a number of criteria will be developed and described in this report. When the system has been given enough criteria it should be able to plan a fully functional schedule without any additional user input. We acknowledge the need to sometimes manually update an already working schedule, and the user therefore must have the ability to manually correct the schedule after the automated process.

In our case, we have chosen to apply the system to a university faculty and its tutors. Therefore the system will operate with the following criteria:

- **Tutors**: Which classes can they teach, how many hours are they required or allowed to have and are they gone on holiday or otherwise occupied.
- **Rooms**: How many students can they contain.
- **Courses**: The timeframe of the various courses.
- **Student sets**: Which courses the students are attending and the number of students in a specific set.

1.1.3 System Definition

The definition of the system is a system to automatically build and organize schedules. It must recieve input from the user containing criteria about tutors, rooms etc. in order to construct a fully functional schedule. Information and schedules are stored locally in a data layer, but the system must also be able to represent the data in a graphical user interface, be able to print schedules as well as display them in a web environment. There will be two types of users who can access the system, an administrator who can create schedules and a guest who can only view them.

The FACTOR Criterion

To make sure the definition contains all aspects of the system, the FACTOR criterion is used to check these. The criterion contains 6 subdefinitions:

- **F**unctionality: Creating and displaying schedules for the user.
- **A**pplication Domain: Secretaries working with planning and administration.
- **C**onditions: The system is developed for English-speaking users that already have some experience with planning schedules. The development deadline is mid-December 2008.
- **T**echnology: A stand-alone application for the user to input data and create a schedule, which can be displayed and printed with ease.
- **O**bjects: Tutors, rooms, students and courses, possibly already existing schedules.
- **R**esponsibility: An administrative scheduling tool.

1.1.4 Context

To illustrate the environment of the system, a rich image has been created. A rich image provides information about in which order a sequence of tasks are performed, how they are carried out, and where conflicts can arise in the given set of tasks. The following two images are rich images of how schedule planning is carried out now, and how the system could be used in the process:

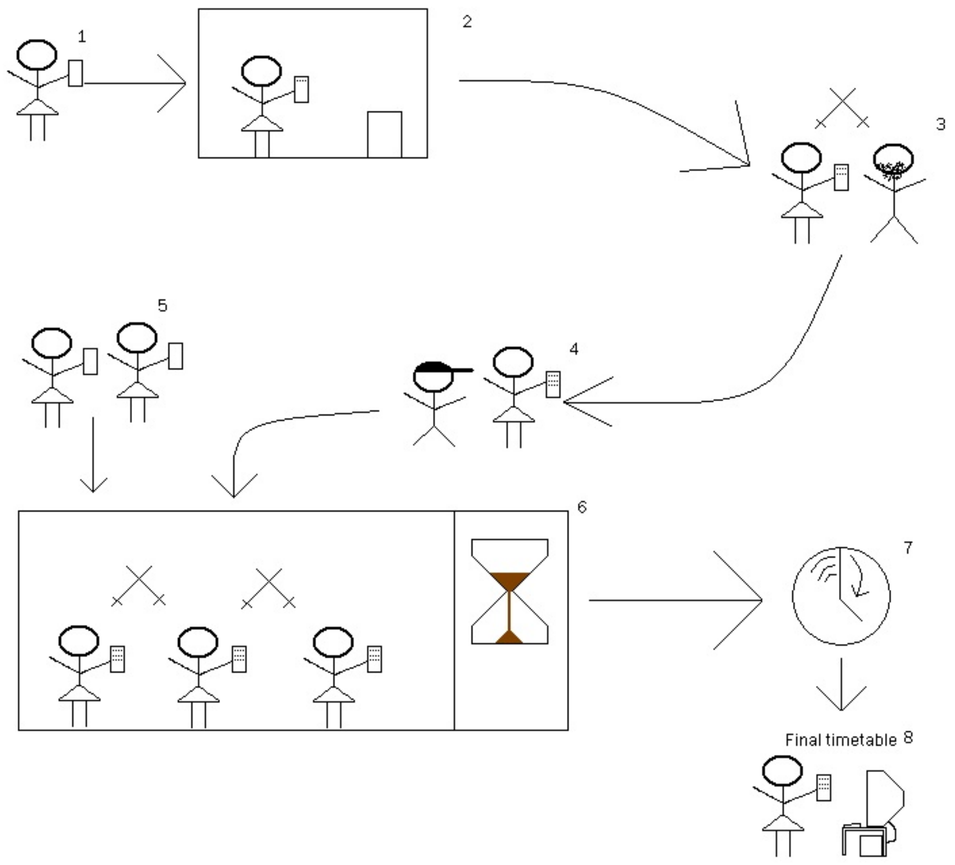


Figure 1.1: The manual process of creating a schedule.

For figure 1.1 we have:
1. The start of the scheduling process.
2. The secretary gathers information about the rooms.
3. The secretary gathers information about the tutors/courses.
Conflict: Some tutors require special rooms and dates where they prefer to work.
4. The secretary gathers information about students.
5. Other secretaries. These secretaries have also gathered information about rooms, tutors and courses etc.
6. The secretaries join together and make a joint schedule from the accumulated information.
Conflict: Getting the different secretary data to fit into the schedule. What information should be prioritized above others?
7. Time goes by.
8. The final schedule is plotted into the computer.

The figure shows how a secretary needs to collect a lot of information about the entities which the schedule is concerned with, and how she has to spend a huge amount of time in a room with other secretaries to make the schedule work. The huge amount of time is needed because a lot of conflicts will arise in the process. For example, you can not have two courses in the same room at the same time. After they have figured out the schedules they need to enter them into a computer, so that they can be displayed on the Internet.

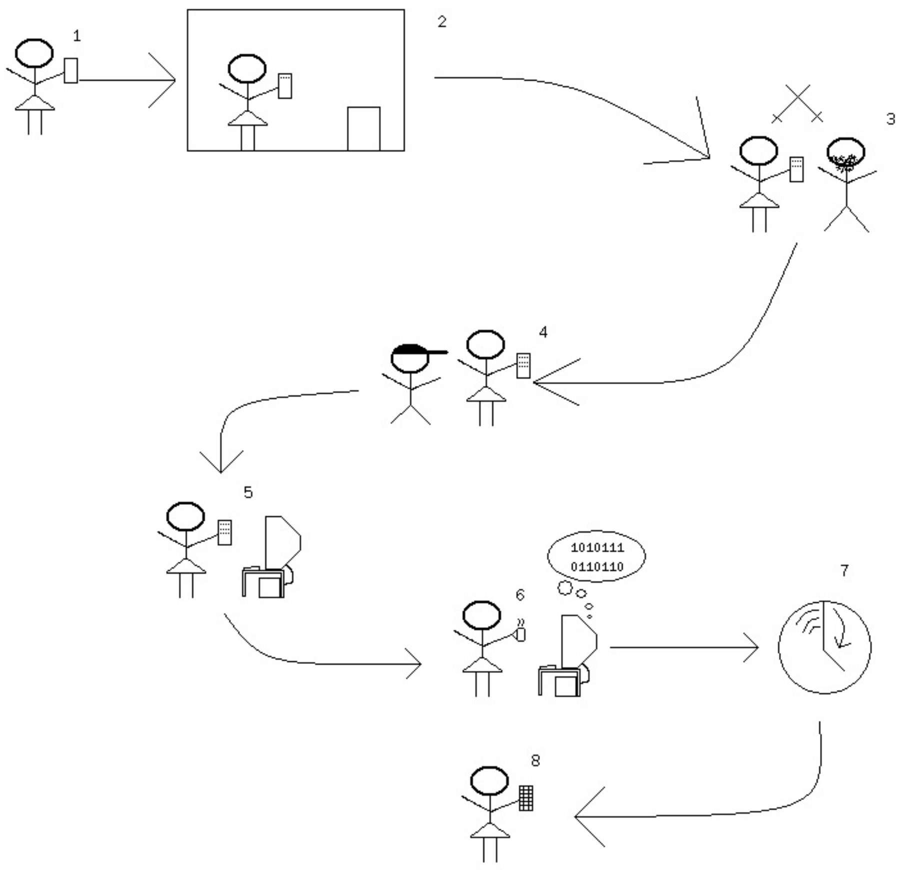


Figure 1.2: The computerized process of creating a schedule.

For figure 1.2 we have:
1. The start of the scheduling process.
2. The secretary gathers information about the rooms.
3. The secretary gathers information about the tutors/courses.
Conflict: Some tutors require special rooms and dates where they prefer to work.
4. The secretary gathers information about students.
5. The secretary plots information into the system.
6. The computer generates a new schedule.
7. Time goes by.
8. The computer finishes generating the schedule.

Just like the previous rich image, the secretary will need to collect data about various entities which the schedule is concerned with. But in this method she will not join other secretaries in a room, instead she will get the data that they have collected, and enter it all into the system. When the data has been entered she will only need to hit a button, and the system will make the schedule for her. The only difference is that the computer will solve the conflicts for the secretaries, and it will take a lot less time.

Problem Domain

The problem domain contains several items related to scheduling:

Courses are attended by students and taught by tutors in a given room at a set timeframe.

Tutors are the persons teaching courses.

Students are the people attending courses.

Rooms are the locations in which courses can be held.

Application Domain

The application domain consists of the employees at the secretariat, namely secretaries who are tasked with creating the daily schedules for the tutors and students with respect to their courses.

1.2 Problem Domain

1.2.1 Structure

This section will describe the overall structure of the problem domain. The structure revolves around the top part of the system, the *Schedule* class. A schedule can be said to consist of three parts:

1. A time frame defining the time interval that the schedule governs, which can be subdivided into smaller units of time. At the bottom layer it consists of the individual time blocks we wish to schedule.
2. Some subjects, e.g. the different entities that the schedule is to be planned for, such as students, teachers and rooms etc.
3. Activities, which serve to bind the two previous parts together. A single activity is bound to a single time block, and several subjects take part in it. Conflicts can arise here, since subjects cannot have more than one obligation at a time.

Clusters

The structure is divided into three clusters, as seen in figure 1.3. The *Timeframe* cluster contains classes describing the time interval in which a schedule can exist. The *Subjects* cluster contains classes describing the various physical objects and persons for which the schedule is planned. The *Activities* cluster contains classes describing the events that the schedule contains, such as lectures, assignments and exams. The individual classes are explained in detail below.

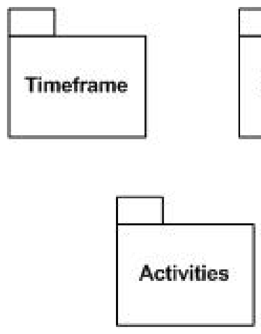


Figure 1.3: Cluster structure of the scheduling system.

1.2.2 Classes

This section will give a description of each class in the diagram in figure 1.4. This description will include definitions of the classes and their properties, as well as behavioral patterns to further elaborate on selected classes that are of special importance. 1.4 depicts the classes and their interactions:

Schedule

Schedule is the most central class, as it represents a complete schedule and serves to bind all the other classes together. It is composed of three things: An aggregation of a single *Timeframe* which defines the time interval that the schedule is concerned with and a number of *Subject* objects, as well as some associated activities that serve as a link between the subjects and the time frame.

Timeframe

The *Timeframe* consists of any combination of *Semester*, *Week* or *Day* classes, since the time frame does not have to be an unbroken interval (e.g. it is possible to have a schedule for two unconnected weeks), although in reality a single *Semester* class will usually be used.

Semester

Semester is used to represent the time interval of an entire semester. It is an aggregation of all the *Week* classes that lie in this interval. Attributes include a start and end date.

Week

Week is straightforward, consisting of an aggregation of 7 *Day* classes. Attributes include a start date, end date and week number.

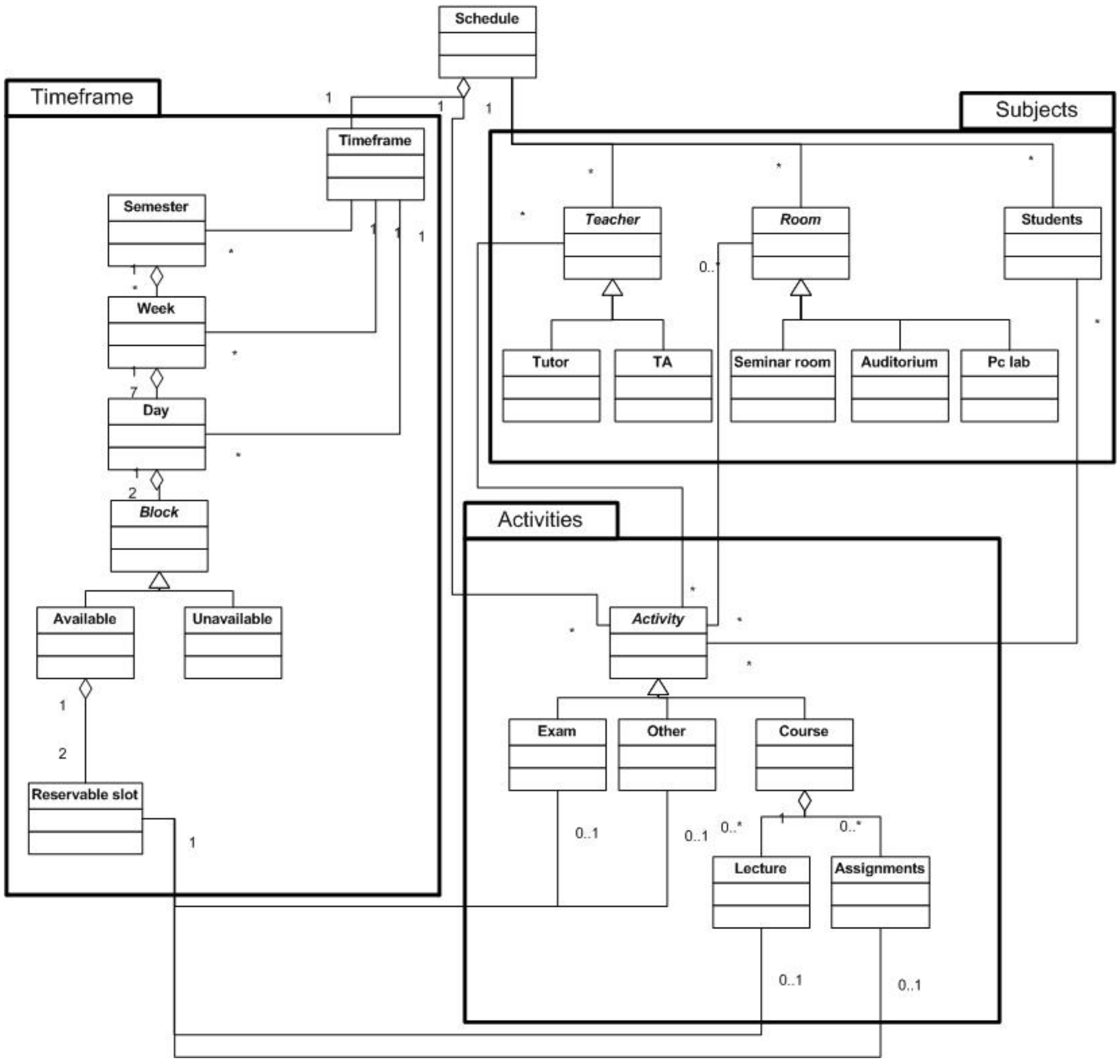


Figure 1.4: Class diagram of the scheduling system.

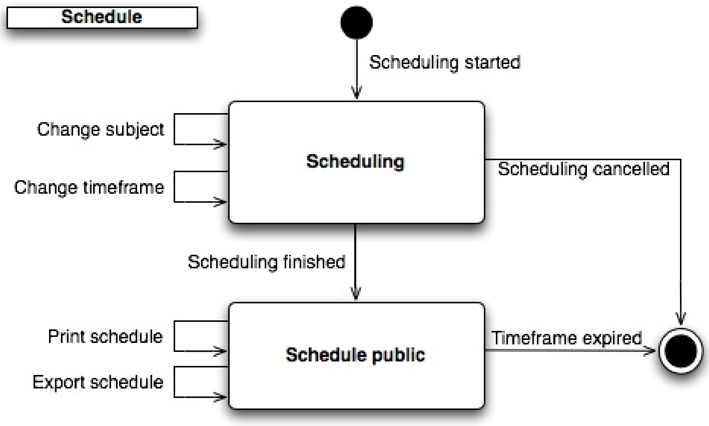


Figure 1.5: Behavioral pattern of the *Schedule* class.

Day

Day is a single day in the schedule. As activities are scheduled either in the morning or in the afternoon, it is an aggregation of two *Block* classes. Its only attribute is a date.

Block

Block is an abstract class that describes a single assignable block in the schedule. A block can be in one of two different states: Either it is available for scheduling (for example on weekdays), or not available (for example on weekends or holidays). This means that a block object will either be of the *Available Block* or *Unavailable Block* class. Unavailable blocks are trivial, since the schedule at all concerned at all with them and they can be ignored. Available blocks are an aggregation of two *Reservable Slot* classes.

Reservable Slot

At the institute, each morning and afternoon block is further divided into two halves, since lectures for the most part only occupy half a block. These slots range from 8:15 to 10:00 and from 10:15 to 1200 in the morning block, and from 12:30 to 14:15 and from 14:30 to 16:15 in the afternoon. Since rooms are a sparse resource, it is more economic to only reserve their use for a single slot instead of the whole block. Each slot has an associated *Activity*, or none in the case that it is free.

Subject

Subject is an abstract class that is the basis for the different types of subjects, namely *Teacher* (itself an abstract class), *Room* (also abstract) and *Students*.

Teacher

The *Teacher* class is a generalization of *Tutor* and *TA* (Teaching Assistant). It holds general information such as a name and a list of the blocks where he is available for teaching. Furthermore, the class has associations with all the activities the teacher is currently involved with (courses as well as individual events).

Tutor

A tutor is an employee of the institute hired to conduct lectures. He may have demands of his own concerning prefered days and hours of the week, which must be taken into consideration when planning the schedule. He may also have preferences for a specific type of room to conduct the lectures in, these are taken into consideration by letting him teach in the specific room type that he prefferes if this is possible. Attributes are those inherited from *Teacher*.

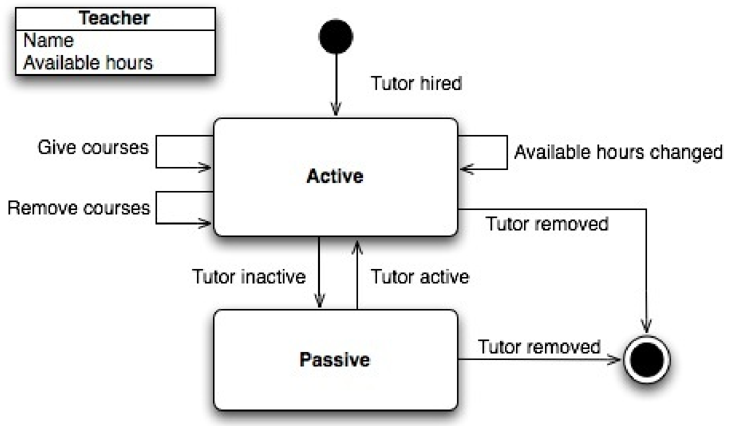


Figure 1.6: Behavioral pattern of the *Teacher* class.

TA

A teaching assistant is hired to assist in the teaching of a course, mostly to help students with problems. Attributes are inherited from *Teacher*.

Room

The *Room* class is a generalization of the classes *Seminar Room*, *Auditorium* and *PC Lab*. A room has a number (describing its location) and a certain amount of seats, which are referred to as its size. It is also associated with all events that it is involved with. Besides that, a room can be occupied or unoccupied. Seminar rooms and the auditorium serve the same purpose of conducting lectures (though tutors may have a preference for one or the other), while PC labs are used to conduct excercises.

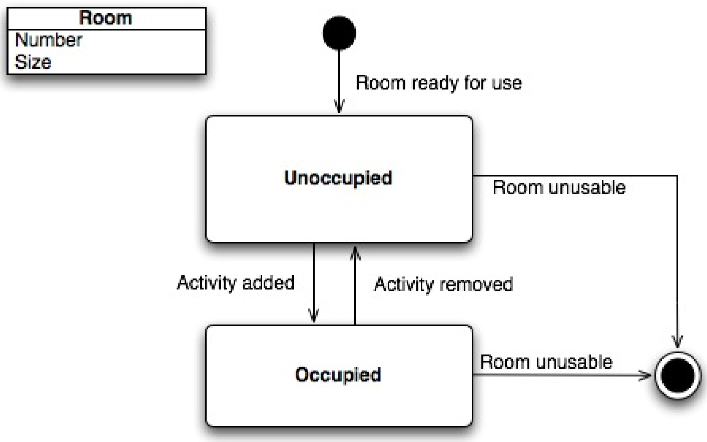


Figure 1.7: Behavioral pattern of the *Room* class.

Students

The *Students* class holds information about all the students at a given semester that follow the same set of courses. Each set is associated with all activities the students partake in. The set is named after the semester (e.g. DAT1 or SW3). It also contains the number of students in the set.

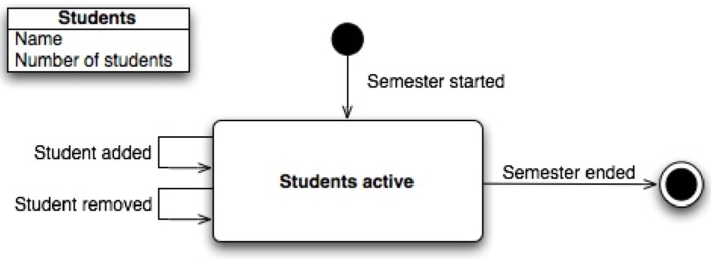


Figure 1.8: Behavioral pattern of the *Students* class.

Activity

The *Activity* class is a generalization of the classes *Course*, *Exam* and *Other*. When the schedule is planned, the activity is associated with a specific time slot. Futhermore, subjects (tutors, rooms and students) can be tied to it. The combination of subjects that are associated with an activity depend on the type.

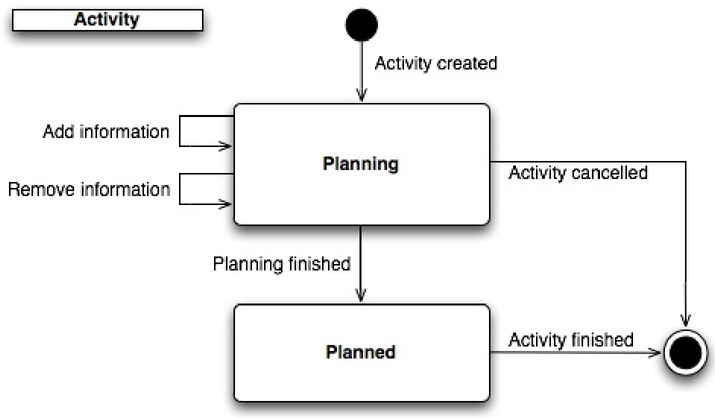


Figure 1.9: Behavioral pattern of the Activity class

Course

Course is a general course that runs over a period of time. It is associated with a number of teachers and students. Each course is an aggregation of individual lectures and assignments, which are associated with time slots. Each course has a name as attribute.

Lecture

Lecture is a single lecture at a specific time. It is assigned to a time slot. Each lecture has the same associations to teachers and students as the course it belongs to, as well as to a specific room.

Exercises

Exercises is the time in a block assigned to working on problems in the project groups. It is assigned to a specific time slot. As *Lecture*, it inherits associations to teachers and students but does not have an association with a room since the students are spread out over the institute instead of gathered in the room used for the lecture.

Exam

Exam requires associations with a room and a number of student sets. The student sets are inherited from the course. Teachers are not always required however, so their association is optional.

Other

Other is a catch-all class for activities that do not quite fit into the other categories. It is useful for dealing with meetings and other events that might clash with normal activities, but are loosely defined. They may have any combination of associations with subjects.

1.2.3 Events

In this section, the various non-trivial events which occur in the problem domain are described. Non-trivial events will be described thoroughly.

- **Semester end**: When the semester ends, all courses become inactive, all students are removed and all rooms become available.
- **Room becoming unavailable**: In the event of renovation or other construction-related work, a room may become unavailable.
- **Room reservation**: When a course is scheduled for a specific day, it also needs a location in where it is to be held. When this is the case, a given room may be reserved for this course.
- **Course cancellation**: If for some reason the assigned tutor or room becomes unavailable, the course may be canceled.

Events are some of the most important things in the objects they belong to. Examples include rooms becoming unavailable and room reservations. Both events belong to the room object. For all the other events the same principle of them belonging to an object holds.

1.3 Application Domain

1.3.1 Usage

Stakeholder Analysis

The stakeholder analysis will be used to identify the persons directly or indirectly affected by the system. Furthermore, it helps to decide which users are to be involved in the project. The primary stakeholders are involved with the application domain in the system definition, as they are the users of the system.

Primary Stakeholders: Secretaries

The secretaries have been identified as the primary stakeholders. The secretaries will be using the system to plan, edit and view the schedules. All the users who view the schedule regularly rely on the secretaries to plan it, therefore the secretaries have the highest degree of responsibility. As the primary users of the system, any changes will affect them directly, and thus they are essential in the development of the system. The secretaries will be involved as much as possible, in order to get a good amount of useful feedback during the project.

Secondary Stakeholders: Tutors

The tutors have been identified as the secondary stakeholders. These users are only indirectly affected by changes to the system, as they are only using it to view the schedule created by the secretaries. However, they have a higher degree of responsibility than the students, as they have a job to fulfill. We have chosen not to involve these users as they will not be using any essential parts of the system.

Tertiary Stakeholders: Students

The students have been identified as the tertiary stakeholders. As with the tutors, these users are only using the system to view the schedules created by the secretaries. Because the students are only attending the courses, they have the lowest degree of responsibility. We have chosen not to involve these users as they will not be using any essential parts of the system.

Actors

The actors that interact with the system are:

- Gatherer
  The gatherer actor gathers all the data from the different parties involved in the schedule, and pass it on to the organizer.

	Gatherer	Organizer	Planner	Editor	Viewer
Gather data	X				
View data		X	X	X	
Organize data		X			
Merge data			X		
Create schedule			X		
Modify schedule				X	
View schedule					X

Table 1.1: List of the actors and their use patterns.

- Organizer
  The organizer receives the data from the gatherer and organizes it in its different parts, thus providing an overview for the planner, while making sure everything is accounted for.
- Planner
  The planner is the actor that plans the actual schedule, by using the information put forth from the organizer. When the planner is finished with the schedule, the schedule is locked to prevent major changes, and made public for the viewer actor to see.
- Editor
  After the planner is finished with the schedule, the only one allowed to make changes to it is the editor actor. The editor can make minor changes to the created schedule. These changes could for instance be to cancel a lecture or move a lecture from one room to another etc.
- Viewer
  After the planner is finished with the schedule, it is made public for the viewer actor to see, which is the only interaction the viewer is allowed to have with the schedule. In case the editor needs to make a change to the schedule, the change is immediately made public, allowing the viewer to see the changes in the schedule without significant delay.

Scenario

The following is a made up example of how our chosen persona might experience using the system. It is a fictional account that reflects the normal work routine she goes through when using the system.
It is the beginning of a new semester. Alice, the secretary, is sitting in her office and has just received word from the board that she can start constructing the schedule for the new semester. Alice has already gathered a lot of information from the various tutors as to when they want to have their courses. She gets comfortable in her chair, adjusts her table, powers up her computer and enters the scheduling system.
After logging in, she starts out with adding all the new tutors who have been hired since the last semester to the system. Luckily, most of the tutors are already in the database and only need slight adjustments to their requirements and requests. One major difference this year is that the institute has moved to new buildings. These buildings have a whole new

set of available rooms, which need to be entered into the system. While doing this, she goes over the old list and deletes the ones that are no longer in use. She enters the active lists of students and courses into the system as well. Finally, she enters the dates into the calendar where lectures are not to be held because of holidays. She instructs the system to start generating a schedule and goes to get a cup of coffee, knowing well how long this will take.
Returning after tending to her other duties, she starts to go over the finished schedule. Discovering a few undesirably placed lectures, she returns to the system to make some amendments to the criteria. After a few tries doing this she is satisfied. She saves the schedule, prints a few paper copies and prepares to publish it on the intranet.

Use Cases

The use cases describe what privileges the various user classes have. The system is designed to interact with five different actors, namely the gatherer, the planner, the editor, the organizer and the viewer. These five classes of actor groups and the various functions are displayed in the following chart. Further explanations of the various functions which are also displayed in the chart will follow.

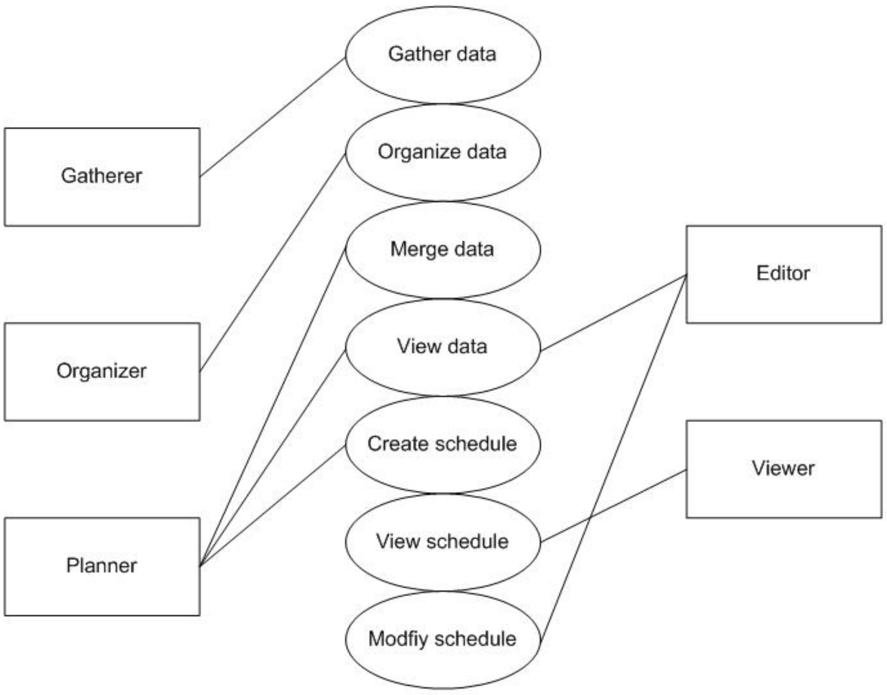


Figure 1.10: Statechart diagram of the use cases.

The different actors will be explained next. When the system is started the gatherer actor is needed. This actors job consists solely of gathering information, which is later used. When the gatherer has completed his or her goal and has gathered all the needed information, the organizer actor is introduced and the gatherer actor discarded. As before this actor also only has one task, to organize the various data collected by the gatherer. When this task is completed the organized data is handed on to the planner actor. This actor has various

tasks. First and foremost the actor merges the different data received from the organizer actor. Dividing the amount of data into lesser subgroups helps the planner actor with its next task, creating a schedule. At any given time the planner actor has access to and can edit the data received from the organizer actor. But after the actor creates a schedule, the planner actors job is done.
At this point two new actors emerge. The viewer actor has one task and one task only. To view the created schedule. This actor can be accessed continuously through the entire system process, after the schedule has been created. The editor actors task is to access the schedule at any point after it has been created and edit where need be. The editor actor can make changes to the schedule if there is a need and can do so during the entire system process.

The statechart digram of use cases, displayed in figure 1.10 is elaborated on next:

- **Gather Data**
  Here the different data is gathered from the various sources. Different data needs to be gathered before the system can continue.
- **Organize Data**
  This use case is used by the organizer actor to organize the data gathered from the gatherer actor.
- **Merge Data**
  The planner actor will merge all the data organized by the organize actor, into lesser subgroups, which will help when creating the schedule.
- **View Data**
  Both the planner- and the editor actor has access to this use case. It contains the merged data from which a schedule can be created.
- **Create Schedule**
  This use case is initiated by the planner actor and will generate the schedule, based on the merged data, organized by the organize actor.
- **View Schedule**
  The viewer actor has this use case as its only task and is mainly used as a mean to gain access to the created schedule without editing it.
- **Modify Schedule**
  The editor actor is the only actor with access to this use case. It is used mainly when changes needs to be made to the schedule.

To give a better idea of our general user, a persona analysis is used.

Persona

Alice is a 34 year old secretary at the Department of Computer Science. Her job is first and foremost to make everything go around. She handles both the students' as well as the tutors' problems. Besides that, she handles the budget for the institute and is in several committees that help various parts of the infrastructure. She has a Higher Commercial Examination and has been in training at the institute for several years before actually starting her current job. Her experience covers over 15 years where she has learned everything from handling websites to arranging schedules, dealing with the economy etc.
Alice thinks that her job is sometimes very stressing and time consuming, but on the other hand she really enjoys it and as she says:

> The day I wake up and say to myself that I really don't wanna go to work today is the day I'll find another job.

Even though she works a regular 8 hour shift from morning to afternoon, her hours often stretch themselves further. Alice has a good knowledge of computers and systems at the institute, though she says herself that she is in no way an expert. Her proficiencies are many, rather than a few where she can excel. She has no problems with new technology and will embrace it if she can see its logic and that it will help her. That being said, she does not aggressively pursue new solutions and new technology.
In her line of work, Alice often interacts with new projects of various nature. She therefore has no problem with getting instructions about how a new piece of equipment works. Even though she sometimes recognizes that she has an interest in learning how new systems work, she also knows that her job takes a lot of time, and that she will not have a lot of free time to get to know the new systems. She thinks that courses where she can get a good and thorough introduction is a really good idea when introducing new systems or software. But as she also says:

> I have no problem learning these new things for myself, if I have a good instruction manual.

Picture source: http://commons.wikimedia.org/wiki/Image:AuroraRobson.jpg

1.3.2 Functions

In this section all the system functions will be described. To give a better picture of how the functions of the system work, a complete function list has been produced. There are four types of functions: Updating, signaling, reading and calculating. All functions need to be categorized within these four. Another thing that needs to be determined is whether the various functions are complex or simple to implement. Simple functions will be regarded as trivial, while complex ones will be described more in-depth.

Function	Complexity	Type
Add tutor	Simple	Update
Show tutors	Simple	Reading
Edit tutor	Simple	Update
Delete tutor	Simple	Update
Show students	Simple	Reading
Add student	Simple	Update
Edit student	Simple	Update
Delete student	Simple	Update
Show rooms	Simple	Reading
Add room	Simple	Update
Edit room	Simple	Update
Delete room	Simple	Update
Show courses	Simple	Reading
Add course	Simple	Update
Edit course	Simple	Update
Delete course	Simple	Update
Show holidays	Simple	Reading
Add holiday	Simple	Update
Edit holiday	Simple	Update
Delete holiday	Simple	Update
Open schedule	Medium	Update
Generate schedule	Complex	Calculation
Show schedule	Medium	Reading
Save schedule	Medium	N/A*
Print schedule	Medium	Reading
Export web version	Medium	Reading

Table 1.2: List of functions and their complexity * = Note that the save schedule function does not have a type. This is because the function does not affect the model state in any way and that it is only included for the convenience of the user.

As seen in table 1.2, the majority of the functions are simple. Showing the different objects is as simple as retrieving a list of them, and the functions to add, edit and delete the objects are trivial, as they are straight-forward operations on simple data structures.
Somewhat more complex are the functions to load and save data, as it will be necessary to convert to and from a suitable format. Showing (as on the screen) or printing a schedule will require parsing all assigned slots and generating a graphical representation of them. Exporting the web version (in the format of a static page) requires generating an HTML representation, but is not otherwise complicated.
The heart and soul of the system lies in generating the schedules. The complexity of this single function dwarfs the others. The function will need to take into consideration a set

of constraints and produce a number of viable candidates, which must then be evaluated from a set of criteria to determine which one is the best suited. The process is costly and fraught with problems, and no simple efficient solution exists. The intricacies of choosing and implementing a useful solution is described in the design document.

1.3.3 User Interface

Usage Goals

In order to determine how our system should be developed usability-wise, a number of key elements are discussed. These are all rated on a scale of importance ranging from irrelevant to very important. Both usability factors and user experience when using the system are taken into consideration:

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

Table 1.3: Usability and experience design criteria for the graphical user interface.

As seen in table 1.3 the usability criteria are described first:
Effectiveness is very important, as the system must be able to complete the task correctly. Efficiency is also very important; we do not want the user wasting time waiting for the schedule to be generated. Safety is important, as the system must ask the user if he or she wants to save their work before exiting as well as have other mechanisms to ensure that there is no data loss. Utility is very important, the system must be easy to use and intuitive. Learnability is less important as we imagine there would be a course used to teach the user how to operate the system. Memorability is very important, as the system is only used a couple of times a year and the user should not need to participate in more than one course to use the system.

In terms of user experience when using the system, the user must feel some degree of satisfaction when using the system, as the user is using it to solve a rather difficult task that would otherwise take up much more time if done by hand. It is irrelevant whether or not the user is having fun, finds it enjoyable or is entertained when using the system. It is less important if the system is helpful, but it should however always be able to give help when needed. It is irrelevant if the system is motivating. The overall look of the system should be aesthetically pleasing, though it is irrelevant if the system is supportive of creativity. The user should feel that it is rewarding to some degree when using the system because it is helping him or her with a difficult task. Last but not least, it is irrelevant if the user finds it emotionally fulfilling to use the system.

Conceptual Model

The system is going to be based on manipulation and navigation, the basic functionality will be for the user to navigate to the desired function, for example inserting or extracting data. The reason why this method is used is that when inserting a vast amount of data, it will create an overview of the current work. For example, if you are entering tutors or rooms it is intuitive which type you are processing. Conversation between the system and the user is also going to take place, this is combined with a safety principle; asking the user if he or she really wants to execute a specific command. For example, when exiting the system, it will ask if the user is sure so they will not experience a loss of data.

Interaction Forms

The system is going to use menus and dialogs as the forms of interaction. The menu is the primary form because this will guide the user through the different parts of the system and into the functions which are needed. The menu will also bring a more intuitive perspective into the matter, as the menus are supposed to tell exactly what their purpose are, which will give the best interaction. The dialog is used when the user is doing something that might endanger what they have been working with so far, for example if the system is told to shut down, it will ask if this is the intention or if overwriting another file is the purpose.

It is imperative, when working with systems, to define how the system should interact with its designated users. A specific interaction form is used to give a better understanding of how the users will react to the different graphical representations of data. When working with interaction forms there are many ways to use these. If for instance a system has numerous complex parts, which act as individual systems, it can be a good idea to use the same interaction form on all of them. In this case a general interaction form has been chosen.

General Interaction Model

The model is a straightforward graphical representation of how the system should interact with the user. The first thing that happens when the user opens the system is that the system will request a password so it can recognize the user. This part will also be referred to as the *login menu*. When the user has gained access to the system, the system takes the user to

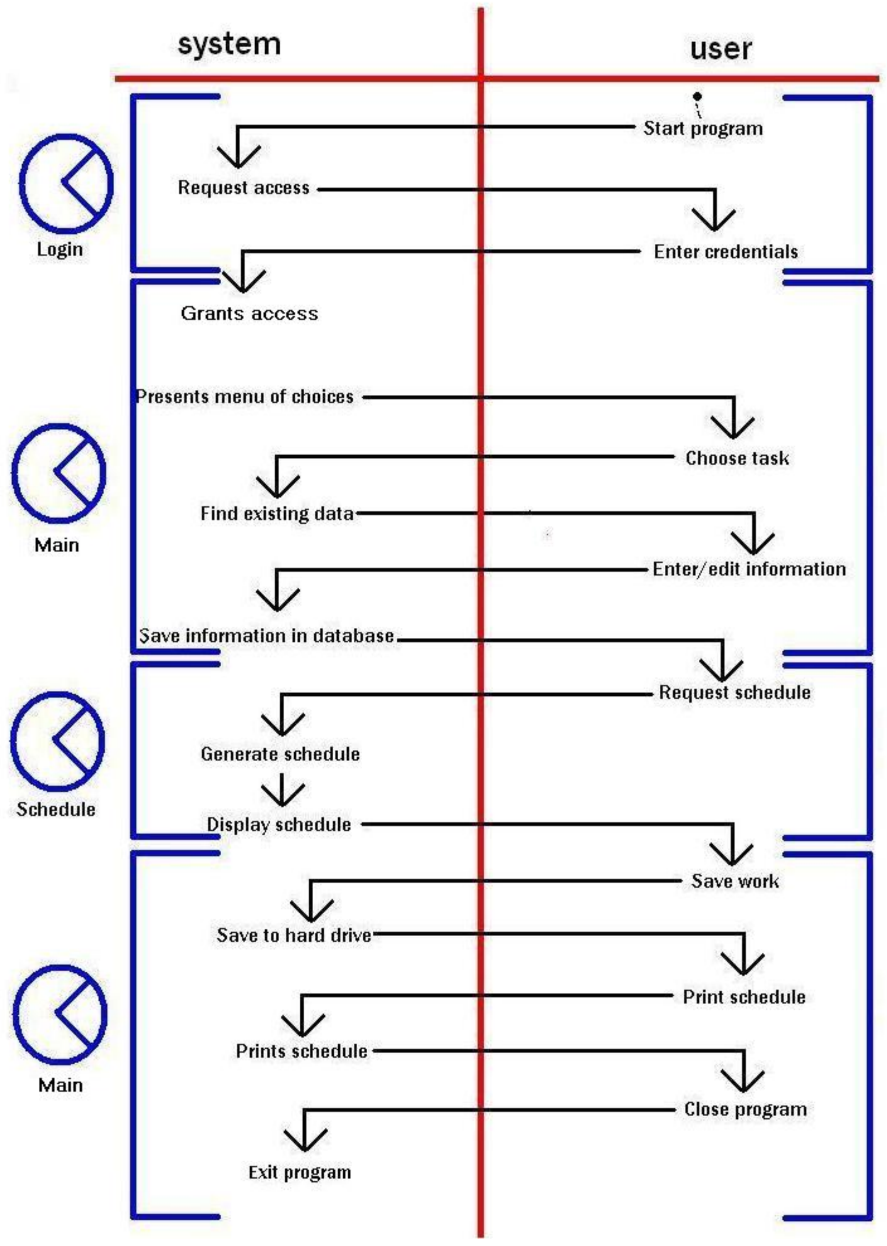


Figure 1.11: This model shows interaction between the user and the system.

the *main menu*. Basic tasks will be accessible from here and the user can edit information variables, save and load schedules etc. If the user chooses to issue the *request schedule* command, the system will enter a submenu called the *schedule menu*. This menu is one of the most important parts of the system, as it is in here that the user will choose the *generate schedule* command. When a schedule has been created the user has a few new options on the *main menu*. For instance, the user can now save the generated schedule, print the schedule, export the schedule to HTML, etc.

In figure 1.11 we can see that we need 3 interaction spaces, namely: The *login screen*, the *menu screen* for inputting, editing data and all the operations for the schedule, such as saving, printing and so on, and the *schedule screen* for viewing the schedule. Many of the possible things to do are located in the menu interaction space, to keep the possible tasks visible to the user at all times, which should provide an overview of the programs capabilities.

1.4 Technical Platform

The system is to be implemented as a stand-alone program, since the system is restricted to generating schedules, and is not meant as a way for tutors and students to access their schedules. Using a stand-alone application also creates a higher security level, as others should not have access to the schedule generating facilities. Since it is a requirement that the system is to be written in C# and use .NET, the stand-alone system should have Microsoft Windows XP or newer installed along with some version of the .NET platform. Since generating schedules can be quite computationally expensive, the hardware should be reasonably up to date to minimize the time spent waiting for a result. This does not mean that it should be necessary to invest in hardware specifically for this matter, any computer that could reasonably be expected to be used in an office setting at the faculty today should suffice.

1.5 Recommendations

1.5.1 Usefulness and Feasibility

We consider that further development would be appropriate, as schedule planning is still to this day mostly done by hand, which is a very demanding process both in respect to work hours and resources. Our argument for continuing the development is that it is believed that the finished system will significantly reduce the amount of time and resources that are needed for scheduling.

The system demands are, in relation to the work and technical environment: The system does not require much from either the user or the technical platform. The system is designed to be relatively easy to access and requires only that the user complete a short course in using the system, before it can be included in daily operations. The system also aims for the user to use the system after a long period of time without the necessity of additional education or training.

1.5.2 Strategy

By and large this project will be developed using an iterative process. This means that when researching or writing new material, every part of the analysis as well as the design document and implementation need to be revisited and reviewed throughout the project to ensure coherence.

1.5.3 Development Budget

We estimate about 230 work hours per person in the group in order to finish the design document and implementation over the next few months. This gives a total of:

$$6 \cdot 230 = 1380 \text{ hours} \tag{1.1}$$

2 Design Document

2.1 The Task

2.1.1 Purpose

The purpose of the system described in this document is to automate the scheduling task for The Department of Computer Science. The scheduling process is automated in order to address one major issue in the manual process: namely to minimize the sheer amount of work hours put into creating a schedule. This is done in order to free work hours for the people involved in the schedule creation process, to boost the efficiency of the process and to avoid possible human errors. The purpose of the system thus indorse strict reqiurements to the efficiency, speed and correctness of the system.

2.1.2 Corrections to the Analysis Document

Teachers

In the analysis document a tutor has been referred to as the person teaching a course, and a teaching assistant (TA) as a person who helps during assignments. Within a design perspective there is no need to have the assistant represented in our system, as he or she has no effect on the outcome of the schedule. Henceforth the person assigned to teach a course will be known as a *teacher*.

2.1.3 Quality Goals

The primary goal of the system is to effectively generate a usable schedule. To accomplish this, we must ensure that the system fulfills certain quality goals. The quality goals will serve as a design specification for our system.

Our choices in table 2.1 are elaborated in further details below.

- *Useable*
  It is important that the system is useable, as it should be easy to use, even though the system is targeted at expert users.
- *Secure*
  It is important to avoid malicious access and use of the system. The generated schedule is to be used for a large number of people and the schedule should always be reliable and correct.

Topic	Very important	Important	Less important	Irrelevant	Trivially fulfilled
Useable		X			
Secure		X			
Effective		X			
Correct	X				
Reliable	X				
Maintainable			X		
Testable		X			
Flexible			X		
Comprehensible		X			
Reusable				X	
Portable				X	
Mergeable				X	

Table 2.1: Table of design criteria

- *Effective*
  It is important that the system is effective as the task of generating a schedule is a very time-consuming process.
- *Correct*
  It is very important that the system is correct, as the system needs to ensure that schedules do not overlap: Teachers are not needed at more than one place at a time, only one lecture is scheduled for a room at a time, and students taking several courses are not scheduled for more than one at a time.
- *Reliable*
  It is very important that the system is reliable. As there is a vast amount of data manually plotted into the system, these may not be corrupted or deleted by crashes or errors. System crashes should however not occur at all.
- *Maintainable*
  It is less important that the system is maintainable as the system is meant to be a complete solution that should not need any patching or maintainance.
- *Testable*
  It is important that the system is testable, as the system is to be tested for funtionality errors throughout the development phase. In the end of the implementation phase, the system should also be tested on the targeted costumer in the usability lab.
- *Flexible*
  It is irrelevant that the system is flexible as it is targeted at a very specific user group.
- *Comprehensible*
  It is important that the system is comprehensible as the system needs to be relatively intuitious and helping, so the user will not need to learn everything from scratch each time the system is used.
- *Reusable*
  It is irrelevant that the system is reusable as the system is made for a specific purpose and platform, so it is not meant to be reused, merged or ported to other systems or platforms.

- *Portable*
  See *Reusable*
- *Mergeable*
  See *Reusable*

2.2 Technical Platform

The requirements for the system match those set forth in the analysis document, though they are elaborated here.

Equipment

A single desktop PC is required. We have only one specific requirement as to its hardware capabilites; since we use the .NET framework, the hardware must be able to run at least Windows XP to a reasonable degree. To ensure a reasonable speed in generating schedules and to prevent unresponsiveness, the target hardware should be of adequate performance. Any system already in use by the secretaries at the institute should be able to fulfill these requirements without any problems. Additionally, the PC should be connected to a printer in order to get paper copies of the schedules.

System Software

Development of the system will be done in C#. For the final system the software requirements will be the operating system Microsoft Windows XP/Vista, with .NET 2.0 support. We do not support the Mono framework, especially since we use Windows Forms as the graphical user interface library, which is not supported by Mono.

System Interface

Since the system is envisioned as a stand-alone application running on a single desktop PC, we have no special requirements concerning this.

Design Language

We use standard UML compliant notation throughout the desgin document, with one exception: In order to reduce the amount of trivial operations in the classes of our class diagrams we have chosen to write private (-) properties as public (+) and with the first letter in upper case, e.g +CourseName. This should be interpreted as if the property is private, but with standard get and set methods, i.e it is public through the use of these methods.

2.3 Architecture

2.3.1 Design criteria and requirements

When defining a systems architecture, it is imperative to discover what design criteria the system has to fulfill and to prioritize these criteria. This will help give the system a more structured outline, as well as set the specific demands that needs to be fulfilled by the system.

The following criteria has been chosen as general criteria for this particular system and prioritized:

1. The system needs to be effective at completing the task it is designed for, namely creating schedules
2. The system needs to do this efficiently and within a limited timeframe
3. The system has to be reliable and as such must not perform unexpected actions
4. The system needs to be flexible under these circumstances
   - Different hardware specifications
   - General interaction with the user
5. The system needs to be intuitive and easy to use in such a way that the user will not forget between sessions
6. The system needs to be pleasant to look at and follow the basic standards of the windows operating system

The prioritation of these criteria is based on the following arguments. As the system is made with the sole purpose as to ease the workload of schedule creation, by generating usable schedules based on input data, it is imperative that the system is effective at this task, as to fullfill its designated purpose which of course is the main priority. The second priority is that the system ends in a foreseeable timeframe, as it would be unpractical to order the system to generate a schedule and then having to wait days for it to finish the schedule. The third priority of the system is reliability, as it is important that data will not be lost by unforseen errors, and that the system as stable enough to counter any foreseen errors, without anything more then minor inconviniences for the user. The fourth and middle priority is flexibility, as the system should be able to run on any machine that meet the system requrements as descriped in section 2.2.The criteria with the least priority is memorability and eastetics, as though it should be in the system, everything else on the list is more important. These criteria is also prioritized to fulfill the basic system development demands: Effectiveness, efficiency, safety, utility, memorability and to some extend learnability. Furthermore the system is to have a degree of standardization with the windows operating system, which will give it a somewhat aesthetically pleasing look.

To further the understanding of the systems development certain architectonic demands needs to be constructed. These will not only give a basis for the development of the system, but will also serve as a starting point for possible changes that needs to be made in later stages of development. The architectonic demands for the system is as follows:

Technology

The system will be developed almost completely from scratch and will therefore only contain a small amount of premade system components. The system will take advantage of some predefined windows components such as the print component, as well as the basic windows form for loading and saving. As defined in the technical platform the system will be based on .NET 3.0 technology with backwards compatability towards .NET 2.0.
This choice is made because of two main reasons:

1. The backwards compatability will allow older machines to run the system
2. The .NET 3.0 technology will allow for the use of more advanced programming technology.

The Human Factor

The people involved in the design process all have a general knowledge of the technical platform as well as how the windows operating system works. The same people also have some knowledge about the programming language C#. New skills regarding the .NET 3.0 framework as well as additional C# and XML programming will have to be acquired during the design and development processes. To some extend a basic knowledge of the mathematical problem behind scheduling algorithms will have to be acquired as well, as it is essential for the development process. In general, the people involved in the design and development process have no experience with earlier systems of this type and little to no experience with making a schedule.

2.3.2 Generic Design Decisions

When working with a system it is imperative to make key choices in the design phase. These choices will affect the outcome of the system and will help to give a more general idea of what ideas and choices that have been put in motion. A list of generic choices have been worked out as the following:

Error Handling

In the event that a critical error should occur while the system is in use, an error message is sent to a specific error handler class in the system. This class reads the error ID, and shows an error message to the user which matches the error ID. At the same time as the message is shown, the error handler class tries to save the data stored in memory to a temporary file so that the data can be recreated without the need for the user to start over. After saving the data, the error handler tries to correct the error. If the error is corrected, the user is asked to save his/her data and restart the system, as the system might have become unstable. If unsuccessful in correcting the error, the error handler will give a new message to the user, and the program will shut down. In this way, the system will send one of these three messages to the user, in case of a system error:

1. A message describing in simple terms, that en error has occured and what the cause of the error was. Such a message could look like this:
   A system error (ID#) has occured. The error was caused by trying to place more lectures then there is rooms at the same time. The system will now try to save your data and correct the error. You should save your data as a backup and restart the system, as the system might have become unstable.
2. A message informing the user whether or not the system was able to save the data and correct the error. This error could look like this:
   The system was able to save your data and correct the error. You should save your data as a backup, and restart the system, as the system might have become unstable
3. Or a message informing the user that the system was unable to do anything. This error could look like this:
   The system was unable to save your data or correct the error. The system have become unstable. Please save your data as a backup and restart the system.

In case of non-critical errors, the system will send error messages to the user, in a shorter and more specific manner. Examples of this is show here:

- If trying to input a name in a number field, the message could look like this:
  Use error (ID#). The input you are giving is invalid. Please input a number in the field.
- If trying to close the system without saving the data, a message could look like this:
  Use error (ID#). You are about to close the system without saving. Are you sure you want to do this? (This show two buttons, one labeled "Save and Quit", the other labeled "Quit without saving")
- If trying to create a shedule without enough data, a message could look like this:
  Use error (ID#). You do not have enough data to create a schedule, please input sufficient data. (a variable determined by the missing data show Rooms, Tutors, Classes ect.) missing.

Progress Bar

The system should create the best working schedule possible, regardless of the time consumed in the process. This does not mean that creating a schedule must take days. To give the user an idea of how far in the process of generating the schedule the system is, a progress bar will be implemented. This bar will give the user a percantage ranging from 0% to 100%. This number will be determined by amount of weeks in the semester for which a schedule is created.

Ease of Navigation

The system needs to be easy to navigate at all times. This is archived by keeping a certain amount of congruity in the input screens, which helps to make the system as clear as possible for the user. The system will make use of both the mouse and the keyboard for navigation, providing a balanced amount of shortcut keys for the expert user. The shortcut keys will be the keys *F1* to *F10*, covering the main buttons on the GUI main screen, with *F1* activating the *help* screen and *F10* generating the schedule. The *tab* key will be used as well, to circle through the data fields on the individual input screens. If the user does not wish to use the shortcut keys, the system can be navigated with the mouse. In both cases, only the keyboard is used for inputing data.

Undo

In most common IT-systems there is a function which allows the user to undo earlier actions. This function is not present in the system, due to the fact that it is simply not needed, as any data that the user might want to undo can be so easily changed by inputting the data anew, that the difficulty of implementing such a function, far outweights the benefits of having it implemented. If for instance a user wants to input data regarding a course into the system, and makes an error, the user will have to re-input that particular piece of data only, instead of starting over or going back.

2.3.3 Component Architecture

The architecture components of the system are layered and consists of four major components: graphical user interface, functions, model and persistence. The graphical user interface component contains all the classes within the user interface. From the graphical user interface component it is possible to call methods downwards in the architecture, e.g. to the function component. Methods are called when a user operates the system. The function component receives and processes method calls from the graphical user interface component. This means calling the correct methods for the model and persistence component at the right time. The function component can use methods from the persistence component if a collection of objects needs to be retrieved, e.g. a full list of rooms. The model component is used to model the systems problem domain, and information from this component is used in the function component. The methods used on the model component are generally used for saving and loading data. When data in the model component has been updated, a method is called to the persistence component to save the modelled data. The persistence component can receive methods from the function and model component, but it can not issue any method calls. The persistence component can send data to a datasource, e.g a database or files - in this case an XML document.

Component Privileges

This systems architecture has some terms connected to it. First whether or not it is a *closed* or *open* architecture. If the architecture is closed then the different components can only access adjacent layers. If the architecture is open it means the different components can access any layer they choose.

There is another term, which also needs to be chosen, namely if the system is *loose* or *strict*. If the system is loose then the different components can use any layers they want,

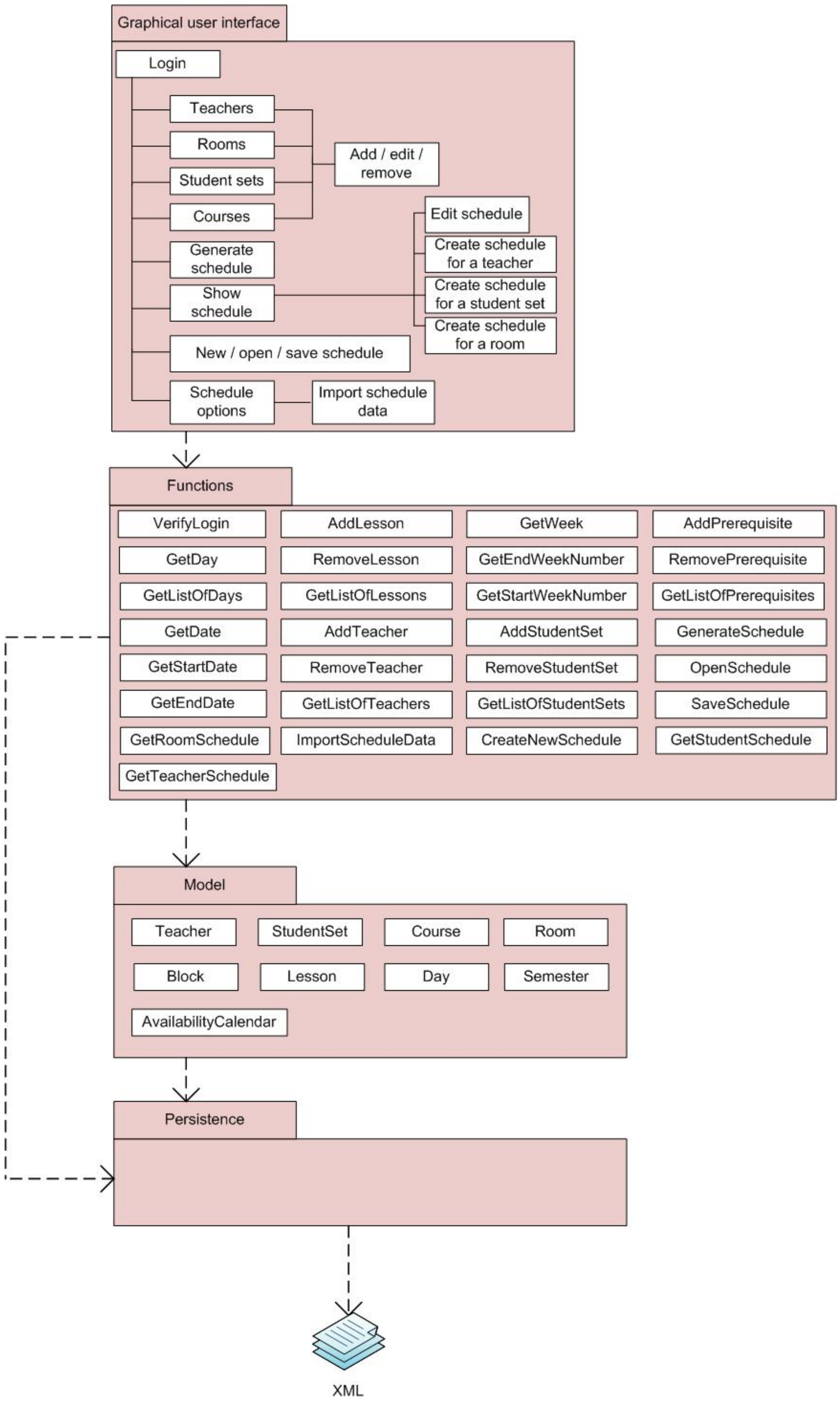


Figure 2.1: Component architecture of the scheduling system

regardless if they are on a higher or lower level. If on the other hand the system is strict, then the different components can only access layers below their own.

In the schedule planning system we operate with an open and strict architecture. These are chosen because the planning system components needs to have access to more than one layer below their own, and in the case of schedule planning, some functions needs to access the persistence layer directly and not through the model layer. The schedule planning system components were also made strict, as we did not want lower level components to be able to access higher level components, as this would not be needed.

Components In-Depth

A more in depth component explanation is given in order to give a better idea of the different components roles and tasks. This section is based on figure 2.1.

Graphical User Interface

The Graphical user interface can be divided into groups, where the groups are based on the similarities in how the user interacts with that part. If you look at the figure, it is noticable that the interactions *Teachers*, *Rooms*, *Courses* and *Student sets* all share a common set of functions, and can thus be considered closely related.
These interactions needs to be addressed before there is enough information to create a schedule. Another group of interactions are the actions regarding the schedule. Whether a new schedule is created, edited, loaded or saved, it shares the common idea of interacting with a schedule. The user can through the graphical user interface chose to create a *New schedule*, which will take the user through the act of inputting data. If the user on the other hand already have some saved data from earlier schedules, which the user can reuse, they can simply choose to import existing data. There is also a *Schedule option*, which contains predefined terms that have been given at the creation of the current schedule. These could for instance be the data concerning holidays and weekends, which needs to be computed before the schedule is created. There is a set of interactions that are focused on showing the user the current schedule. When examinating the schedule the user will have access to look at both the main schedule, but also more customized schedules, which has more focus on what the individual user needs. An example could be what lectures *one* teacher has, or what courses are held in *one* particular room.
Within the *show schedule* you will also find the *edit schedule* interaction. This is where the user will go if corrections needs to be made to an already existing schedule. Of other noteworthy interactions with the graphical user interface, the *login* comes to mind. The login is the first thing that meets the user, when the system is booted up. Interaction with the login revovles around gaining access to the program. If the user has the correct username and password, they can gain access to the whole system. If the user does not have this, they can choose to login as a guest with restricted access.

Functions

The functions are situated a layer below the graphical user interface. It is here that all the functions of the system are located. Many of these functions revolve mainly around adding, editing or removing variables (*Teachers*, *StudentSets*, *Rooms*, *Courses*). These functions are necessary as they tell the layer below to create different objects and what these objects contains. There are also functions which deals with the date and time of the given schedule. These can set the dates for when a newly created schedule will start and end, and get the number of weeks the schedule will run for.
The *login* function is there to verify if the correct username and password is given. This means that the *VerifyLogin* has to access the Persistence layer, to check if this is the case.
When the function *ImportScheduleData* is activated it also has to access the persistence layer for information regarding a previously created schedule.
Other functions centers around adding criteria to courses and create special circumstances for the start and end of the various courses in the semester. There is also a function that handles possible conflicts in the schedule. This function does for instance make sure that a teacher cannot have two courses at the same time. Different functions for the different custom schedules are also necessary, since the user has to have access to different data sets. Functions like *GetTeacherSchedule* helps create a schedule with only one teacher as the focus.

Model

When a function is activated a new object will be created. This is where the Model layer begins. The different functions will often communicate directly with the object. If for instance a function for making new rooms is activated, it require a new object, and the object *Room* is then created and used by the function. Another object that needs to be created is *Block*. Blocks are used for dividing a schedule week into smaller spaces, so that courses can be plotted easily.

Persistence

The persistence layer contains all of the functions that are used when communicating with the data system. In this case the data system is represented in the form of XML documents. The persistence layer will take information, when prompted by the function layer, from the XML documents and load them into the system. When the system is done working on the data it prompts the persistence layer, which then saves the data to the XML document.

Architecture Examples

To give a general understanding of how the architecture of this system works two examples of component interaction is given.

Example one:

In the first example the user wants to create a new *Teacher* to add to the system. The user starts by logging into the system. This initiates the *VerifyLogin* function that quickly goes to the persistance layer to get the right information. The persistance layer contacts the XML file and gets the data it needs and the user gains access to the system. Next the user goes to the *Teacher* interaction and chooses to create a new one. The function *AddTeacher* is now activated and goes to the model layer and asks for a new object. The model layer creates an object, after which the data about the new teacher is created and exists in the system. The data is later stored in an XML documents.

Example two:

In the second example the user wants to start creating a schedule. All data needed has already been inputted. The user has already gained rights to use the system, so there is no need to login. The user now presses the *Create schedule* button, and the user is prompted for a start and end date. Afterwards the user presses the *Generate schedule* button, which activates the *GenerateSchedule* function. This in turn initiates the creation of a schedule with the currently inputted data gathered from the XML file through the persistence layer. The function also takes an object called *AvailabilityCalendar*, which has already been setup with specific dates for when Courses can not be plotted (Weekends, holidays etc.) The system now tries to create a schedule with the given input. During this time a function called *IsInConflict* might be activated. This happens if there are some parts of the schedule which can not be worked out, for instance if a teacher has two courses at the same time etc. After some time, and if all goes well, the user can then see the newly created schedule.

2.3.4 Exemplary Design

When the *GenerateSchedule* function is activated, the system attempts to plan a useful schedule from the entered data. Since this process is very complex, and easily the most difficult to implement in the entire project, the following use case design will hopefully shed some light on the issue.
The scheduling problem we are facing is a complex computational problem, and no simple solution exists. In order to be able to implement a scheduling algorithm, taking into consideration the various constraints, we have designed an approximative algorithm. While faster in execution time and easier to implement than other, more sophisticated algorithms, it is a tradeoff for its ability to generate well-functioning (or in the worst cases, barely functioning) schedules. The algorithm attempts to brute-force a schedule by iterating through a semester chronologically, attempting to plan a single week at a time. Conflicts in the planning are handled by attempting to plan a week over from scratch, using different starting parameters. The same goes for each semester. This gives the algorithm a limited ability to work around conflicts, though if they prove persistent it will simply give up after a given number of tries.

The process of generating schedules concerns two classes: Semester and Schedule. A sequence diagram displaying the progression of method calls between the two can be seen in figure 2.2. A description of each step in this process follows below.

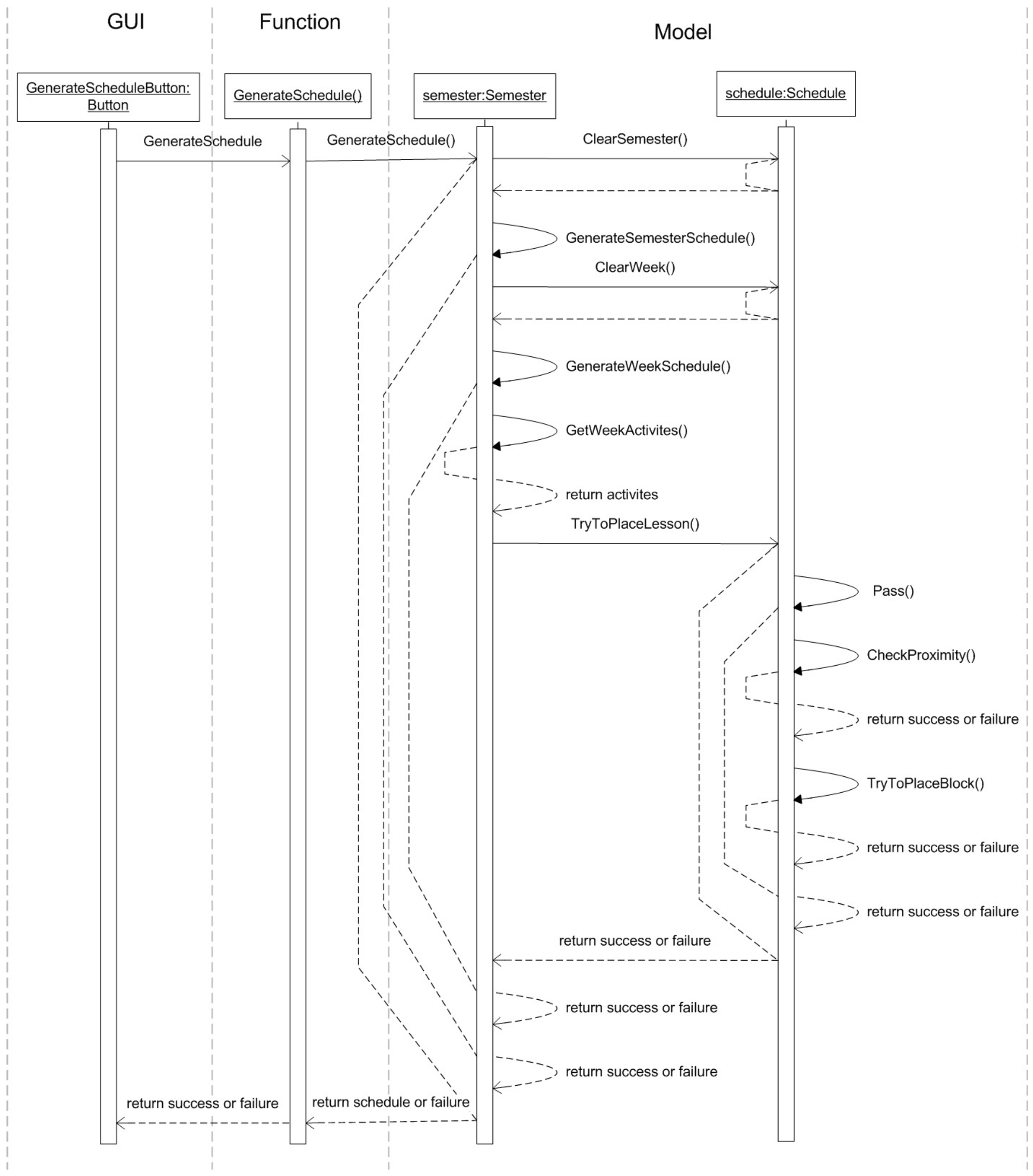

GUI
Function
Model
GenerateScheduleButton: Button
GenerateSchedule()
semester:Semester
schedule:Schedule
GenerateSchedule
GenerateSchedule()
ClearSemester()
GenerateSemesterSchedule()
ClearWeek()
GenerateWeekSchedule()
GetWeekActivites()
return activites
TryToPlaceLesson()
Pass()
CheckProximity()
return success or failure
TryToPlaceBlock()
return success or failure
return success or failure
return success or failure
return success or failure
return success or failure
return success or failure
return schedule or failure

Figure 2.2: Sequence diagram for the *GenerateSchedule* function, showing the nesting of method calls and the interaction between the two objects. The lifetimes of the methods are marked on the left.

The methods below (and their associated classes) are based on a series of algorithms (in pseudocode below) that were designed to handle the problem in the manner described above. The pseudocode fragments that accompany the methods will serve as the base from which the code is to be written during the implementation phase.

GenerateSchedule

The *GenerateSchedule* method, a member of *Semester*, is called from the function layer when the user wishes to start generating a schedule, with the purpose of attempting to create a schedule. If this succeeds the finished schedule is returned and a failure notification otherwise. First, a new *Schedule* object is created. An attempt is then made to create a schedule, by first clearing it (*ClearSemester*) and then attempting to plan it (*GenerateSemesterSchedule*). This process is repeated, should a candidate for schedule fail, until either a viable schedule is created or a set number of tries have been exhausted. Each time a new candidate is attempted, the starting conditions are changed, to ensure that the schedule is laid in a new manner each time, hopefully resolving conflicts by trying out new variations.

Algorithm 2.1 GenerateSchedule()

```
1: semestertries = 0
2: result = false
3: while result = falseand semestertries < 7 do
4:     ClearSemester()
5:     tries = tries + 1
6:     result = GenerateSemesterSchedule(semestertries)
7: end while
8: return result
```

ClearSemester

All associated lessons of the blocks in the schedule are cleared, in order to remove lessons that have been left behind after earlier attempts to plan the semester.

GenerateSemesterSchedule

The *GenerateSemesterSchedule()* method, a member of *Semester*, is responsible for iterating through all weeks in the semester and trying to plan them individually. The idea behind the process is similar to how candidates for semester schedules are generated: First, the week in question is cleared (*ClearWeek*) to ensure that there is no activites left over from earlier passes over this week. Then, *GenerateWeekSchedule* is called. Should a single week fail, the week is tried again with different starting parameters, as above. This continues until a set number of tries have been exhausted, by which point the schedule is abandoned, returning an indication of failure.

ClearWeek

All associated lessons of the blocks in a specified week, specified by start day (by index into the list of *Day* objects) are cleared.

Algorithm 2.2 GenerateSemesterSchedule(semestertries)

```
1: weektries = 0
2: result = false
3: for d = 0 to semester.numberofdays, d = d + 7 do {Iterate through all days that lie at the begin-
   ning of a week.}
4:    while result = falseand weektries < 7 do
5:       ClearWeek(d)
6:       tries = tries + 1
7:       result = G(d, (semestertries + weektries) mod 7)
8:    end while
9: end for
10: return result
```

GenerateWeekSchedule

The *GenerateWeekSchedule*, a member of *Semester*, is responsible for plotting all lessons within one week. It returns success if this is possible, or failure if it is not. It does this as follows: First, a list of all activities that are to be plotted in this week is retrieved (*GetWeekActivites*). The list is sorted to ensure that courses with the largest number of attendants are given the highest priority. The method traverses the list, and attempts to place each individual lesson in a block in the schedule (*TryToPlaceLesson*). The lessons that are successfully placed are removed from the global list of all lessons to be placed in a semester. Lessons that fail to be placed because of conflicts are left in the list, so that they may be pushed forward to coming weeks. Once the list of lessons to be placed has been exhausted, the function returns either success or failure, depending on how many lessons failed to be placed (A critical value is set to prevent that a huge number of lessons are pushed forward. This can be chosen more or less arbitrarily, in the pseudocode below it is set to 2).

GetWeekActivities

GetWeekActivites, a member of *Semester*, returns a list of lessons that are to be placed in the current week. These are generated from the remaining number of lessons per semester and the maximum number of allowed lessons per week, also taking into account prerequisite courses (that is, courses that must be completed before a given one).

TryToPlaceLesson

TryToPlaceLesson, a member of *Schedule*, attempts to place a single lesson within a given week. This is done by doing two passes over the week, starting from a given day. The two passes have different criteria for placing lessons, namely that the first pass will ignore blocks marked with limited availability, as well as unavailable of course, to ensure that we prioritize available blocks. Should the first pass fail however, the second will be envoked. This pass also take blocks of limited availability into consideration, so that lessons that can not be placed in the generally available blocks are placed in the less desirable timeslots. If both passes fail, the method returns failure. Otherwise it will terminate once a suitable block is found.

Algorithm 2.3 GenerateWeekSchedule(weekstart, semestertries)

```
 1: failed = 0
 2: result = false
 3: used[] = NIL {The list of used activities}
 4: startday = tries
 5: activities[] = GetWeekSchedule()
 6: Sort(activities[]) {Sort the list by number of attendants, from highest to lowest}
 7: while activities $\neq \emptyset$ do
 8:    if TryToPlace(first[activities],weekstart,startday) then
 9:       List-Insert(used, first[activities]) {Add the activity to the list of used ones.}
10:    else
11:       failed = failed + 1
12:    end if
13:    List-Delete(activities, first[activities])
14: end while
15: if failed > 2 then {See if the critical number of lessons failed to be placed.}
16:    return false
17: else
18:    while used $\neq \emptyset$ do
19:       usedlessons[course[first[used]]] = usedlessons[course[first[used]]] + 1 {Increment the
          number of used lessons for this course.}
20:       List-Remove(used, first[used])
21:    end while
22:    return true
23: end if
```

Algorithm 2.4 GetWeekActivities()

```
1: activities[] = NIL
2: for each course c in courses do
3:    n = Min( perweek[c], totallessons[c] - usedlessons[c])
4:    for i = 0 to n do {Create this number of new lessons}
5:       course[a] = c
6:       List-Insert(activities, a)
7:    end for
8: end for
9: return activities
```

Algorithm 2.5 TryToPlace(Activity a, weekstart, startday)

```
1: if Pass(a, weekstart, startday,1) = false then {First pass}
2:    if Pass(a, weekstart, startday,2) = false then {Second pass}
3:       return false
4:    end if
5: end if
6: return true
```

Pass

Pass, a member of *Schedule*, is responsible for carrying out the aforementioned passes. Starting from a given day, it will proceed through all blocks in a linear fashion until it is able to place the lesson in a conflict-free block. If this is not possible, it returns a failure notification. This is done as follows: During each iteration of days, *CheckProximity* is called to ensure that the same course is not placed on adjacent days, as it is a requirement

that there is at least a one day break between lessons. If this is satisfied, *TryToPlaceBlock* is called, first for the morning, and then for the afternoon block. If successful the method returns success, and failure if all days are expended with no result.
In the pseudocode below, the availability of a given block in the semester is defined as an enumerated numerical value (1 = unavailable, 2 = limited, 3 = available), and the passes as well (1 = first pass, 2 = second pass).

Algorithm 2.6 Pass(Activity a, weekstart, startday, pass)

```
for n = 0 to 7 do
    index = day + (weekstart + startday) mod 7
    if CheckProximity(index, course[a]) = true then
        semesteravailable = false
        teacheravailable = false
        if pass = 1 then
            if availability[morning[days[index]]] = 3 then
                semesteravailable = true
            end if
            if teacheravailability[morning[days[index]]] = 3 then
                teacheravailable = true
            end if
        else if pass = 2 then
            if availability[morning[days[index]]] > 1 then {Eg. limited or available.}
                semesteravailable = true
            end if
            if teacheravailability[morning[days[index]]] > 1 then
                teacheravailable = true
            end if
        end if
        if semesteravailable and teacheravailable then
            if TryToPlaceBlock(a, morning[days[index]] = true then
                return true
            end if
        end if
        semesteravailable = false
        teacheravailable = false
        if pass = 1 then
            if availability[afternoon[days[index]]] = 3 then
                semesteravailable = true
            end if
            if teacheravailability[afternoon[days[index]]] = 3 then
                teacheravailable = true
            end if
        else if pass = 2 then
            if availability[afternoon[days[index]]] > 1 then {Eg. limited or available.}
                semesteravailable = true
            end if
            if teacheravailability[afternoon[days[index]]] > 1 then
                teacheravailable = true
            end if
        end if
        if semesteravailable and teacheravailable then
            if TryToPlaceBlock(a, afternoon[days[index]] = true then
                return true
            end if
        end if
    end if
end for
return false{Return false when all blocks are exhausted.}
```

CheckProximity

CheckProximity, a member of *Schedule*, checks the preceding, current and following day for other lessons of a given course to ensure that there is proper spacing between lessons. It returns success or failure depending on the outcome of this check.

Algorithm 2.7 CheckProximity(index, course c)

```
1: for i = index-1 to index+1 do
2:     if activities[morning[days[i]]] contains c or activities[afternoon[days[i]]] contains c then
3:         return false
4:     end if
5: end for
6: return true
```

TryToPlaceBlock

TryToPlaceBlock, a member of *Schedule*, is responsible for checking for possible conflicts between a potential lesson given as a parameter, and the activities already assigned to this block. This is done by iterating over all the activities that are already assigned, and checking whether conflicts exist between the teachers and student sets belonging to the courses. If none are found, a suitable room is selected. Courses may have preferences for a specific type of room. This, along with the number of attending students is used to select a room. If a suitable room is available, the potential lesson is added to the list of activities of the block, and the method returns success. Should the process fail at any of the steps, however, it will return failure.

Algorithm 2.8 TryToPlaceBlock(activity a, block b)

```
1: if no activity in b has any studentset in a then
2:     if no activity in b has any course in a then
3:         if no activity in b has any teacher in a then
4:             for each room r in rooms do
5:                 if type[r] = preferredroomtype[a] and b does not use r then
6:                     if size[r] $\leq$ numberofstudents[course[a]] then
7:                         room[a] = r
8:                         List-Insert(activities[b],a)
9:                         return true
10:                    end if
11:                end if
12:            end for
13:        end if
14:    end if
15: end if
16: return false
```

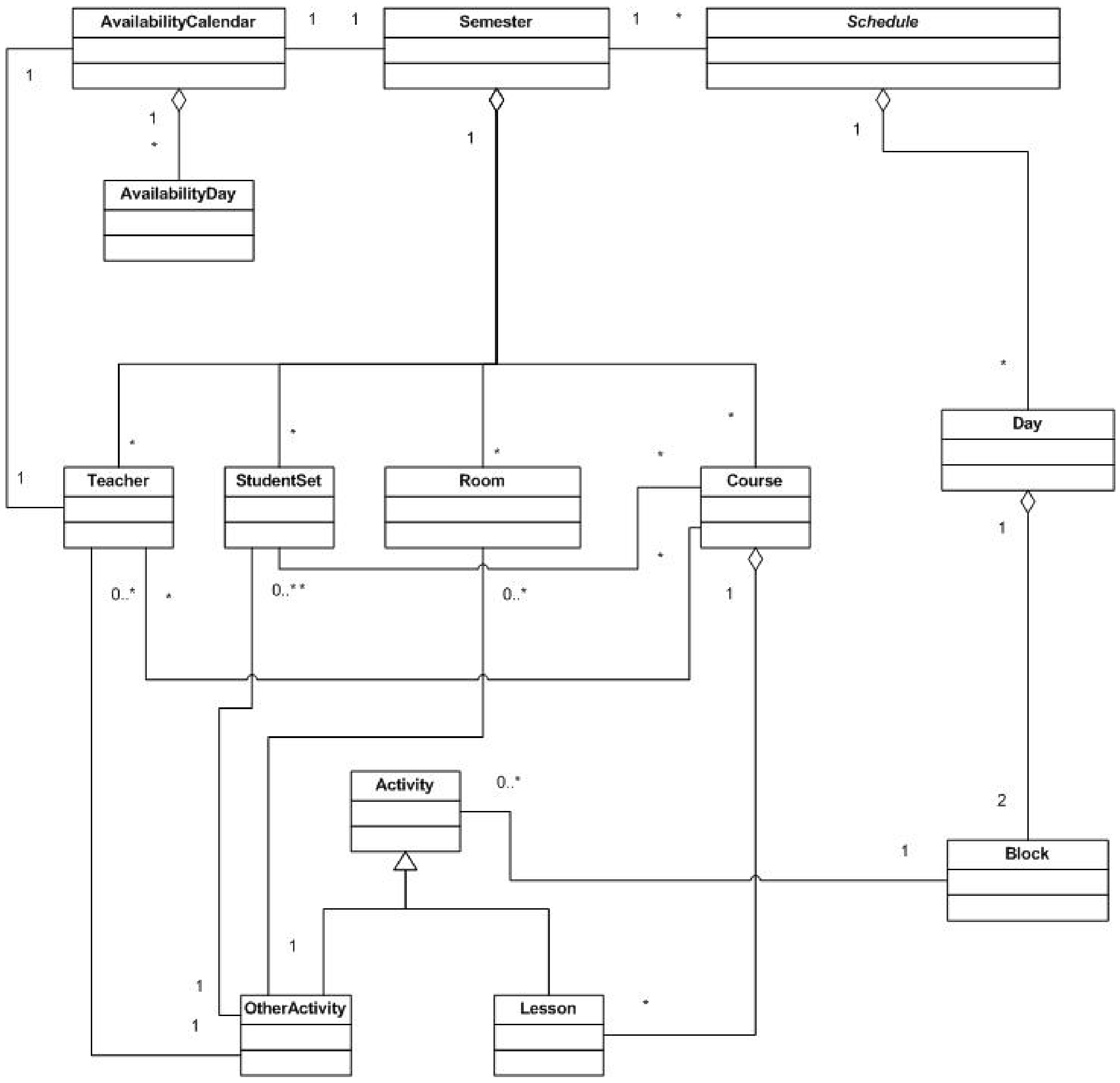


Figure 2.3: The design class diagram

2.4 Components

2.4.1 Structure

Class Descriptions

Here follows the description of the classes illustrated in the class diagram in figure 2.3. This section will describe the purpose of each class as well as its attributes and most important methods. This means all trivial methods such as getters and setters as well as the class constructors are left out. Since the system architecture has undergone major changes since the analysis document all classes, even the triviel ones, are described.

Semester
Purpose: Represents the time of the entire semester. The timeframe is defined by a starting date and an ending date. Semester then holds a list of all the days in between and including those dates. Semester also holds lists of all Teachers, Students, Rooms and Courses at the current semester. The most important method of semester is to generate the schedule and display different variations of the schedule.
Attributes: *DateTime* startDate, *DateTime* endDate, *AvailabilityCalendar* availability, *Schedule* globalSchedule, *List<Schedule>* schedules, *List<Teacher>* teachers, *List<StudentSet>* students, *List<Room>* rooms, *List<Course>* courses
Methods: Add and remove methods for all lists, GenerateSchedule, GenerateWeekSchedule, GenerateSemesterSchedule, Initialize, Instance

Schedule
Purpose: Represents any schedule created or displayed. After the semester schedule has been created it is possible to create instances of Schedule which can display a schedule for a specific timeframe, teacher, student set, course or room.
Attributes: *Day[]* days, *DateTime* startDate, *DateTime* endDate
Methods: GetStudentSetSchedule, GetRoomSchedule, GetTeacherSchedule, ClearSemester, ClearWeek, CheckProximity, TryToPlaceBlock, TryToPlaceLesson, Pass

Day
Purpose: Represents a day in the schedule respresented by a DateTime. A day consists of two blocks for respectively morning and afternoon.
Attributes: *DateTime* date, *Block* MorningBlock, *Block* AfternoonBlock

Block
Purpose: Represents either a block of hours in the morning or in the afternoon. A block holds a list of activities which represents all the courses and other activities held at the institute in that particular block.
Attributes: *List<Activity>* activities
Methods: Add and remove methods for activities

Activity
Purpose: Represents a single activity in a block. An activity can either be a lesson or a custom activity. Futhermore each activity is assigned a room so the possiblity to change rooms for each activity is left open.
Attributes: *string* Name, *Room* AssignedRoom

Lesson
Purpose: Represents a single lesson of a course. A lesson can be parted into two types of activity, which is either a lecture or the appurtanant exercises.
Attributes: *Course* owner, *LessonActivity* firstActivity, *LessonActivity* secondActivity

OtherActivity
Purpose: Represents all other activities than courses. These activities could be examns, optional lectures etc. These activities can get teachers and students assigned to them.
Attributes: *Teacher* assignedTeacher, *StudentSet* assignedStudentSet
Methods: Add and remove methods for teachers and students

AvailabilityCaldendar
Purpose: Represents a calendar consisting of an array of AvailabilityDay. The array contains as many days as the timespan of the semester.
Attributes: *AvailabilityDay[]* days

AvailabilityDay
Purpose: Represents an availability day which have two states for respectively morning and afternoon. These states determine whether activities can be placed in the morning and afternoon blocks of the current day.
Attributes: *ActivityState* morning, *ActivityState* afternoon

Course
Purpose: Represents a course that must be attended by a set of students. It also holds a list of the teachers who are able to lecture the course, as well as the the room they prefer to lecture in. A course can be parted into a total number of lessons which must be filled into the schedule. Also a week may only have a predefined number of lessons of the same course. This number is defined as lessons per week. The number of lessons assigned serves as a counter, that keeps track of the number of lectures in the schedule. There are some cases in which one course must be finished before another one can begin. In these cases the prerequisite courses will be stored in a list to avoid conflicts.
Attributes: *string* Name, *string* Acronym, *uint* TotalNumberOfLectures, *RoomType* PreferredRoomType, *List<Teacher>* teachers, *List<StudentSet>* students, *List<Activity>* activities, *List<Lesson>* lessons, *List<Course>* prerequisiteCourses
Methods: Add and remove methods for all lists

Teacher
Purpose: Represents a teacher at the institution. A teacher is represented by his name and the username which he uses at the institutes intranet. Teacher also holds a list of courses which he may lecture. Furthermore each teacher has his own calendar. This calendar holds information about the blocks where he is unable to lecture, and the blocks where he would prefer not to lecture.
Attributes: *string* Name, *string* Username, *AvailabilityCalendar* availability, *List<Course>* courses
Methods: Add and remove methods for courses

StudentSet
Purpose: Represents a group of students at the institution. Such a group consists of a name, the number of students and a list of the courses they are attending.
Attributes: *string* Name, *uint* NumberOfStudents, *List<Course>* courses
Methods: Add and remove methods for courses

Room
Purpose: Represents a room at the institution. A room has a name and a size which is the number of seats in the room. Besides that a room has a type which can be an auditorium, seminarroom or pclab.
Attributes: *string* Name, *uint* Size, *RoomType* Type
Methods: isAvailable

2.4.2 Presentation Model

This section will provide an understanding of the graphical user interface (GUI) and how the different interaction spaces relate to each other. In the class diagram below all classes are interaction spaces. The diagram shows all the interaction spaces between the user and the system, what attributes they hold, both for input and output, and what kind of actions they can perform, such as adding a new object to the system.

As seen in figure 2.4 a lot of the interaction spaces are contained in the Main interaction, and these are the larger interaction spaces, while the smaller interaction spaces are navigated. All the contained interaction spaces can only be active one at a time. Almost all of the interaction spaces are built on a simple manipulation and navigation interactiontype, as you navigate among different objects, in this case the inputs for the system, and then manipulate these objects after the criteria, determined by the user for schedule generation. The main idea behind the design is to only have a number of small popup windows, and keeping the larger screens in a primary window. Examples of these types will be elaborated in section 2.4.3.

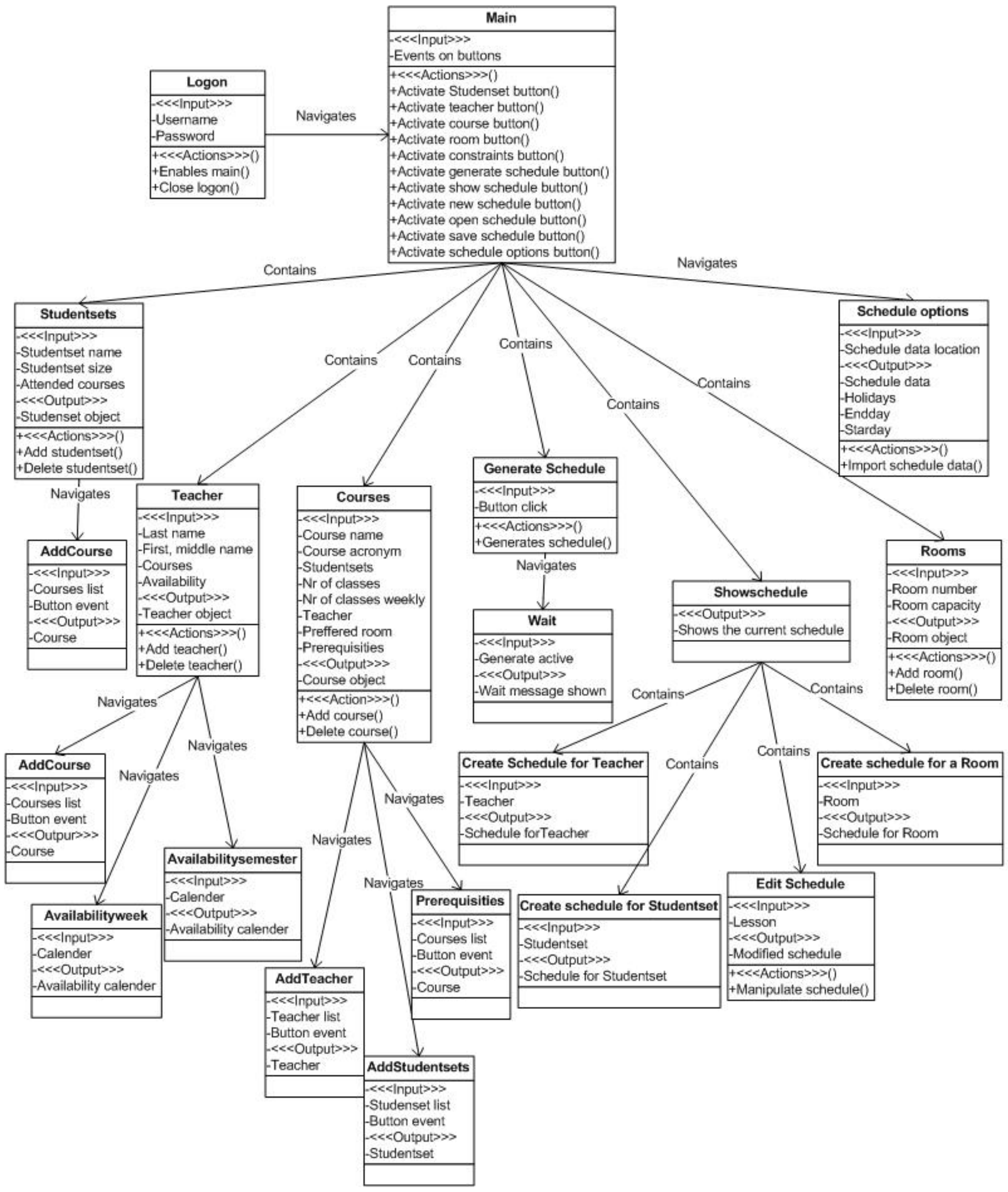


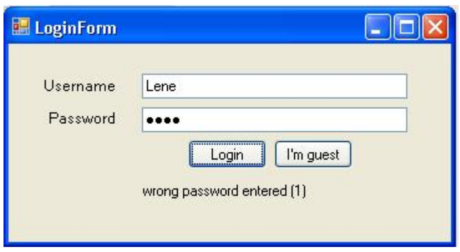
Figure 2.4: A diagram of the GUI. All classes are interaction spaces

2.4.3 Interaction Elements

In this section some of the screens intended to be used in the final system will be shown and explained, especially the interaction type and how it is supposed to work.

The login screen seen in figure 2.5 is one of the screens that does not follow the navigate and manipulate interaction type.

Instead it uses the conversation interaction type, i.e it asks the user for some basic information and then grants access to use the system. The screen itself consist of two textboxes to take in the username and the password, two buttons, one to confirm login and another for


LoginForm

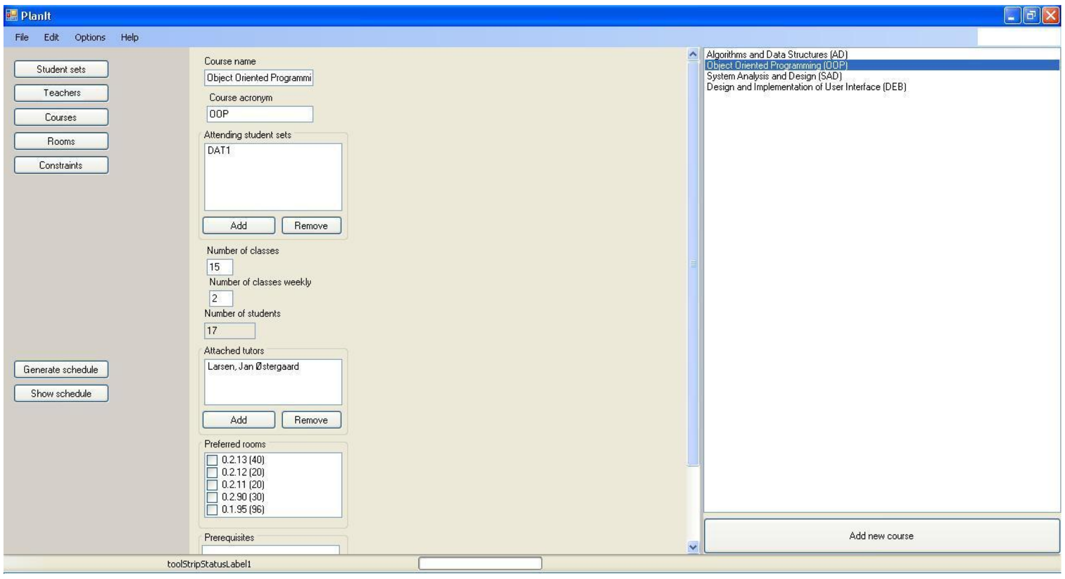
Username: Lene

Password: ••••

wrong password entered (1)

Figure 2.5: The login screen of the scheduling system

guests to use. The last thing is a label which tells the user if the password or username is wrong, and how many times a wrong login has been tried. The login screen is not af part of the use case diagram from the analysis document, figure 1.10, as it is only system related.


PlanIt

Course name: Object Oriented Programmi

Course acronym: OOP

Attending student sets: DAT1

Number of classes: 15

Number of classes weekly: 2

Number of students: 17

Attached tutors: Larsen, Jan Østergaard

Preferred rooms:
- ☐ 0.2.13 (40)
- ☐ 0.2.12 (20)
- ☐ 0.2.11 (20)
- ☐ 0.2.90 (30)
- ☐ 0.1.95 (96)

Prerequisites

Algorithms and Data Structures (AD)
Object Oriented Programming (OOP)
System Analysis and Design (SAD)
Design and Implementation of User Interface (DEB)

toolStripStatusLabel1

Figure 2.6: The screen for inputting information about courses, intended to be used in the final system

The course input screen seen in figure 2.6 is built on the navigate and manipulate interactiontype, this is emphasised by the main window where you navigate among the different input screens. In the course screen, the list of courses in the box to the right lets you navigate through all the different courses already created, or you can create a new one. This is seen as the organize data, view data and modify schedule use cases in figure 1.10 from the analysis document. When one is selected (or created) the content which is covered by the course is able to be manipulated. When none of the courses on the list on the right of the screen are marked, the buttons controlling the content are disabled, forcing the user to choose or add a new course. The basic structure of the course screen is also used in the other contained screens to create consistency, to make it simple to use when learned. The list box to the left is the same throughout the contained interaction spaces.

The screen in figure 2.7 helps the user to add a student set to a course. This is seen as the merge data in the use cases diagram in figure 1.10 from the analysis document, it is used to connect the different data such as courses and student sets. A box contains the list of student

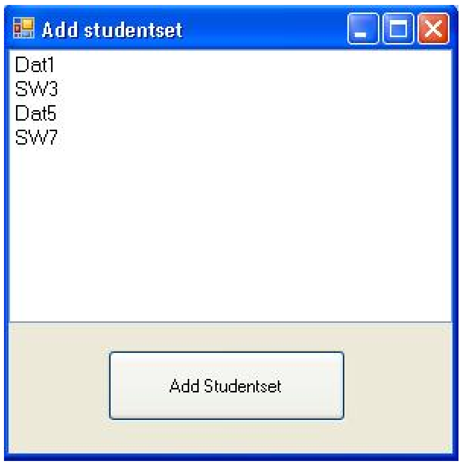


Figure 2.7: The add studenset to course screen

sets already created and a button to add a student set to a specific course. The screen follows the conversation interactiontype, just like the login screen, the user chooses a student set by marking it, and then simply adding the studentset by clicking the button labled "*Add studen set*". The same type of screen is used whenever it is needed to add something to an object, such as a course to teacher or student set, or a student set and teacher to a course, in order to preserve consistency.

The create schedule and view schedule use cases from 1.10 is contained within the main window, and will not be shown as individual screen. The generate function itself will not have a window, because it's just a function to plan a schedule. The view schedule will display a screen with a calender where all the days with courses will be marked.

2.5 Programming

Implementability

Looking at the class diagram and the explanation of the algorithm in section 2.3.4, the system is considered fully implementable. There is no unreasonable design criterias to the system, nor any unobtainable components or development tools. The system is to be implemented using a standard PC with a development environment supporting C#. A couple of examples will be given to explain how parts of the system is to be implemented.

Data Contract Serialization

The source code below explains how to save the data to XML files. The *DataContractSerializer* class from the .NET Framework Class Library will be used to accomplish this. The primary reason for this is that it is easy to convert an objects properties and relations to an XML format using serialization. By using the *DataContract* and *DataMember* attributes a root node can be defined with underlying child nodes.

```
[KnownType(typeof(RoomType))]
```

```
 2   [DataContract]
 3   public class Room
 4   {
 5     [DataMember]
 6     string Name{ get; set; }
 7
 8     [DataMember]
 9     int Capacity { get; set; }
10
11     [DataMember]
12     public RoomType Type { get; set; }
13   }
```

Line 1: The *KnownType* attribute is used to tell the serializer what type of object it needs to know about when serializing.

Line 2: The *DataContract* attribute is used to define our class *Room* as a root node in the XML document.

Line 5: The *DataMember* is used to define the property *Name* as a child node under *Room*.

Line 8: The *DataMember* is used to define the property *Capacity* as a child node under *Room*.

Line 11: The *DataMember* is used to define the property *Type* as a child node under *Room*.

```
1          public static void Serialize<T>(T obj, string filename)
2          {
3              using (Stream fs = new FileStream(filename, FileMode.Create))
4              {
5                  DataContractSerializer ds = new DataContractSerializer(typeof(T),
                       null, int.MaxValue, false, true, null);
6                  ds.WriteObject(fs, obj);
7              }
8          }
```

Line 1: The function *Serialize* is made generic so it can serialize an object regardless of its type. As parameters it takes an object of a generic type, and the name of the XML file it writes to.

Line 3: A new *FileStream* is initialized with the name of the XML file and the *FileMode* enumeration. The Create value specifies that the system should create a new file or overwrite an already existing one.

Line 5: Creates an instance of the class DataContractSerializer which controls how an object is encoded into an Xml file. The *DataContractSerializer* constructor takes six parameters, where the first is the type of the object to be serialized. The second parameter is a list of known types, but since these are declared using class attributes it is set to null. The third parameter is the maximum number of objects to serialize, which is set to the maximum value of an int. The fourth parameter determines whether to ignore extension data of the type being serialized, which is set to false. The fifth parameter determines if object references are preserved upon deserializing, which is set to true. The sixth parameter can take a *DataContractSurrogate* object, but since there is no need for a customized serialization process this is left as null.

Line 6: The *WriteObject* function is called on the *DataContractSerializer* object. It takes a *FileStream* object and an object as parameters, and serializes the object to an XML file.

```
1          public static T Deserialize<T>(string filename)
2          {
3              using (FileStream fs = File.OpenRead(filename))
4              {
5                  XmlDictionaryReaderQuotas quotas = new XmlDictionaryReaderQuotas()
                       ;
6                  quotas.MaxDepth = Int32.MaxValue;
7                  XmlDictionaryReader r = XmlDictionaryReader.CreateTextReader(fs,
                       quotas);
8                  DataContractSerializer ds = new DataContractSerializer(typeof(T));
9                  return (T)ds.ReadObject(r, true);
10             }
11         }
```

Line 1: The function *Deserialize* is made generic so it can deserialize an object regardless of its type. It returns a generic type thus allowing the creation of any type of object from the XML file. As parameters it takes the name of the XML file to deserialize.

Line 3: A new *FileStream* is initialized with the *OpenRead* function, which takes the name of the XML file as a parameter.

Line 5: An object of the type *XmlDictionaryReaderQuotas* is initialized. This will be used in the initialization of the *XmlDictionaryReader*.

Line 6: The *MaxDepth* property determines the maximum nested node depth in the XML file. This value is set to the maximum value of an int to ensure the entire file is deserialized.

Line 7: The *ReadObject* method is called on the *DataContractSerializer* object with the *XmlDictionaryReader* object as the first parameter. The second parameter indicates whether the name of the object is checked with the name of the class.

The primary function of the system is to generate a correct and satisfying schedule for the semester. To do this is is vital that the algorithm needed to accomplish this is implementable. In section 2.3.4 the algorithm has been described in detail and all the functions have been fully elaborated. This thus serves as documentation for its implementability. Furthermore for the system to be functional it is also neseccary to have data persistence. This persistence is maintained through the implementation of the XML serializer and the storing of data to XML files which has been explained above. Based on these statements it is the overall conclusion that the system is fully implementable.

2.6 Evaluation Plan

2.6.1 Purpose

In section 2.1.3 the quality goals of the scheduling system have been determined. The two most important goals was making the system very correct and reliable. However this is not something we can test with a test subject as a user, this must be tested using white- or blackboxing in conjunction with the correctness and effectiveness of the system. We can however test if the system is useable and comprehensible, by using user evaluations. As we only have one real test subject with great insight into the scheduling subject, it is imperative that a formative evaluation process is chosen. The evaluation test is placed rather late in the development process and it is therefore a validation test.

2.6.2 Main Question

The main purpose of the evaluation test is to validate whether the system meets the requirements specified about general usability. In our case this means testing utility and memorability in order to see if the system is useable and comprehensive. In short: "*Can the user use the system without significant trouble and remember how he/she accomplished the given tasks?*"

2.6.3 User Profile

This evaluation is aimed towards the first-time user, e.g one who has not used the system before. The user should have some basic computer skills as well as a general understanding of scheduling. Participating in the evaluation as a test subject is a secretary from the department of computer science.

2.6.4 Evaluation Method

In order to speed up the process of doing the evaluation test we have chosen to use the *Instant Data Analysis* (IDA) model. We will therefore need 1-2 observers who take notes of the evaluation while it is running, one test leader who guides the test subject through the exercises and one facilitator who does not participate in the test, but helps gather the data once the evaluation with the test subject is over, in conjunction with the observers and the test leader.

2.6.5 Exercises

All aspects of the system will be tested, this includes exercises doing the following: Logging in, inputting new data (new teachers, rooms, studentsets, etc), editing existing data in the system and editing a schedule after it is created.

A few examples of exercises will be given:

Exercise 1 Monday, August 25th, 2008. New students are arriving in a week to start at the department of computer science. You must therefore create the new student sets to represent these people in the schedule system. Dat1 has 13 students and SW3 has 37 students.

- Create the new student sets for Dat1 (13 students) and SW3 (37 students)
- Assign both of the new student sets to attend the algorithms and datastructures (AD) course

Exercise 2 During the summer vacation a new building has been built to house the increasing number of new students at the institute. It has been named "cluster 6" and it has a 1st and 2nd floor like the existing building. In the new building there is also a new auditorium, the room number is 6.1.1 and it has the capacity of 96 students.

- Add the new auditorium named 6.1.1 with a capacity of 96

Exercise 3 Most of the courses are already in the system from last year, but a senior of the staff has gone on pension and a new teacher has been hired.

- Remove the teacher Finn Jeppe Jeppesen from the schedule system
- Add a new teacher named Peter Müller Jørgensen to the system and assign him to teach the AD course

Exercise 4* *(The test leader turns off the computer screen prior to starting this exercise)* Having in mind the tasks that you have just completed, please quickly run through (think aloud) the steps of which buttons you pressed and where you typed in information for each exercise.

*This exercise is meant to be the very last exercise

2.6.6 Location and Equipment

When using the IDA model it is not needed to have recording equipment, though using a HCI-lab will still help as the observers needs a place to observe the test without interfering, therefore the test will take place in a HCI-lab. Pseudo-data for creating a schedule is required.

2.6.7 Data Assessment

Data regarding the test subjects thoughts while using the system (thinking-aloud) must be recorded by the observers as well as actions taken on the screen.

2.6.8 Contents of the Evaluation Report

1. Method
   Purpose
   Test subjects
   Test course
   Location and equipment
2. Usability problems
   Problem list
   Detailed description of the problem list
3. Conclusion
4. Appendix
   Exercises
   Screenshots

2.7 Test Plan

In section 2.1.3 the two most important goals of the system was that it be very correct and reliable. This is something we can make sure is fulfilled using program tests in the development phase. During the latter phases of the development process, it becomes necessary to perform program tests to ensure that the software is working as intended. For this project, a mixture of methods will be used to perform tests. More specifically, a level of testing will be chosen for individual classes based on a number of criteria.

These are: Their complexity relative to each other (favoring those with greater size and number of interacting subroutines), how straightforward they are to implement based on this document (favoring those where a greater amount of tweaking has to be carried out during implementation), and how prone they are to error (to ensure that our bug extermination process is streamlined towards the most critical parts of the software).

2.7.1 General Testing

Classes that are simple e.g trivial structs and classes with a shallow level of complexity) are not unit tested, but only subjected to removal of errors using a debugger. During development, all code is debugged using the built-in debugger in Visual Studio. This is the only error elimination the trivial classes will go through.

2.7.2 Unit Testing

Unit testing is employed to test the correctness of a number of core classes, selected based on the above criteria. These are, corresponding to the classes of the preceding class diagram:

- Semester
- Teacher

- StudentSet
- Room
- AvailabilityCalendar
- Course

When conducting unit tests, one has the option of choosing between white-box testing (where the focus is on the internal structure of the code, and where each unique path through the code is identified and tested) and black-box testing (where focus is on the external interface of a unit, and different inputs are supplied and the outputs verified). In this project, black-box testing will exclusively be used, as identifying and testing all possible paths through the code is too time-consuming.
Developing classes will take precedence over their respective tests, e.g focus will be on writing the classes before the tests are written. The purpose of the testing is to act as a verification tool to ensure that the code is working as intended. In other words, the test cases will not be used to drive the development of the classes, merely as a tool to stamp out errors later.
For the tests a set of appropriate inputs have been chosen to supply the test framework. These will include appropriate as well as inappropriate values to ensure that exception handling is behaving correctly. The results generated from the classes will then be verified to see whether or not something has gone wrong in the process. The different tests will allow for pinpointing the points of failure and in which units they occur.
For the practical part of the testing process the NUnit testing framework will be used, as this will allow for easy construction and running of tests.

3 Implementation

This document will deal with the transition from design to a functional system. This transition includes some changes made to the design, to make the implementation easier or at least possible. These changes will be explained at first and afterwards there will be an in-depth explanation of the most important functions in the system. This will include descriptions as well as source code.

3.1 Design Changes

The changes to the system design are primarily based on the class hierarchy and the methods implemented by the some classes. Most of these changes were discovered as the implementation process progressed and complications arose. In this section these changes are described.

The first change that has been made concerns the class diagram as shown in fig. 3.1. Here Schedule has been made an abstract class which the two specializations GeneralSchedule and SpecificSchedule inherits from. This change has been made to distinguish between the general schedule which will hold the methods for generating the schedule, and a specific schedule which will only be a used for viewing.

Below it will be described which of the classes and what methods have been reworked. Only the most important of these methods will be shown here, as simple methods such as getters and setters are self-explanatory.

Semester
Changes: *Semester* is no longer responsible for the generation of schedules, as these methods have been moved to the schedule classes instead. A new attribute has also been added for the general schedule.
Attributes: *GeneralSchedule* globalSchedule
Methods: Has the same methods as described in the design document except the schedule generation methods.

Schedule
Changes: *Schedule* has been made an abstract class who generalizes *SpecificSchedule* and *GeneralSchedule*.
Attributes: *Day[]* days
Methods: A method GetDay which return a day from the array days based on a DateTime.

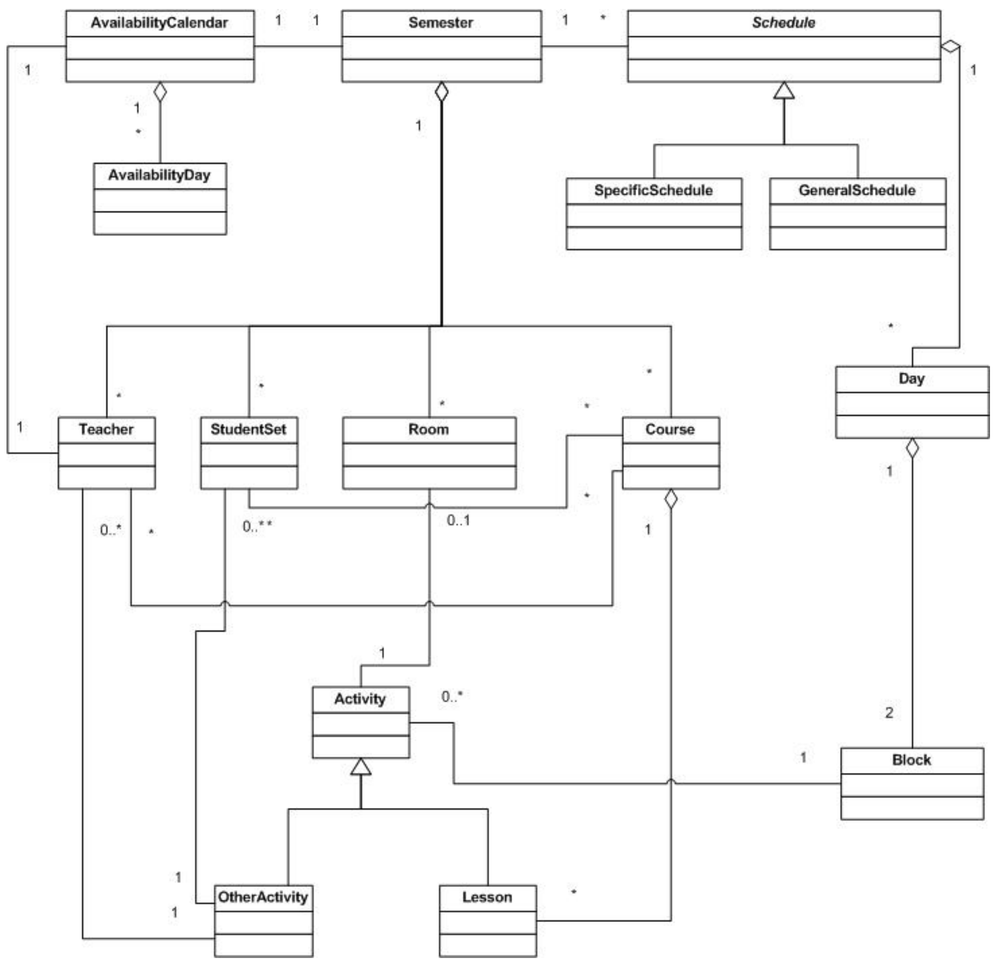


Figure 3.1: Component architecture of the scheduling system

SpecificSchedule
Changes: This respresents a specific schedule created from a certain criterium. This criterium can be a certain teacher, student set, room or course.
Attributes: *string* Name
Methods: A constructor method for each of the criteria.

GeneralSchedule
Changes: This respresents the general schedule for the semester. GeneralSchedule are also responsible for the schedule generation methods.
Methods: GenerateSchedule, SemesterSchedule, WeekSchedule, TryToPlace, Check-Proximity, Pass, TryToPlaceBlock, TryToAssignRoom, ClearSemester, ClearWeek,

The next change is in section 2.4.2 to the GUI class diagram. Some of the classes have been merged in the system, and others have been extended with small GUI classes for different input, which were not taken into consideration in the design document.

3.2 Important Methods

This section will describe the most important funtions of the system, with details about what they are for, how they work, and why they are important. The descriptions will include partial or complete sourcecode, with the exeptions of setters and getters.

3.2.1 AvailabilityCalendar

The *AvailabilityCalendar* class holds all days for the schedule with their individual availability. This is done with an array of AvailabilityDay with the same length as the number of days in the schedule. When the semester is created, every day is set to available and a holiday list is created. Right before the schedule is to be planned by the generate method, the two lists of days (holidays and availability days) is merged into one list, containing every day of the semester, all of which have had their availability set by the user, either in terms of holidays or by general availability (weekends and the like).

When the *Semester* class is first created, the *AvailabilityCalendar* class is initializes by getting the length of the schedule as well as the start day in the constructor, and creating an array of days that match the schedule:

Listing 3.1: AvailabilityCalendar.AvailabilityCalendar()

```
public AvailabilityCalendar(uint size, DateTime _startDate)
{
  AvailabilityArray = new AvailabilityDay[size];
  this.startDate = _startDate;
  // Initialize the AvailabilityDays in the array
  for (int i = 0; i < AvailabilityArray.Length; i++)
  {
    AvailabilityArray[i] = new AvailabilityDay((uint)i);
  }
}
```

Where the array of days consists of *AvailabilityDay*(s):

Listing 3.2: AvailabilityDay.AvailabilityDay()

```
public AvailabilityDay(uint _dayNumber)
{
  this.dayNumber = _dayNumber;
  StateMorning = ActivityState.Available;
  StateAfternoon = ActivityState.Available;
}
```

The holidays is then added to the lists of holidays by way of *get* and *set* methods, and every day have their availability set according to their place in the week by the user. The availability of each day is changed using the *ChangeStatusOfAvailabilityDay* method:

Listing 3.3: AvailabilityCalendar.ChangeStatusOfAvailabilityDay()

```
public void ChangeStatusOfAvailabilityDay(uint dayNumber, ActivityState status
    , bool isMorning)
{
  if (isMorning == true)
  {
        AvailabilityArray[dayNumber].StateMorning = status;
      }
      else
      {
        AvailabilityArray[dayNumber].StateAfternoon = status;
      }
    }
```

3.2.2 Semester

The *Semester* class represents a semester, and is implemented as a singleton class as it was decided during the design phase that the system should only operate on one schedule at a time. *Semester* has three main areas of responsibility: Keeping track of the timeframe and availability during the semester, maintaining lists of the entities from which the schedules are generated (Teachers, courses, etc.), and storing a general schedule containing everything as well as subject-specific schedules.
Since it is intended that there should only be one active semester at a time, the *Semester* class has been made a singleton. The class deviates from the classic singleton design pattern in that the class is supplied a start and end date on creation, since the timeframe is defined when the semester is created, and cannot be changed later. Days are stored as arrays of length *numberOfDays*, with index 0 corresponding to the start date. This is done by calling the *Initialize()* method:

Listing 3.4: Semester.Initialize()

```
    public static bool Initialize(DateTime startDate, DateTime endDate)
    {
      // Abort if we already have an instance running.
      if (instance != null)
      {
        throw new Exception("Semester is already initialized!");
        return false;
      }
      // If not, make one
      instance = new Semester(startDate, endDate);
      return true;
    }
```

This function calls the constructor, which sets the timespan and initializes the semester availability calendar:

Listing 3.5: Semester.Semester()

```
    private Semester(DateTime startDate, DateTime endDate)
    {
      TimeSpan ts = endDate - startDate;

      //TimeSpan is the number of days _BETWEEN_ two days, therefore we add 1:
      numberOfDays = ((UInt32)ts.Days)+1;
      this.startDate = startDate.Date;
      this.endDate = endDate.Date;

      availability = new AvailabilityCalendar(numberOfDays, startDate);
    }
```

After this has been done, the *Semester* instance can be retrieved through *Instance()*, and it can be destroyed by calling *Destroy()*.
The timeframe is defined when *Semester* is instantiated, in the form of a start and end date (*DateTime* classes). *Semester* has a few methods to make the timeframe available to other parts of the system, these include getters and methods to convert between *DateTime* dates and the numbered indices used to look up days in the array storing them.

Handling of the lists of teachers, student sets, rooms and courses is done through a number of methods for each type. Their function is to provide a straightforward interface to the lists, as the lists are used extensively by other parts of the system. For each of them, *Semester* has methods to add and delete elements as well as a method that returns a reference to the lists so they can be manipulated directly.

Semester also stores all schedules: One global schedule containing everything that is planned for the semester (In the form of a single instance of *GeneralSchedule*) and a list of schedules that only concern a certain subject (*SpecificSchedule*), so that it is possible to show schedules that only contain activities involving eg. a certain teacher. When the user instructs the system to generate the schedule, the *Semester.GenerateSchedule()* method handles this:

Listing 3.6: Semester.GenerateSchedule()

```
public bool GenerateSchedule(BackgroundWorker w, DoWorkEventArgs e)
{
  GeneralSchedule newSchedule = new GeneralSchedule(numberOfDays, w, e);
  bool result = newSchedule.GenerateSchedule();
  // Did we succeed?
  if (result)
  {
    globalSchedule = newSchedule; // Use the newly created one
  }
  return result;  // Let the caller know how it went.
}
```

3.2.3 Generating Schedules

The code described below is spread out between a few classes in the source code, namely *GeneralSchedule*, *Semester*, *Block* and *Course*. The non-trivial functions involved in schedule generation are presented in full below, with some annotated descriptions to aid in their understanding. The source code can be found in full on the enclosed CD, aling with the generated documentation of all classes and methods.
The methods involved in generating schedules are based on the pseudocode described in section 2.3.4. The ideas behind the algorithms remain the same, and the descriptions in the preceding section are still valid to some extent. During the implementation phase, the algortihms were expanded and improved upon. The general structure of the code is the same with a few important changes:

- The *GetWeekLessons()* method, a member of *Semester*. This method returns a list of lessons that are to be held in the coming week. In turn, it constructs the list by calling corresponding methods in *Course*.

- Conflicts are detected throuh the *IsInConflictWith()* methods, defined under both *Block* and *Course*.
- A list called *leftoverLessons* under *GeneralSchedule*, which stores lessons that could not be placed in a week for the next.

As well as several minor changes that are described below.

Generating schedules from the data that is entered into the program is carried out by *GeneralSchedule*. The size of this array is supplied to the constructor by *Semester*. The two additional parameters that it takes enable the scheduling process to be run in a seperate thread, ensuring that the GUI is still responsive and that the process can be interrupted by the user. This is done by using the *BackGroundWorker* class (A part of Windows Forms). The first thing *GeneralSchedule* does (on creation), is to store the two arguments needed to use the *BackGroundWorker*:

Listing 3.7: GeneralSchedule.GeneralSchedule()

```
public GeneralSchedule(uint numberOfDays, BackgroundWorker w, DoWorkEventArgs
    e)
: base(numberOfDays)
{
  worker = w;
  eventArgs = e;
}
```

Through the call to the base constructor, the array of *Day* instances is initialized:

Listing 3.8: Schedule.Schedule()

```
public Schedule(uint numberOfDays)
{
  this.numberOfDays = numberOfDays;

   // The size of the array is this number
  days = new Day[numberOfDays];

  // initialize all the Day instances in the array
  for (int i = 0; i < days.Length; i++)
  days[i] = new Day();
}
```

The next thing that will happen is that *Semester* will call the *GenerateSchedule()* method, which will start the process:

Listing 3.9: GeneralSchedule.GenerateSchedule()

```
public bool GenerateSchedule()
{
  int tries = 0;
  bool result = false;
  // Keep trying to generate a satisfying schedule,
  // each time with different starting parameters (tries)
  while (!result && tries < 7)
  {
    ClearSemester();
    result = SemesterSchedule(tries);
    tries++;
  }
```

```
        // return either success or failure, depending on the outcome
        return result;
    }
```

The method will keep calling *SemesterSchedule()* until the process succeeds, or the number of tries are expended. The tries variable serves to ensure that the schedule is planned differently each time. The value is the starting day used when each week is planned (0 = Monday and so on). This means that should a schedule fail to be planned, then the assigned activities will be sufficiently shuffled by this iteration. In cases of deadlock, this might be enough to create a functional schedule.
SemesterSchedule attempts to plan a single semester. Should it fail, the method returns false, otherwise true:

Listing 3.10: GeneralSchedule.SemesterSchedule()

```
    private bool SemesterSchedule(int semesterTries)
    {
        // Starting at the first day, plan the one week at a time.
        for (int currentDay = 0; currentDay < days.Length; currentDay += 7)
        {
            int tries = 0;
            bool result = false;

            // Try some times with different parameters, until we get a successful
                week schedule.
            while (!result && tries < 7)
            {
                ClearWeek(currentDay);
                result = WeekSchedule(currentDay, (tries + semesterTries) % 7);
                tries++;
            }
            // If we fail to plan a single week, give up and return failure.
            if (!result)
                return false;
        }
        // We've run out of time, check to see if we managed to place all lessons
        // This is done by checking if we have any left over ones, or if the
            semester has ones we have not received yet
        if (leftoverLessons.Count > 0 || semester.GetWeekLessons().Count > 0)
            return false;

        return true;
    }
```

The method iterates through all weeks, and tries to plan each week by itself (by calling *WeekSchedule()*, which returns true on success and false on failure). Each week is planned based on a "new tries" parameter. This functions in much the same way as the previously described variable, except only for one week at a time. Thus, should a week fail to be planned, the algorithm will try again starting on the next day (wrapping around when the end of the week is reached). Should all the tries fail, the method returns false. Should the process succeed, the method checks two things: Whether there are any lessons left from the last week, and if there are any lessons that remain to be planned in the semester. If not, everything has been placed successfully and the method returns true.
WeekSchedule attempts to plan a single week in the schedule:

Listing 3.11: GeneralSchedule.WeekSchedule()

```
private bool WeekSchedule(int weekStart, int startDay)
{
  uint nFailed = 0;    // The number of activities that could not be placed

  List<Lesson> lessonCandidates = semester.GetWeekLessons();  // Get the new ones that should be placed

  List<Lesson> usedLessons = new List<Lesson>();         // The lessons that have been placed.
  List<Lesson> unusedLessons = new List<Lesson>();       // The lessons that failed.

  lessonCandidates.AddRange(leftoverLessons); // Add the unused lessons from the week before

  // Sort them by attendance, so we give priority to those with the most students
  lessonCandidates.Sort(
    delegate(Lesson a, Lesson b)
    {
      return -a.Owner.GetNumberOfStudents().CompareTo(b.Owner.GetNumberOfStudents());
    }
  );

  // Try to place them.
  foreach (Lesson l in lessonCandidates)
  {
    if (TryToPlace(l, weekStart, startDay))
    {
      usedLessons.Add(l);
    }
    else
    {
      unusedLessons.Add(l);
    }
  }

  // Abort if too many failed.
  if ( unusedLessons.Count > 2)
  {
    return false;
  }

  // Else, if there is a small number of unplaced lessons left, save them for the next week.
  leftoverLessons.Clear();
  leftoverLessons.AddRange(unusedLessons);

  // If we succeed in placing them, increment the number of placed lessons of the courses
  foreach (Lesson l in usedLessons)
    l.Placed();

  return true;
}
```

It starts out with retrieving a list of lessons that are to be placed in the coming week from the *Semester* instance, and adds in any leftover lessons from the preceding week. As the method tries to place them, they are placed in one of either the *usedLessons* or *unusedLessons* lists. The list of lesson candidates is sorted to place the lessons with the most attendants first (the minus sign in front of the call to *CompareTo* in the delegate method ensures that the sorting order is reversed). Then, the method attempts to place each individual lesson by a call to *TryToPlace()*. When all lessons have been processed, the method returns either true or false depending on the number of failed lessons: If they fall within some threshold value (in the code above set to 2), it returns false. Otherwise, the lessons that failed are added to the list of leftover lessons to be placed in the coming week, and the successful lessons are marked as placed (the *Placed()* method increments the number of placed lessons of that particular course).

GetWeekLessons() constructs a list of all the courses that are to be held in the current week:

Listing 3.12: Semester.GetWeekLessons()

```
public List<Lesson> GetWeekLessons()
{
  List<Lesson> list = new List<Lesson>();

  // Go through all courses and get theirs
  foreach (Course c in courses)
  {
    list.AddRange(c.GetWeekLessons());
  }

  return list;
}
```

This is done by calling the corresponding *GetWeekLessons()* method for all courses, appending their lessons to the global list and returning it:

Listing 3.13: Course.GetWeekLessons

```
public List<Lesson> GetWeekLessons()
{
  List<Lesson> list = new List<Lesson>();

  // If all lessons have been assigned, or the prerequisites have not
  // been, return the empty list
  if (HasFinished() || !CheckPrerequisites())
    return list;

  // Otherwise, get the number of lessons left (adjusted for the amount per
      week)
  int numLessonsToPlace = Math.Min( (int)(TotalNumberOfLectures−
      numberOfLecturesAssigned), (int)LecturesPerWeek );

  // Create the lessons and add them to the list
  for (int i = 0; i < numLessonsToPlace; i++)
  {
    Lesson a = new Lesson();

    a.Name = this.Acronym;  // Name it after the course
    a.Owner = this;

    list.Add(a);
  }

  return list;
}
```

The above method first checks whether the course has already finished, or if there are any outstanding prerequisite courses that have yet to finish. The first is done by comparing the number of placed lessons with the total number of lessons that are to be held. The latter is done by calling *HasFinished()* on all prerequisite courses to check if they have finished. If the conditions are fulfilled, the number of outstanding lessons are calculated based on the number of lessons per week and the remaining lessons. They are constructed and the list returned.
Placing a single lesson in a week is done by *TryToPlace()*:

Listing 3.14: GeneralSchedule.TryToPlace()

```
private bool TryToPlace(Lesson lesson, int weekStart, int startDay)
{
  // Check for user cancellation
  // If invoked, just return success so we quit fast
  // All lessons will be "placed" fast, but the schedule is disposed of anyway
     .
  if (worker.CancellationPending)
  {
    eventArgs.Cancel = true;
    return true;
  }
  if (!Pass(lesson, weekStart, startDay, PassType.First))
    if (!Pass(lesson, weekStart, startDay, PassType.Second))
      return false; // If they both fail, return false;

  return true;  // Otherwise, we succeeded
}
```

TryToPlace() simply performs two passes over the week, with different criteria for placing lessons as described in the pseudocode. If both fail, it returns false, otherwise true.
This method is also the place where user cancellation is checked. If the cancel event has been triggered, the method returns true, which normally indicates that the lesson was successfully placed. This way, the schedule generator will quickly burn through the semester and finish. We don't care about the resulting schedule as it will be discarded anyway.

Pass() runs through the week and tries to place the current lesson in the first available block:

Listing 3.15: GeneralSchedule.Pass()

```
private bool Pass(Lesson lesson, int weekStart, int startDay, PassType pass)
{
  // Get a reference to the availability calendar for the semester
  AvailabilityCalendar calendar = semester.GetAvailabilityCalendar();

  for (int n = 0; n < 7; n++)
  {
    int index = weekStart + (startDay + n) % 7;

    // Check whether other lessons in this course lie nearby
    if (!CheckProximity(index, lesson.Owner))
    {
      // Morning block
      // Figure out if the block is marked available by the semester
      bool semesterAvailable = false;
      bool teacherAvailable = false;

```

```
          if (pass == PassType.First)
          {
            if (calendar.GetAvailabilityDay(index).StateMorning == ActivityState.
                Available)
              semesterAvailable = true;
          }
          else if (pass == PassType.Second)
          {
            if (calendar.GetAvailabilityDay(index).StateMorning == ActivityState.
                Available || calendar.GetAvailabilityDay(index).StateMorning ==
                ActivityState.Limited)
              semesterAvailable = true;
          }

          // And if it is marked available by the teacher(s)
          ActivityState teacherState = lesson.Owner.IsTeacherAvailable((uint)index
              , BlockType.Morning);
          if (pass == PassType.First && teacherState == ActivityState.Available)
            teacherAvailable = true;
          else if (pass == PassType.Second && (teacherState == ActivityState.
              Available || teacherState == ActivityState.Limited))
            teacherAvailable = true;

          // If it is available, and the teacher is too, proceed.
          if (semesterAvailable && teacherAvailable)
          {
            if (TryToPlaceBlock(lesson, days[index].Morning))
              return true;
          }

          // Afternoon block
          // Figure out if the block is marked available by the semester
          semesterAvailable = false;
          teacherAvailable = false;

          if (pass == PassType.First)
          {
            if (calendar.GetAvailabilityDay(index).StateAfternoon == ActivityState
                .Available)
              semesterAvailable = true;
          }
          else if (pass == PassType.Second)
          {
            if (calendar.GetAvailabilityDay(index).StateAfternoon == ActivityState
                .Available || calendar.GetAvailabilityDay(index).StateAfternoon ==
                 ActivityState.Limited)
              semesterAvailable = true;
          }

          // And if it is marked available by the teacher(s)
          teacherState = lesson.Owner.IsTeacherAvailable((uint)index, BlockType.
              Afternoon);
          if (pass == PassType.First && teacherState == ActivityState.Available)
            teacherAvailable = true;
          else if (pass == PassType.Second && (teacherState == ActivityState.
              Available || teacherState == ActivityState.Limited))
            teacherAvailable = true;

          // If it is available, and the teacher is too, proceed.
          if (semesterAvailable && teacherAvailable)
          {
            if (TryToPlaceBlock(lesson, days[index].Afternoon))
              return true;
          }
```

```
        }
      }

      return false;
    }
```

The method runs through all days in the week (first the morning block, then the afternoon block), the starting day is determined by the two aforementioned tries variables. First, the method checks whether there are any other lessons in the course nearby (as lessons are required to be more than one day apart in a week) by calling *CheckProximity()*.
Then, the method determines if it is allowed to place the lesson in the current block. This is done by first checking semester and then teacher availability (taking into account whether it is the first or second pass). If both are available, the method continues, otherwise it skips forward to the next block. If the availability conditions are met, the method will call *TryToPlaceBlock()*. If the lesson is successfully placed, the method returns true. Otherwise, it skips forward to the next block. If the entire week is used without a positive result, the method returns false.

CheckProximity() determines whether any adjacent days have lessons in the same course as a given lesson. This is simply done by iterating through the three days that make up the range we wish to keep clear and checking each individual block:

Listing 3.16: GeneralSchedule.CheckProximity()

```
    private bool CheckProximity(int index, Course course)
    {
      // Check all three days in the range
      // Make sure that we do not run outside the bounds of the array
      for (int i = Math.Max(index - 1, 0); i < Math.Min(index + 1, days.Length); i
          ++)
      {
        if (days[i].Morning.ContainsLessonInCourse(course))
          return true;

        if (days[i].Afternoon.ContainsLessonInCourse(course))
          return true;
      }

      return false;
    }
```

TryToPlaceBlock() attempts to place a given lesson in a given block:

Listing 3.17: GeneralSchedule.TryToPlaceBlock()

```
    private bool TryToPlaceBlock(Lesson lesson, Block b)
    {
      // Conflicts?
      if (!b.IsInConflictWith(lesson))
      {
        // If no, then find a suitable room

        List<Room> availableRooms = semester.GetListOfRooms();

        // Try the rooms that match the course preference
        foreach (Room r in availableRooms)
        {
            if (r.Type == lesson.Owner.PreferredRoomType)
            {
              if (TryToAssignRoom(r, lesson, b))
                return true;
            }
          }
        }

        // If we reach this point, either there was a conflict or we could not find
            a room.
        // Anyway, we failed.
        return false;
      }
```

Checking for conflicts is done by calling the *IsInConflictWith()* method (described below). If no conflicts exist, it is necessary to find a suitable room for the lesson. The method retrieves the list of rooms, and runs through them until one passes the call to *TryToAssign-Room()*, which checks if the room size can accomodate the number of students that take part in the lesson and whether it is available. If there is a conflict or no room could be found, the method will return false. Otherwise, it will return true when a room is found.

IsInConflictWith() checks whether a lesson candidate is in conflict with the other lessons that have already been assigned to the block:

Listing 3.18: Block.IsInConflictWith()

```
      public bool IsInConflictWith(Lesson candidate)
      {
        foreach (Activity a in activities)
        {
          if (a.GetType() == typeof(Lesson))
          {
            if (((Lesson)a).Owner.IsInConflictWith(candidate.Owner))
              return true;
          }
        }

        return false;
      }
```

The method simply runs through all lessons currently assigned to the block and determines whether their courses are in conflict with each other by calling *IsInConflictWith()* on them:

Listing 3.19: Course.IsInConflictWith()

```
      public bool IsInConflictWith(Course c)
      {
        // Check student sets
        foreach (StudentSet s1 in GetListOfStudentSets())
          foreach (StudentSet s2 in c.GetListOfStudentSets())
          {
            if (s1 == s2)
              return true;
          }

        // Check teachers
        foreach (Teacher t1 in GetListOfTeachers())
          foreach (Teacher t2 in c.GetListOfTeachers())
          {
            if (t1 == t2)
              return true;
          }

        return false;
      }
```

IsInConflictWith() checks the lists of teachers and student sets from both courses against each other. If a match is found, a conflict exists and the method returns true. Otherwise, if the two courses do not conflict with each other, it will return false.

3.2.4 Specific Schedule

When a schedule has been generated by *GeneralSchedule()*, it contains all lessons for all student sets etc. that have been planned. In order to produce readable schedules that only contain the information that pertains to a single subject, several instances of *SpecificSchedule* can be created. There are four overloaded versions of the constructor for this class, allowing one to extract a schedule that only contains lessons for a single teacher, student set, room or course. Shown below is the constructor for constructing schedules for courses (The other three are similar.):

Listing 3.20: SpecificSchedule.SpecificSchedule()

```
      public SpecificSchedule(Course course, GeneralSchedule generalSchedule)
      : base(Semester.Instance().GetNumberOfDays())
       {
        subject = course;

        // Go through each day and copy the relevant activities
        for (int currDay = 0; currDay < semester.GetNumberOfDays(); currDay++ )
        {
          // First the morning
          List<Activity> activities = generalSchedule.GetListOfDays()[currDay].
              Morning.GetListOfActivities();
          foreach (Activity a in activities)
          {
            if (a.GetType() == typeof(Lesson) && ((Lesson)a).Owner == subject)
            {
              // Add it to the specific schedule
              this.days[currDay].Morning.AddActivity(a);
            }
          }

          // Then in the afternoon
          activities = generalSchedule.GetListOfDays()[currDay].Afternoon.
              GetListOfActivities();
          foreach (Activity a in activities)
          {
            if (a.GetType() == typeof(Lesson) && ((Lesson)a).Owner == subject)
            {
              // Add it to the specific schedule
              this.days[currDay].Afternoon.AddActivity(a);
            }
          }
        }
      }
```

The method creates an empty schedule (through the base constructor in the *Schedule* class), and then it iterates through all days in the general schedule, adding lessons that match the subject to the new schedule.

3.3 Unit Testing

As described in section 2.7, unit testing using the NUnit Framework (version 2.4.8) has been used to test the core classes of the system and their methods. Below is a table of all test classes, how many tests they contain and how many of these tests passed or failed. After the table a detailed runthrough of the tests in the different classes will be given.

Name of test class	Number of tests	Tests passed	Tests failed
PlanItCourseTest	8	8	0
PlanItAvailalbilityCalendarTest	3	3	0
PlanItRoomTest	1	1	0
PlanItSemesterTest	4	4	0
PlanItStudentsetTest	2	2	0
PlanItTeacherTest	2	2	0

3.3.1 PlanItCourseTest

This is a class testing the *Course* class.

- AddLessonTest: This test whether or not a lesson will be added or removed currectly to or from a course, and if the list of lessons is empty if no lessons have been added, and if all lessons have been removed.
- AddPrerequisiteTest: This test whether or not a course is correctly added or removed to or from another course as its prerequisite, and if it produce an empty list of prerequisites when all prerequisites have been removed from a list, or if no prerequisities is added.
- AddStudentSetTest: This test whether or not a studenset is added or removed to or from attending a course, and if a list is empty after the last studenset has been removed from the course, or if no studentset has been added.
- AddTeacherTest: This test whether or not a teacher is added or removed to or from being attached to a course , and if the list is empty when creating the course, or when all teachers have been removed.
- GetWeekLessonsTest: This test whether or not the functions return the number of lectures for a week, both if there are still lessons to be held, and if all lessons have been placed.
- InitTest: This test the starting values for the different course types, such as if name, acronym, etc. have been saved correctly.
- IsTeacherAvailableTest: This test if the function IsTeacherAvailable produces the right activitystate answer. So if the activitystate is limited it will return limited.
- ResetLecturesAssignedTest: This test whether the function resets the number of lectures that already have been assigned back to 0.

3.3.2 PlanItAvailalbilityCalendarTest

This is a class testing the *AvailabilityCalendar* class.

- AddTest: This test if two holidays can be added two an availalbilitycalendar with different availability and dates, and if these are saved correctly.
- ChangeStatusTest: This test if the availability of a dey already in the schedule can be changed, testing both afternoon and morning.
- InitTest: This test if a availalbilitycalendar is initialized correct according to the length.

3.3.3 PlanItRoomTest

This is a class testing the *Room* class.

- InitTest that checks if changing name, size and type of rooms works.

3.3.4 PlanItSemesterTest

This is a class testing the *Semester* class.

- AddTest that checks if courses, rooms, student sets and teachers are added correctly to the semester's lists of these.
- RemoveTest that checks if courses, rooms, student sets and teachers are removed currectly from the semester's lists of these.
- DestroyTest that tests whether the *Destroy* method does as expected; that is null the instance of the
- Semester class and clear all lists of teachers, courses, student sets and rooms.
- GetDateTest that tests whether the *GetDate* method returns the correct DateTime when given an uint daynumber of the day we wish to retrive, and whether the *GetDayIndex* returns the correct uint daynumber when given the DayTime of the daynumber we wish to retrive.

3.3.5 PlanItStudentsetTest

This is a class testing the *StudentSet* class.

- InitTest that tests if the constructor assigns the correct values to a new student set.
- AddCourseTest that creates a new course and checks whether it is added correctly to the student set's list of attended courses.

3.3.6 PlanItTeacherTest

This is a class testing the *Teacher* class.

- InitTest that tests whether assigning new firstname, lastname and username works correctly.
- AddCourseTest that creates a new course and checks if it is added correctly to the teacher's list of attached courses.

3.4 Program Documentation

During the implementation phase, the source code was documented using XML comments. These were used with Doxygen to automatically generate documentation in the form of HTML files. It contains class and member descriptions, hyperlinked source code and collaboration graphs to illustrate the relationships between classes. The documentation can be found on the enclosed CD.

4 Usability Evaluation

4.1 Method and Purpose

The purpose of the evaluation is to obtain a realistic list of usability problems, in order to improve the design of the system. In section 2.1.3 the quality goals of the scheduling system have been determined. As the test is meant to be a formative evaluation it is placed rather late in the development process and it is as such a validation test.

The intention is to evaluate the parts of the system which involves data collection from the user. In section 2.1.3, usability and comprehensability were given a high priority in order to ensure that the user found the system easy to use, intuitious and helping if needed. Thus it makes sense to try to evaluate this part of the system to check that the specified criteria is met. While it is difficult to evaluate these somewhat relative criteria because of lack of a reference point, the system can be assumed to comply better with the criteria, after the usability issues found in the test have been corrected. Explicitly, the intention is to evaluate functionality such as creating new student sets, rooms, teachers and courses and attaching teachers to courses and vice versa.

4.1.1 Procedure

In order to speed up the process of analysing the user evaluations, the *Instant Data Analysis* (IDA) method was chosen. This method greatly decreases time consumption compared to the traditional method of doing analysis of video and logfiles after the test, while still discovering a lot of the problems found using the traditional method. IDA was chosen due to time constraints.

Instant Data Analysis

IDA is a method which can greatly decrease the time spent on the analysis part of a usability test by taking advantage of the fact that a thinkaloud test session already involves a test monitor and one or more data loggers, who during the execution of the test gain insight into key problems experienced by the user. The procedure for carrying out a usability test using IDA consists of a test phase and a analysis phase. The test phase is just a typical think-aloud test session. The analysis phase begins immediately after conducting the test, with a brainstorming and data analysis sessions, which is set to take around one hour. In this session the most critical problems are discussed, categorized by severity (as critical, serious or cosmetic), while at the same time an IDA facilitator writes them on a blackboard. The idea is that the most critical or prominent problems are still fresh in the memory of the test monitor and data loggers, however, the notes from the data loggers are used as a reference. After the brainstorming session the IDA facilitator summarizes the contents of the blackboard into a categorized list of usability problems, which are then documented in the evaluation report.

4.1.2 Test Subjects

During the design phase of the system, users of the system have been expected to be educated in the use of the system, prior to interacting with it. A quick run through of the basics of the system will therefore be given before beginning any exercises to the test subject. Throughout the analysis document, the user has been described as a secretary at the department of computer science. Because of this, a single secretary from the institute of computer science has been recruited to participate in the user evaluation.

4.1.3 Test Procedure

The test subject is placed in the laboratory with access to the workstation with a pre-installed copy of the *PlanIt* scheduling system. The test leader explains the concept of the think-aloud method, the general purpose of the test and gives a quick run through of the system, in order for the user to gain some experience with the system, prior to using it. Afterwards the test leader hands the exercise list to the subject, and he or she is asked to start every exercise by reading the task out loud, subsequently explaining to the test leader how he or she plans to carry out the given task. Thereafter the test subject must state clearly when he or she starts, and then continuosly explain what he or she is thinking, what he or she is doing and what responses he or she expects from the system as a result of input. Meanwhile, two observers each write a preliminary problem list. After the system test is done the test subject is briefly interviewed in case he or she wants to reflect on the experience or other key issue with the system.

4.1.4 Location and Equipment

The usability test will take place in the usability laboratory in the Computer Science building on Selma Lagerlöfs Vej 300. The laboratory has a sealed room with a workstation for the test subject and leader and a room for the observers with equipment for recording video and audio. Equipment will not be used for recording data, but to ease the work of the observers so they can follow everything that happens.

Needed equipment:

- Workstation with Windows XP or Vista with Microsoft .NET framework 2.0 or newer
- Overhead camera aimed at the test subject
- Screen logger, capturing the PC display monitor signal

4.2 Usability Problems

Presented below is the list of identified problems in the usability test. If the problem description is preceded by a term other than *General*, it implies that the problem was found to be specific to a certain window. In this case, the term refers to the appropriate screenshot in the appendix (section 8.2).

After problems have been identified, they will be categorized into the follow three categories:

- *Cosmetic*: The user is delayed for a couple of seconds or less, showing a mild irritation or is confronted with information that deviates slightly from his expectations.
- *Serious*: The user is delayed for 30 seconds or more, showing obvious signs of irritation or is confronted with information that does not comply with expectations.
- *Critical*: The user is unable to complete the task, showing strong signs of irritation or is confronted with information that is critically mismatching expectations.

4.2.1 Problem List

Table 4.1

	Problem	Categori
1	*General:* Generally too much empty space, especially between buttons and controls	Cosmetic
2	*General:* The button for adding new objects, such as student sets or rooms is placed too far from other buttons	Serious
3	*New Schedule:* When creating a new schedule, not selecting mondays or sundays will make the selection "jump" back to the present day	Critical
4	*New Schedule:* The error message that appears when choosing an incorrect day for the start or the end of the semester, is often overlooked	Cosmetic
5	*General:* When inputting data into the system, the user is uncertain whether the data is saved	Serious
6	*General:* "Schedule Options" is not intuitively named	Serious
7	*Courses:* The user expected to be able to see the capacity of the preferred room	Cosmetic
8	*Teachers:* Not clearly defined whether the username should be written with or without the "@cs.aau.dk"	Cosmetic
9	*Courses:* The user expected the courses to contain a box for links to the course website	Serious
10	*General:* Lacks indication of which sub-window you are currently working with	Serious
11	*Teachers, Student sets, Rooms, Courses:* The empty textboxes (before you select a teacher/student set/room/course) confuses the user	Cosmetic

4.2.2 Detailed Description

1. ***General:*** **Generally too much empty space, especially between buttons and controls.**
The user thought that there was generally too much empty space throughout the program. This resulted in the attention being drawn towards the upper left side of the screen, thereby missing the list and "Add new" buttons in the right side.

2. ***General:*** **The button for adding new objects, such as student sets or rooms is placed too far from other buttons.**
When given the task of adding a new object, either a student set, teacher, course or a room, the user had alot of trouble finding the "Add new" button the first time. Most likely due to it being placed far away in the rightmost bottom corner of the screen, far away from any other buttons.

3. ***New Schedule:*** **When creating a new schedule, not selecting mondays or sundays will make the selection "jump" back to the present day.**
When given the task of creating a new schedule, while picking start and end date of the semester, the user did not notice the selection jumping back to the present day when not selecting a monday or sunday. Instead the user expected the selection to stay on the selected day and was rather confused when this happened.

4. ***New Schedule:*** **The error message that appears when one of these prementioned "jumps" occurs are hard to notice.**
When given the task of creating a new schedule, while picking start and end date of the semester, when the user selected an incorrect day, he/she did not notice the error message that appeared. The error message was too insignificantly displayed.

5. ***General:*** **When inputting data into the system, the user is uncertain whether the data is saved.**
When given the task of creating a new object, being either a student set, course, room or a teacher, after inputting data and pressing the "Add new" button, the user is uncertain whether the data has been saved before he/she pressed "Add new".

6. ***General:*** **"Schedule Options" is not intuitively named.**
When given the task of editting general avaialbility, the user would aimlessly look through the program's different windows to find somewhere to do this. The user did not think that "Options" described the action of editting availability and adding holidays.

7. ***Courses:*** **The user expected to be able to see the capacity of the preferred room.**
When given the task of creating a new course and giving the particular course a *Preferred room type*, the user was confused if room type referred to a specific type of room or a specific room with a given capacity.

8. ***Teachers:*** **Not clearly defined whether the username should be written with or without the "@cs.aau.dk".**
When given the task of creating a new teacher, the user was uncertain whether he/she should write "@cs.aau.dk" in the username or if it was automaticly included.

9. ***Courses:*** **The user expected the courses to contain a box for links to the course website.**
A course at the department of computer science normally has a website for containing exercises, and when given the task of creating a new course the user expected there to be a box to include this kind of information.

10. ***General:*** **Lacks indication of which sub-window you are currently working with.**
When given the task of creating objects in the program, either student sets, rooms etc, the user lacks indication of which part of the program he/she is currently working in. The user would sometimes click the corresponding button, for example the "Student sets" button repeatedly in order to make sure that he/she was in this window.

11. ***Teachers, Student sets, Rooms, Courses:*** **The empty textboxes (before you select a teacher/student set/room/course) confuses the user.**
When given the task of creating objects in the program, either student sets, rooms etc. The user was not sure if he/she was able to fill out the empty textboxes that appears before selecting an already created object or pressing the "Add new" button to create a new one.

4.3 Conclusion

The scheduling system named PlanIt has been tested at the Department of Computer Science at the University of Aalborg in the usability laboratory. The test was carried out with one test subject who was asked to think-aloud about his or her thoughts and expectations during the test while solving different tasks in the system.

It was discovered that the PlanIt system had several usability problems with overall structure and layout, namely too much empty space in the system, buttons not being grouped together correcly or some specific buttons not having the expected name. The more serious problems tended to relate to error messages being too insignificant, ie. not being displayed properly and therefore making the user miss them. The one critical problem that was discovered was due to a misconception, or bad description, of how to navigate the calendars when choosing start and end date for the schedule.

It was found that the menu bar in the left side of the system, the general naming of buttons and the very easy and effective form of inputting data worked rather well with the user. It was believed that correcting the problems found in the test could go a long way in making the system's overall usability better.

5 User Manual

This section is a guide on how to properly use the system, and is meant as a How-To manual for the end user. The guide will refer to various screenshots from the system, and consists of a *user* part and a *guest* part, as there is significant differences from the two login types. All screenshots are shown with test data added, to ease the understanding of the user.

5.1 Normal User

This section will guide the normal user through the system.

5.1.1 Login

The first screen the user will see when starting the system is the login screen. Here the normal user can enter his/her username and password, or enter the system as a guest user. Upon entering a correct username and password, the user will be presented to the main screen, where the buttons *New Schedule* and *Load Schedule* are available.

5.1.2 Main Screen

On the main screen, the user will have access to the *New Schedule* and the *Load Schedule* options. Clicking on the *Load Schedule* button will let the user find a previously saved *PlanIt* file to load into the system, providing access to most of the other options in the system. Clicking on the *New Schedule* button will show the schedule calendar screen to the user.

5.1.3 New Schedule Screen

On this screen, the user is asked to determine the timespan of the semester that the user wishes to plan a schedule in. The first day of the semester needs to be placed on a Monday, and the last day of the semester needs to be placed on a Sunday. If the user selects a date that is neither a Monday or a Sunday, or that is otherwise an illegal choice of date, the *schedule calender* screen will display one of the following errors:

- Start date must be a Monday.
- End date must be a Sunday.
- Start date must be before end date.
- End date must be after start date.

As the error message is displayed to the user, the date on the calendar is moved to the current date. When two legal dates have been selected, the *OK* button is enabled.

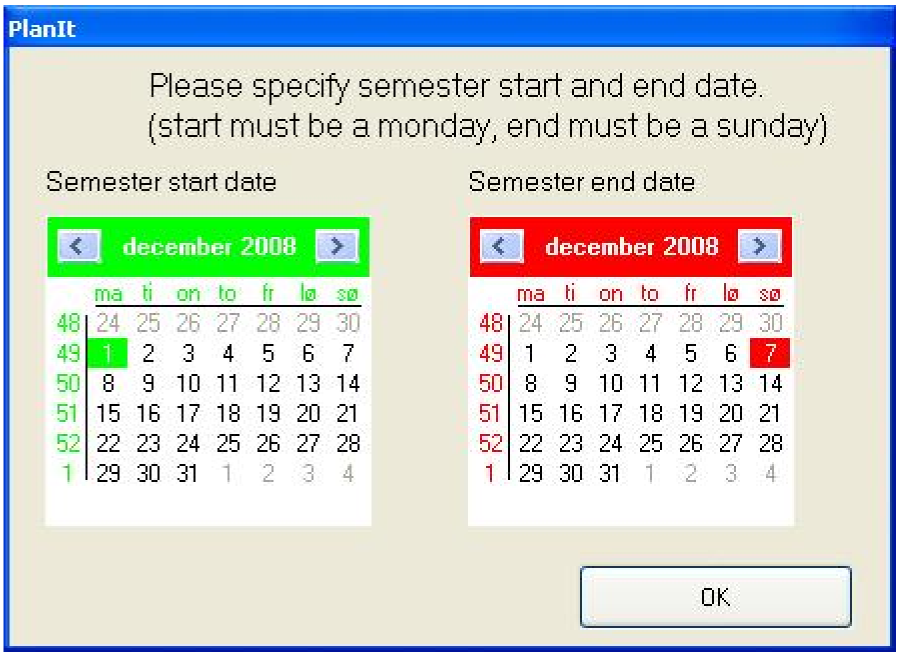

PlanIt
Please specify semester start and end date.
(start must be a monday, end must be a sunday)
Semester start date
Semester end date
december 2008
december 2008
OK

Figure 5.1: The *New Schedule* window. This is used when creating a new schedule to specify at which dates the schedule starts and ends.

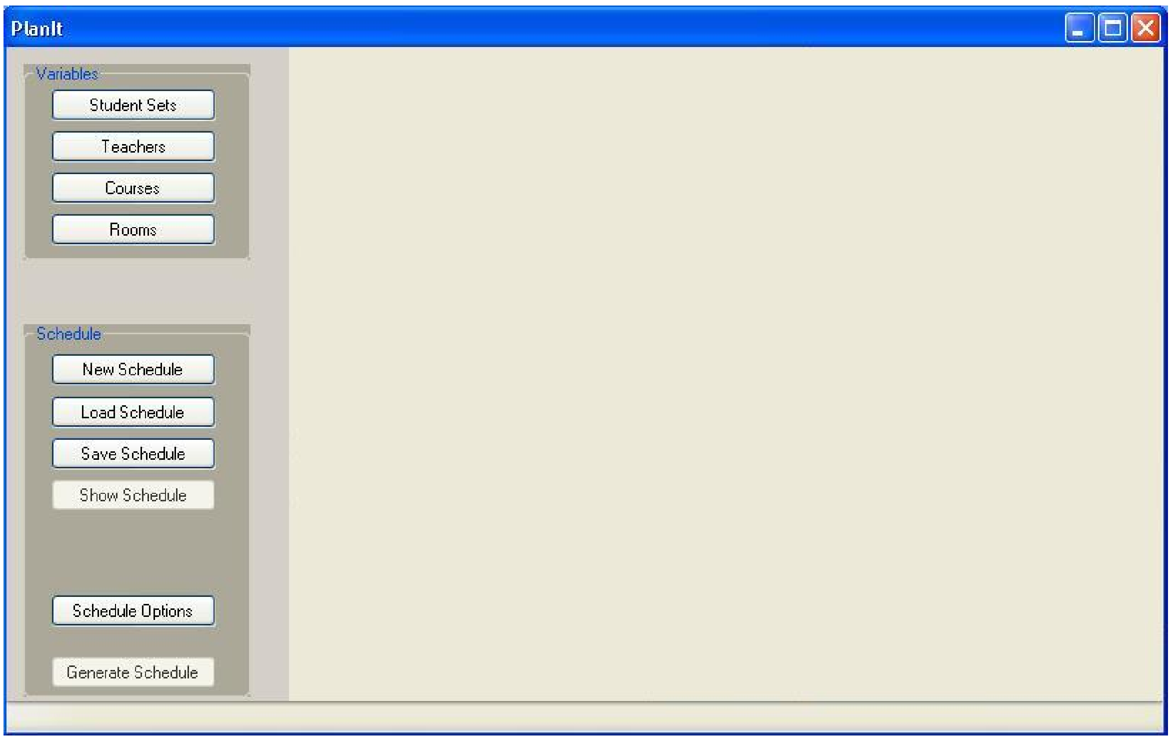


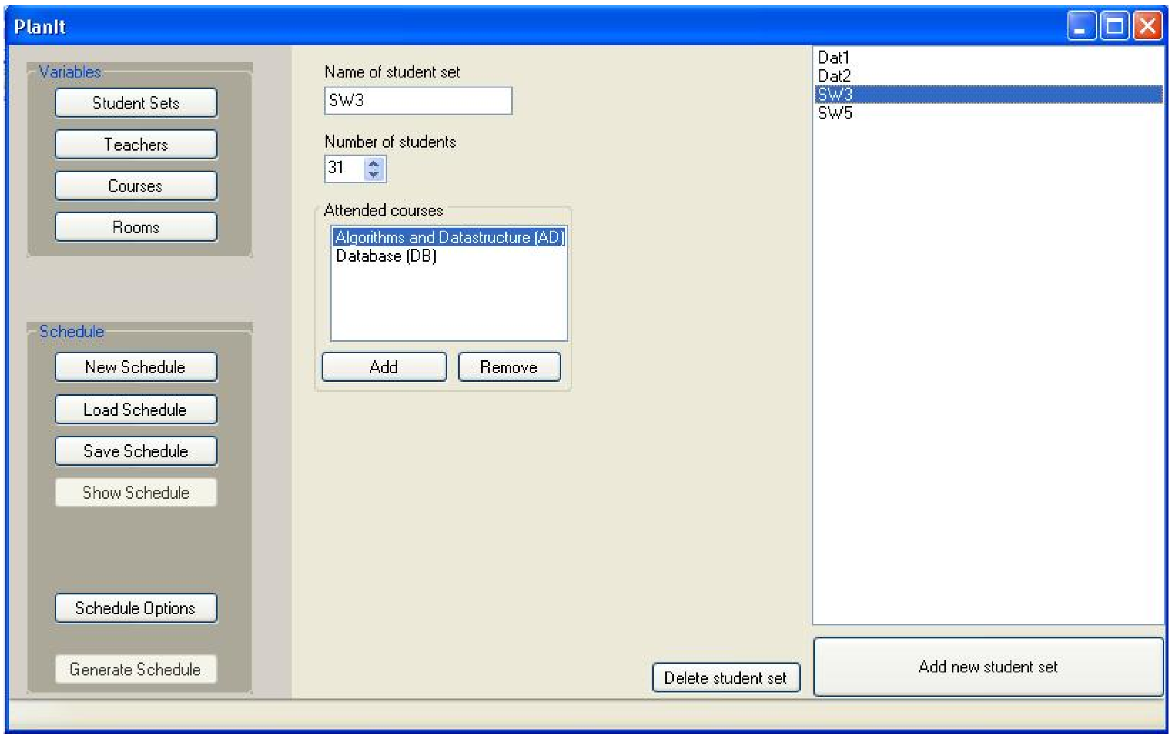
Figure 5.2: The *Main* window. This is used to navigate the system.

5.1.4 Main Screen - Continued

Upon coming back to the main screen all options have been enabled except for the *Generate schedule* and *Show schedule* options. The *Generate schedule* option will be enabled after clicking the *Schedule options* button and the *Show schedule* option will be enabled after clicking the *Generate schedule* option.

5.1.5 Add a Set of Students

To add a set of students to the semester, the user need to click on the *Student Sets* button on the main screen, and then the *Add new student set* button in the bottom of the student set screen. This will create a new set of students, where the amount of students in the set and the courses that the set will be following can be added (if created at the given time, see the section 5.1.7.



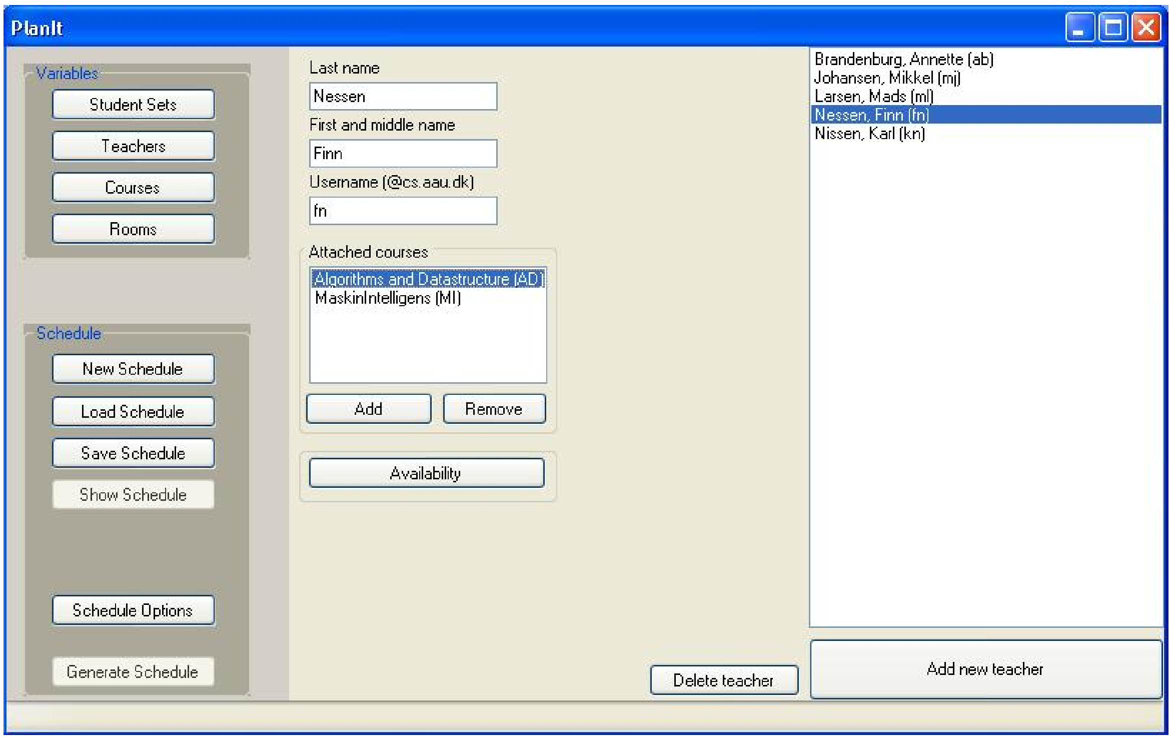
Name of student set: SW3
Number of students: 31
Attended courses:
Algorithms and Datastructure (AD)
Database (DB)

Dat1
Dat2
SW3
SW5

Figure 5.3: The *Student sets* window. This is used for creating and editing student sets.

5.1.6 Add a Teacher

To add a new teacher to the semester, the user need to click on the *Teachers* button on the main screen, and then the *Add new teacher* button in the bottom of the teachers screen. This will create a new teacher, where the fields for editing the teacher will become available. In the teacher screen, the courses that the teacher will be teaching can be added to the teacher (if created at the given time, see the section 5.1.7) and the days when the teacher is able to teach can be added to his/her personal calendar, by clicking the *Availability* button on the teacher screen.



Last name: Nessen

First and middle name: Finn

Username (@cs.aau.dk): fn

Attached courses:
- Algorithms and Datastructure (AD)
- MaskinIntelligens (MI)

- Brandenburg, Annette (ab)
- Johansen, Mikkel (mj)
- Larsen, Mads (ml)
- Nessen, Finn (fn)
- Nissen, Karl (kn)

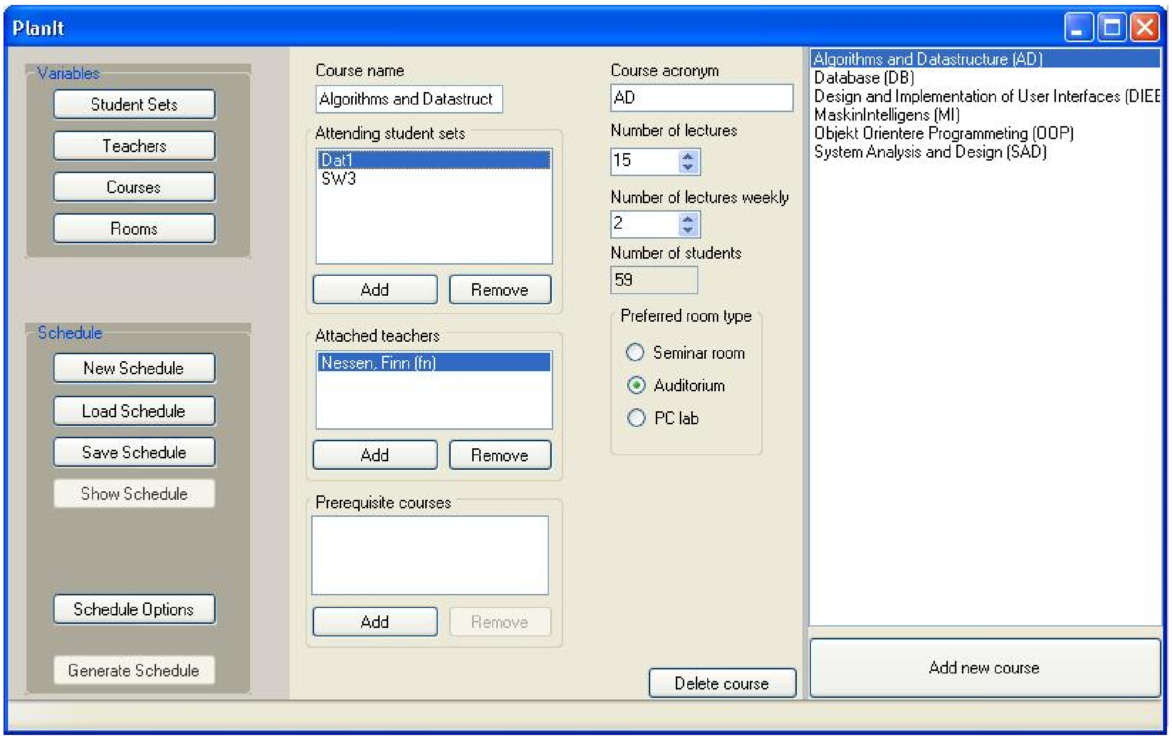
Figure 5.4: The *Teachers* window. This is used for creating and editing teachers.

5.1.7 Add a Course

To add a new course to the semester, the user need to click on the *Courses* button on the main screen, and then the *Add new course* button in the bottom of the courses screen. This will create a new course, where the name of the course and the acronym can be added, as well as how many lectures the course will be spanning and how many lectures a week the sets of students should have. This screen also allow the user to assign the course to the different sets of students (if they are created at the given time, see section 5.1.5, to assign one or more teachers to the course (if they are created at the given time, see section 5.1.6, and assign courses as prerequisite courses to the course, if the students following the course need another course before having the selected one. The course screen also shows the user the total number of students following the course, and allows the user to choose a room type that the course prefers, in case the teacher or course should have any special demands.

5.1.8 Add a Room

To add a new room to the semester, the user need to click on the *Rooms* button on the main screen and then the *Add new room* button in the bottom of the rooms screen. This will create a new room where the user can set the name of the room (e.g. 2.1.13), the capacity of the room and the room type.



Course name: Algorithms and Datastruct

Attending student sets:
- Dat1
- SW3

Attached teachers:
- Nessen, Finn (fn)

Prerequisite courses:

Course acronym: AD

Number of lectures: 15

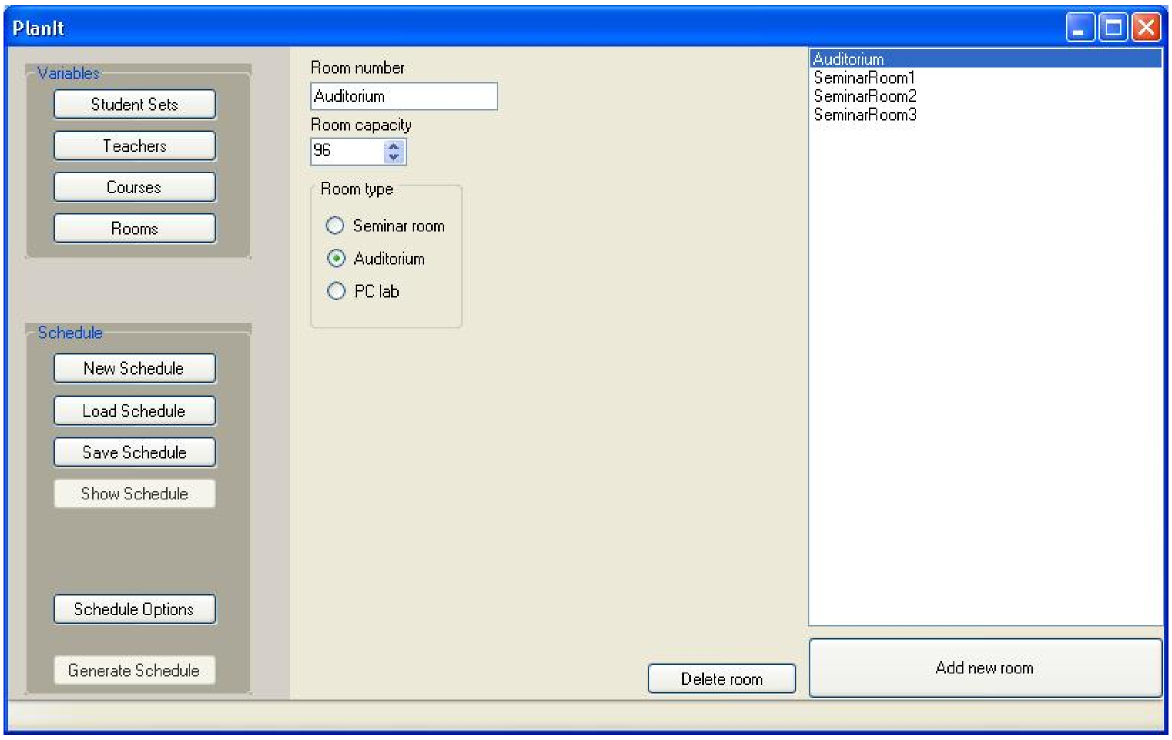
Number of lectures weekly: 2

Number of students: 59

Preferred room type: Seminar room [✓ Auditorium] PC lab

- Algorithms and Datastructure (AD)
- Database (DB)
- Design and Implementation of User Interfaces (DIEE
- MaskinIntelligens (MI)
- Objekt Orientere Programmeting (OOP)
- System Analysis and Design (SAD)

Figure 5.5: The *Courses* window. This is used for creating and editing courses.



Room number: Auditorium

Room capacity: 96

Room type: Seminar room [✓ Auditorium] PC lab

- Auditorium
- SeminarRoom1
- SeminarRoom2
- SeminarRoom3

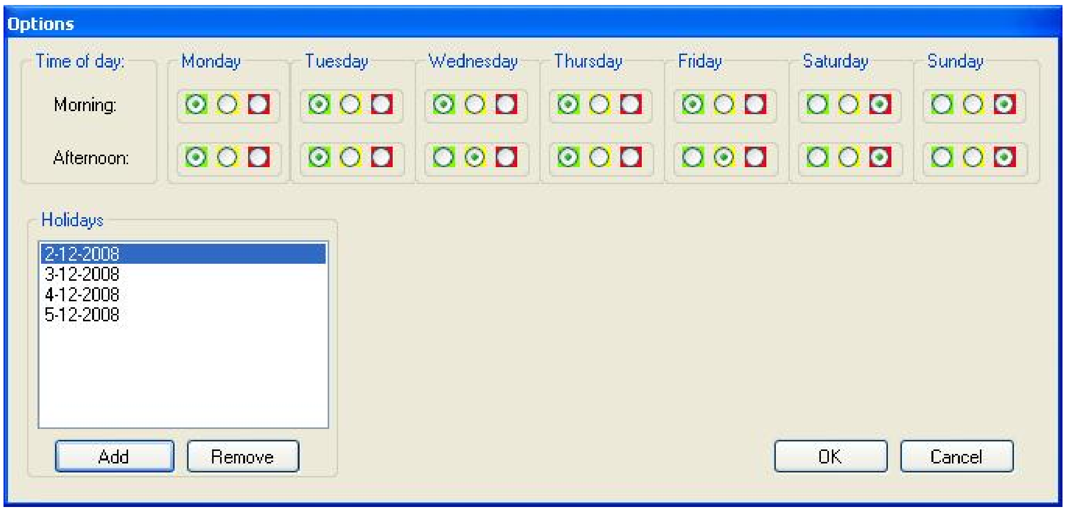
Figure 5.6: The *Rooms* window. This is used for creating and editing rooms.

5.1.9 Removing a Set of Students, Teacher, Course or Room

To remove a set of students, a teacher, a room or a course, the user simply needs to navigate to the corresponding screen (student sets, teachers, courses or rooms), select the desired element from the list at the right and press the *Delete ...* button in the bottom right corner, next to the *Add ...* button. This will automatically remove any associations to the selected element, as well as remove it from the planned semester. If the user wants to keep the selected element and just wants to remove an association to it (e.g. a teacher assigned to a course), the user simply needs to find the corresponding screen, select the element on the corresponding list and click the *Remove ...* button.

5.1.10 Schedule Options

The schedule options let the user determine which days of the week the semester should be planned in and which days in the semester should be holidays. The color codes for the days are: **Green** means available for scheduling, **yellow** means undesirable and **red** means unavailable for scheduling. For example; if the user wanted the schedule to have Wednesday afternoon and Friday afternoon clear of courses, these days should be set to yellow in the afternoon in order to have the system try to place courses somewhere else other than these days, and only place courses on Wednesday and Friday if no other times are available. Whereas Saturday and Sunday should be set to red for both days, as they are the weekend and as such should be kept clear of courses at all times.

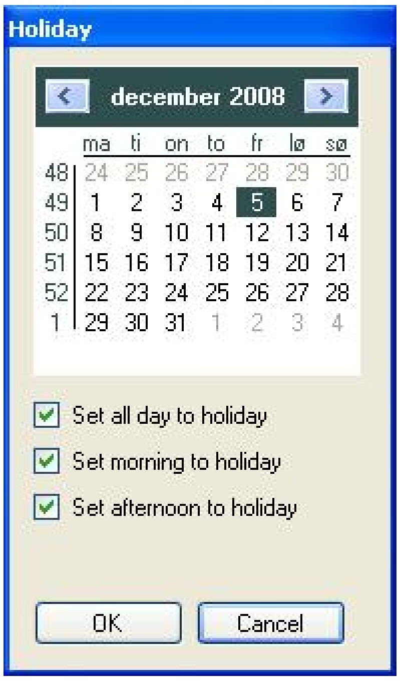

Options

Time of day:	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday	Sunday
Morning:	[✓ 🟩] 🟨 🟥	[✓ 🟩] 🟨 🟥	[✓ 🟩] 🟨 🟥	[✓ 🟩] 🟨 🟥	[✓ 🟩] 🟨 🟥	🟩 🟨 [✓ 🟥]	🟩 🟨 [✓ 🟥]
Afternoon:	[✓ 🟩] 🟨 🟥	[✓ 🟩] 🟨 🟥	🟩 [✓ 🟨] 🟥	[✓ 🟩] 🟨 🟥	🟩 [✓ 🟨] 🟥	🟩 🟨 [✓ 🟥]	🟩 🟨 [✓ 🟥]

Holidays
2-12-2008
3-12-2008
4-12-2008
5-12-2008

Figure 5.7: The *Schedule Options* window. This is used for editing general availability of the semester schedule as well as a list of holidays. A window with the exact same layout is also used for editing availability for teachers.

To add a holiday to the schedule, the user should click the *Add* button below the list of holidays, and a calendar will be shown to the user. On this calendar the user select a day within the semester, choose if the day should be a full holiday, or if it should only be a partial holiday (morning or afternoon), and then press the *OK* button. The holiday will then be added to the list of semester holidays and a new holiday can be added.



	ma	ti	on	to	fr	lø	sø
48	24	25	26	27	28	29	30
49	1	2	3	4	5	6	7
50	8	9	10	11	12	13	14
51	15	16	17	18	19	20	21
52	22	23	24	25	26	27	28
1	29	30	31	1	2	3	4

- ✓ Set all day to holiday
- ✓ Set morning to holiday
- ✓ Set afternoon to holiday

Figure 5.8: The *Holiday* window. This is used for adding a holiday to the list of holidays in the *Schedule Options* window. It is also used in the corresponding version used for teacher availability.

5.1.11 Generate Schedule

To enable this option, the available days for planning the semester first need to be set. To let the system plan the schedule, the user simply need to click the *Generate schedule* button on the main screen, wait for the process to be completed and then press the *OK* button when it finishes.

5.1.12 Show Schedule

To enable this option the semester first needs to be planned. For the user to see the finished and planned semester, the user simply need to press the *Show schedule* button on the main screen, and the planned semester will be shown.

5.1.13 Load Schedule

To load a previously made PlanIt file, the user should click on the *Load schedule* button on the main screen, find and select the file that the user wishes to load, and click the *OK* button. This will load the chosen file into the system, and the user can continue working on the data.

5.1.14 Save Schedule

To save a file, the user should click the *Save schedule* button on the main screen, decide on and write a name for the saved file and click the *Save* button.

5.2 Guest User

This section will describe the differences between the normal user login and the guest login

5.2.1 Login

As a guest user, you enter the system by pressing the *I'm a guest* button.

5.2.2 Main Screen

When entering as a guest user, only the *Load Schedule* option is available. Upon loading a previously saved file the *Show Schedule* option becomes available.

6 Conclusion

A system that can produce a schedule has been developed. The system was developed with several features, including both sending and receiving data from XML files, have a working graphical user interface which the user could interact satisfyingly with, planning schedules with the use of a greedy algorithm and set holidays for both teachers and a semester among others. This development was accomplished through a series of phases, starting with thorough analysis of the problem- as well as the application domain which ended in the analysis document. The analysis helped find the user group as well as narrowing down the problem so that it could be covered by the extend of the report. Based on what was learned from the analysis phase, a software solution was designed and described in the design document. Here several design criteria were created. The system architecture was also defined during the design phase. The design document served as the basis for our implementation effort.
In the implementation phase the designed solution was developed into a working system, which fulfilled a satisfying amount of the goals set forth during the analysis. Unit testing and usability evaluation was carried out in order to find and correct errors and problems within the system and its graphical user interface. Furthermore a user manual was written in order to guide and help the new users.

7 Student report

7.1 Introduction

This chapter will concern itself with how the different aspects of the entire process worked out. It is meant as a retrospective and as a mean to help define the good and bad things which ocurred. This chapter will analyse how the members working with the project interacted with each other, the new learning methods that they were required to learn, and how they interacted with the system itself. All of this will be described in several different sections each covering their specific area. Each section will have a general description about the issues handled. There will also be descriptions of what thoughts were made throughout the different processes and how these affected the work and shaping of the system. For each section there will also be a small discussion and conclusion, which will be used at the very end where final thoughts will be given. The Student report is meant as both a way to describe the huge task it has been to develop a system fitting the needs of the users, but also to discuss the ups and downs of the entire process.

7.2 Analysis

This section will concern itself with the analysis document, which is the first document of the report. The content of this document was to be an analysis of the problem we faced as well as a possible solution. Looking back at many of the problems, we have had both superficial and in-depth discussions of various solutions. We have also been given a broad set of tools to deal with various situations. We will here try to give the reader a deeper understanding of how these discussions and tools helped us form a solution we saw fitting.

FACTOR

At the very beginning of the project we stated a system definition. This definition is a rough description of the system using some of the tools we have learned from "Object Oriented Analysis & Design". The general idea with the system defintion is to provide both the developers and the customer with a fundamental definition of the systems capacity. As part of this definition we sat up the FACTOR criterion, which describes all the primary aspects of the system.

During this process we had a discussion whether the system should be web-based, stand-alone or client-server. We discussed back and forth and came up with some pros and cons of these different solutions:

Web application: The reasons for a web-based application were that both showing and editing the schedule would be an integrated part of a website. This would make it possible for teachers to report in absent themselves, thus making the schedule more dynamic. There would also be no need for schedule formatting and exporting, since the viewing would also be part of the integrated web solution. The cons of this solution was that we had not recieved any lecturing in web programming, which would be a great constraint.

Stand-alone: The arguments for a stand-alone application were that the schedule editing would be more controlled. Only persons with direct access to the application would be able to make changes in the schedule. Thus for the teachers to make any changes they would have to go through the secretaries. Afterwards the schedule could easily be exported to a website for teachers and students to see. This solution would therefor be focused more on the authenticity of the schedule. Another pro was that we had recieved lecturing in windows forms programming, which would easy the programming of the interface.

Client-server: The client-server idea was quickly discarded because it would be too troublesome to use. Users would be to annoyed having to download a piece of software just to view their schedule. Besides that it did not really have any pros that the other solutions could not provide.

After discussing the pros and cons of these solutions, we chose to make the application stand-alone. Though all of us would have prefered the web application, none of us felt we had the necessary experience with web programming. Time was also a great factor in this decision as the programming of a web interface would be very time consuming compared to a windows forms interface. By choosing the more time friendly solution in form of a stand-alone system, we believed it more likely to end up with a working application.

After having stated the system definition we have not really used it since, which of course does not mean it has not been important. It has served well as delimitation for the project whenever we have considered various solutions. Our mutual opinion is that it has been really helpful, in situations where drastic changes are being proposed, to keep the project on course.

The Problem Domain

In our analysis of the problem domain we made several class diagrams. The purpose of these class diagrams was to get a clear understanding of the reality that made up the problem. Though most of us had a general idea of how scheduling worked, none of us really expected the level of complexity that we suddenly faced.

In the creation of the diagrams we used the so-called bottom-up method. First we found the different objects and events that made up our problem domain. We then worked our way up trying to abstract from the individual objects and make out common phenomenon. When all the necessary phenomenon had been identified we would make out relations between these.

In the beginning we had a hard time indentifying the objects and events in our problem domain. We used a lot of time discarding several class diagrams we had made, untill we were left with one we were satisfied with, which is the one seen in figure 1.4. Especially the relations between many of the classes seemed complicated to us as we always seemed to miss some important elements or connections. A problem that often repeated itself during the analysis of the problem domain was the lack of abstraction. Those of the group members who had experience with programming found it hard not to think in terms of implementation when creating the class diagram.

Most of our frustrations revolved around the representation of time and activities in the schedule. We had a hard time figuring out how to express the timeframe phenomena, which is also why it has been removed in later versions. Apart from that we also had some discussions concerning the structure of the classes. Some of these discussions regarded an abstract super-class for Teacher, Room and Students. The reason for this would be they were all in a way a subject of a schedule. In the end however we could not really explain why we needed it and it was discarded.

Afterwards we have looked back at the process and tried to analyse what went wrong. The conclusion we came to are that we have simply been to focused on the details. Instead of taking the abstract approach to the analysis of problem domain and trying to describe this, we tried to describe an implementable solution instead. Our problem domain thus became more of a design rather than an analysis. In future projects we will strive to be more abstract minded during the analysis.

7.3 Design

This section will concern itself wih the second part of the report, namely the design document. The design documents main purpose was to expand the system analysed in he analysis document. The design phase also focused upon creating various crierias for creating a working system. Where the analysis document concerned itself with the problem and how a system solution could be created, the design document focuses on the chosen solutions as well as propose an implementable design.

Scheduling Algorithm

This section discuss the choices we made concerning the scheduling algorithm. This was probably the most complex part of our entire project and the reason why scheduling is even a problem. As it is today there is no really efficient algorithms for planning schedules. Here we will discuss why we chose the algorithm we did and what we could have done instead.

The schedule planning problem can be thought of as a graph coloring problem. In our case each vertice would represent an activity and each color would be a block in the schedule. The edges between the vertices are all the conflicts that occours between the activities. Our goal would now be to see that not two vertices of the same color is connected. If this can be done a solution exists. Below is described different algorithms for solving this problem.

Brute force algorithm: The simplest approach to the graph coloring problem is the brute force method. This method will simply try all possible combinations untill a solution has been found or terminate if no solution exists. However this method is very computationally expensive and often provides the worst possible solution in matter of time consumption.

Applied to our problem we would consistently run through all the activities and assign an available block. If this fails at some point and conflicts occured we would go one step back and assign that block differently. This would be repeated untill a usable solution was found or terminate in case no solution was found.

Greedy algorithm: Another approach to the problem would be to use a so called greedy algorithm. Such an algorithm would find an optimal solution at each local point, which would hopefully yeild a globally optimal solution in the end. This is rarely the case, but the greedy algorithm provides a good approximation of an optimal solution and is faster than the brute force method.

Applied to our problem we would run through the activities and assign the best block available based on an order assigned to them. The assignment of blocks would continue untill a solution was found or it terminated.

Cost-based solution: In our discussions of various algorithms we also talked about the possibility of adding a cost-based element to the algorithms. Each solution would then get a cost based on the viability of the solution. Each change made would affect the cost in some way and in the end we would end up using the solution with the highest cost. This would also enable us to adjust the cost modifier of the various variables according to the type of solution we wanted.

This could easily be combined with both the brute force algorithm and the greedy algorithm. When a solution had been found using either of these methods, we could perfect it using our soft contraints and calculate a cost based on the viability of the solution.

Genetic algorithm: After the discussion of a cost-based addition to our other methods, we ended up discussing the possibility of a genetic algorithm. Using such an algorithm we would generate a number of random schedules. Each of these schedules would then be evaluated using a set of criteria and the fitter solutions would be selected. These solutions would then be paired together, and used to produce a set of "children". The "children" produced would be mutated in some way to create new solution which resembles their parents. This process would be repeated untill some predefined criteria was reached, which could e.g. be a given cost based on the viability of the solution or a fixed number of iterations.

In our solution we chose to implement the greedy algorithm. This decision was made mainly due to time contraints and because it fitted the needs of the project. Our primary goal in the project was to develop a working schedule planning software and not find an optimal solution in terms of algorithms. Thus we chose a simple and fast solution which provided us with a usable solution.

System Architecture

In this section we will discuss the reasons for our choices of architecture and the discussions we have had concerning these.

One of the key items in the design document is the system architecture. This part is used mainly for setting some criterias and define what the most important issues the system should solve. These criteria have been created as an expanded and merged entity derrived from the *Goals for use* and *Quality goals* sections. Those criterias and why we chose those from all of the existing, will be described here.

> Effectiveness
> The most important thing for the system was defined to be that it needed to be effective at its task. The reason why we chose this as the most important criteria was that we above all else wanted to generate a good and working schedule.
> Efficient
> Another criteria which was valued very high was that the system needed to be efficient. This criteria was chosen because we acknowledged that eventhough the most important thing was to create good schedules, it was also important that the system would to it within a suitable timeframe.
> Reliable
> The third most important thing was that the system was reliable. This of course was chosen as we found it very important that the system would not perform unexpected actions or even crash.

These three criterias are the most important for our system. Of course there are also criterias associated with the interaction with the user, as we found this to be very important as well. Besides that we also wanted the systems user interface to follow some basic standards and be somewhat similar to the design of the standard Windows form.

All of these criteria were what we chose to create our system from. There were several other criterias that could perhaps be included. Testable for instance, meaning the ability to test the system was valued quite high for us as well, but we chose not to include it, because we felt that if the criteria regarding the user interactions with the system, were fulfilled satisfyingly, then the system would be testable per definition. We also decided on not to focus on safety and security as we previously defined that the system would not be a web application, but a stand alone system working on one computer. We therefore agreed that security issues should not be necessary to deal with in the first place.

In general most criteria we valued high are covered in those we chose. As many criteria either associated themselves with the interface and the user interactions, we felt that making sure the system followed certain standards as well as made the system simple would solve all of these. The other half of criteria are mostly associated with how the system is running. We felt that by making sure the system was efficient, reliable and effective that we covered all the main issues that might occur while interacting with the system.

Component Architecture

During the design phase it was also neccessary to determine how the system would interact with itself and its user. This was done in the component architecture. We wanted the architecture to have access to XML documents, as we had chosen these as our way of saving the schedules and data. Therefore we quickly became aware that a persistence layer was well suited for saving and loading data from the XML documents. Above the persistance layer we found that having a model layer with objects was a good choice. If needed the models could go one layer down to the persistence and ask for data whenever needed. Of course the models also needed functions that told them what to do and these functions were put into the function layer, above the model layer. Above the functions layer we found the user interface. This was where the user would interact with the system and issue functions. This architecture was well liked in the group as it fulfilled how the group saw that the different components should interact with each other. It was important though, to define what priveliges the components had. We found that having an open architecture was good for the system as it meant that components could access other layers. We were though concerned that it would cause problems with rights, so we assigned the component architecture to be strict. This meant that the components could only access layers below their own. This was a good choice as we later found that some functions did not have to interact with the model layer, but could go directly to the persistence layer, saving time in the process.

Components

This section will concern itself with the reworked class diagram from the analysis. After reviewing the diagram we found several inconsitencies and thus decided to start all over. The reason we chose to do this instead of updating the already existing class diagram, was that the entire system structure was deemed unfit and that it would be faster to create a new structure from the start. This structure was still somewhat based on the analysis of the problem as we kept most of the classes, which were most likely not about to change. Instead we focused our effort on a new structure which should give a better description of the relations. Here we will describe what changes has been made and why we have chosen to do so.

Our first decision was to split apart all classes and discard those we doubted were necessary. Thus we changed the whole timeframe concept by removing the timeframe object, as we realised the time of an entire semester could be described by only using days. Thus we discarded a lot of unnecessary classes and as a result we gained a simpler class structure. This was of much benefit to us in the later implementation phase.

We also realised that we needed a class for all the information outside the schedule. To solve this problem we chose to let semester represent both the time of the entire semester and all the different objects connected to it such as students, teachers, courses etc. By doing this and letting schedule associate to semester we could maintain a lot of different schedules at the same time. This was the primary reason we chose to change the entire functionality of the semester class.

Another change was the addition of calendars to maintain availability. This feature added a lot of complexity to the solution as each teacher would need to have a calendar of their own. However we felt that this solution would be the simplest way to solve a problem as difficult as availability for every teacher. We had discussed other solutions as well such as two dimensional arrays the length of all days in the semester. This array would then hold boolean values according to the teachers availability. Although this would also be an easily implementable solution we felt the calendar approach would provide us with a better understanding of the problem.

The reworking of the structure and the different classes gave us a clearer structure and a better model of the system. It also left us with a much more implementable solution.

7.4 Implementation

In the start of the implementation phase, three of the groupmembers did most of the sourcecode writing, and layed down the fundamentals for the system, while the other groupmembers were building the layout of the graphical user interface.

After the fundamentals had been established, the workload of designing and writing the sourcecode was distibuted to the various groupmembers according to their individual programming skills in C#, with the most experienced developers working in pairs with the less experienced.

After one week of programming in this manner, the pairs split up, and would then be working individually. This was done because the less experienced developers had come up to speed with the other developers, alowing them to focus on other parts of the system by themselves, and because the number of classes that needed to be programmed was growing in numbers and complexity, in such a way so that the resources of the developers would be better spent, by working on seperate parts, instead of helping each other with the same part. We named this method of programming, *buddy programming*, as the former programming pairs would still be working in roughly the same areas of the system, and still be helping each other with various problems that would occur, but they would not be sitting in pairs, nor would they use the same computer or program the same functions.

This distribution of work was keept for the rest of the implementation phase, with the different developers getting classes of varying degrees of complexity assigned, as they gained more and more experience with programming. As prevoiusly mentioned, the developers would be interacting with eachother by either working alone or with their programming buddy. Furthermore a lead developer was chosen, who had the responsibility to assign work assingments to the different developers, if they were unsure of what to work on, after having finished an assigned class or system part.

The implementation phase was in many aspects as much a learning phase as a development phase, as (as previously mentioned) the developers had varying degrees of programming experience from the begining, even though all had some knowledge of either C# or other programming languages. This meant that all the developers were required to learn the C# syntax, the use and programming of *Windows forms*, and how to use *Visual Studio*. Furthermore, the programs *NUnit* and *Doxygen* were used for testing purposes and sourcecode documentation respectively, so a certain amount of knowledge of the programs and testing methods was needed.

Throughout the implementation phase the groupmembers interacted with the system by iterative and repetitive functionality testing, while trying to simulate usage of the system.

Throughout the implementation phase a variety of choices were discussed, mainly in regards to what classes would have the different responsibilities (see 3.1). The classes *Semester* and *Schedule* in perticular were changed, to ease the implementation phase. The decision to change the classes were made, based on the discussion that while we could have kept the previous class design, the distribution of responsibilities would have become illogical and a nuisance in the programming. Changing the class responsibilities however would provide an increased overview of the system as a whole, and ease the programming.

Another major change was the decision to only save the entered data from crashes and errors, when the user activily pressed the *save* button, instead of the data being saved automatically as described in section 2.3.1. This decision was made because of time constraints and because it was argued that it would be more prudent to secure the system from crashes and errors, thus avoiding the need for automatic saving, instead of implementing automatic saving and risk an unstable system.

A less essential change in the system, was made in regards to how the system should handle errors. In section 2.3.1 error handling was explained using a single main error handler class. This was changed to the individual classes having responsability for handling errors in their respective area of the system, as this was much easier and faster to implement than a central error handler that should handle the errors.

The last noteworthy change in the system, was the cancellation of the expert keys *F1* to *F10* as descriped in section 2.3.2, due to time constrains.

To conclude on the implementation phase, it should be noted that the group have worked well together without any major obstacles or difficulties in the workflow. Communication between the individual groupmembers have been supportive and all members of the group have contributed to the implementation phase in accordance with their programming skills. It should also be noted that given more development time, the *automatic save* and *expert keys* features could be implemented.

7.5 Testing

Unit Testing

In this project, unit testing has been used as a way of validating the correctness of our classes and their methods. But because of time constraints only the core classes, such as *Teacher*, *Course*, *StudentSet*, and *Room* along with *Semester* and *AvailabilityCalendar* were tested using the NUnit framework. During the process, the tests did not only validate the classes - we also discovered programming errors and corrected these. This led to the conclusion that NUnit tests might be mondane and trivial to write and run, but they give great insight to how the different classes work together and is also a good tool for discovering errors.

Usability Evaluation

During our test we discovered that we had a few problems; the test leader was a little too helpful and some of the exercises contained errors, despite these problems we did gain a good understanding of the most critical and serious usability problems in the system. Some of the discovered usability problems were somewhat expected, however completely new and unforseen problems were also found. During the test we also became aware that the test subject expected more features than we had provided, for example the test subject would have liked a textbox for inputting course webpage links under the Courses. The test did not only show problems or errors, we also found and confirmed positive things about the interface. We learned that user evaluation is very essential when developing a system, it helps discover potential usability problems which the developers might not see because they know how it all works behind the scene. But it also gives the delveopers an idea of whether the users can use the system in the way that was intented.

7.6 Future work

This section will contain some of the future work which could be implemented to potentialy make the system better:

Corrections of usability problems
If more time was available one of the first issues which would have been looked upon would have been the usability problems we encountered during the test we performed with our user group. These issues are mostly related to the graphical user interface and with the interaction with the graphical user interface.
Viewing specific schedules
The viewing of specific schedules was never implemented in the graphical user interface. The schedule that was implemented into the graphical user interface contained all the data for every student set and it would have been nice if we could see data for a specific student set.
Web format export
One of the biggest things we would implement if we had more time, would be the web export option. We acknowledge the need for teachers and students to be able wo view there schedules on the internet, instead of through the program directly. **More effective algorithms**
Given more time other means of planning schedules would definatly have been researched indepth. The greedy algorithm that we currently use might not neccesarily be the most effective schedule planning algorithm, and we acknowledge that the potential for more success in other algorithms exists.
Manual corrections
One of the things that was originally suppose to have been implemented in the system, was the ability to make manual corrections to the created schedules. As currently is, the schedules now can only be created and not edited, this was something we would like to change, as we see the need to add, delete or edit in the created schedules. **Administration of users**

As of right now there is only 2 users in the system, the guest user and the aministrator user. If the system were to be expanded to cover more users, these needed personal accounts as well. At the moment such a feature does not exist in the system and we acknowledge the need for such.
Autosave of files
Right now the system will save data when typed, but these data will not be saved into a file unless chosen so. For various reasons we realize that having the program to autosave into a file will be neccesary when working with vast amount of data, it will also ease the work of the user, by saving data even in the case thatt something unexpected should happen.
Schedule printing
As of now the graphical user interface does not have an option to print a created schedule. This is an option that of course would be implemented. The option to print the schedules would help external sources with seeing when courses where planned for them.

7.7 The process

This section will handle the other aspects of the project. Namely the cooporation of the group members and how the work started with a few ideas, evovled with new tools and ended with a system. Different roles will be discussed here as well and how these affected the workload. Different means of organization which affected the process, either in a good or bad way will also be discussed. This section will end with how improvements could have possibly affected the process in order to make it more efficient.

7.7.1 The social aspect

The members of the group associated with this report had several advantages during the various stages of the report. One of the biggest advantages in the group was that almost all of the group members had coorporated on other reports before. This meant that the group-members already knew each other and knew each individual's strenghts and weaknesses. This also meant that certain aspects of the report was completed fast and efficiently because the group did not have to discuss who did what and who was good at some aspects and bad at others as this was already known. This definatly meant that the phases like for instance the implementation phase was structured and efficiently done. Besides the fact that the group members already knew each other, they also had a similar idea of how the system should look.
From the very beginning the group members shared many aspects which helped both in the analysis and especially in the design document. A good example of this is the GUI of the system. From the start all the members agreed on a dividing of the menu from the input screens. The concept of lists when managing data was also widely approven.
The social environment when working in the group has been mostly good. The group members work well together, either alone, in small groups or in the entire group. The social environment outside the workplace has also been really good, with group members doing various activities together.

Evaluating the social aspect

The group worked well in general. It was definatly a positive thing that the group members already knew eachothers strenghts and weaknesses. Whenever a problem arose the group was also fast to discuss how that problem should be handled and generally did not take long to decide on which road to take.

7.7.2 New tools

In this project alot of new tools have been introduced. These span from a new programming language, to new testing programs, not to mention the wide array of analysis and design tools there have been introduced. For the analysis part of the project the course *"System Analysis and Design"* taught us how an analysis document should be written. Another course *"Design and Implementation of User Interfaces"* was also introduced, though not used as much as *"System Analysis and Design"*. A standard was given and the group had to follow this. An example of a new tool that was taught could be the *FACTOR* criterion. This tool gave a quick way to find out how the system should act and followed a set of predefined categories.
Later on in the design phase a new standard was given for the design of the system. The course *"System Analysis and Design"* was once again a sturdy supplier of new methods and tools for developing a good design document. *"Design and Implementation of User Interfaces"* was now used alot more and provided tools both for testing and implementation of the system into a working program.

After the design phase had been completed the implementation phase started. In this phase most new knowledge was accumulated from the programming course *"Object Oriented Programing"*. A new coding language C# was also introduced and the implementation would be performed with this. Another tool related to C# is the visual studio debugger. This tool is used for debugging and has been invaluable in the testing phases of the project. Another new program that was learned was Doxygen. Doxygen is a tool designed for making automized program documentation and was widely used while documenting the program code. The program NUnit was used for testing the central units of the program. NUnit was used because it was very easy to use and test the different units implemented in the system.

Evaluating the new tools

Of all the new tools that were learned the C# programming language was definatly the most usable. Without it the program would have looked alot different. Programming documentation was also made much better with the Doxygen program. The new techniques and methods introduced through the courses were varied in value. While some were total eye-openers and defined key aspects of the system, others did perhaps not play an as important role as others, but their relevance for the project remained. It was especially in the early stages of the project methods like the FACTOR criterion quickly gave a set of ideas that quickly became concrete choices. In the design phase a set of demands were required to continue with the design. These demands, although they already had been touched in the analysis part, were crucial for the entire system and without the tools that were learned the group would have been stuck here.

7.7.3 The progress

As earlier stated this project has been divided into different parts. The allocated time for this project is around four months of work. Of course there have been numerous courses taking time from actual project work. Some courses did not have project work at all, while others had half lecturing, half project work. The different parts of the project are summarized into: The analysis part, the design part, the implementation part and the testing and evaluation part. While these different parts have been touched before, a more indepth explanation of them will be given. The project started out with the analysis of the system that was chosen, in this case the scheduling system. This phase was very dependant on the new tools, which were taught. Basiacally it meant that progress happened when there were course work.

The finalization of the analysis document took the better part of a month to complete. When the analysis phase was completed a review was held, where an outside party would look the analysis document over and give both positive and negative feedback.

The design document was started immediatly after the completion of the analysis document. This time eventhough there still was project work related to the different courses, there was more time to work on the project outside of courses. Eventhough the design document had about the same amount of time available as the analysis document, due to generally less courses in the semester, there was more time to work on the project than before. The design document deadline was also around a month and a second review was held after its completion where an outside party would give feedback.

The implementation phase lasted roughly three weeks and was started after the design document had been reviewed. There was of course alot of functions and bugs in the program at the time where the implementation phase was ended, but it had been chosen that there would only be allocated three weeks for the implementation and that the remainder of the time available was needed on the testing and evaluation. During the implementation phase the work was handled assignment wise, where either a single groupmember or small groups would take for instance a class or a function and write the code for this. During the entire implementation phase, comments and program documentation was created alongside the coding.

At this point there were two weeks left before the final deadline, where the was project needed to be done. During these two weeks the testing and evaluation took place. Obviously the testing was completed first, since the results of the tests would be crucial for the discussion in the evaluation. The testing process was conducted during one day with the help of the *"Instant Data Analysis"* method. The tests were performed with one test subject, the main user of the program. The tests were focused on how the user interacted with the programs Graphical User Interface. The test gave many new things to consider in the evaluation, both about what the user found good as well as bad, but also how some things potentially could be done diferently. After the testing was completed and documented, the evaluation began.

Each phase was discussed and evaluated. The main focus was on how each part of the report turned out and how that was in conjunction with what was envisioned. Improvements were also an important part as the time constraints did not allow the project to be completed. Therefore it was vital to discuss what parts of both the report and the program that needed improvement.

Evaluating the progress

The progress could easially be divided into two stages, the review part and the more free work part. While the first part covers both the analysis and design documents, the free work part covers all that happened after the design phase. In both the analysis and design phase a deadline was given, for the group to be done before this deadline, this resulted in they could only work on first the analysis document and later the design document. Because of these deadlines it was necessary to complete tasks one at a time. At the same time the courses only taught so many new things each time that if you wanted to do it faster, you would have to go further into the course material. This was of course both an annoyance as well as a bottleneck. But the progress was constant during the analysis and design phases. Both review deadlines were reached and the standards given out in the beginning were fulfilled.

In the later stages of the project, there was only one deadline, the day where the project was due. This denote that the group had alot more freedom to choose what they wanted to work with. This coupled with the fact that most of the courses were over, this entail that the group had alot more time to complete the tasks they would choose. This also emtail that the group could relax a bit more, which in return gave more energy in the implementation phase. While the free work part had more phases, those phases were worked on simultaniously and were more acting like one large phase. With the exception of the Student report, which was done mostly in the last two weeks of the project. The work progressed in general well and there was new parts of the report done each week.

7.7.4 Roles in the group

During this project, different roles for different group members have existed. While some of these roles were defined in the very early stages of the project, while others were made much later. These different roles will be described:

Project secretary

The project secretary's job was to write down whenever anything important happened in the group. Every time there was a meeting either internally in the group or with for instance the group's advisor, the secretary would write down any key notes worth mentioning for later reference. The secretary would also be responsible for any contact with advisors, test users and handles in general all communication to and from the group. The role of the secretary was given to one member of the group, but the role was given to another group member after about a month. This was done because the group at the time thought it best to switch the secretary role, so that more group members could experience that aspect of the group.

The second secretary stayed in that role for the remainder of the project, because the general satisfaction was high and none saw a reason to switch.

Project leader

At the beginning of the project a project leader was elected by voting. The project leaders role was first and foremost to supervise the project work. This included handing out different tasks for the other group members. The project leader also acted as the moderator whenever a meeting took place. It was also the project leaders task to make sure that group members did what they were supposed to do and met their deadlines and in general showed up on time. The reason why a project leader was elected in the first place was that earlier the group had had a democracy where every group member had a saying. Eventhough this had its advantages, the group wanted to know how it would work with a designated project leader. This leader was of course chosen by voting in the group. The project leaders tasks were tedious at times, because it was not the leaders task to babysit the rest of the group members or to hang above their heads constantly to be sure they worked.

Project vice-leader

The project vice-leader was created to make sure that if the project leader was not available, there would be a person to take his place, should the need arise. The vice-leader had the same tasks as the project leader, but was only active if the project leader was unavailable. The reason for creating this role was that the members wanted to make sure that there always was one group member in charge of the situation.

Programming leader

During the implementation phase, where the system was coded, a programming leader was elected. The programming leaders main task was to know exactly what classes and functions needed to be coded. Making sure that everyone always had something to work at, was also among the programming leaders tasks. The programming leader was created because the group agreed on that there was a need for a firm structure when coding. The programming leader had a good knowledge of the programming language and also acted as a helping hand whenever someone was stuck.

Evaluating the roles in the group

The different roles elected in this project have had varrying results. The secretary role started out shaky at best, but once a new group member gained the role of secretary, the role fulfilled its requirements to the fullest. Communication with outside entities like the advisor was very good and the summarizations of the various meetings where thorough and explanatory.

The programming leader was also mostly good. The idea of having one person that knew exactly what was needed and what classes to code etc. was very nice during the implementation phase. Keeping the other group members occupied at all times, was sometime a tedious task. For instance was some classes very hard to implement and the focus was not always high. At times the programming leader could help them, but sometimes he did not have time and the members had to find out what to do for themselves. The programming leader was overburdened alot of the time, which at times was stressfull. But without the programming leader the programming phase would have been alot more chaotic and progress would surely be affected by this.

The project leader role was intended to make the workflow go smoother in the group, as well as keeping track of the other group members. The role of the project leader was at first defined very loosely, meaning that the group was not entirely sure what the project leader should do. Therefore a sort of trial period was created, where the project leader could try out different ways of leading. After a while the leader was evaluated and given a concrete set of rules to follow. These rules were used throughout the remainder of the project. The leaders role was first and foremost to supervise the other group members and to maintain a certain level of seriousnes in the work environment. While the first of the two was done well in the earlier stages, and less well in the later stages. Supervising especially in the analysis and design phase was well done, and all group members either always had work or could ask for work. In the later stages though, due to the more open ended work form as well as the fact that there were less courses, made the entire group loosen up. Immediatly after the design phase, the programming leader took over for the duration of the implementation phase. Afterwards during the test and study report phases the project leader came back and took over again. Another important task that the project leader was assigned was to make sure that the group would either be on time for work, or know why they could not come on time, if they did. While the project leader almost always knew if someone would be late and why this were, the task of making sure that people were on time failed miserably. Through the entire project people would show up later than was agreed. This was worst in the more free work part, where there no courses were. A punishment system was even invented to make sure that people would show up, but it did not work. Some aspects of the project leader worked well, but others failed.

The vice-project leader role worked out well. The reason for making this role was to make sure that if the project leader was unable to perform his work, that there was one who could take his place. Due to the fact that the project leader was very active, the vice-leader did not get to act alot. But when the need arose, the vice-leader stepped up and performed well.

7.7.5 Improvements and what would have been different

This part will be about exactly what did not go so well in the process of this project.

One of the things that needed a vast improvement is the lack of being on time for group work. This had been a recurring factor in the entire project and eventhough different ways of dealing with it had been tried, none had succeded. This will deffinatly be a thing that will be looked serious upon in future projects.Ways to perhaps deal with this issue would be to make sure that the group members got up when they were supposed to instead of sleeping. The problem is that in the end it is the individual group members own responsibility to be on time and that not being on time hurts the entire group. So if for instance the group leader should have the task of making everyone arrive on time, then the responsibility would be shifted making it the project leaders responsibility. This would certainly mean that group members would show up on time, but it would strain the project leader.

Another thing that needed an improvement was the confusion around the different tools. At times when a new tool had been taught, the group had a hard time seeing how the tool should be used and what relevance it would have to the project. The general procedure when this happened was that the group would use the new tool regardless and perhaps not fully understand its functions. A way to improve this problem for future projects could be to have group members read older reports to get additional feedback.

The work moral in general was something that could also need some work. The amount of work that was done each day was very varying. This could of course have been a bit more static. In the early stages of the project, because there was alot of courses, there was not much time for actual project work. Serious work were conducted mostly during the different courses' work hours. In the later more free stages, with less courses, the work effort was varying at best. Sometimes nothing would be done in an entire work day. Sometimes alot would be done. Group members also tended to work at different times, such that perhaps half of the group was doing work while the other was not. When trying to do serious work it was of course hard to do so if the person next to you was relaxing with a game or other activities. There is no doubt that the project would have been done alot faster if there had been serious work from day one. One way to possibly improve the work moral was to make dedicated offdays, where the group members would not work at all, but relax and socialize. This would of course also mean that there would be dedicated work days, where there were work from start till end.

The project leader role also needed some improving. A certain agreed set of rules should have existed from the very beginning of the project. This would definatly be something that will be changed for future projects. The intention of the project leader was not that the role would act as a boss, but more of a representative from the group. It could appear that a more strict project leader could solve this problem as well as many of the earlier stated problems. That would also mean that the role of project leader had to be reevaluated.

7.8 Final thoughts of the process

This section will handle all the previous sections and try to compress all the thoughts and ideas described. This is done to give a general aspect of the entire process.

The overall result of the entire process has been mostly positive. A well written report as well as a system have been created, discussed, tested and evaluated. The group members have functioned well together and have each complimented the process with their own skills and abilities. Although obstacles were encountered during the project, these were discussed thoroughly and means to make sure they did not reappear were created. These means combined, though not always successful, into a set of rules for the group. These rules made for instance sure that group members would be on time for group work, respect the roles defined in the process etc. These roles functioned well in general, but some of the roles like the project leader needs improvement. During the process, alot of new methods and tools were acquired. Some very effective, others not so much. Some were straightforward and easy to learn while others were harder and took considerably more time. These tools were taught from the various courses that progressed during the semester. Without these courses no real progress would have existed, as they served the base as well as help for what was required to be in the different documents. The review deadlines that were present during this project worked well as they made sure that the group would get a serious start on the project. Eventhough the reviews themselves were not always a pleasant experience, having deadlines for both the analysis as well of the design documents was very good to have. All in all the progress was constant, every week new work had been done to the report, from start to finish. The group overcome most problems simply through discussion. Another positive thing was that during these discussion the group would not dwelve too long and would quickly choose one way rather than staying one place too long. The new tools helped tremendiously with the progress as well and the courses served as important teachers and helpers for the tools. The group roles were in general good, small mistakes were made, but nothing critical for the group work. The process had some recurring problems though, like group members sometimes lacking motivation to do serious work, as well as being on time. Although none of these problems were solved during the process, both were discussed thoroughly and different means to try and solve these problems were created.

Appendix 8

8.1 Exercises

This appendix contains the exercises carried out within the usability evaluation report in chapter 4.

Exercise 1
A new semester is about to start and you must therefore create a new schedule in the system. The semester starts at 1/8-2008 and ends at 31/1-2009.

- Create a new schedule with a starting date of 1/8-2008 and an ending date of 31/1-2009.

Exercise 2
One of the courses that is going to be taught in the new semester is Algorithms and Datastructures (AD). The course is normally attended by a large number of students and should therefore be held in an auditorium.

- Create a new course with the name "Algorithms and Datastructures" and acronym "AD".
- Change the preferred room type of AD to auditorium.

Exercise 3
New students are arriving in a week to start at the department of computer science. You must therefore create the new student sets to represent these people in the schedule system. Dat1 has 13 students and SW3 has 37 students.

- Create the new student sets for Datalogi1 (13 students) and Software3 (37 students)
- Assign both of the new student sets to attend the Algorithms and Datastructures (AD) course

Exercise 4
During the summer vacation a new building has been built to house the increasing number of new students at the institute. It has been named "cluster 6" and it has a 1st and 2nd floor like the existing building. In the new building there is also a new auditorium, the room number is 6.1.1 and it has the capacity of 96 students.

- Create the new room named 6.1.1 with a capacity of 96 and a type auditorium.

Exercise 5
In order to accommodate the increasing number of students, a new teacher has been hired to teach Algorithms and Datastructures. Her name is Magda Lene Kristensen and her username is MLK. Every wednesday Magda goes fishing and she is therefore not available to teach at these days.

- Create the new teacher named Magda Lene Kristensen and with username MLK.
- Assign her to teach Algorithms and Datastructures.
- Edit her availability to be unavailable (red) on wednesdays (both morning and afternoon).

Exercise 6
Throughout the semester there are meetings and other activities planned for wednesday afternoon, however it is possible to sometimes have courses here if everything else fails.

- Edit the general avaialbility for the schedule so that wednesday afternoons get a limited (yellow) availability.

Exercise 7
Christmas without any holidays is not fun, atleast december 24th, december 31st and january 1st (09) should be holidays.

- Add the three dates to the list of holidays for the schedule.

Exercise 8
(Turn off the computer screen prior to starting this exercise) Having in mind the tasks that you have just completed, please quickly run through (thinking aloud) the steps of which buttons you pressed and where you typed in information for each exercise.

8.2 Screenshots

This appendix contains the screenshots for the usability problems within the usability evaluation report in chapter 4.

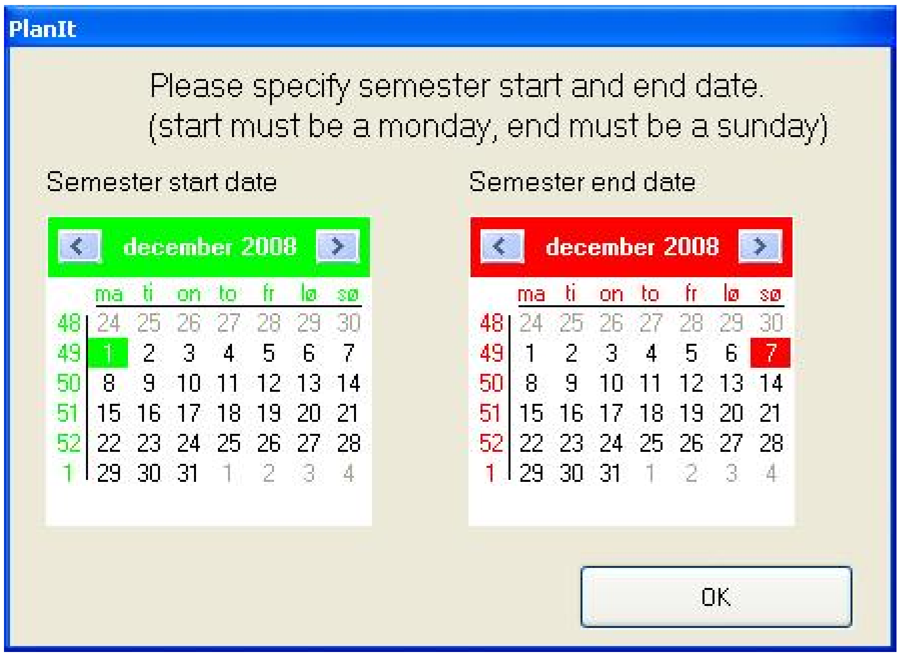
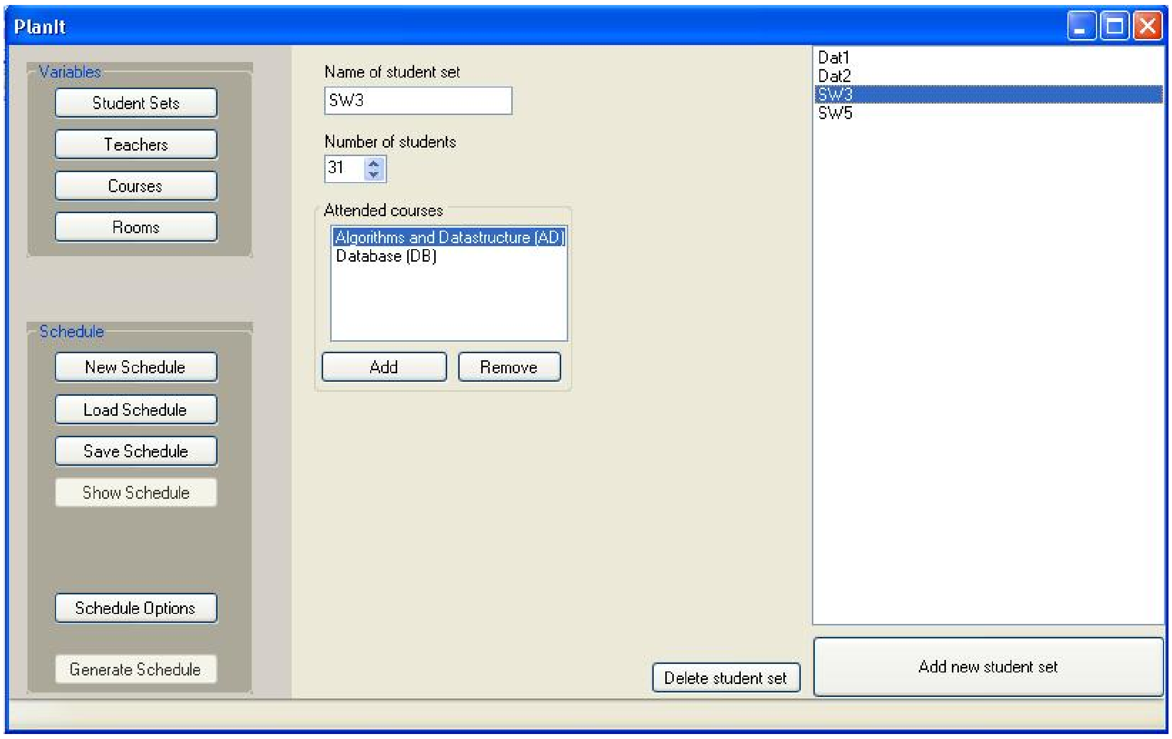

PlanIt
Please specify semester start and end date.
(start must be a monday, end must be a sunday)
Semester start date
Semester end date
december 2008
december 2008
OK

Figure 8.1: The *New Schedule* window. This is used when creating a new schedule to specify at which dates the schedule starts and ends.

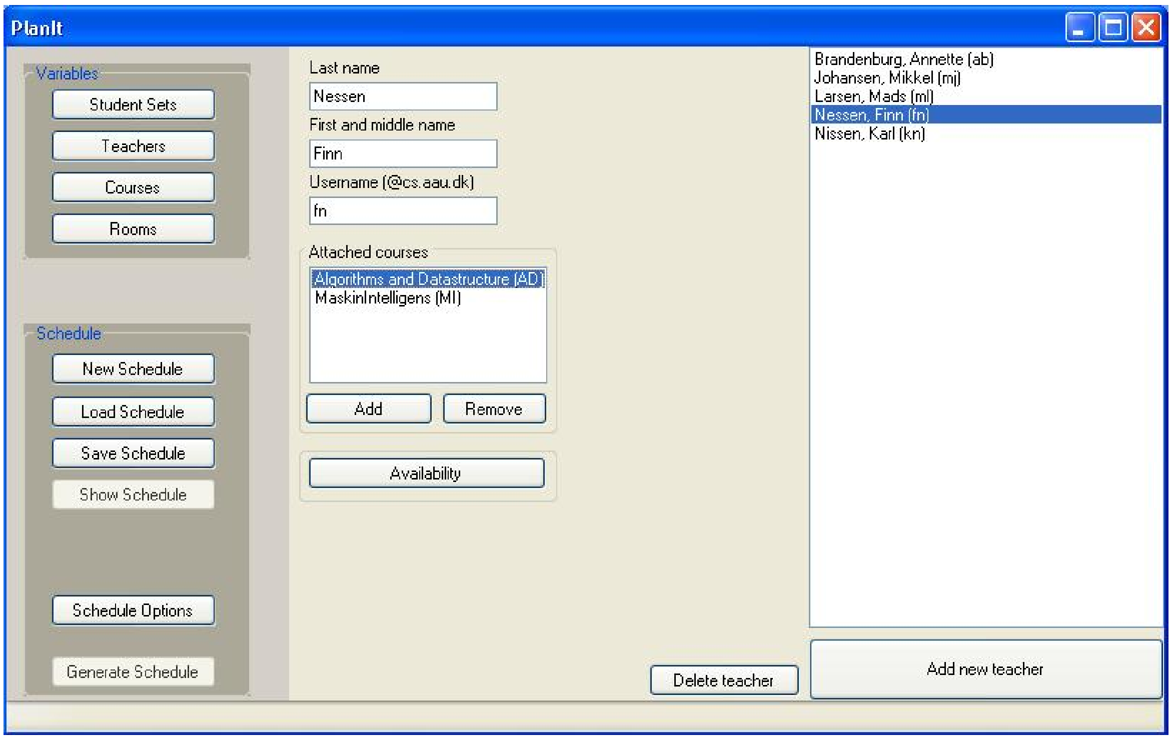


Name of student set
SW3

Number of students
31

Attended courses
Algorithms and Datastructure (AD)
Database (DB)

Figure 8.2: The *Student sets* window. This is used for creating and editting student sets.

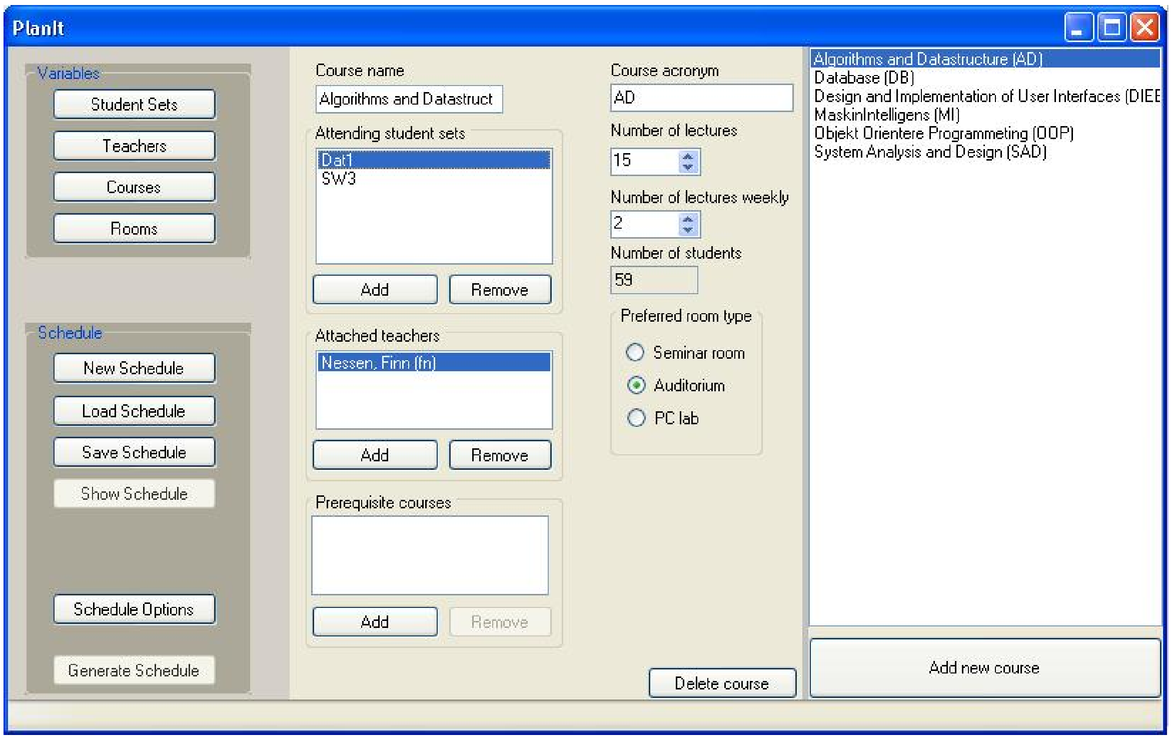


Last name: Nessen
First and middle name: Finn
Username (@cs.aau.dk): fn

Attached courses:
Algorithms and Datastructure (AD)
MaskinIntelligens (MI)

Brandenburg, Annette (ab)
Johansen, Mikkel (mj)
Larsen, Mads (ml)
Nessen, Finn (fn)
Nissen, Karl (kn)

Figure 8.3: The *Teachers* window. This is used for creating and editting teachers.



Course name: Algorithms and Datastruct
Course acronym: AD

Attending student sets:
Dat1
SW3

Number of lectures: 15
Number of lectures weekly: 2
Number of students: 59

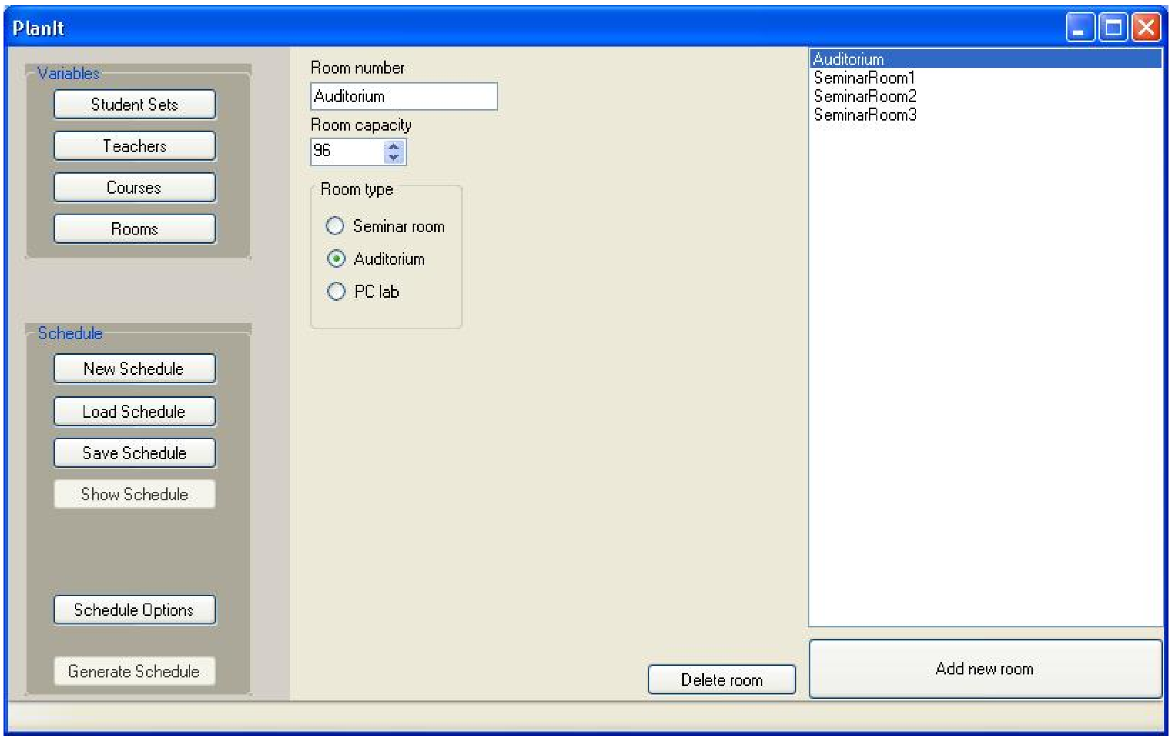
Attached teachers:
Nessen, Finn (fn)

Preferred room type: Seminar room [✓ Auditorium] PC lab

Prerequisite courses:

Algorithms and Datastructure (AD)
Database (DB)
Design and Implementation of User Interfaces (DIEE
MaskinIntelligens (MI)
Objekt Orientere Programmeting (OOP)
System Analysis and Design (SAD)

Figure 8.4: The *Courses* window. This is used for creating and editting courses.



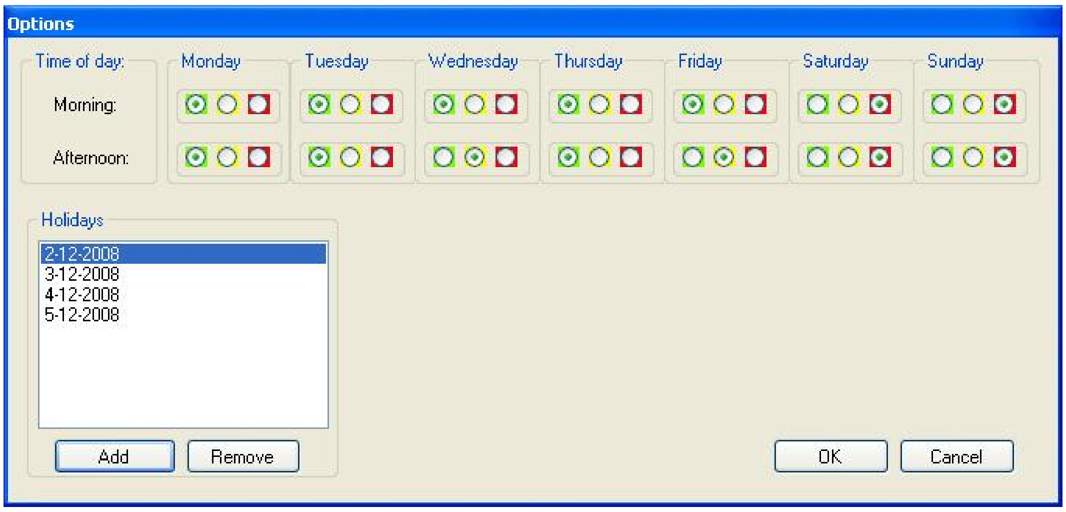
Room number: Auditorium

Room capacity: 96

Room type: ( ) Seminar room [✓ Auditorium] ( ) PC lab

Auditorium
SeminarRoom1
SeminarRoom2
SeminarRoom3

Figure 8.5: The *Rooms* window. This is used for creating and editting rooms.

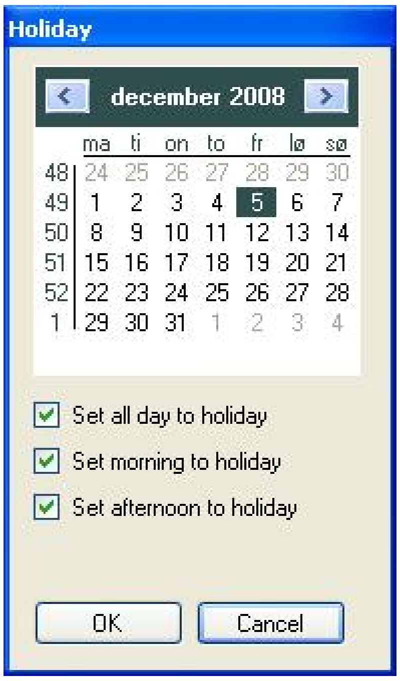


Time of day:	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday	Sunday
Morning:	[✓ 🟩] 🟨 🟥	[✓ 🟩] 🟨 🟥	[✓ 🟩] 🟨 🟥	[✓ 🟩] 🟨 🟥	[✓ 🟩] 🟨 🟥	🟩 🟨 [✓ 🟥]	🟩 🟨 [✓ 🟥]
Afternoon:	[✓ 🟩] 🟨 🟥	[✓ 🟩] 🟨 🟥	🟩 [✓ 🟨] 🟥	[✓ 🟩] 🟨 🟥	🟩 [✓ 🟨] 🟥	🟩 🟨 [✓ 🟥]	🟩 🟨 [✓ 🟥]

Holidays

2-12-2008
3-12-2008
4-12-2008
5-12-2008

Figure 8.6: The *Schedule Options* window. This is used for editting general availability of the semester schedule as well as a list of holidays. A window with the exact same layout is also used for editting availabilty for teachers



Holiday

december 2008

	ma	ti	on	to	fr	lø	sø
48	24	25	26	27	28	29	30
49	1	2	3	4	5	6	7
50	8	9	10	11	12	13	14
51	15	16	17	18	19	20	21
52	22	23	24	25	26	27	28
1	29	30	31	1	2	3	4

- [✓] Set all day to holiday
- [✓] Set morning to holiday
- [✓] Set afternoon to holiday

Figure 8.7: The *Holiday* window. This is used for adding a holiday to the list of holidays in the *Schedule Options* window. It is also used in the version used for teacher availability.