

MyCal

creating a calendar system



Fall 2003

SW3, Project group S301A



TITLE:

**MyCal:
Programming in the large**

SUBTITLE:

Creating a calendar system

PROJECT PERIOD:

SW3,
4. september - 19. december 2003

PROJECT GROUP:

S301A

GROUP MEMBERS:

Brian Jørgensen,
qte@cs.auc.dk

Eckhart Pedersen,
corner@cs.auc.dk

Kristian Bødker,
boedker@cs.auc.dk

Sutharsan Narayanasamy,
qmar@cs.auc.dk

Tinus Nortsved
tinus@cs.auc.dk

SUPERVISOR:

Laurynas Speicys
laurynas@cs.auc.dk

NUMBER OF COPIES: 8

NUMBER OF PAGES: 55

SYNOPSIS:

This report documents the requirements, design and implementation of a personal calendar system. The first part describes the requirement specification for the design and implementation. The design part covers design of the main software architecture and central parts of the system. The implementation part covers structure and details of selected parts of the system. Finally test of the system and description of the development process is presented. We found that the developed program meets the specified requirements and we succeeded in developing a large system compared to the time available to us and our previous experience with developing large programs.

Preface

This report is written by the group s301a, as a part of the SW3 semester, fall 2003 at Aalborg University. The outline for this semester is called "Programming in the large", and focuses on the processes involved in developing software systems of some considerable size. Spanning from different planning methods to actual development, configuration management and testing policies that can be applied during and after the development of the actual source code.

This report is documenting the development of the MyCal calendar system, a client/server based system, targeted at users needing a computerized calendar accessible from different locations. The actual program, and its source code can be found on the accompanying CD, together with a short description of how to install the server. At the end of the report there is a number of appendixes with additional information concerning some of the chapters.

The development of our program was split into two iterations and a prototype phase. The prototype was created to make the client server connection work, while the iteration development should make it possible to reevaluate our design. See the development process chapter for further information on this subject.

Throughout the report, source code, pseudo code and queries are written in monospaced font. If a code line is longer than the page allows, the break is indicated by a "\n" at the breakpoint.

References

Specific bibliographical references are of the following form: [3], and the actual entry is described in full in the Bibliography.

In the project we have used information from material and lectures from the semester courses SWA, OOP and SWP and the following sources:

- MySQL documentation webpage [10] - used primarily as a reference manual.
- Object-Oriented Analysis & Design [9] - used primarily during the development of the class and sequence diagrams.
- SUN's JavaDoc [13] - used throughout the programming phase to look up interface information to the used packages from sun.

Brian Jørgensen

Eckhart Pedersen

Kristian Bødker

Sutharsan Narayanasamy

Tinus Norstved

Contents

- 1 Introduction** **1**
- 1.1 Practical motivation 1
- 1.2 Academic motivation 1

- 2 Requirements** **3**
- 2.1 Business case 3
 - 2.1.1 Example: A student at a university 3
- 2.2 Requirement specification 4
 - 2.2.1 Single-user issues 4
 - 2.2.2 Multi-user issues 5
 - 2.2.3 GUI requirements 5

- 3 Design** **6**
- 3.1 Software architecture 6
 - 3.1.1 Server 7
 - 3.1.2 Client 8
- 3.2 Database 10
 - 3.2.1 Appointments 12
 - 3.2.2 Users 12
 - 3.2.3 Calendars 13
 - 3.2.4 Task list 13
- 3.3 Protocol 14
 - 3.3.1 XML data 14
 - 3.3.2 Error messages 14
 - 3.3.3 Login procedure 14
 - 3.3.4 Retrieving and manipulating appointments 15
 - 3.3.5 Change in the appointment handling 17
 - 3.3.6 Retrieving and manipulating tasks 18
 - 3.3.7 Retrieving and manipulating categories 19
 - 3.3.8 Manipulating user information 19
 - 3.3.9 Managing groups and administrators 20
- 3.4 Meeting wizard 21

4	Implementation	22
4.1	Server	22
4.1.1	Database implementation	25
4.1.2	Administration system	27
4.1.3	Server configuration	28
4.1.4	Server logging	28
4.2	Client	29
4.2.1	Login dialog	29
4.2.2	Main window	30
4.2.3	Meeting wizard	33
4.2.4	Importer	36
4.2.5	Exporters	37
4.2.6	User Interface	38
5	Testing	40
5.1	Ad hoc testing	40
5.2	Unit testing using JUnit on central classes	41
5.3	Acceptance testing	42
6	Development process	44
6.1	Development cycle	44
6.1.1	Planning	44
6.1.2	Development phase	45
6.2	Development tools	46
6.2.1	Configuration management	46
6.2.2	Testing	47
6.2.3	Code documentation	47
6.3	Reflections	47
7	Future work	49
7.1	Security	49
7.1.1	Improving security with SSL	49
7.1.2	SQL Injection	50
7.1.3	DoS attacks	51
7.2	Reminders	51
7.3	Recursive events	52
7.4	Optimization	52
7.4.1	Server optimization	52
7.4.2	Scalability	53
8	Conclusion	54
A	ER-model	56
B	Database tables	57

C Code review	60
D Protocol Test	62
E User guide / manual	63
F Class diagram	64

Chapter 1

Introduction

During the start of this semester we (as a group) were required to choose a topic for our "programming the large" project. We had some initial ideas on which types of project we would be able to make within the given timeframe and at the same time would be complex enough to match the requirements specified in the semester description. We ended up choosing to develop a calendar system. Our choice of project topic is due to a mixture of both practical and academic motivations.

1.1 Practical motivation

One of the reasons behind our choice of project was that we with a calendar system had the chance of making a system that is widely useable by many different people.

Calendar software in general can be utilized for keeping track of different appointments, and if a centralized approach is selected, this calendar information can be accessed from many different locations. The fact that the information is stored and processed by computers implies that there is possibility to represent the information in different manners.

1.2 Academic motivation

Another reason for our choice was that creating a calendar system allowed us to use many of the technologies covered by the courses on the SW3 semester. We especially wanted to have a project in which the database was a central part and not just a simple storage system, in the same manner as a standard file system. A calendar system also seemed to have an easily defined basic functionality and we would be able to focus more on making the actual system instead of having to figure out what functionality to build in. Furthermore a centralized calendar system allowed us to work with some interesting programming issues, such as:

- Network
- Threads
- GUI

At the start of the project none of the group members had more than superficial experiences using Java. Since we were required to use an object oriented programming language, and we had a course on using the Java programming language, it was practically implied that we should use Java for our project. Java does, however suit our purpose well, due to the portability of Java applications, and it would be a logical choice even without the previous reasoning. Futhermore Java provides us with a good foundation for working with the previous described programming issues through its large standard library.

Chapter 2

Requirements

2.1 Business case

This section will discuss our business-case, explaining the intended use of our calendar system. We will briefly describe our target group and what situations the calendar system should be used in. Thereafter we will give a concrete example for the use of our system. The business case will be used as a foundation for the requirement specification in the next section.

Our calendar system was supposed to be used as a personal calendar, which could be used to keep track of appointments. We wanted to include some functionality to share event information with other users, so it could be used to schedule events for groups of people. This functionality was only to support the scheduling of simple appointments and events, and not the possibly more complex needs when scheduling appointments in a company and where room and equipment accessibility has to be taken into account.

The rest of this section will be an example of the possible situations that our calendars system could be used.

2.1.1 Example: A student at a university

The user in our example is a student at a university, who uses the MyCal system to keep track of his personal appointments, e.g., exams, dentist appointments, family parties etc. These personal appointments are private, and the user doesn't want other users to be able to see them. The user accomplishes this by setting the visibility attribute of these appointments to "private". The appointments should be able to contain general information, like location, start time, end time, category and others.

University group

The user is a member of a study group with five persons and this group also uses the MyCal system to manage group events and meetings. The group creates a calendar as a new user with the username "MyUniGroup". Every person in this group is granted administrator rights on the group calendar, so that they all can view, edit and create events in the calendar.

Appointments that only apply to the university group are marked as "private" so that only the members of the university group, can view them (due to their administrator rights).

Appointments concerning external contacts

In some cases the university group may want to create appointments, that are visible to some specific persons outside the university group. This can be done by setting their visibility attribute to "group" and granting group rights to the external contacts. Yet again some other events may be public and thereby visible to every MyCal user on the same server who has chosen to merge with the MyGroup calendar.

Frisbee club

The example user is also a member of a Frisbee club, which has meetings each Tuesday and some Thursdays. In order to inform the members of club meetings, the club uses the MyCal system. The club has created a user called "FrisbeeClub", and in this calendar the club officials (with administrator rights) create appointments on the days where the club has meetings. The appointments are marked as "public". Anyone interested in playing Frisbee can merge with the "FrisbeeClub" calendar and see which days the club has meetings.

Tasks

Some entries in the calendar can't fit into the event category, e.g., if a user wants to have an entry with a due date and the possibility to mark the entry as completed. The system should allow users to have a list of tasks with entries that could or could not have specific due date. Tasks are only accessible to the user owning them.

2.2 Requirement specification

The calendar system is primarily meant as a private calendar, that the user can access from different physical machines. It also contains multi user functions, such as merging with other users calendars, and creation of groups that can share a calendar. This functionality requires a client-server structure, to allow multiple users to access the same information from different locations.

2.2.1 Single-user issues

The user should use the client to connect to the server via a network (or using localhost as the server address), and then login with his username and password. When authorized the user must be able to manipulate calendar events (title, description, location etc.) and tasks using the client GUI. These changes must be saved on the server, so that the information is accessible from other machines. The calendar information is to be represented to the user in a visually pleasing manner and allow the user to change between different main views with different information detail levels. The user should be able to assign a category to an event, and this should allow the user to visually distinguish between events of different categories.

2.2.2 Multi-user issues

For the calendar system to be useful for groups of people, it must be possible for the client to access other calendars and display their contents in the clients GUI. This process will from now on be referred to as “merging”. In order not to reveal sensitive information, a user must grant other users privileges, enabling them to view different types of events. Three levels of visibility are needed, called “private”, “group” and “public”. This means that merging with a calendar will not show “private” events, if the user isn’t an administrator for that calendar. Likewise, merging will not reveal “group” events if the user isn’t either an administrator or a member of the group in the calendar. Only public events may be shown to all users when merging.

In order to change data in a calendar, the user must have administrator privileges for the calendar. The management of privileges has to be decentralized, that is it should not be necessary to contact the system-administrator in order to change access privileges for a user. In practice, this is accomplished by letting the calendar-administrator(s) manage access privileges.

2.2.3 GUI requirements

To present the information in a user friendly fashion, we specified some requirements for the design of the client GUI. These requirements included the following visual components:

- The main viewer - showing either events for a day, week, month or year
- A list with all future events
- A menubar that provides centralized access to the most important functionality
- A toolbar that provides quick access to important functionality and navigation
- A small navigation panel that allows for quick navigation to specific dates or in a specific view.
- A task list that provides an overview of tasks
- A small selector to select which calendars to merge with.

These components should always be accessible, unless a dialog is shown. The main viewers should have different level of details when showing events, and the day and week view should be able to display multiple events that take place simultaneous.

There were further requirements concerning the use dialogs for settings, meeting wizard and other functionality, but these are too trivial to detail here.

Chapter 3

Design

Prior to writing actual code, it is necessary to specify what the desired result should be like. This design process is important, as it is the basis for the code development and thus determines what the program will be like. The functionality is not the only part of the program that can be designed, often it is desirable to design the code structure itself too.

This chapter will detail the final design of our program structure, while keeping the focus on the more interesting and central parts. This should make it easier to understand the ideas behind the actual implementation of the code, and give a better understanding of the system structure.

3.1 Software architecture

Before describing the smaller parts of the system, we need to have an overview of how the total system is going to work; what parts it consists of and how they communicate with each other. On the highest level our system consists of a server program that communicates with multiple clients. The clients could be running different client programs, eg. a web client or, as we are going to use, a Java GUI-based client. The general structure of our system is shown in Figure 3.1.

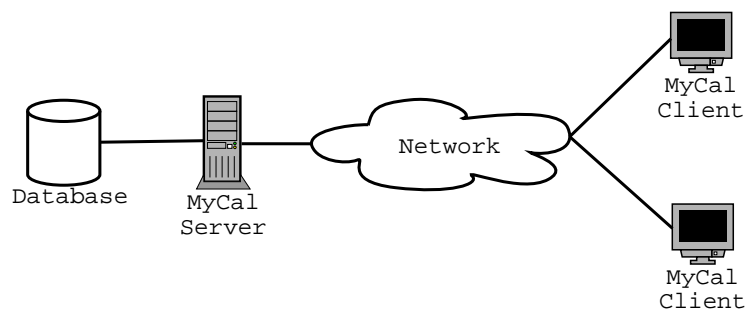


Figure 3.1: Overview of software architecture

From this overview it becomes clear that the system can be split into 3 parts; server, client and database - since most of the data management can be handled directly in the database.

3.1.1 Server

The central part of our system is the server, allowing clients to work on data in the database. Other client programs could communicate with the server as long as they communicate in a manner the server understands. The server could also exchange its storage system from the selected with another DBMS or a file based storage solution.

To support this modular architecture of the server, it is split up into different layers, each taking care of its specific task. These layers are however only theoretical, since we aren't going to create a complex communication protocol between the different layers, so exchanging a part would require changing some code in other parts. (Figure 3.2)

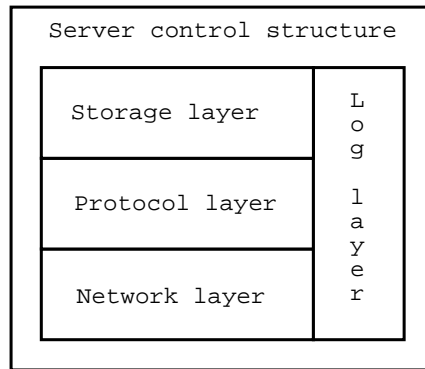


Figure 3.2: Server consists of four different layers

The server accepts input at the network layer level, which is responsible for handling connections to clients, receiving and sending data. Exchanging this layer could add compression and encryption of all data sent from server.

When the network layer has opened a connection to a client and has received input, it sends this input to the next layer, the protocol layer which determines what actions should be performed given the input. These actions could be to authorize a user against a list of valid users or to list events in a database. The protocol layer is the only one to communicate with the next layer - the storage layer.

The storage layer handles all manipulation of data in the storage. It could be exchanged, as described, with a different form of storage than a database. Instead of using the API provided by the storage system, we have chosen to create this wrapper layer, not only to make exchange of the storage system simpler, but especially to simplify the methods supplied by the API.

The last layer handles logging and is used by all layers, directly or indirectly. Each layer can log actions e.g. queries performed on the database, to help debugging or track down performance problems, command execution on the protocol layer and traffic statistics on the network layer. Again it could be exchanged with other logging methods, e.g. to database, off-site storage, informing administrators by e-mail on certain errors.

Sequence diagram

The server part of the program can also be described by Figure 3.3, which is a sequence diagram of how the classes call each other.

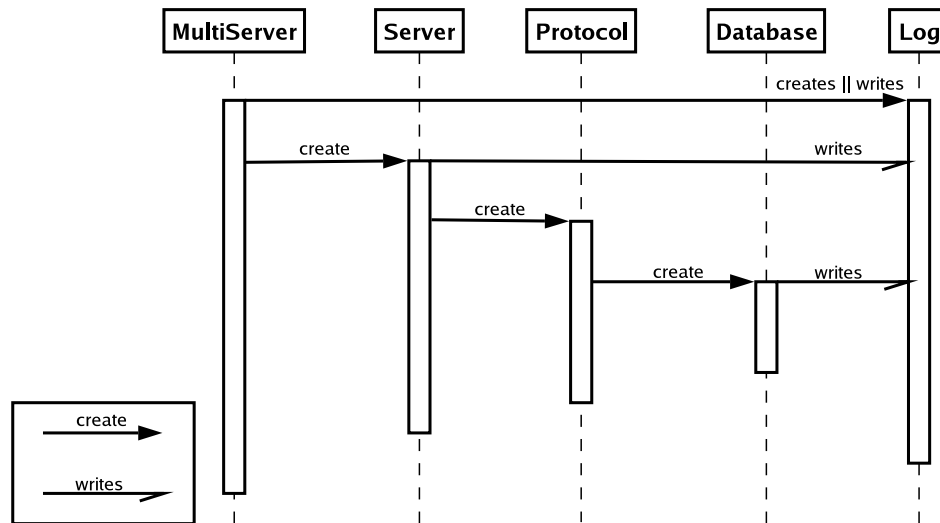


Figure 3.3: Server sequence diagram

3.1.2 Client

In the same manner as with the server architecture, the client can be split up into smaller layers. (Figure 3.4)

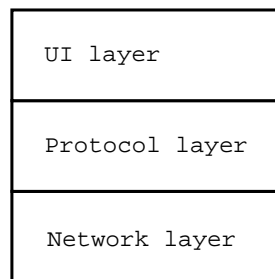


Figure 3.4: Client consists of three different layers

When the client program starts, the user will be presented with the user interface (UI) layer. This layer will decide what actions to take, based on the input of the user. The actions could be "show list of appointments from the current month", which the UI layer then translates into "ask protocol to receive list of appointments from server, show list of appointments".

The protocol layers translates the requests from the UI layer into protocol commands and then sends it to the server through the next layer, the network layer. The protocol layer is also responsible for translating error messages returned from the server into messages for the

UI layer. In the following sections we will go through the in depth design of the database, protocol, meeting wizard and class diagram of the program.

3.2 Database

In the following chapter we will describe the design of the database of our system and some of the reasoning behind this design. We will start with the requirements we had for the database, and how these ideas and expectations led to the actual running database. A view of the entire ER-model can be found in Appendix A.

In several places throughout the database an id field is being used as the primary key to an entity. This is done in order to have an easily distinguishable handle to an entity e.g. a category, so that several users can have entities with the same name without creating problems when one user chooses to remove or change his instance of this entity.

Our database design primarily focuses on the following items

- Several users can have several appointments.
- Users can view events that other users own.
- Several users can own the same appointments.
- Users should be able to choose whether other people should be able to view certain appointments.

Secondary focus is on

- Users can have private tasks.

We start with a user and an appointment and model the relationship between these two entities. Several users can have several appointments, and several appointments can be owned by several users. (Figure 3.5)

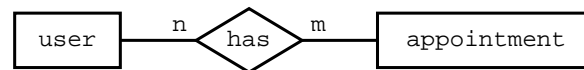


Figure 3.5: User / appointment relationship

Instead of mapping several users to one appointment, we choose to let a calendar own the appointment and then allow users to be administrators of the given calendar. Each user can have his own calendar for personal appointments and at the same time administrate a calendar with group appointments. This results in the following ER-model. (Figure 3.6)

This only allows users to view appointments, which they indirectly own through calendars where they are administrators. To allow users to view appointments from other users calendars and even at different levels, we add a groupmembership relationship between user and calendar and a views relationship between user and calendar as well. (Figure 3.7)

The relationship between users and appointments is now finished, and we add attributes to the ER-model. This is discussed in the following section.

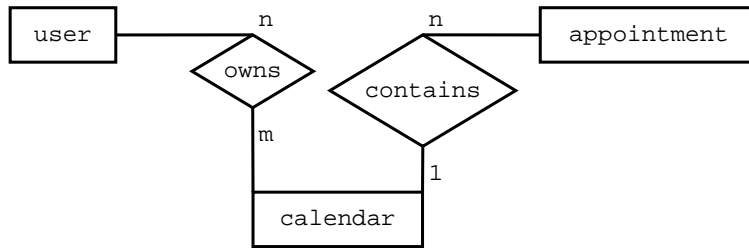


Figure 3.6: User / calendar and calendar / appointment relationship

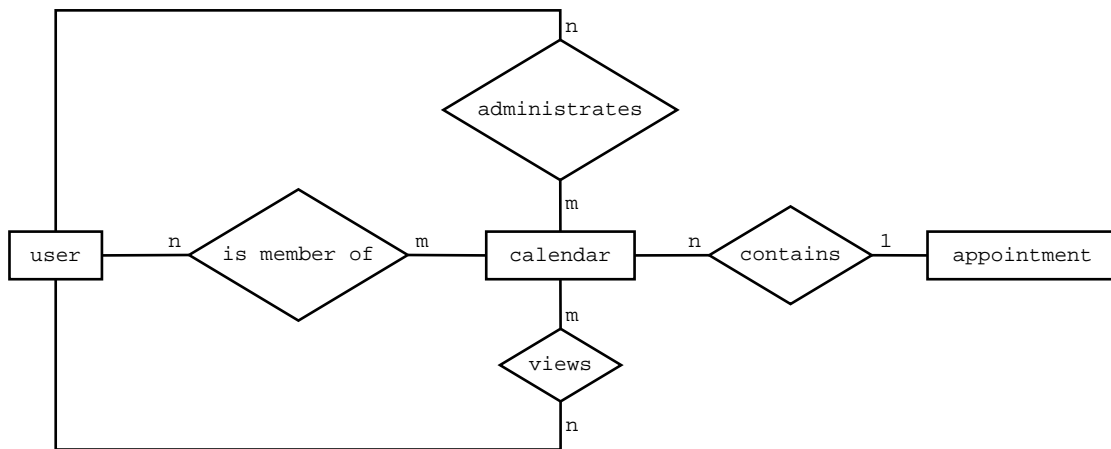


Figure 3.7: Relationship between users that can view and administrate calendars.

3.2.1 Appointments

The first set of attributes we add to the appointment entity, in order to support the concept of having several users to view appointments depending on whether they are administrators, members of the group or just outside users, a visibility attribute describing whether the appointment is public (all users can see it), group appointment (members and administrators can see it) and finally private (administrators only).

To identify the appointment uniquely we add an ID as key.

- **id** - this is introduced in order to have a unique handle for each appointment.
- **title** - each appointment can have a describing title, for easier recognition.
- **start time** and **end time** - to display the appointment correctly in the view of a calendar - each appointment has a start and end time describing what date and time of day it occurs.
- **location** - description of where the appointment takes place.
- **note** - user specified notes for appointment, could include what to bring, how to get to the location, who is going to attend etc.
- **allday** - boolean flag describing whether the appointment is an allday appointment. Examples of allday appointment could be birthdays and holidays.
- **busy** - appointments can be transparent and thus help a user to decide whether to disregard an appointment and participate at another. This could be a conference with business meetings in the afternoon.
- **calendar** - name of the calendar to which the appointment belongs - used to select all appointments in one calendar.
- **category** - instead of giving each user 20 calendars for each type of appointments the user might want to use, we enable the user to specify a category and thus group appointments into different groups.
- **visibility** - a field used to set the visibility. The possible values are `private`, `group` and `public`, where `private` is default.

Instead of having the category as an attribute we place it in the ER-diagram as an entity with an “is in” relationship to the appointment as shown in Figure 3.7. This allows the user to choose from a list of already used categories without having to create category information for each appointment.

3.2.2 Users

- **username** - used to uniquely identify each user.
- **password** - used to authorize the users.

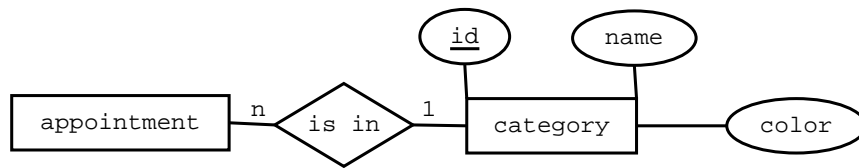


Figure 3.8: Relationship between appointment and category.

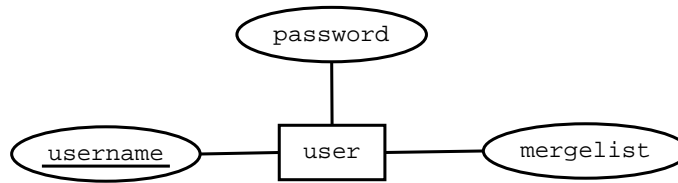


Figure 3.9: Relationship between users that can view and administrate calendars.

- **mergelist** - a list of calendars from which the user wants to be able to merge appointments into his own calendar, and whether the individual calendars are currently merged or not.

3.2.3 Calendars

- **name** - unique identifier for calendar.
- **description** - when a user chooses to merge with a calendar, he can get a long description of what the calendar contains, e.g., “TV-Guide for the Simpsons and Southpark.” instead of just the calendar name “tvguide”.

3.2.4 Task list

Users could place tasks as appointments in the calendar, but this would make it necessary to browse through their calendar to find all tasks. Using tasks as appointment entries gives problems with specifying status of tasks (whether they are completed) and where tasks should be placed if they don’t have a due date. We have chosen to separate tasks from appointments and add extra to and remove not used attributes from it.

- **id** - unique identifier.
- **username** - owner of task.
- **title** - short title with task summary.
- **note** - description of what needs to be done for the given task to be completed.
- **done** - when a task is completed, the user can flag it as completed and let the client program display it as striked out or perhaps not show it at all.
- **due date** - users can specify a date where a task is supposed to be completed.

3.3 Protocol

The protocol used for communication between the server and clients is a very central part of our program, and thus this section is going to explain what commands a client can send to the server and what responses it can be given.

3.3.1 XML data

With our third iteration of the protocol, we have chosen to transfer appointment data, our most complex data structure, as a XML string, which the client can parse. In a future iteration all communication of data should be done using XML. This will minimize the need to restrict users from having certain characters, that have been used as argument separator for protocol commands, in their appointment titles, descriptions etc. Ideally almost all of the data should be send as XML, but since this involves a lot of trivial implementation we have chosen to prioritize other aspects of the system higher, however this should most certainly be a part of a future iteration

3.3.2 Error messages

The protocol must return error messages if something goes wrong, so that the client can react appropriately to the given error, e.g. by showing an dialog with the error to the user.

Permission denied..

This error message is returned if something failed due to privilege problems.

Unknown error..

If a command fails to perform its task and none of the other error messages match - e.g. it fails determining if the user has administrator privileges due to problems querying the database, the protocol returns this error message.

Apart from returning these error messages, the protocol commands can return messages informing the client that a command succeeded and what effect it had - e.g. how many rows were deleted.

3.3.3 Login procedure

login

Our first requirement for the protocol is that clients shouldn't be allowed to run commands on the server without being authorized as users. To handle this, the protocol can be in different states.

- **WAITING** - When a client opens a connection to the server, the protocol instance goes into this state. When the protocol is asked for an output, it responds with the server name and version, to let the client know which server it is communicating with.

- **AUTHORIZE** - When the client has received the server version information, the protocol goes into the AUTHORIZE state, where the only valid input is login - if this login fails the protocol terminates the connection.
- **LOGGEDIN** - If the client authorizes correctly, the protocol goes into its main state that handles manipulation and retrieval of data.
- **EXIT** - When the connection to a client is going to be closed the protocol changes to its final state EXIT. The server that communicates with the client can then close the connection when the protocol instance reaches the EXIT state.

MyCal Server version 1.0

User: john|secretpassword

Figure 3.10: Server response is in bold text.

If the authorization succeeds, a success message is returned.

User authorized

If the authorization fails, the protocol returns an error message, where after the connection is closed.

User authorization failed.

logout

If a client wants to close the connection it can send a "Bye." command, which makes the server change state to EXIT, and the connection is closed.

3.3.4 Retrieving and manipulating appointments

After having enabled clients to log in and out of the system, the next phase is to allow users to manipulate data on, and retrieve data from server. In our first iteration the following five commands were added to the protocol.

addEvent()

When a user wants to add an appointment to the database, the protocol sends all the attributes to the server separated by pipes (|). The protocol splits up the input string at every pipe and uses the attributes again to insert the appointment entity into the database.

Before inserting the appointment into the database, the protocol must ensure that the user has privileges to add and edit appointments in the calendar specified as target for

insertion, so that users without the proper privileges can't add and edit appointments in calendars they aren't supposed to.

The `addEvent()` command of the protocol has the following syntax:

```
addEvent: id|calendar|title|start time|end time|visibility|allday|busy \  
         |location|category id
```

And an example of the command in use could be:

```
addEvent: 0|TV-Programs|Simpsons: Christmas special|1072202400|1072206000 \  
         |public|false|false|TV room|3
```

The sent id attribute is ignored, since it is assigned a new id automatically by the database, it is left in to keep the same syntax between `addEvent()` and `updateEvent()`

updateEvent()

This command updates appointments that already exist in the database and it differs mostly from `addEvent()` by having to check that the user both has administrator privileges to the source and target calendar. Otherwise the user could either move appointments from or insert appointments into calendars where it shouldn't be possible.

deleteEvent()

This command deletes an event identified by an id. The deletion is only performed if the user has the correct privileges, and this is checked in the same manner as with `addEvent()`.

The syntax of the command is very simple:

```
deleteEvent: id
```

And an example could be the following, requesting the appointment with the id, 3:

```
deleteEvent: 3
```

listEvents()

the `ListEvents()` command is used by the client to receive all events for a specified time interval from a list of calendars. The time interval is specified with two timestamps, start and end time, and the calendars are given by a list of calendars seperated by ":".

The resulting appointments are send back to the client as a XML-document containing nested event-entities in an eventlist container entity. The structure of such a result is shown on fig 3.11 and the syntax of the command has the following form:

```
ListEvents: start time|end time|calendar1:calendar2:...
```

```

<?xml version="1.0" encoding="iso-8859-1"?>
<eventlist amount="8"
  <event></event>
  <event id="1" title="Simpsons Episode 123" allday="false" busy="true">
    <calendar value="TV-Programs" />
    <description>Remember to bring:<br />- Popcorn<br />- Beer</description>
    <starttime value="1070215200000" />
    <endtime value="1070218800000" />
    <visibility value="public" />
    <location value="TV Room" />
    <categoryid value="1" />
  </event>
  <event>
    ...
  </event>
</eventlist>

```

Figure 3.11: An example of a result from the ListEvents() command

3.3.5 Change in the appointment handling

In our first two iterations the `listEvents()` method returned only ids and titles, since we expected the client to get appointment data using a specific `getEvent()` method to get the complete dataset for each. This however turned out to be very inefficient in situations where we needed the client to get a lot of events e.g. when displaying all appointments from a month or a year.

Instead we chose to send all of the data when the client requests the list of appointments. This, although a list with 1000 appointments would typically has a size of 3-400 kilobytes, is still faster than the old method involving the `getEvent()` command. This is due to the fact that many `getEvent()` calls would produce higher database load (more individual queries), and overhead from sending protocol commands.

3.3.6 Retrieving and manipulating tasks

In agreement with our business case, tasks are binded directly to one user, and can't be accessed for read or write by others. This makes the task of managing privileges easier for the following commands, since they can rely solely on the username of the active user and don't have to look up calendar and usernames sent from the client.

addTasks()

This method creates a new task in the database with the current user as owner. Since there is no possibility of creating tasks other than with the current user as owner, there is no need to check privileges other than the initial authorization of the user (username and password).

The syntax of the protocol command is:

```
addTask: id|title|note|done|duedate
```

updateTask() and deleteTask()

These commands respectively delete and update a task. Before this is done the protocol must check that the current user is allowed to manipulate the particular task, by verifying that the user owns the task with the specified id.

The syntax of `updateTask()` is similar to the `addTask()` method, and `deleteTask()` only needs an id.

cleanTaskList()

The `cleanTaskList()` command removes all tasks in the database that have been marked as completed. The same functionality could be achieved by letting the client loop through the task list and delete every task that is marked as completed using the `delTask()` command, but this has shown not to be as efficient as the current method much due to the increase in database load and protocol activity.

listTasks()

When listing tasks, the protocol can skip all tasks that don't belong to the current user. As a help for the client, this command must sort the list of tasks in the following order:

1. Tasks with a due date are listed first, where tasks with earlier due dates are on top
2. Tasks without a due date are placed in an unsorted order after the other tasks.

The list consists of a pipe seperated id string, which the user can use to retrieve data for each task with the `getTask()` command. In a future iteration this command should output complete data set for all the tasks, preferable as an XML-string, instead of just a list of ids.

getTask()

This command requests a task from the server. The task is identified by an id, and the server returns the task as a pipe separated string. This command is used combination with `ListTasks()` in order to show the tasks list to a user. The client must first utilize the `listTasks()` method to get a list of ids, and then call the `getTask()` method with each task id, and thus retrieving all the task information for each specific task.

3.3.7 Retrieving and manipulating categories

addCategory()

This command adds a category to the database, as a category within the specified calendar. When the user wants to add a new category, a call to the `addCategory()` method with the calendar name, the category title and the rgb color values as arguments is made. The data is sent as a pipe separated string. The server-side protocol then has to check whether the user is allowed to add a category to the specified calendar. The protocol returns "Category added.." if successful.

update- and deleteCategory()

These commands respectively update and delete a category specified by id. before the deletion or update is executed, the server-side protocol has to check whether the user is allowed to make the changes. If the user has the right privileges the change is brought to effect, and the protocol returns respectively "Category updated.." or "Category deleted.."

3.3.8 Manipulating user information

At some point, the user will want to change and save information stored about his or her profile on the server. We supply 3 commands for this, one for changing the password used for authentication and two for saving and retrieving the list of calendars to merge with when using `listEvents`.

changePassword()

The first command we added in the second iteration was the `changePassword()` command. Since we already know the user will be logged in when calling this command, we don't need to ask for the old password, which usually is the case when changing passwords. The protocol returns "Changed password.." if the change succeeded.

getMergeList() and saveMergeList()

The `listEvents()` command takes a list of calendars to include appointments from in the list of appointments it returns. This list of people to merge with is specific for one user, so the list is stored on the server and the client must be able to get the list using the protocol.

The protocol must return a list of which calendars the user has chosen to place on the merge list and which of these calendars actually should be merged. The syntax for this command is similar to the syntax of the returned data of `saveMergeList()`.

The syntax of `saveMergeList()` is as follows:

```
saveMergeList: calendar1(:true)|calendar2(:true)|calendar3(:true)|...
```

The parts inside parentheses are optional, and the ... Illustrates that any number of arguments is possible. Each calendar in the merge list is separated by a pipe, and the calendars that are selected to be merged are appended with a ":true".

3.3.9 Managing groups and administrators

`listAdmins()`

To check which users are administrators for a calendar, the client can make a call to the `listAdmins()` method with the calendar as argument. In order not to reveal unnecessary information about other users, the method returns `null` if it was called by a user, who is not an administrator for the calendar himself. Otherwise the protocol returns a string list containing the usernames of the administrators.

`listUsers()`

This command returns a list of all users on the server, and this is primarily used by the client when a user wants to delegate privileges to other users.

`listMembers()`

The `listMembers()` command returns a list of users that have group-privileges on a specific calendar. In order not to reveal unnecessary information about other users, the method returns `null` if it was called by a user, who is not an administrator for the calendar himself. Otherwise the protocol returns a string list containing the group members usernames.

`add- and delete Privileges()`

these commands is used to add and delete privileges. The syntax of the two are the same and the commands are complementary. The syntax is as follows:

```
addPrivileges: calendar|user|privilege
```

In which `calendar` is the calendar to grant access to, and `privilege` is one of the two privileges "group", "admin". When an administrator wants to grant other users rights in a calendar, the client makes a call to the `addPrivileges()` method, specifying the calendar where the user should have the rights, the username for the user and the desired access level. The protocol has to ensure that the user issuing the command actually is an administrator for the specified calendar. The protocol returns "true" if successful, and otherwise "false".

3.4 Meeting wizard

Early in the project we decided to create a feature, which would allow a user to arrange a meeting with other users of our system, by searching the calendars of the participants for available time-slots. When designing the meeting wizard, we specified a set of specific constraints that the user should specify for the wizard to find the wanted time-slots. This implied specifying on what days of the week, dates, time of day and how long the meeting should be. Thus the requirements are as follows:

- **Search in multiple calendars**

The wizard should be able to search for time-slots in multiple calendars, which are specified by the user. The initial idea was that the wizard should send a message to the involved calendars, which should notify the other users of the meeting request. This idea was dropped however, since we didn't want the wizard to be able to create events in other users calendar, without their permission.

- **Search for all busy events**

The wizard should use all events in the specified calendars that are marked as busy, to find available time-slots. Since all events are marked "busy" per default, it should be safe to assume that events specifically not set to busy are not important for the search.

- **Search for time intervals within a specific time of day**

The user should be able to specify what time of day the arranged meeting should take place. The wizard should then disregard all time intervals that are outside the specified time of day.

- **Search for time-slots on specific weekdays**

The user should also be able to specify on what weekdays the meeting should take place. This can cause some problems, if the user specifies a set of dates, that doesn't contain the selected weekdays. We chose not to do anything about this problem however, as we thought that a user most likely would not set conflicting restraints on search parameters. If the user would do this anyway, the wizard would simply not be able to find a fitting time interval

Conclusion

The question on how to keep this feature useful without inserting appointments in other users calendars remains. We have thought of two possibilities; The user can either use this tool to find a time, and then contact the involved persons one by one to ensure that the meeting will take place. In addition the user can include an exported *.ics file, to make it easier for the other users to create the appointment in their calendar. The other possibility is to create an appointment in a calendar, that is shared by all involved users. For this to work properly, all involved users have to merge their calendars with the shared one, but this is something the users have to agree upon.

Chapter 4

Implementation

This chapter will deal with the implementation of our program. The intention of this chapter is to give a better understanding of the overall program structure, while also detailing specific components and features in our program. We will therefore focus on the implementation of some of the more complex and interesting parts of the system. This will make it easier to understand how specific problems were solved, while also learning more about the general program structure.

4.1 Server

When the server starts, it sets up a listener on the port specified in the server configuration file. When a client tries to open a connection to the server, the main loop opens a socket connection and moves the connection into a separate thread, which is responsible for communication with the client and handles processing of protocol commands (see Figure 4.1). The reason for the main loop not to handle this directly, is that it would block the server from handling more than one user at a time. In chapter 7 we will discuss other methods for handling multiple client connections for greater performance.

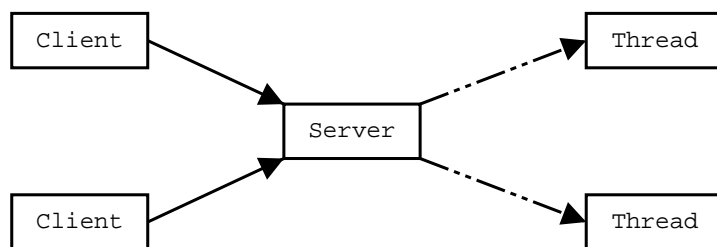


Figure 4.1: For each client connection the server creates a new thread.

When a connection is established and set up in a handler-thread, the server and client communicates using a ping-pong approach, where the server responds with a message or data output each time a client sends a request. The flow of communications is sketched on Figure 4.3 and in the pseudo code listed in Figure 4.2. The solid lines represents a “call” while the dotted lines means that it creates an instance of the object.

When the connection is opened, as many commands as needed can be executed, as long as the connection is kept open. When an input string from the client has been parsed and executed by the protocol, the execution time, IP-address of client, executed command and traffic amount to and from the server is logged using the log class.

```
while (input = input from client)
do
    output = protocol parse(input)
    return to client: output

    log(command)
end
```

Figure 4.2: Pseudo code for handling client communication

Escaping characters

When reading the input line from the client we use the `java.io.BufferedReader.readLine()` method, which reads an input from the user until it reaches a newline character (carriage return). To be able to send comments for tasks and appointments containing newlines, quotes and backslashes (\) to the protocol, we send these characters as other characters e.g. `"\t"` which is interpreted as a tab-character, is replaced with `"\\t"`.

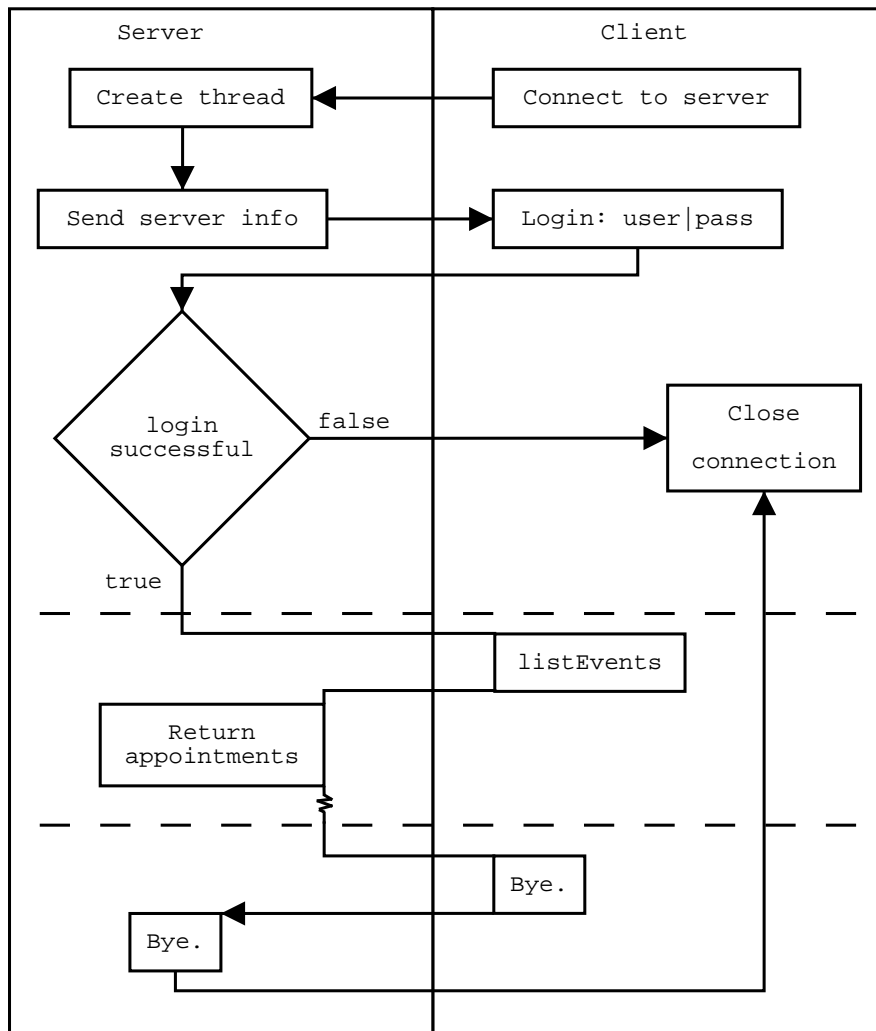


Figure 4.3: Communication flow between server and client. The section between the dashed lines indicate that several commands could be executed. Arrows indicate flow direction.

4.1.1 Database implementation

We have used MySQL as DBMS for our server, because of its ease of setup and use, and because it was introduced to us in our Databases and Software Architectures course [12].

After having designed the ER-model for our database, we translated the model into a set of MySQL tables as shown in Appendix B, with all entities, attributes, keys and relationships covered. To use the database in our protocol, we have created a list of queries.

Check if a provided username and password is valid

The authorization of a user is done by querying the database table for users with the username attribute equal to the provided username and a password attribute equal to the MD5-checksum of the provided password. If the query returns a row then the user login is successful. If user john tries to login using the password "pass", the resulting query is:

```
SELECT 1 FROM users WHERE username="john" AND password=MD5("pass");
```

Determine if a user is administrator in a calendar

Users can only add, edit and delete content in calendars they have administrator rights for, so we need to determine whether the current user is in the list of administrators associated with a calendar. To determine if user john is administrator of the calendar "TV-Programs" the following query will return one row if true and zero if false. Since the query needs to work on several tables we use aliases for the table names, instead of entering the full table name with each attribute.

```
SELECT 1 FROM calendars c, administrators a WHERE a.username="john" \
AND a.calendar=c.name AND c.name="TV-Programs";
```

Listing appointments from a list of calendars to merge with

The largest and most complex query in our system is the one that selects appointments from calendars that the user has chosen to merge with and that the user has privileges to see and finally within a given time interval. In the following example the user john lists appointments from his own calendar and the calendar "TV-Programs" within the time slot TIMESLOT_START and TIMESLOT_END.

```

SELECT * FROM appointments ap, administrators ad, groupmembership g
WHERE (timestart < TIMESLOT_END AND TIMESLOT_START < timeend) \
AND ( \
  (ap.visibility="public") \
  OR ( \
    "ap.visibility="group" \
    AND (ap.calendar=g.calendar AND g.username="john") \
    OR (ap.calendar=ad.calendar AND ad.username="john") \
  ) \
  OR ( \
    ap.visibility="private" \
    AND ap.calendar=ad.calendar \
    AND ad.username="john") \
) \
AND ( \
  ap.calendar="john" \
  OR ap.calendar="TV-Programs" \
) \
GROUP BY ap.id \
ORDER BY timestart;

```

The query first removes all the appointments that don't fit into the time interval. All the appointments that fit are then checked for privileges to remove all the appointments marked as group appointments, e.g., in case the user isn't a group member of that appointments related calendar. Finally it selects appointments from the calendars that the user has chosen to merge with and sorts them by their start time.

The boolean expression that the query uses to find all appointments within the given time slot, selects all appointments where the start time is before the end of the time slot and where the end time is after the start time of the start of the time slot. Figure 4.4 shows appointments that the expression would select.

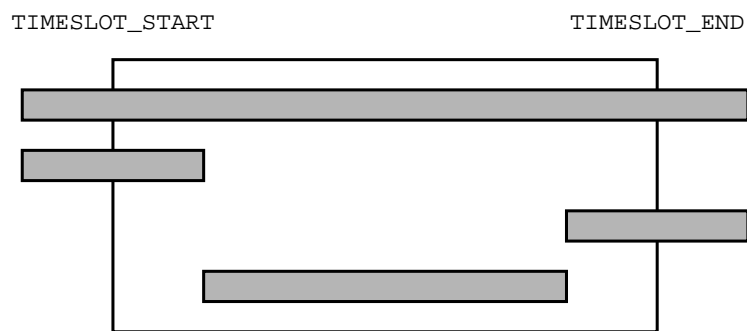


Figure 4.4: Different valid positions for appointments at the time slot. TIMESLOT_START is the start of time interval and TIMESLOT_END is the end.

4.1.2 Administration system

When setting up our calendar system in a large scenery with many users, the administrators of that specific server will need a set of tools to efficiently maintain it instead of directly working with appointments, users and relationships in the database. We have created an example of the features an administration program could have.

We have chosen to work directly with the database and the log file created by the server, instead of extending the server protocol or even create a separate administration protocol and communicating with the server through this. The last option would have enabled us to disconnect users from the server, see live statistics on how many users are currently connected and what they are doing. Using our logging system and client implementation it is not possible to track specific user sessions. Having the option to track sessions would help server administrators keeping an eye on potential malicious users.

Our simple administration tool is implemented in PHP and runs on a web-server with access to the database-server, in our test setup it has run on the same computer running the database and MyCal-server. The tool enables the server administrators to

- Add new users.
- Delete users and their calendars if they are the only administrator on them and the appointments related to those calendars.
- View a sorted list of appointments and the possibility to delete them.
- View a summary of the log file including:
 - How much data has been transferred to and from the server.
 - A sorted list of how often different protocol commands have been executed.
 - A sorted list of what IP-addresses has executed the most commands.

If the server and its administration system should be extended with ease of administration in mind, the following functionality would have the highest priority:

- Banning of users from certain IP-addresses.
- Disabling user accounts without deleting them.
- The possibility for a user to request an account on the server. The administration system would then require having a list of users requesting an account with the option to activate the specific ones. Another solution would be, to let the user to request an account, by supplying an e-mail address that the system would send an activation e-mail to. After the user has replied to this e-mail the system could activate the account.
- Extending the system to allow setting a quota on how much data each user can use on the server - maximum number of appointments, tasks etc. - Though it should only be used on servers with a limited capacity compared with the user load. Even with a limited quota the users shouldn't experience this quota as a problem.

- Assigning connection session ids to each new connection to help tracking users.
- See when a user was last active either by updating this attribute in the user table each time the user executes a command on the server or when he logs in, or by having the administration system search backwards through the log file to find when each user was last active.

4.1.3 Server configuration

To let the server administrator configure the server in an easy way, the server loads a configuration file at startup. In this configuration file, the administrator can specify:

- **port** - which port the server should listen for connections on - defaults to 4444
- **logfile** - path to file where the server should write log entries to.
- **loglevel** - how much data to log - further described in the next section.

Example of server configuration file

```
port=4444
logfile=/var/www/html/mycalserver.log
loglevel=1
```

4.1.4 Server logging

As described in the server software architecture description in Section 3.1.1, the server must log activity on the server. We have created a logging utility class that handles all logging on the server side. This class is used from different parts in the server architecture. These are server main loop and threads for handling clients and database connectivity.

When the server has loaded the configuration file it tries to open the specific log file. If this fails the log class will default to logging on standard output.

When the server starts, it sends a startup message to the log class which then determines if the action should be logged, depending on the log level set in the server configuration file. Equally when a client opens a connection to the server, this is logged. These two actions are logged if the server is running with log level 2 or above.

If the server is running with log level 1 or above, the server logs all client attempts to run protocol commands. The execution time, command and parameters, traffic size and client IP-address are logged for debugging purposes and getting an overview of server usage.

The highest level of logging is level 3, where all database queries are logged. Again this would only be of use for developers, unless the server administrators is trying to track malicious users, who are trying to compromise the server. This log level will produce a considerable amount of output, and it will quickly be hard to get an overview if the log file isn't put into a log rotation cycle.

4.2 Client

In this section we will describe how our client program is structured and how the complex parts of the client implementation have been done. The final client program is depicted in Figure 4.5.

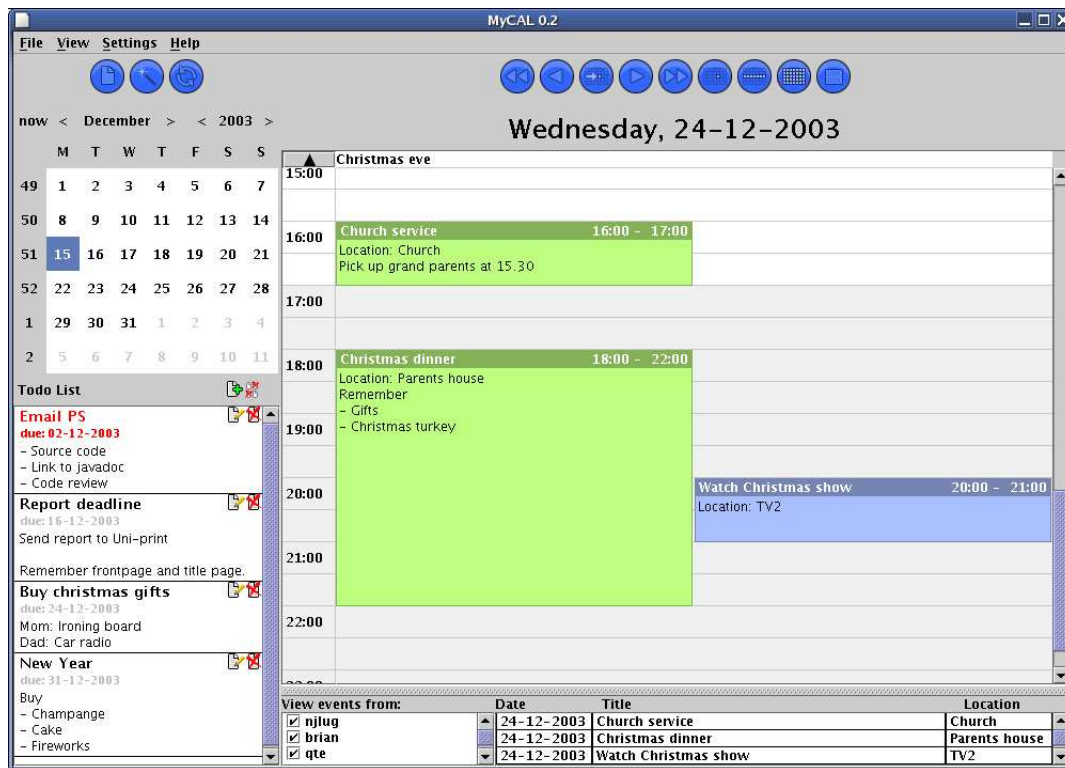


Figure 4.5: Client program main window. Showing 3 events in day view and task list

4.2.1 Login dialog



Figure 4.6: The login dialog that the user is presented with at startup

When the client program is started, the user is presented with a login dialog where the user can enter what username and password to login on the server with. This login information is then saved in the client-side protocol until the program is closed. If the client wants to change to another server, the login dialog contains a button that opens the server settings dialog. It is not possible to close the login dialog and continue to the program,

```

#MyCAL Configuration file for client
#Mon Dec 15 11:52:11 CET 2003
port=4444
server=h142.s.cs.auc.dk
username=brian

```

Figure 4.7: Example of client program configuration file.

without having provided correct login information. (Figure 4.6). The login dialog loads information from a configuration file, to let the user save last used server hostname, server port and username.

If the user changes the server hostname and port to connect to in the server settings dialog, or checks the "remember username" checkbox, the client saves the settings to the configuration file. The configuration file can easily be changed by the user by opening the file in a text-editor. If the configuration file contains information that the client program can't understand it defaults it the values specified in the source code.

4.2.2 Main window

When the user has logged in the main window is opened, which consists of a main class, that handles creation of sub objects of other classes presented in the same window. The main window consists of the following components: (see Figure 4.8 for visual reference)

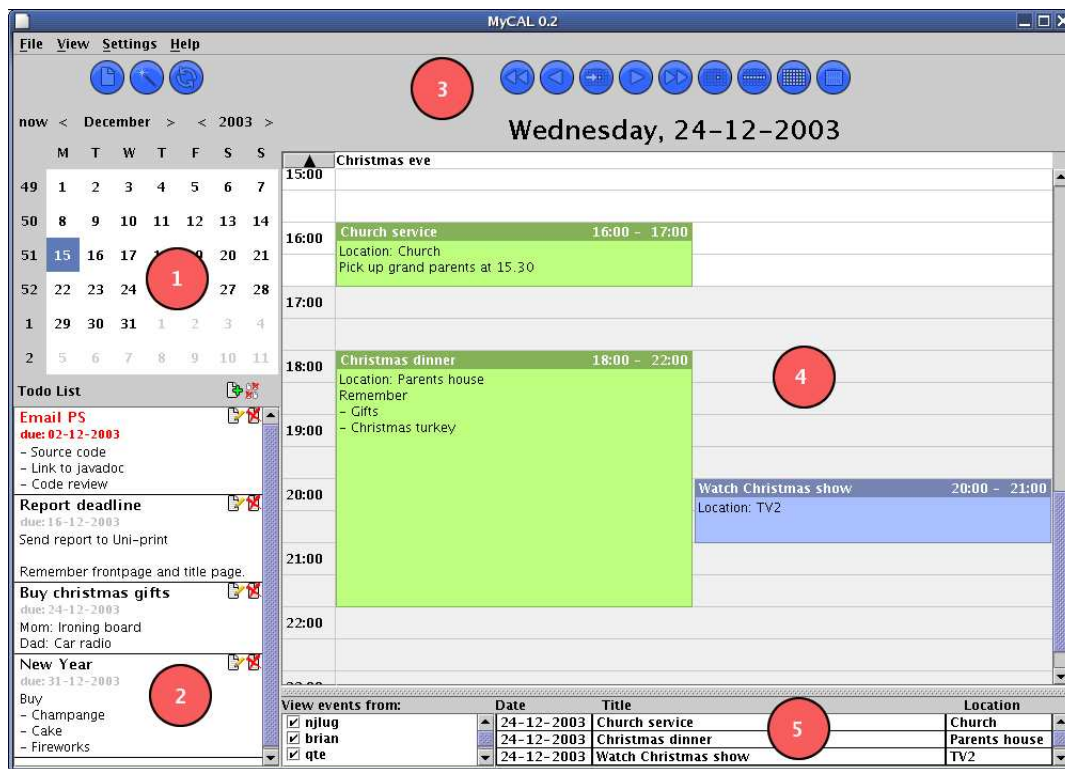


Figure 4.8: Client program main window, shown with numbers explained in this section

1. **Minimonth** - the user can use the minimonth component to quickly browse to a day, week, month or year, instead of having to browse using the some what heavier viewer component. When a user clicks on an entry in the mini month, the viewer component changes to the selected viewer type and moves to the selected time.
2. **Task list** - contains tasks of the current user, and when the user adds or deletes a task the task list component is updated.
3. **Toolbar** - enables the user to browse in the viewer component and change between different viewer types. Apart from interacting with the viewer component, it also calls the setting dialogs, event editor and meeting wizard.
4. **Viewer** - consists of 4 sub components (day, week, month and year view), that each presents the user with a different view of the calendar.
5. **Event list** - is separated into two sub components; a list of events with start date after current time and a check list of calendars which the user has selected to merge with.

Further explanation of the different components, icons and visual effects is given in the user manual (Appendix E).

Class diagram

Figure 4.8 shows how the GUI client looks when initialized. The functionality in that Figure is described here using a class diagram. When it is initialized, then the main part is the views, while the minor part is the task list. The default viewer is the DayView, and the changing of views is handled both internally among the viewers and by the MiniMonth, Toolbar and the MainMenuBar. As the legends say, that changing is represented by a “call”. The same type of connection is found between the event editor, and all the class that “calls” it. The contents (equal to its appointments) of the viewers is decided by the CSCalProtocol, so all the viewers depend on that class. EventEditor modifies the contents of the database through the CSCalProtocol and does therefor also depend on it. The Meeting Wizard is only called from the toolbar, and it depends on the CSCalProtocol, and when it is succesfull in finding a time slot, it will call the EventEditor.

The task list works similar to a viewer, in the sense that it depends on the CSCalProtocol, and the class that modifies the contents of the task list, works like the EventEditor.

4.2.3 Meeting wizard

The wizard consists of a dialog where the user selects the constraints after which the program sends the information to another class that does the calculation. The result is then sent to a panel where the user can select the time slot he prefers. A visual representation and a usage guide is found in the help documentation delivered on the CD.

The wizard algorithm must take all the requirements into account when it is searching for an available time slot. The easiest way to show how it works, is by using the same drawings that we constructed the algorithm from. We will use the legends shown in Figure 4.10, where

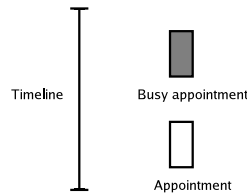


Figure 4.10: Legends

an all-day appointment is considered as a 24 hour event on the day(s) it affects. First we will show what the wizard considers when searching for appointments.

Example

We have an example user that has some appointments, both busy and not busy (see Section 3.2 for a further explanation of the difference). The wizard gets the list of appointments and starts with checking whether the day in question is selected by the user. If the day is selected it starts with finding the appointments that are in the chosen time interval. If it encounters an appointment that is marked “busy”, it places it in a new time-line, as shown in Figure 4.11. This happens for all the users that are selected by the user then. The next thing that

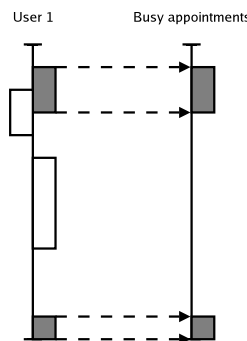


Figure 4.11: Example

happens is that the wizard creates a merged time-line, that contains all the busy events from the different users. This is shown in Figure 4.12. This time-line is then inverted to create a time-line that holds fictive appointments that don't interfere with the other users. And last but not least, the appointments in that time-line are checked to see whether they are equal or larger than the specified duration.

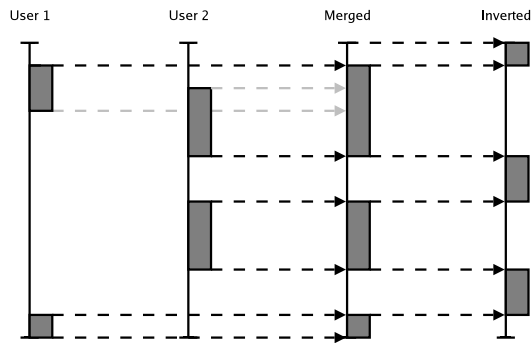


Figure 4.12: Algorithm

Algorithm

The algorithm will be described in a simplified version in pseudo code in this subsection.

```

FindMeetings(start time, end time, duration, calendars, days) {
    setStartDate(start time);
    setEndDate(end time);
    Appointments = listAppointments(start time, end time, calendars);
    while (StartDate != EndDate) {
        if (day is in days) {
            findTimes();
        }
        StartDate++;
    }
}

```

The only purpose of this method is to take arguments from the dialog where the user specifies constraints, to keep track of what day it is and then call the methods that contains the real algorithm. It takes the following arguments: start and end time of day (e.g. 8AM-4PM), duration of meeting, calendars and days of week to include in search and finally time interval to search through.

The actual algorithm is in the findTimes method, which is called whenever the date, that the wizard is searching on, is the the array of days.

```

findTimes() {
    candidates = findCandidates();
    if (candidates != null) {
        if (allday isn't in candidates) {
            if (candidates[0].startTime > day.startTime) {
                diff = candidates[0].startTime - day.startTime;
                if (diff >= duration) {
                    start = day.startTime;
                    end = candidates[0].startTime;
                }
            }
        }
    }
}

```



```

        Appointments[i].startTime < day.endTime &&
        Appointments[i].endTime > day.startTime) {
            candidates = allDay;
        } else if (Appointment[i].busy &&
            Appointment[i].startTime > day.endTime &&
            Appointment[i].endTime > day.startTime) {
                candidates = appointment;
            }
        }
    }
    return candidates;
}

```

The method that finds the appointments that are in the time interval specified as the start and end time of the day. It starts by checking whether an appointment is an all-day appointment and whether it is busy. If it is, it places an all-day mark in the candidates array. Then it checks for all appointments that are busy, and are in the time interval. If all the conditions are met, then it places the appointment in the candidates array.

When the FindMeetings class is done it sends the start and end array to a class which creates a dialog where the user can select what time he would like to have the meeting.

4.2.4 Importer

Importing appointments is a feature included to make it possible to use appointments created using other applications. This is an important feature since the users will be able to move appointments from other systems to MyCal without being forced to create each event manually.

The single most important issue here is what format to choose, when deciding on what the importer should be able to handle. We decided to use the iCalendar format, since it is widely available, and offers everything from weather reports, sporting events and other events targeted at the general public (see [2] for further info).

The specifications of the iCalendar format, which was used to create the parser, can be found at [5]. However the parser only supports the most important tags, and those that match the functionality of our program.

When a user chooses to import a *.ics file (see the help section for info on how to do that) then the file is sent to the parser which is responsible for recognizing the tags it knows and extracting information from it. It reads the specific file line by line and writes all the recognized tags to a string with each tag separated by a newline character. All tags, except the time tag, are written directly to the string. The time tag can appear in several different ways (see the website [5]), and to uphold consistency the parser converts this to this format: YYYYMMDDTHHMM. The “T” acts as a separator between the date and time information. All tags outside “BEGIN:EVENT” and “END:EVENT” are ignored.

The next step in the parser is where information is extracted from the newly created string, to create an array of events. Each event begins with an “BEGIN:EVENT” and ends with “END:EVENT”, so the parser runs through a for loop, for each event in the string. Time is again a factor that needs to be taken care of, since an event might be missing an end-time

tag and a duration. Events of that kind are being treated as all-day events. If end-time is missing but duration isn't, it calculates the end time from the duration. If both are specified, duration information ignored. In case there isn't any start-time tag, it is set to the current time and the title of the event is set to start with "INVALID EVENT: NO START TIME SPECIFIED." in order to get the users attention.

When the entire file has been parsed, the resulting array of events is displayed in a dialog, enabling the user to select which events to import. These events are then added to a calendar of the user's choice, selected from a drop down list of all the calendars the user has admin rights.

4.2.5 Exporters

To allow users to move appointments from our MyCal system to another system, e.g. to their website, other applications or another MyCal server, we have created a series of export methods that export to various formats. We have chosen 3 formats - iCalendar, HTML¹ and XML - and are going to describe the output of each exporter and how it can be used.

The user chooses which format and file to save export data to, and the program then sends this information to the selected export method together with a list of appointments, more specifically future appointments from a merged list of calendars.

iCalendar Exporter

The MyCal client program includes an importer for iCalendar files and we have implemented an exporter as well, so a user can export events to the iCalendar format and import them again on another computer - or store them as backup.

The iCalendar exporter should be used as the primary format when working with imported and exported lists of appointments, since it can be used with a range of other Calendar applications e.g. Mozilla Calendar [11], Apple's iCal [2], Microsoft Outlook [3] and Lotus Notes [8]. Furthermore the iCalendar format is used on many calendar enabled cellular phones.

HTML Exporter

The second export format is the HTML format. The resulting HTML file can be placed on the personal website of the user to present his or hers schedule.

The output includes a list of appointments separated by an horizontal line, each with the title, start and end time printed. If a location and description is set for the appointment, they are printed as well. With a CSS² document the list could be presented to the user to match the layout of the website.

¹HTML: HyperText Markup Language

²Cascading Style Sheet

XML Exporter

The XML document output from the XML exporter can also be placed on a website and styled with an XSL³ document. Compared to the HTML output the XML can be used in other applications, where the developers wish to create a parser for our format. An example application that could benefit from this output could parse the XML document and pop-up a reminder warning 5 minutes before the appointment starts.

4.2.6 User Interface

The entire user interface is created using Swing and without using any GUI tools. The fact that this was the first encounter any of us have had with Swing has resulted in a lot of problems during the first part of the client programming. Most of these problems were rather trivial and were overcome by using the javadoc ([13]) and some of the tutorials from www.java.com, but yet other problems were not so easily overcome.

An example of these problems is that when adding a JTextArea to a scrolling pane, the view is automatically changed to the part with the JTextArea. This has and still does create some undesirable effects in some views. Especially in DayView, where we wanted the initial view to show the time from 08:00 in the morning and forward. But this doesn't occur with some configurations of appointments.

Day-view and Week-view

The day view and week view components are the core event viewing components in the client. Together they allow the user to get a clear overview of events while still being able to get detailed information on specific events. We will therefore discuss their structure and development process in detail now, as far as they seem non-trivial.

We decided to start developing the day-view, as we thought that the week-view could be created by slightly modifying an existing day-view component. The first approach to develop a day-view was to use the JTable class to create a time-grid background, and then somehow show the events inside the table's cells. We quickly discovered that the JTable and JTableModel class were not easy to customize however, and thus we decided to create the grid ourselves by using JPanel objects with a border.

The next problem was to display event boxes on top of the created grid. We decided to use layered panes, where one layer would contain the grid (the gridpane), and another layer would contain the actual event boxes (the eventpane). This resulted in another series of problems: As the whole view was inside a scrolling pane, the components `setPreferredSize()` methods were used to determine the size of the viewport content. When using a layout manager, to make use of the `setPreferredSize()` method, a layered pane would treat components as if they were all on the same layer. This made it almost impossible to both set a proper size for the components and to have them be displayed on top of another. The solution to this problem might be best described as a workaround, as it is not certain that it is the intended way to solve the problem, or if the problem situation ever has occurred to the java developers

³XSL: Extensible Stylesheet Language

anyway. In essence, the solution is to use the BorderLayout layout manager on the layered pane. Adding the gridpane and eventpane to the layered pane, without specifying where in the border layout the components should be positioned, gives the desired result. The components can fill the viewport horizontally, while keeping their height at a specified size.

When we had successfully placed the eventpane on top of the grid, we needed to find a way to display several simultaneous events. The solution to this problem was to create a pane that would contain several subpanes that would act like columns in a table. These columns would then again contain the actual event boxes. To prevent confusion, we added a restriction of 10 columns. Originally we had intended to make it possible to see the other events (if any) in some way, but later we had to dismiss this idea due to limited time.

After we had created a usable day view, we created a new class called Day, which would be used in both day and week view. This class contained the layered pane and all its contents. The DayView and WeekView classes merely create a scrollpane, add column and row headers (place for allday events and timetable respectively) and add the right number of Day objects to the main viewport of the scrollpane.

Chapter 5

Testing

Testing is one of the most important parts when developing a program, as a program which was not tested properly tends to have many bugs. Testing can occur at various stages in the development process and can be executed to test different aspects of the program. This chapter will explain what types of testing we used and how the tests were executed.

5.1 Ad hoc testing

Ad hoc testing is the testing that is done after developing some small quantity of code, to quickly find any errors in the new code. It is an integral part of development, because when practiced it gives the possibility to catch a lot of errors, compiletime as well as runtime errors. We will discuss our way of doing ad hoc testing in this section.

We primarily used ad hoc testing as an integration test, so when we had developed a new or modified an existing part in our system, we would try to build the whole program, instead of only building the class that we had modified. After fixing all compile-time errors we ran the whole program to check for any errors and unexpected behavior. This made it possible not only to find errors in the new developed code, but also to check for design errors and unexpected side effects.

After removing all found errors, the code was submitted to our configuration management system. Thus the newly written code was not only tested by the author, but possibly also by any other developers who were testing their code. When developing some of the main components we actually asked the other people to update and test it with their own code, to ensure that we had followed our predefined design.

This kind of testing is not systematically enough to ensure an error free program, but it is a great improvement to the approach of developing components entirely independent from the rest of the code. It also gives the developer a greater understanding of the program, because if the program has run-time errors, it might have been caused by an incorrect method call, or something similar that doesn't invoke a compile-time error.

5.2 Unit testing using JUnit on central classes

”JUnit is a simple framework to write repeatable tests” [7] - this framework is in most cases used as an environment for unit testing. Developers use this environment to measure how far they are in the development phase, or simply to test a program for side effects. Another good thing about ”JUnit” is that if a test case/suite that is complex enough and deals with all the important classes is created, the program structure can be created from the test cases. All the developer has to ensure is that the program follows this structure is to run the ”JUnit” test. So one could actually argue that JUnit is very important to the project, because it proves that the requirements set up are being followed by the program and that there aren’t any side effects - if the test case/suite is good enough.

We didn’t know of JUnit in the beginning of our project, and hence didn’t write a test case, nor a test suit. When we were nearing our project deadline, we were told that the tests we had done, so far weren’t the best way of documenting that our classes worked as intended. The ad hoc testing described before this section, and acceptance testing described in the next section deals with those tests. So we decided to test the CSCalProtocol, which is an essential part of our program, plus that also implies testing the Protocol and the Database class. But since we hadn’t made a test case in the beginning of the project, we wrote it from our requirements. This gave us a list of test cases that needed to be successful if the classes should follow the requirement list. The test checklist is added as Appendix B, and it describes what is needed to run the test and what the expected result was. After we had created this test checklist we created a class that extended JUnit test case and used the methods from it that matched the return values of our methods, these extended methods were:

- assertEquals(String, String)
- assertTrue
- assertFalse
- assertNull
- assertNotNull

To quickly sum up Appendix B, then we have four kinds of users, an admin user, a co-admin user, a group member and a public member. They have been named according to what their purpose is, so the user who is an admin is called AdminUser etc. The admin user has three kinds of events, a private event, a group event and a public event. The idea behind this, is that it should be easy to understand what kind of events the other users should be able to see, and wether anyone of them should be able to edit, create or delete events.

The easiest way of showing how we tested our `CSCalProtocol` is the following example, where a user is trying to log onto the server:

```
CSCalProtocol p = new CSCalProtocol();
p.setLogin("AdminUser", "AdminUserPW");
assertTrue(p.login());
```

This is a fairly simple boolean test, which fails if the server doesn't recognize the user, or his password. Another example is how we used the `assertEquals` method to determine if what the server replies is a "denied" message. The `CSCalProtocol` replies with Strings when a user tries to modify the database, e.g. when a user tries to add an category:

```
assertEquals("Category added..",
    p.addCategory("AdminUser", "newCategory", 100, 100, 100)
);
```

The same test fails if the user is trying to add a category to another users calendar:

```
assertEquals("Permission denied..",
    p.addCategory("UnknownUser", "newCategory", 100, 100, 100)
);
```

By using the test methods mentioned in the beginning of this subsection we have then tested all the methods mentioned in appendix B. The outcome of the test was written in the test checklist as the "actual result", but only in the cases where it differed from the expected result.

It is at this point that JUnit proves to be a very useful tool, because there were some unexcepted result, and after having corrected the code, we could just run the test again. By using this test case we discovered some bugs, and we fixed most of them. Those left behind did not have an impact on the program execution, and they didn't provide any security risks. Besides finding bugs the test case also made us rewrite some of the methods in the `CSCalProtocol` and the `Protocol`, because their return value were useless.

After conducting this test case we all agreed on the fact that JUnit, despite looking like a simple replacement for if/else statements, is a powerful tool, that can be used to create a better program. And we intend to use it the next time we write in Java.

5.3 Acceptance testing

Acceptance testing is the final process of testing. This is usually done by the customer to see if the program matches the requirements the customer has specified. In our case, since there was no customer involved in the project, we had to make this testing ourself, and look at the program from a user's perspective.

Furthermore the requirements of our calendar system were not specified clearly and detailed at first, and the requirements were re-evaluated during the entire development process. Since we had to play the role of both developer and customer, this re-evaluation was probably a little too easy sometimes.

In a real developer-customer relation there would most certainly be more conflicts due to the difference in interests. A real customer would probably not have agreed to change the requirements list as much as we did. However, a real customer would probably have created a more specific and detailed requirements list, where the developers would not have been allowed to make so many changes. This combined role of developer and customer resulted in a not so realistic acceptance testing, since the duality of our roles allowed us to make changes in the direction of least resistance.

Our acceptance testing consisted of running through the list of requirements and verifying that the program followed the specified behaviour. This included verifying that the business case was covered by the calendar system. Some of the acceptance testing consisted of verifying goals concerning usability and the general "look and feel" of our client. This part of the acceptance testing has proven to be almost impossible to do properly, as we both had the role of developer and customer.

Chapter 6

Development process

The development process can have a big impact on the quality of the product, as it influences the team's knowledge about the project and the current status. This knowledge makes it possible for the team to work as a unit and create a consistent product. If the team does not work together properly, the individual units are not certain to integrate properly, and bugs can be harder to spot and can require a greater effort to fix. That is why we will take a closer look at our development process in this chapter.

6.1 Development cycle

6.1.1 Planning

In the initial phase of the project we discussed our visions for the system and started with some brainstorming over a general structure. We did not choose a specific development model to work with at this point, but we layed out a plan on how to proceed together.

We agreed to develop a prototype for the system, which was to include only the most basic functionality. The prototype was to be developed strictly to gain more knowledge on the key concepts, that would be necessary for our project. These concepts included java swing, network and database programming, which most of us had never worked with before. In addition we wanted to ensure that our layout for the protocol architecture would work properly.

When the prototype was done we decided to start from scratch again, and to use the gained knowledge to create a real product. We wanted to do some more detailed planning, this includes designing the code structure in detail, creating a priority list and time schedule for the various components. As a part of the code structure design, we wanted to create a detailed class diagram that would give an overview of all the classes we would need. This class diagram was to include detailed information on fields, constructors, methods and how the classes interacted with each other. The planning phase was supposed to last one week, and thereafter we wanted to start the code development. In hindsight this approach could be interpreted as the use of an evolutionary development model.

The development of the code was to be split up in three iterations as follows:

1. Starting from scratch and developing basic functionality
2. Development of advanced functionality
3. Work on minor details, finishing the program

After each iteration we wanted to summarize on our current status and re-evaluate our plans for the program. This would make it possible to use any newly gained knowledge to improve the program structure, and if necessary, to realize which features to remove, add or change.

6.1.2 Development phase

After the initial planning we started the development of the prototype. The team members chose which part of the program they wanted to work with, instead of actively assigning tasks. There were frequent discussions about the structure of various components, and so everybody was able to influence the program structure and visual layout.

When the prototype was done we summarized briefly on what we had learned creating the prototype and started working on the real product. We started with some more detailed planning, to ensure that the individual components would integrate properly, and that the finished product would actually do what we wanted it to do. We created a priority list for the individual components we would need, and estimated the time it would take to complete these components. We then brainstormed the layout and functionality for the various visible components in the client, so that everybody would know how to create the individual components.

We had planned to create a detailed class diagram that would give an overview of all the classes we would need. Actually this was not done before we had written the very most of the code however, as we did not know how to properly design a program structure in detail. In addition we had only little experience with programming in Java. Therefore, the work on the diagram was stopped shortly after the idea had come up, as we were not sure how to create a good structure.

Since some of the team members had only worked with certain parts of the prototype, not everybody was familiar with each of the concepts we tried to gain knowledge about, except for the protocol system, which was discussed frequently. Instead of exchanging the knowledge that we gained on the various parts before starting the development of the real program, we helped each other out when needed during development.

When the prototype was almost completed, we participated in a code review, which made us redesign a few things. A description of the code review is in Appendix C. After finishing the prototype and planning of the further development we started on the first iteration. We wanted to include basic event viewing functionality (day view, event list, task list) and navigation (navigation panel, menu, toolbar). The actual development proceeded similarly to the development of the prototype, as everybody selected which part of the program they wanted to work on. There was slightly less communication concerning the development of the various parts, since we all had a better understanding of the intended overall program structure and design. The ER-model was also redesigned to include all the entities we needed in our system.

In the second iteration advanced functionality (e.g. meeting wizard, calendar merging) was included, and the various components were set to use the new ER-model. A lot of time was spent redesigning parts of the structure and making the structure more consistent.

The third iteration was mostly spent trying to fix bugs. A lot of these bugs were caused by the lack of a specific class design, so that some of the parts did not integrate as smoothly as they could have. A php server administrator interface was also created, to allow the server administrator to manage users.

Testing was first started when the second iteration was done and we already had spent some time on the third iteration, due to the fact that we did not want to take any time from code writing.

6.2 Development tools

The use of tools can make development much easier, as it can reduce the effort wasted on otherwise trivial and perhaps repetitive issues. Selecting the right tools should not be the only concern though, as the correct use of these tools can be of great significance too. This section will thus detail what tools we used and how we used them.

6.2.1 Configuration management

A configuration management system keeps track of changes and different version for individual files, while keeping them all in a central place. This allows the team to work on the various parts of the project simultaneously, ensuring that everybody has access to the latest work. It also adds safety to the development, as it is possible to retrieve older version of files and thus any usually fatal errors and mistakes can be corrected.

We have chosen to use CVS to manage our source code, as we had a positive experience with CVS in previous projects. In our planning phase, we have worked out a routine on how to use CVS, to ensure the maximum benefit from this tool. This routine requires the developer to go through the following steps after creating new code:

1. Update the local work-copy of the tree (fix potential clashes)
2. Make a complete build of the whole program(with no compile-time errors)
3. Run the program and look for runtime errors and side effects in changed and related code
4. Commit the code to CVS (look for possible clashes and fix if present)

This routine was created to ensure that the central version was always buildable, so that retrieving the latest code would not result in any new build or runtime errors. In addition, this routine would require each developer to actually integration test any new parts of the code. This approach is of course not sufficient in itself to ensure a bug free program, but it adds some additional safety.

This routine was used almost always when committing new code, however there have been a few exceptions from some of the team members. This did not always lead to bad code in the central version, and when it did the problem was usually fixed after maximum 10 minutes. In general the use of CVS was handled quite well, and CVS was of great use to us, as it made it possible to delegate work efficiently. The CVS stats system was also useful keep track of changes in the code and to determine who had committed what code.

6.2.2 Testing

Since testing is an important part of programming, it is also important to select the right testing tools. Some tools make it possible to test a programs components with relative ease, as they allow for automation of tests or have other benefits.

We used JUnit to test one of our central components, the protocol, as it is easy to use and makes it possible to automate tests. But since we didn't use it at the beginning of our project we didn't benefit as much from it as we could have. This was due to the fact that we were occupied with many technical problems concerning the programming. In hindsight we would have been better off focusing on setting up JUnit on many more classes than focusing on some of the rather trivial technical difficulties. And at the same time it would be easier and more efficient to develop the test cases in parallel with developing the actual code. This is explained more detailed in section 5.2.

6.2.3 Code documentation

Code documentation is an important part of developing a program, as undocumented code often is hard to understand and use. Good code documentation can make it possible to quickly grasp the concept of some code. This is not only useful when trying to understand some given code later on, but can be great asset at development time, since it allows the developer to quickly see how to use some given code.

We have used javadoc to document our code, as it is a tool that can create a good documentation with relatively little effort. Most of the time javadoc was added to the code when a given class was finished, but we had no specific policy on the use of javadoc, and thus some classes were lacking documentation until late in the development. This sometimes made it necessary to analyze some code, in order to understand how to use some class or method. This was not a frequent problem however, but it could be avoided with a simple agreement upon how the documentation process was to be handled.

6.3 Reflections

The prototype development was supposed to give us a better understanding of the key concepts we would need for our project. We did gain basic knowledge about the different concepts, but several problems arose later on during the development of the real program, which took some efforts to solve. In general we found the prototype to be useful for understanding the basic concepts for the development of a given system.

During development we came to realize that our planning had not been completed, and we had somewhat underestimated the time we would need to complete the first iteration. We rarely wrote down our decisions when reevaluating our design, and this sometimes caused an inconsistent structure due to misunderstandings. In addition we did not have enough experience with creating systems as large as our calendar system. Thus we had to try out different approaches to some problems, before we were able to determine the best solution to some problems.

In the second iteration a lot of effort had to be used on making the structure of the program consistent, as the class diagram was not done properly and could not be used. One of the parts we tried to make more consistent was the use of timestamps and calendar objects, which at some point was left unfinished however, as more serious issues were discovered. Thus we did not have enough time to include all the features we wanted, which is why features like recursive events and a reminder system were not included in the final version of the program.

When the development neared its end, a lot of bugs and issues remained. To keep track of these, we created a list of issues, where we would add new issues as soon as they were discovered and remove them as soon as they were solved. We realized that there was no time for the third iteration and thus it was dropped.

Systematic testing was started very late in our project, and we only had the time to test our protocol. Some issues could possibly have been avoided or spotted earlier, if we had started testing sooner, preferably at project start. We have agreed to use some sort of systematic testing from start on for our next project, as we believe that it can increase development efficiency when done consequently.

Our use of CVS as a configuration management system was very positive, as it allowed us to delegate work in an efficient manner. The CVS stats system made it possible to keep track of the development activities of the individual team members. Due to our positive experiences with CVS we have agreed to use it as our configuration management system in future projects.

Chapter 7

Future work

In this chapter we will describe some possible extensions and additional features that could be included in the system in later versions. Some of these are improvements of already present functionality and yet others are new functionality that would improve the value of the system.

Many of these extensions were discussed during the very early brainstorming phase and then rejected due to time limitations of the project, and yet other emerged during the development phase.

7.1 Security

One of the aspects of our system that could need more attention is security. When developing client / server solutions, there is a rather large possibility of receiving corrupt or even malicious data, in our case as protocol commands. In the following section we will try to describe some of the things that could be done to improve the security of the server and therefore also in general some of the methods that could be used to compromise a server.

7.1.1 Improving security with SSL

One way of greatly improving the security of the calendar system is to implement some sort of encryption standard, so that all of the communication between the client and the server would be practically unreadable to an outside attacker. As of right now the username and password is sent as plain text, and a potential attacker would be able to catch this by the process of "eavesdropping"¹ on the network using any of a wide range of network tools and utilities. These acquired user credentials can be used later to harass the user (e.g. deleting appointments or marking private calendar entries public or even steal his account) or to use a collection of sniffed passwords to a DoS attack (described later). A way of implementing encryption with relative ease is to use the `javax.net.ssl` package provides by sun. This package provides methods of creating an encrypted socket connection in a similar manner as normal sockets, and it is also provides the following security features:

¹Eavesdropping: sniffing TCP/IP packets

- Integrity Protection - ensuring that the data has not been altered during transport.
- Authentication - ensuring that the communicating machine is who they seem to be.
- Confidentiality (Privacy Protection) - the communication is encrypted during transport, and cannot be deciphered without great expense.

7.1.2 SQL Injection

Another commonly used technique used to attack servers that uses databases is called SQL injection. SQL injection is the process of delivering malicious input to an application that was not intended by the developer, designed to pass SQL code into a query. This can be used to manipulate information in the database, or to access privileged data.

In an earlier iteration of our protocol, it was possible to authorize as any user without having the password of that user using SQL injection.

We will use this exploit as an example of how SQL injection can be done, and which countermeasures that can be taken to prevent such openings.

Exploiting the login protocol command

This example is from an earlier iteration of our system, and the exploit described is no longer possible, due to the changes described at the end of the section. The section is based partly on information from [6].

When a user logs into the system using the protocol, the following protocol command is issued in this example using john as username and pass as the password.

```
User: john|pass
```

The protocol then translates this command and its arguments to the following SQL query.

```
SELECT 1 FROM users WHERE username=''john'' AND password=MD5(''pass'');
```

Our task as the malicious user is to log into the system without providing a password and performing a query on the server. As the protocol uses the first argument as the active user internally, so we choose not to inject our SQL query here but instead do it in the password argument. We want to skip the password authentication so we escape from the password equals statement and instead use a boolean OR so even if the password doesn't match it logs in. We escape the MD5("password argument") statement, by entering a quote and a parentheses followed by the OR statement. To ensure that the rest of the query isn't performed and perhaps fail we comment out the remaining SQL query.

```
User: john|'') OR 1; --
SELECT 1 FROM users WHERE username=''john'' AND password=MD5(''') OR 1;\
-- ');
```

The protocol then accepts the user as logged in. The user could then inject a separate SQL query that manipulates the database. The following injection deletes all users from the database.

```
User: john|'') OR 1; DELETE FROM users; --  
SELECT 1 FROM users WHERE username=''john'' AND password=MD5(''a'') OR 1;\br/>DELETE FROM users; -- ');
```

To prevent users from doing this, we have in the escaped quotes (” becomes \”) current version in the protocol, when a user tries to login. In a later iteration prevention of SQL injection when a valid user is logged in, and to prevent more advanced injections the escaping of dangerous characters has to be added all through the protocol code.

7.1.3 DoS attacks

Another possible way of attacking the system is a so-called DoS - Denial of Service - attack. A DoS attack is the action of flooding a target system with automatically generated requests and thereby creating so much load on the service that the regular users of the system cannot access the service or experience major inconveniences due to longer response times or similar effects.

DoS attacks come in a wide range of different variations and some of these are usually very hard to distinguish from actual load peaks created by many users using the system on the same time.

In our system however many of the strategies used in simple DoS attacks can with rather simple rules be recognized as an unusual user pattern. These actions include extreme use of the meeting wizard and the listevent method of the protocol. And if such suspect activities occurs the implicated user accounts could be temporarily disabled. And/or all requests from the suspicious network node could be ignored for a limited period of time.

Examples of other potential attack patterns could be:

- The same user credentials being used from many different notes within a very small timeframe.
- Many different users login from the same network node, again within a very small timeframe.

In order to implement a security policy able of recognising and countering some of the above mentioned attack patterns, a lot of monitoring of a working system is necessary in order to determine what should be considered normal behaviour, and this is far past the scope of our project.

7.2 Reminders

Another natural expansion of our system could be to add a reminder service, enabling the user to set up reminders to a specific appointment. The handling of such reminders (checking when to send which reminders and the actual sending of these) should be done by the server.

Email would be a logical choice as the primary reminder method since the sending of email is rather simple and that there is very little recourses involved (both financial and computational), but sending emails as reminders has one major disadvantage. If the user

currently away from a computer or other device capable of receiving emails, he will not get the reminder in time.

Another reminder media could be SMS. It is very common to have access to a cellular phone capable of receiving SMS's even when away from the office, and since these messages are more prone to catch the user's attention and thereby delivering the reminder message in time, SMS could be a grate choice. Unfortunately sending SMS' involves paying fees to the telephone operators and this may discourage users from using such a reminder service. Perhaps the best solution is a combination of the two, and letting the user decide which kind of reminder he or she would like to receive, email for free or SMS for a small fee.

7.3 Recursive events

Some events occur on a regular basis, and it can be tedious to create an event for each occurrence in a calendar. This problem could be solved by supporting recursive events, i.e. events that occur on a regular basis. This would be a possible enhancement for our calendar system, as it would allow the user to specify rules for the recurrence of the event.

7.4 Optimization

The main goal of this project has been to develop a system with the main focus on functionality as apposed to speed. But during the project we have found several places in which optimization is possible and some places where it is also needed if the system is to run on a large scale. We have chosen to discuss some of these optimizations under further development since we choose to prioritize optimization lower than features such as meeting wizard and import export functions.

In later iterations however these optimizations should be implemented in order to increase the performance, especially of the server.

7.4.1 Server optimization

Server optimization is the most important optimization since the server by nature is to serve multiple clients at the same time and therefore subject to heavier loads during peaks.

In our program a thread is created for each user who connects to the server, and this thread is destroyed upon logout or timeout. This is a rather costly way of doing this due to the cost of creating and destroying threads. Especially when the average lifetime of these threads is very low.

Thread pooling

One way of improves multi user performance is to implement a strategy called a thread pool. A thread pool strategy means that the server spawns a number of threads on start-up, and then assigns one of these threads to each new user. This strategy reduces the creation and destruction of threads and thereby improving the performance of the server, and further

more it improves the response time of the server, since the user gets an already created thread assigned, as opposed to waiting for the thread to be created.

Thread pooling can be categorised in to two types; static and dynamic pools. The major difference between these two is the manner of which the threads is created. In the simpler static version the pool is created upon initialization and the number of threads does not change. This implies that in periods of low usages there are a lot of unused threads taking up system resources. On the other hand static thread pools are by far the easier of the two types to implement. Dynamic thread pooling distinguishes itself from static pooling by the fact that the number of threads is not constant, but adapts to the current load of the server. This dynamic approach take up less resources during times of low activity, but uses more resources when scaling up or down the number of threads in the pool.

Thread pooling in one of the two forms or similar strategies is commonly used in web-server applications and other types of servers handling many requests within a small time-frame. A resonable extension of our server would be to implement thread pooling or some of the similar strategies to improve the server performance under periods of multiply users.

Based on information from [1] and [4]

7.4.2 Scalability

When dealing with many simultaneous users, the server performance is the most critical, since the client software isn't affected by the server having multiple connections (assuming the server response time doesn't increce much) on the other hand the server load increases by each new connected user.

Vertical scalability

Vertical scalability of a system, is the ability of the system to increase its capacity by adding resources in the form of more computational power or memory. Our calendar server will due to its rather linear functionality scale up rather well until the time the generation of the output is faster than network connection of with the server is connected. Beyond this point adding processing power will not improve the overall performance of the system.

Horisontal scalability

Horizontal scalability of a system is the ability to distribute the load of the system upon more individual machines in order to increase its capacity while still working as one logical unit. This is a strategy has many practical applications in systems working with a cluster of machines working parallel or is utilizing some sort of load balancing.

Our calendar server does in its current state not scale well horizontally, it is possible to spread the system over two logical units one working as the actual server, and the other running the database system, but beyond this point there is a need of a rather complex redesign of the whole system in order to work.

Chapter 8

Conclusion

During this project we were able to develop a calendar system, which can be used as a personal calendar. The calendar allows users to store their appointment information in a central location and to access this information as long as they have a connection to the server containing the information. In addition it is possible to share appointment information with other users, and the calendar system can be used to arrange meetings with other users. To ensure the privacy of the user, there are three levels of access for appointments, so that it is possible to keep some appointments private while sharing others.

We chose not to create a detailed design, since we lacked the experience with designing system structures. In addition we wanted to have a development process that would allow us to learn more about developing systems of this size, to be able to create a better design in future projects. A more detailed design could have proved useful in some situations however, as it would have allowed us to keep our system structure more consistent during development.

In our project group some communication problems occurred during the project, but since the group consisted of only five persons they were easily solved. In a larger group however such problems are harder to solve and more effort should be used to avoid these problems. One way of doing this would be to write down the decisions made by the group in more detail. This information could then be used to avoid the repetition of discussions.

During development we came to realize that we would not have enough time to include all the features, and thus some of the features, like a reminder system and support for recursive events, were not developed. These features had been designed, see Section 7.2 and 7.3, but were removed from our design plans due to time limitations.

In spite of these design changes and time limitations, we were able to create a calendar system within the time limit, which can be used as described in our business case, and which with minor changes could be considered a finished product.

Bibliography

- [1] David Brown. A simple multithreaded web server, December 2003. <http://java.sun.com/developer/technicalArticles/Networking/Webserver/>.
- [2] Apple Computer. Apple - ical, December 2003. <http://www.apple.com/ical/>.
- [3] Microsoft Corporation. Publishing calendar information on an intranet or the web, December 2003. www.microsoft.com/technet/prodtechnol/office/office2000/reskit/ork2000/html/70t3_4.asp.
- [4] Brian Goetz. Java theory and practice: Thread pools and work queues, December 2003. <http://www-106.ibm.com/developerworks/java/library/j-jtp0730.html>.
- [5] Network Working Group. Rfc 2445 - internet calendaring and scheduling core object specification (icalendar), December 2003. <http://www.faqs.org/rfcs/rfc2445.html>.
- [6] Mitchell Harper. Sql injection attacks - are you safe?, December 2003. <http://www.sitepoint.com/article/794>.
- [7] Junit, December 2003. <http://junit.sourceforge.net/#Documentation>.
- [8] Craig Lordan and Dick McCarrick. Notes 6 technical overview, December 2003. http://www-10.lotus.com/ldd/today.nsf/lookup/notes_rnext_technical_overview.
- [9] Lars Mathiassen, Andreas Munk-Madsen, Peter Axel Nielsen, and Jan Stage. *Object-oriented analysis & design*. Number 87-7751-150-6. Marko Publishing ApA, Aalborg Denmark, 1. edition edition, 2000.
- [10] MySQL. Mysql reference manual, December 2003. <http://www.mysql.com/documentation/>.
- [11] The Mozilla Organization. Calendar - frequently asked questions:, December 2003. <http://www.mozilla.org/projects/calendar/faq.html>.
- [12] Albrecht Schmidt. Databases and software architectures. Course website, December 2003. <http://www.cs.auc.dk/~al/teaching/2003/dbsa/>.
- [13] Inc. Sun Microsystems. Java TM 2 platform, standart edition, v1.4.2 api specification, December 2003. <http://java.sun.com/j2se/1.4.2/docs/api/index.html>.

Appendix A

ER-model

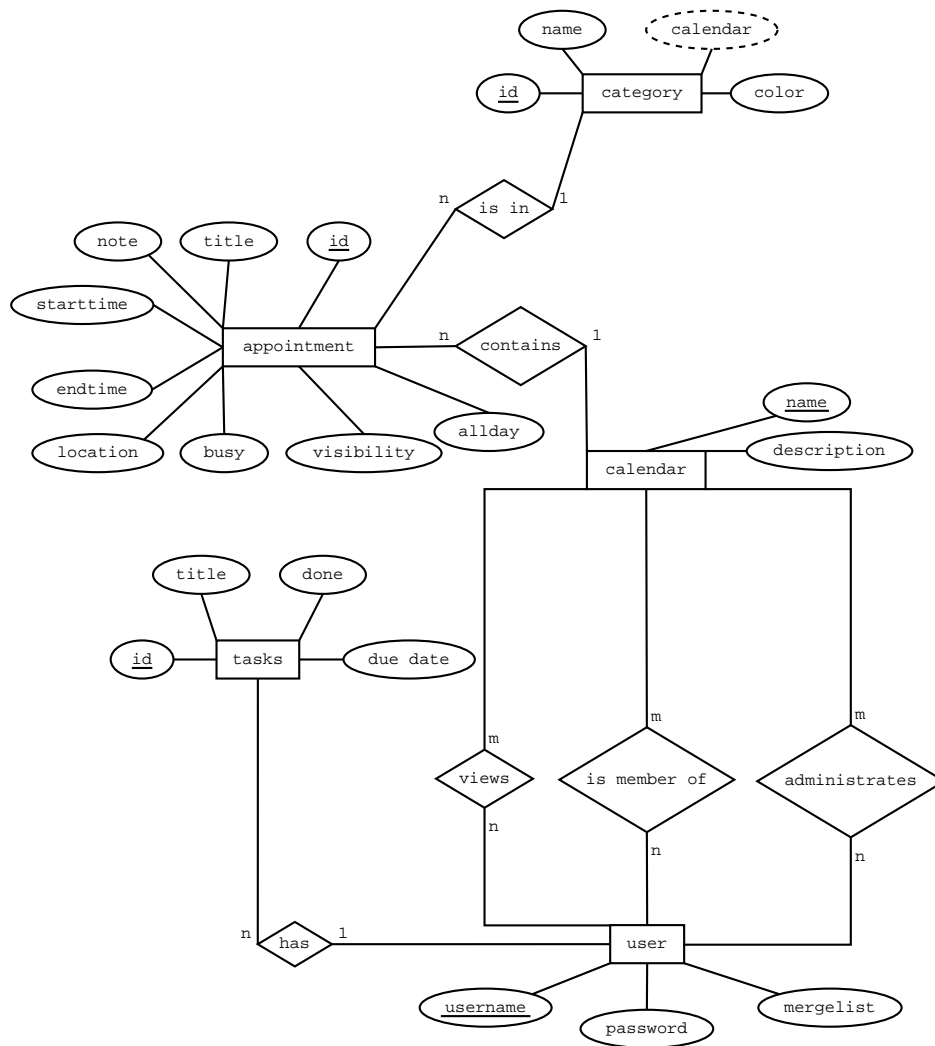


Figure A.1: Complete ER-model. Dashed lines around the calendar attribute for the category entity indicates a derived value through the appointment from the calendar.

Appendix B

Database tables

administrators

Field	Type	Key	Default	Extra
username	varchar(200)	PRI		
calendar	varchar(200)	PRI		

appointments

Field	Type	Key	Default	Extra
id	int(11)	PRI	NULL	auto_inc
calendar	text			
title	text			
note	text			
timestart	bigint(20)		0	
timeend	bigint(20)		0	
visibility	enum('private','group','public')		private	
allday	enum('true','false')		false	
busy	enum('true','false')		true	
location	text			
categoryid	int(11)		0	

calendars

Field	Type	Key	Default	Extra
name	varchar(255)	PRI		
description	text			

categories

Field	Type	Key	Default	Extra
id	int(11)	PRI	NULL	auto_inc
calendar	text			
name	text			
color_r	tinyint(3) unsigned		0	
color_g	tinyint(3) unsigned		0	
color_b	tinyint(3) unsigned		0	

groupmembership

Field	Type	Key	Default	Extra
username	varchar(200)	PRI		
calendar	varchar(200)	PRI		

tasks

Field	Type	Key	Default	Extra
id	int(11)	PRI	NULL	auto_inc
username	text			
title	text			
note	text			
done	enum('true','false')		false	
duedate	bigint(20)		NULL	

users

Field	Type	Key	Default	Extra
username	varchar(255)	PRI		
password	text			
mergelist	text			

Appendix C

Code review

As a part of the PE-course SWP we made a code review in cooperation with some of the other students on the semester. The code review process was first discussed in class, and later we had to turn in some vital parts of our code for reviewing. We chose to hand in the following parts of our program at the time of the code review:

- Server including database, protocol and main class (MultiServer).
- Client including CSCalProtocol, ClientEvent and ProtocolTest.
- Event utility class.

We selected these parts because they were the most complex and important parts at the time.

The code review itself consisted of two sessions. One in which we were to review another group's code, and one session in which our code was being reviewed. We sent a checklist with our handin of the source code, which the other group could follow to review all the areas we wanted feedback on.

- The code is adequately and correctly documented.
- The code follows applicable standards for structure, style, naming conventions, portability, and so on.
- The code provides appropriate tests for input parameter validity and plausibility.
- All code is executable (can be reached) and logic is consistent and correct.
- The program and all of its loops will terminate.
- Variables are uniquely named and defined before use.
- Initialization is complete and correct.
- All external variables and data structures are declared.

- All allowable variable values (including zero ?) are within range.
- Interrupts, user, operator precedences, etc., are obvious.
- The possibility of intermediate data results (e.g., underflow or overflow) has been checked and resolved.
- All mixed-mode computations and comparisons have been checked. This includes assignment of variables with different width.
- The code satisfies the design it implements.
- The code is maintainable.
- Documentation agrees with the implementation.

Process wise we benefited most from the part in which we were reviewing, since this review was done in a more correct fashion than the review of our code. The review of our code was more unstructured and suffered a bit under the fact that the not all of the members of the reviewing group was actively taking part in the review. However we still got our attention directed towards some issues concerning our code and coding style pointed out.

Appendix D

Protocol Test

Test description, test cases and test data can be found on the accompanying CD under:

"\test\index.html"

Appendix E

User guide / manual

The user guide for both server and client can be found on the accompanying CD under:

"\manual\index.html"

This file contains information on:

- Installation and setup of server
- Setup of test users
- Maintenance (adding users ect.)
- Client user help

Appendix F

Class diagram

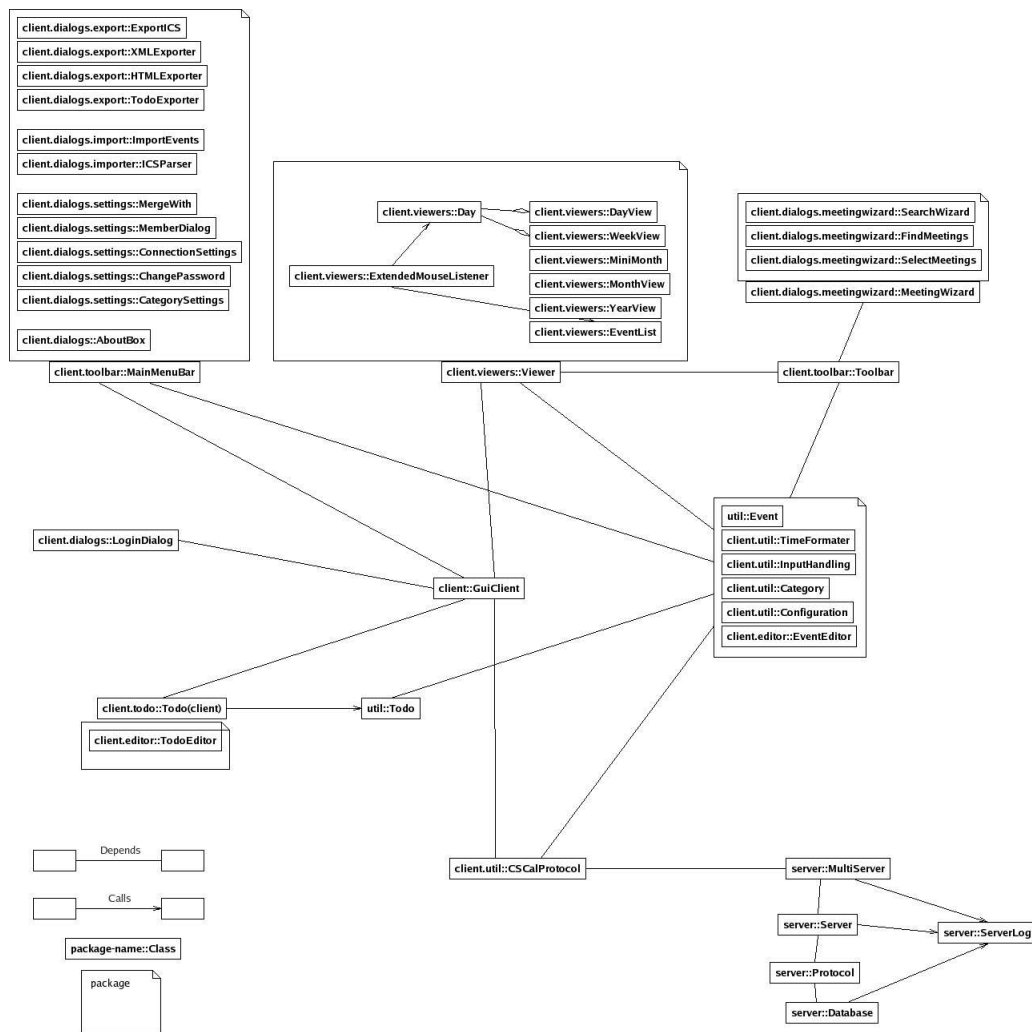


Figure F.1: class diagram