IT 13 026

Examensarbete 15 hp
April 2013

Goal-Oriented Collision-Free Schedule

Emanuel Berg

Institutionen för informationsteknologi
Department of Information Technology

UPPSALA
UNIVERSITET

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

Goal-Oriented Collision-Free Schedule

Emanuel Berg

The education of to-be physicians at Akademiska sjukhuset, Uppsala, includes practical services. The students are divided into groups that each has its own goals. The goals specify (1) what services that group's students should perform, and (2) for each service, a minimum number of times each student should attend that service.

It is only possible to perform any service at certain occasions: each occasion offers slots, to be filled by students. The occasions make up a calendar.

The challenge is to distribute the students over the calendar, so that the goal is achieved for each student and service. No occasion is overpopulated, and no student is due to attend two (or more) occasions that collide in time.

The algorithm to solve this sets up a table with occasions (expanded horizontally by their number of slots) as the x-axis, and dates (expanded vertically by two: the day parts) as the y-axis. Then, distribution of students is done top-down, left-right. Collision is avoided by having students only appear once per row. Overpopulation won't happen as the allocation of students is done explicitly to slots, not to occasions in general.

MS Access forms make up the UI. My thoughts when I set them up was that each form should boil down to a single purpose, but include everything to fulfill that purpose (and nothing else). Also, I setup an intuitive flow of movements between forms, and I made an effort to setup mnemonic shortcuts (and tab chains) as to minimize mouse use.

Handledare: Per Söderberg
Ämnesgranskare: Olle Gällmo
Examinator: Olle Eriksson
IT 13 026
Tryckt av: Reprocentralen ITC

Contents

1 Introduction

As part of their medical education at Uppsala University, the to-be physicians perform services.

1.1 Services

To make sure that a student acquires a certain skill level within some field, the student must carry out the associated service a number of times. This is the *minimum* participation requirement of that service.

Although it is desirable that students exceed the minimum requirement (gaining even more experience), there is also a *maximum* number of participations defined for each service: more practice than that is deemed unnecessary.

1.2 Occasions

Typically, for each service, there are two occasions every day: one *early*, and one *late*.

During such an occasion, a number of students can carry out that service. In the lingo of this document, the number of students that can participate at an occasion is the *slots* of that occasion.

1.3 The problem

We want to distribute the students over the occasions so that:

- each student carries out a service n times, so that $s_{min} \leq n \leq s_{max}$, for all services s
- no occasion is overpopulated with students
- no student is associated with two or more occasions that take place at the same time (in effect, at the same date *and* at the same day part)

2 The algorithm

An algorithm is executed to distribute the students over the occasions. The algorithm is best described by an example:

		type	1					2					3					4				
		slot	1	2	3	4	5	1	2	3	4	5	1	2	3	4	5	1	2	3	4	5
date	**part**	**first**	1	5	10			2	6				3	7				4	8	9		
12/15/11	e		1	5	10	x	x	2	**6**	x	x	x	3	**7**	x	x	x	4	8	9		
	l		2	6	1	x	x	3	**7**	x	x	x	4	**8**	x	x	x	5	9	10		
12/16/11	e		3	7	2	x	x	4	**8**	x	x	x	5	**9**	x	x	x	6	10	1		
	l		4	8	3	x	x	5	**9**	x	x	x	6	**10**	x	x	x	7	1	2		
12/17/11	e		5	9	4	x	x	6	**10**	x	x	x	7	1	x	x	x	8	2	3		
	l		6	10	5	x	x	7	1	x	x	x	8	2	x	x	x	9	3	4		
12/18/11	e		7	1	6	x	x	8	*2*	x	x	x	9	*3*	x	x	x	10	4	5		
	l		8	2	7	x	x	9	*3*	x	x	x	10	*4*	x	x	x	1	5	6		
12/19/11	e		9	3	8	x	x	10	*4*	x	x	x	1	*5*	x	x	x	2	6	7		
	l		10	4	9	x	x	1	*5*	x	x	x	2	*6*	x	x	x	3	7	8		
12/20/11	e		1	*5*	10	x	x	2	*6*	x	x	x	3	*7*	x	x	x	4	8	*9*		
	l		2	*6*	1	x	x	3	*7*	x	x	x	4	*8*	x	x	x	5	9	*10*		
12/21/11	e		3	*7*	2	x	x	4	*8*	x	x	x	5	*9*	x	x	x	6	10	*1*		
	l		4	*8*	3	x	x	5	*9*	x	x	x	6	*10*	x	x	x	7	1	*2*		
12/22/11	e		5	*9*	4	x	x	6	10	x	x	x	7	*1*	x	x	x	8	2	3		
	l		6	10	5	x	x	7	*1*	x	x	x	8	*2*	x	x	x	9	3	*4*		
12/23/11	e		7	*1*	6	x	x	8	*2*	x	x	x	9	*3*	x	x	x	10	4	*5*		
	l		8	*2*	7	x	x	9	*3*	x	x	x	10	*4*	x	x	x	1	5	*6*		
12/24/11	e		9	*3*	8	x	x	10	*4*	x	x	x	1	*5*	x	x	x	2	6	7		

rows Dates and day parts. The day part is either *early* or *late*, so a date has two day parts.

columns Services and occasion slots.

cells Students. Students are represented as unique, sequential integers.

2.1 Input

The input to the algorithm consists of three parts:

2.1.1 Calendar

The first part is the *calendar*. In the above figure, the calendar is represented by the x-axis (the types of service occasions that students may be present at) and the y-axis (the dates and day parts, when those occasions take place).

Also included in the calendar are the number of students that may be present at any individual occasion (the **slots** of the occasion, displayed below the **type** span on the x-axis).

2.1.2 Students

The *number* of students that are to be distributed over the calendar is also passed as input to the algorithm.

2.1.3 The maximum goals for each service type

What is not shown in the figure is that the algorithm checks that no student exceeds the maximum number of participations defined for any service. In that case, allocation of that student is not done, even if the student is available at that time.

2.2 Output

The output of the algorithm is a distribution of the students over the calendar. That distribution is shown in the figure above as integers. That allocation is a potential schedule; however, depending on the student group service set (the goals defined for the number of participations for each service), the distribution may be discarded. If this happens, a messagebox will tell the user that the goals were not possible to fulfill, and the user will be asked to reduce the requirements for this particular student group.

2.3 What does the figure say?

Point a finger at any student (an integer in a table cell). Look left (all the way to the **date** and day **part**) to see *when* that student is supposed to do something. Backtrack right to the student. Look up (all the way to a service **type** span) to see *what* that student is supposed to do.

With a finger at a student, look up to see in what **slot** a student is placed in. Note though, from a human point of view, it does not matter what student gets what slot. That data is only used by the algorithm as not to overpopulate any occasion.

An **x** indicates that there isn't that many slots at that occasion.

The figure also shows why the algorithm works: if each student occurs at most once per row, there are not any collisions. (A collision is when a student is scheduled for two activities, that occur at the same time. This should never happen.)

2.4 A closer look

When reading this section, it is beneficial to keep an eye on the example figure. Although the algorithm is not complicated at all, understanding will come around ten times as fast, when simultaneously comparing text and figure.

The algorithm's start point is: the first student; the first slot of the first service type; on the first date, *early* (if available, else *late*). Early is obviously before late, and dates are sorted chronologically, as is intuitive. While the remaining components – the students, and the service types and their slots – are consistently enumerated (starting at one (1), incremented by one, so no gaps), those integers are just handlers for the algorithm to iterate – in effect, it doesn't matter what student gets number two or three, what service type is number one, etcetera.

This start point is the top-left cell in the figure, which shows the digit one.

From here, the algorithm progresses forward in time, by step of *day part.* If it is early, it gets *late*; if it is late, it gets *early the next day.* For each step, student assignment is made, and the student counter is incremented by one.

If the algorithm runs out of students, it simply loops back to the first (who has the digit one). In the example figure, there are ten students. Already in the first column, it is visible how the work student loops back to one.

This continues until time is up: either there is no "late" at that day, or, it is late, and there is no tomorrow. (The former is what happens in the last row in the figure.) When this happens, again, the algorithm loops back, this time, in *time*, to where it started off. The difference is, it also *increments the service type by one.*

Then, the algorithm simply do everything again, and it will proceed until it can't increment the service type anymore, because it is at the very last service type. At that point, instead, it loops back the service type to one, and *increments the slot number*. Note that, once that has been done, it is back at incrementing service type, not slot, until the next time this happens.

If the algorithm can't increment the slot number, because it is at the very last slot number, there is nothing to do: if there isn't a feasible schedule at this point, for the current configuration (and this algorithm), there will never be one.

2.5 Runtime technicalities

Student allocations in the figure are presented in different font styles. Also, there is a row labeled **first** that is probable cryptic. This is metadata that relate how the algorithm is actually implemented in VBA-code.

2.5.1 The first row

Student allocation integers flow downward. The **first** indicator is simply a way for the computer to track where to start for each column. Once started, there is not much of a challenge: just increase one for each step down the ladder.

2.5.2 The font styles

The font styles indicate the *mode* of the algorithm, at the time that particular allocation was made.

ordinary digit – 1 – This assignment has been made to reach the *minimum* requirement of student 1 of the particular service type.

tilted digit – *1* – Ditto *maximum.* In the VBA-code, this is `max forward`.

bold digit – **1** – The algorithm previously refrained from making this allocation, as the minimum requirement for student 1 of this service type had been fulfilled. However, as, later on, there were (minimum) allocations of other students (at least one such case) in this column, the algorithm might as well backtrack and allocate student 1, as that opportunity had otherwise been wasted. (`max backward`)

underlined digit – $\underline{1}$ – If no minimum allocation has been done in an entire column, this column is moved out of action, so its presumed **first** can be used for better purposes in another column. (There is an example of this in the figure.) But, at a later point, that empty column was moved back to its original place and filled (for the maximum requirement, obviously, otherwise none of this would have happened). (`max empty column`)

2.6 After the algorithm

After the algorithm is executed, a test subroutine is invoked to see if the minimum goals were reached for each student and service. A failure does not mean that the algorithm failed – rather, the goals were set too high (and this is communicated).

3 Constraint programming

Another way to solve this problem would have been to use the methods and tools of a branch of computing known as *constraint programming.* To do so would have meant a less “hackish” and frustrating implantation process; as for the end result, it would have been more robust, and less error-prone.

The foundation of constraint programming is that a solution to a problem is acquired not by algorithmic, step-by-step computation, but by *search.* Instead of telling the computer “do this, do that, and when you’re done, we have a solution”, a *model* is made that describes the problem scope; then, *constraints* are posted that define what must hold for a *solution*; finally, the computer invokes the generic search algorithm and, depending on the problem (and the problem instance), delivers a solution (or, if desired, a solution set). [14, p. 18]

As compared to the traditional algorithm approach, this method is more flexible – if requirements were to change (which is very likely), instead of re-writing the algorithm, the programmer would simply change a constraint (for example, if the student participation requirement for some service was reduced). Or, on the flip side of it, if the context was to change – say, it became possible to carry out service at weekends – again, this would not require an algorithm rewrite. Only, this time around, rather than changing the constraints, the programmer would change the model (in effect, add a Saturday instance to the calendar data structure).

It is less demanding (no algorithm design, implementation, and maintenance), more robust (adding a constraint won’t mess up what has been setup thus far), and less error-prone (without the re-writes, there isn’t any need to test for, and correct, new bugs, possibly introducing new bugs, and so on).

The most common example, often used to illustrate the principles of a constraint problem, is the Sudoku puzzle. There is a square board, and a span of integers ($[1, s]$, where s is the side of the board). From the integer pool, the player places single digits at each square of the board. This is the model of the constraint problem. Then, a set of constraints defines what distributions of digits over the squares are Sudoku solutions.

This is all there is to it: there is no step-by-step algorithm what to do, and every possible solution (if there are many) are (in this case) as valuable, provided they are solutions: they are in compliance with the stated constraints, *and* do not violate the model. (Violating the model is, once implemented, impossible: this is one reason the model is so important – it can be used to impose implicit constraints, which will always hold but won't require computer resources.)

Here, I'd like to mention that the skilled constraint programmer may apply several advanced methods to either reduce the search space, or impose a more efficient search. Search space is represented as a search tree, so branching is a key factor, as is traversal. The interaction between constraints, and between constraints and the model, as well as the quality of the model itself – everything matters to the outcome, so, although in general much easier than explicit algorithms, at a certain level of complexity, even constraint programming will pose a challenge. [4, p. 352] This could be compared to SQL: with a sound database (the model), retrieving information is child's play. However, even so, given enough data, and complicated enough queries, queries have to be optimized in order to get acceptable runtimes.

To summarize, constraint programming, from a user (not developer) perspective, focuses on the two *endpoints*: the model (the start) states formally what is, and what can be done; the constraints (the end) state the definition of a solution.

Constraint programming if often made available as a *module* to an established, non-CP programming language. But note: that language doesn't have to be declarative — in fact, that's what the module hopes to bring to it. However, it helps if the language has OO support. C++, for example, puts much more weight on modeling than does C. For sure, this will facilitate incorporating a constraint programming module, that itself is model-oriented (even more so than C++). [11, p. 439]

In sharp contrast to our solution to this brand of scheduling, as users of a constraint programming tool, we'd not concern ourselves at all with the actual implementation of the algorithm. Instead, search algorithms have been generalized (and thus thoroughly tested) to encompass a huge domain of problems, including, I think, this one. (At the time we started this project, I was less familiar with constraint programming: I didn't make the case that we should use it.)

4 Database theory

The MS Access' database belongs to the family of *relational* databases. A relational database is a database in which data is stored flat, in tables (two-dimensional matrices made up of rows and columns, although with some special properties). [13, chapters 5 and 11]

A table is called a *relation* in database theory (though in MS Access, a table is still called a **table**); likewise, a column is an *attribute* (a **field**), and a row is a *tuple* (a **record**).

The tables of a relational database each describe different entity *types.* If something doesn't have such a table prototype, as for the database model, it doesn't exist. Each entity relates to other entities that are defined in the same way. This is an intuitive model that corresponds to the way humans perceive the outside world.

Although relational database tables are made up of columns *and* rows once populated with data, it is only the composition of columns that actually defines a table, and thus any possible entity instance. Just as the tables, the columns have names; they also have metadata such as data type, references to other tables, and more, to accompany them.

The rows of the tables are not rows in the sense that the columns are columns; rather, they are bare placeholders for data (whereas the columns actually set the domains for such data). That is, no metadata with respect to the model accompany the rows.

4.1 History, purpose, and SQL

The father of the database relational model — to organize data in flat tables, that are interconnectable — is Edgar Codd. [7, pp. 377–387] Back then, there were several solutions adrift as to how to store, modify and retrieve data, and typically each such solution had its own adjacent data definition/manipulation language. The simplicity of the relational model, and in particular the flat data representation, paved the way for SQL as the industry's de facto standard language for database interaction.

4.2 Normalization

Database normalization specifies a ladder of *normal forms.* Each form imposes constraints on a single table. The forms are cumulative in the sense that, say, the third normal form (**3NF**) implies the constraints of the preceding normal forms down the ladder (namely, **1NF** and **2NF**). [12, pp. 233, 237]

The constraints of the normal forms typically relate to the interdependencies of the table attributes, and, especially, the set of attributes that constitutes the primary and secondary keys of the table.

For a database to adhere to a certain normal form, all of its tables must be (at least) in that form. But, even though normal forms are cumulative, normalization is not done in steps: the **3NF** is usually the point where there is minimal data redundancy, and interacting with such a database does not suffer from insertion, modification, or deletion anomalies — in effect, the database is considered "normalized" at that point — and, if the database designer can get there at the first attempt, all the better!

At first glance, it is tempting to compare database normalization with the *standards* of a programming language or a class of operating systems. It would seem that both normalization and such standards attempt to shape up a technology that, left alone in the wild, has spun out of hands. However, this similarity is superficial. Normalization is much more abstract and theoretical in nature, but has nonetheless had a huge practical impact. Standards, on the other hand, are often pragmatic: for example, several brands of a programming language are present, and the standard attempts to unify them, without too much provoking any of the factions (an example of this is, curiously, SQL [13, p. 169]). In other cases, a "new" standard may be the most inclined not to break any existing code that "adheres" to the previous standard (where in fact, very few care about either standard one bit: an example of this is C [5, chapter 12]).

4.3 My experience

During the course of this project, I didn't feel the need for normalization in the sense that I experienced anomalies interacting with the database. Rather, dealing with the database tables, I was often confused as to what information was expressed by a particular table. I often got the impression that tables were used simply as placeholders for data, rather than expressing some piece of information that could be stated explicitly. Although perhaps not the most pressing goal of normalization, normalization almost always implies dividing large tables into smaller, and this would have made for a more streamlined and shaped-up database.

5 AI, and greedy search

A *greedy* algorithm is a search algorithm with applications not the least in the field of AI, where it is referred to as *hill-climbing.* [10, p. 127] Like many search algorithms, the greedy algorithm traverses a search tree where nodes represent partial or complete solutions. In search of a particular solution (or one that is better than the current), the algorithm makes moves from node to node: *what* move to make (in effect, how to make that decision) is to a great extent what distinguishes one search algorithm from another.

As for the greedy algorithm, it employs a system of rank (or score): of its alternatives, it moves to the node with the highest rank. In general, such an algorithm performs well, but in some cases it could fail, or end up with a suboptimal solution, because of its limited view. It gets stuck at a *local maximum*, and won't reach (see) the global, optimal solution: a belt of lower-rank nodes lies between the optimal solution and its current, local maximum node. [8, chapter 16]

5.1 A greedy implementation

Let's assume the same datastructures — the calendar — and the same student representation (sequential, distinct integers, starting at 1). Also, assume the same solution formalization: associations between *students* and calendar (service) occasion *slots.* Third, assume the slots are simply iterated: only now, for each slot, a greedy choice is made as to what student should fill that slot. Now, how could that greedy algorithm have been implemented?

The first step would be to eliminate (from the entire group student set) all students that are already doing something else at the time of the occasion.

For each remaining (available) student, the greedy algorithm would calculate a score based on certain conditions. For example, each students could start with at score of 100. Then,

(1)	the student is assigned the service [at least once]	-7
(2)	(for each such assignment)	-2
(3)	the student is assigned the *minimum* requirement	-20
(4)	(for each assignment above that)	-5
(5)	the student is assigned the *maximum* requirement	*disqualify*
(6)	the student is assigned one more shift today	-10
(7)	for each shift the same *week*, prior to today	-4
(8)	ditto *month*, prior to this week	-2

The greedy algorithm would pick the student with the highest score. After that, everything would happen again, only for the next slot (and so on).

This is an AI, greedy, search algorithm. It is **AI**, since the score system emulates (or, rather, *implements*, with 100% consistency, and 0% flexibility) how a human would think:

(A)	"All students should at least *try* each service, even if there isn't time to acquire any real skill."	(1)[1]
(B)	"Students should get the same quantity of practice."	(2) (4)
(C)	"Students should acquire skills, experience, and confidence."	(3)
(D)	"Students shouldn't do the same thing over and over again, instead, it is preferable they recuperate and/or assimilate what they have learned."	(5)
(E)	"Individual student activity should be distributed, so the student is given time to digest his or her experience, discussing it with fellow students, and so on, to be able to perform better the next time around — also, that will make for a more robust, persistent experience, that he or she won't instantly forget."	(6) (7) (8)

1. This column indicates how a (human) thought has been formalized into a building block of the AI algorithm (as shown in the previous listing). However, as for the AI algorithm's end result, it was my intention that *all* seven point deductions (as well as the one disqualification) more or less should contribute to all five goals.

It is **greedy**: for each occasion, it picks the student with the *highest* score, *without considering* how the next slots are to be filled.

Last, it is **search** as it selects one candidate from a set of possible students.

Another AI way to solve this problem would be to define goals for what makes up a solution, and then define a score system to rank the solutions. The AI would train on randomized input sets: a high score would, in the AI mind, make the steps just taken more probable to lead to a good solution, and thus more likely to be employed (and the other way around for poor schedules).

5.2 The iterative algorithm (of this project)

The iterative algorithm operates on the calendar slots on three nested levels: the innermost level is *time*, incremented by day part; the middle level is *service type*, iterated in the order that the services were setup in the group service set; last, the outermost level is *slot number*, incremented by 1.

Iteration covers all slots: virtually, it transforms the calendar representation into a *sequence* of slots. The first student–slot assignment made is student number 1, who gets the first slot. Thereafter, for any slot, allocation is made to the student with number $s+1$, if the student with number s was (or could have been) assigned the *preceding* slot. If there are only s students, iteration starts anew, and student number 1 is assigned.

So, there isn't any search; nor are there any decisions that resembles human (or machine) problem solving. Although assignments are made by increasing student numbers $(1, 2, 3, ...)$, that is not greed: those digits does not reflect any properties of the students (most certainly, they do not indicate rank). Instead, the digits are tokens to keep the students apart: the students could have been designated $A, B, C, ...$ just as well, the only requirement being that the tokens are incrementable.

What's more, the algorithm actually does consider the future. Whenever a student has surpassed the maximum requirement, [s]he is not assigned a service on that occasion. This is no different from the AI algorithm. However, after the AI algorithm disqualifies the student, *it picks another:* the one with the highest score, not disqualified. The iterative algorithm cannot do this; that would lead to collisions later on. So, the slot is left empty.

The iterative algorithm is not an AI algorithm. It does not have any personality or behaviour that could be compared to how a human would cope with any one instance of this problem. The algorithm is the result of a programmer (or perhaps even more so, a mathematician or logician) solving a *generic* problem, by observing certain patterns, understanding them enough to make them work to his or her advantage.

6 Testing

Testing is often put forward as a way to find bugs at an early stage. It requires little effort and may pay off huge: not having to retract shipped copies, or publish patches, and so on. [1]

Even more so, this holds when testing is compared to *formal verification*, the more scientific (rather than engineering) approach, where a huge analytic effort produces a result that is often hard to grasp. By contrast, testing will reveal bugs that can be fixed instantly upon detection. Also, testing tests the real thing. Formal verification requires a model which may be wrong. It only proves that the model is correct, not that the application is.

The key aspect to testing is to actually do it. Already at that point, there is a huge advantage compared to not testing at all. Beyond that, it is uncertain that more refined test methods actually produce better results (in effect, find bugs that slipped through the net the first time around). There is also the *volume* factor: if a simple test method can be employed massively, it is probably preferable to a more refined method, that can only cover patches of the test field.

If a plethora of test methods are employed, each test should have an explicit purpose, and/or a distinct test scope. For example, one test could enforce that every line of code of the application is executed at least once. Depending on the execution flow, this could require several invocations. To achieve this, we again benefit from modular code: each function should be called, and the return value fetched; each procedure should be invoked and brought to its conclusion; and each interface should be covered in full, including optional parameters. Beneath that, it gets more fine-grained, as the control logic — iteration and branching — must be taken into account. Although probably not necessary, a directed, cyclic graph could illustrate the execution logic and flow. This test — that every line of code can execute without an error — is intended to track bugs that are not syntactical, and thus will compile, but once executed, will either bring a halt to the action *or* (worse) further down the road produce a bogus result.

Another way to test focuses on input data. This method is tangential to the notion of software as a black box mapping inputs to outputs. If volume testing is possible, brute force with random input data shouldn't be shunned at. Input data must be valid, but mustn't necessarily make sense: with volume, in time, what makes sense will be tested as well.

If a more sensible approach is desired, testing could be based on input cases, that are setup manually. With the student data of this student database, such cases could be the empty set (of zero students), a singe student, all students, and so on. Cases that might strike as unrealistic or even impossible should not be avoided, as long as they are valid: on the contrary, those border cases can reveal shortcomings that sensible inputs cannot, and indeed, the purpose of testing is to *break* the examined application, thus revealing the bug that made it possible.

As for this project, we didn't do any intelligent testing. Whenever the execution of the algorithm was a success, we assumed that the schedule made sense.

6.1 Good habits

Of course, better than finding and correcting bugs is never to introduce them in the first place. Although near-impossible to achieve in full, sound code and work habits will significantly improve the quality of any piece of software.

6.1.1 Code looks

Sound code has as consistent style. This includes naming, indentation, and spacing, and also applies to comments (if used). Such code is a lot more readable, which will benefit not only future readers, but also the person who is writing the code, at that selfsame moment, as well as in the later, debug phase.

As a specific example, consider the implicit constructions that are possible in some programming languages: for example, in C, they programmer can disperse with the curly brackets that delimit scope, if the scope is but a single line of code. While compact code is beneficial to the trained eye, such usage is error-prone if the code is ever to be extended or modified (which is very likely).

A consistent style will also increase productivity, as it will reduce the amount of thinking when actually typing (as it is already clear *how* to write things, the remaining concern is *what* to write). Consistency will reduce the number of lookups into other sections of the code, as the programmer knows in what manner variables, functions/procedures, and so on, are named: if the programmer knows him- or herself, his or her guess as to the name will in most cases be correct. Such lookups are disruptive: even more than the actual time loss to do it, they will drain the programmer's energy as they imply a visual and mental halt and re-orientation.

6.1.2 Code habits

If the programmer finds him- or herself habitually looking up things in different sections of the code, this might indicate poor design and in particular lack of modularity. Instead, the code should be the unification of self-sufficient, but communicating, components, with clear interfaces (function/procedure names that semantically describe what is done, with just a few parameters each, all likewisely named). Apart from being much easier to actually write than the other way around, obviously, such code is much less bug-prone. If bugs nonetheless are introduced, detecting and correcting them — without the introduction of new bugs — will be much easier if the code is modular.

To achieve a clear, modular code may be done in a classroom, where UML boxes and arrows are drawn on whiteboards. However, there is another approach, which in my experience is superior. Instead of trying to grasp the big picture at an early stage, I build any piece of software bottom-up. If I get the details correct, once all components are connected, the overall result will be to my liking as well.

For example, if I find that a stack of preconditions accompany a function or procedure, I change the data type of the troublesome parameter to a user-defined type, that way making preconditions implicit by an explicit domain infringement, that is defined but *once*. If there is an obvious domain infringement, but for practical reasons it is desirable to stick to a built-in data type (for example, a signed real for a digit expressing the temperature in Celsius, although anything below absolute zero won't make sense) such input should be validated first thing in the function or procedure body. Such a check requires close-to zero coding effort, and its drain on run-time performance will be unnoticable; on the other hand, it could expose a bug that would produce bogus data and be very hard to find.

Code is written by humans, and read by humans. The compiler then translates the code, in steps, to machine instructions: near-unreadable, even to programmers. This should encourage programmers to write code in the most *human* way as possible. Naming, including interfaces, should reflect (even describe) *what* is done to *what* pieces of information, not *how* this is achieved from a programming language or otherwise machine point of view.

For example, say that a reference to a week day is needed. Here, it is tempting to use an integer for the seven days of a any calendar week. However, that is machine thinking. A programmer will start to wonder: "Is that 0 to 6, or 1 to 7?" An American might think that the week begins with Sunday, while a Swede won't doubt that the week begins with Monday. Here, an enumeration with symbolic constants is a more robust solution.

Speaking of constants, preprocessor constants should be avoided as they disable type checking; symbolic, typed constants, on the other hand, should be used excessively: never put the same digit (that quantifies the same thing) twice is any piece of code. Several instances of hard-coded data will make any change risky: at best, it will take time to find all instances; at worst, not finding every occurrence will make for inconsistent data, or, a sloppy search-and-replace can change some other data item, that denotes something completely different, because it at the time (by chance) held an identical (but unrelated) numerical value.

7 The code already there

The code at place when I started to work on this project was not professional: the person who wrote it did not have a programmer's education or any field experience to make up for it. That said, I don't want to be dismissive because that person probably had some altogether different area of expertise. [s]he should not be blamed, but rather the person who assigned him or her the job.

During this project, I visited the hospital on countless occasions, and it is my clear impression that people there were stressed out. One of the reasons is that, instead of carrying out research and educating within their respective fields, they were fiddling with computer systems. Although their computer systems were more or less dysfunctional, even more so were their management of human resources.

7.1 Hungarian notation

As an example, they use Hungarian notation[2] for naming variables and GUI elements. Thus, a string to hold a greeting is not named `greeting`, but `strGreeting`. This is machine, not human thinking: besides, VBA is typed, so a typecheck is performed anyway. Especially to those not used to it, Hungarian notation reduces code readability and is disruptive to workflow.

Decades ago, when computing wasn't strong enough to support typed programming languages, type errors were everywhere: perhaps *then* it was beneficial to use Hungarian notation. Note though, this should not be confused with today's dynamically typed programming languages: they are so *by design* (for whatever reason), and to use Hungarian notation would probably be to "reinvent the type".

In MS Access (and the Visual Basic languages), I think the rationale for using Hungarian notation actually has nothing to do with type. Rather, the *forms* of both MS Access and VB projects tend to hold lots of GUI elements, each of which has a global name. When writing code, those names could be hard to remember; what's more, as they aren't declared in the code (but in the "GUI GUI"), they aren't readily visible. On the other hand, most often the programmer will remember their *types*: button, text field, etcetera. If

2. Wikipedia, *Hungarian Notation*, accessed Mars 4, 2013: `https://en.wikipedia.org/wiki/Hungarian_notation`

so, and if Hungarian notation is used, the programmer could simply (for a button) type `cmd`, and the autocomplete function of the MS Access or VB editor would present viable alternatives. (As for me, I think autocomplete popups are disruptive and I disabled them first thing.)

7.2 Visual Basic for applications (VBA)

If a programmer is asked what [s]he thinks of VBA, [s]he will often express dislike and even ridicule. Actual experience is not a prerequisite for this from-the-holster reaction. But, as for me, if I only had to assess the *programming language* VBA (leaving aside the rest of MS Access) I'd say this reputation is unfair. Probably, those preconceived notions are due to legacy implementations of Basic, which lacked a lot of features that had to be made up for by spaghetti code, virtually making the code impossible to maintain, reuse, or extend. However, those features are available in VBA: explicit `goto` is not needed, and it is possible to write modular code. Also, for those inclined to C or Lisp, the syntax of Basic (including that of VBA) is cumbersome: while true, that's aesthetics. Last, VBA is Microsoft, it is proprietary and close source, and it is not portable. Personally, I wouldn't recommend VBA to anybody (for those and other reasons), but the programming language is not the reason (or a valid excuse) for any crude application. Also, VBA is well integrated — almost *too* well so — with the UI editor and the database parts of MS Access.

7.2.1 The MS Access editor

Instead, my main concern is the built-in *editor* used to write VBA code. That editor is somewhat configurable, but not programmable; compared to a professional editor, it is a toy. Mastering a professional editor, and to fully configure it down to one's last preference, is a process of hours of work and years of active use. But, once done (or proximately so), degrading to the likes of the MS Access editor is to experience a distinct, tactical deficiency.

7.2.2 Shortcuts

What I missed the most from my regular editor (in very stiff competition) were numerous shortcuts to navigate the cursor through code: `forward-word`, `backward-word`, `beginning-of-line`, `end-of-line`, and so on. Those shortcuts have since long entered my muscle memory; I invoke them without thinking. [6, pp. 20, 22] Although a small subset of the shortcuts I use is actually present in the MS Access editor, those are bound to different keys: even though I adapted, in particular, it annoyed me to have to *reach* for the cursor keys instead of the much better solution that involves keys in the middle of the keyboard.

Some functionality that I'm used to from my regular editor is indeed present in the MS Access editor, only *not entirely so*, or in a somewhat other fashion (typically, those were bound to different shortcuts, as well). This made for confusion and misunderstandings. In general, nobody should be made to use any other than his or her preferred tools, and even more so in computing where portable and tool-independent alternatives are everywhere.

7.2.3 Editor looks

Visually, the MS Access editor is fairly configurable. The Windows color scheme can be reversed to produce a black background. A black background will reduce eye strain, as less light enters the eyes. But, unlike the light of the sun, the light from a monitor conveys information that must be decoded in/by the brain. If a programmer experience eye discomfort (or even pain) from staring into a monitor for too long (while maintaining a high degree of concentration), [s]he is virtually incapable of carrying out any work. Also, the use of large, clear fonts, as well as the elimination of any visual noise (redundant decorations, or a blinking cursor) will help. Last, the use of a projector (instead of a monitor) is a huge improvement, due to the increased font size and eye distance. Also, straight, upright ergonomics will "fool" the body and mind of the programmer that [s]he is alert, aware, and confident.

7.2.3.1 Highlighting

The purpose of syntax highlighting is to reduce *reading*, and increase *seeing.*

Highlighting can be defined as regular expressions, and implemented in much the same way as the front-end of a programming language compiler, with a *lexical analyzer*. [3, chapter 12]

The most important aspect of highlighting is to actually have it, in the editor. Once there, it can be refined, but the gain of any such second, third, and fourth steps won't be comparable to the first, to get it.

How will an individual color highlight of a word affect the human eye's ability to make out that word? An experiment should include the black background color, as it in large is the contrast that makes a word stick out.

Also, to what extent is the eye capable to appreciate different shades of the same color? For example, several shades of green can be used to convey lots of information, but it could just as well be either unnoticable, or worse, confusing (the programmer expects green to be one thing, only sometimes, "the same" green is something else).

Here, note that the eye is better at detecting green than red and blue, due to a more sensitive cone for green (than those for red and blue) at the back of the eye. [2, pp. 18, 19] Perhaps, this is a measure of evolution, that helped paleo-men in search for food.

There is also the human emotional response to a color. Red sticks out, but perhaps overly so – it would indicate (in a programming setting) an error, or some danger.

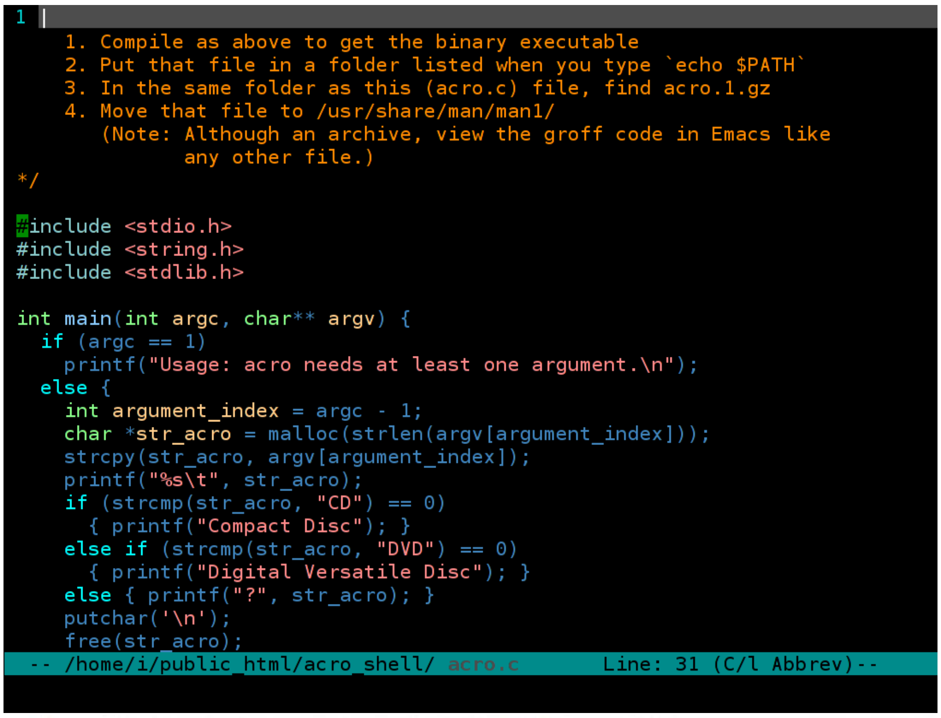
For sure, comments should have a color that puts them apart from everything else (that is, code). To put comments in a medium-bright red would probably achieve this, as the code highlight probably doesn't use red that much. However, apart from the danger of making the comments stick out to much, and emotionally conveying errors or dangers, there is the habit of some programmers to comment out code. [9, p. 86] So, the comment highlight mustn't only stick out, it must also be *less* visible (or less instantly so) than normal code (that isn't commented out), because commented out code is intently put out of action, and shouldn't be considered until it, possibly, is brought back to life at a later stage.

As a rule of thumb, highlight (mainly colors, but also including creative use of fonts, sizes, and styles – for example, the **boldface**) should not make a single word stick out in a paragraph (or code block). Intuitively, the reader should look at the beginning of any paragraph, as if [s]he were to read it. However, when the reader naturally (by reading) gets to a particular word, sensible formatting will convey extra information and make for a more pleasant, less bumpy, and thus more efficient reading experience.

Highlighting a programming language should focus on what are keywords of the programming language, and what are data, branching conditions, etcetera, that is specific to the current application. That way, the programmer will instantly see what to possibly change — for example, in search of a bug — because, obviously, [s]he isn't going to change phrases that are fixed constructs of the programming language itself. Only rare constructs should be made to really stick out: for example, preprocessor directives in C.

7.2.3.2 Comparison

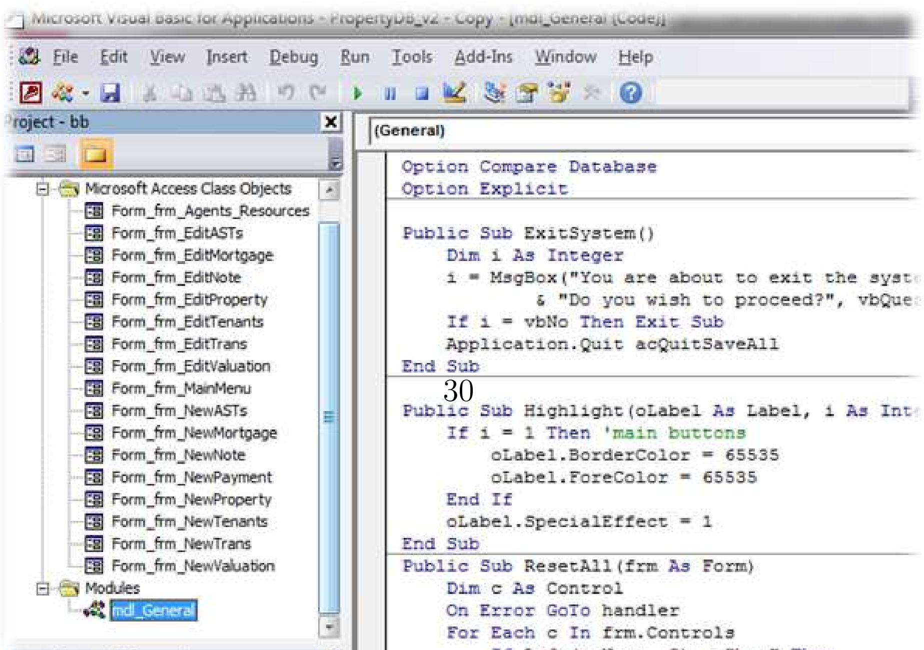
I include below a screenshot of the optimal programming interface I found (or, more precisely, setup), so far. Note the minimalist, tabbed terminal; the black background; the large, clear, and monospace font; the highlighting; the lack of redundant graphical elements (including scrollbars, and the mouse pointer, as the mouse isn't used); and the block, non-blinking cursor. Although I could make for a pleasant ride in MS Access, there is no way I (or anyone else) could make it look like the screenshot below. For comparison, I include a screenshot from the MS Access 2010 editor. While I used the 2003 flavour, the GUI looked almost identical.



```
1 |
    1. Compile as above to get the binary executable
    2. Put that file in a folder listed when you type `echo $PATH`
    3. In the same folder as this (acro.c) file, find acro.1.gz
    4. Move that file to /usr/share/man/man1/
       (Note: Although an archive, view the groff code in Emacs like
              any other file.)
*/

#include <stdio.h>
#include <string.h>
#include <stdlib.h>

int main(int argc, char** argv) {
  if (argc == 1)
    printf("Usage: acro needs at least one argument.\n");
  else {
    int argument_index = argc - 1;
    char *str_acro = malloc(strlen(argv[argument_index]));
    strcpy(str_acro, argv[argument_index]);
    printf("%s\t", str_acro);
    if (strcmp(str_acro, "CD") == 0)
      { printf("Compact Disc"); }
    else if (strcmp(str_acro, "DVD") == 0)
      { printf("Digital Versatile Disc"); }
    else { printf("?", str_acro); }
    putchar('\n');
    free(str_acro);
 -- /home/i/public_html/acro_shell/ acro.c        Line: 31 (C/l Abbrev)--
```


Microsoft Visual Basic for Applications - PropertyDB_v2 - Copy - [mdl_General (Code)]
File
Edit
View
Insert
Debug
Run
Tools
Add-Ins
Window
Help
roject - bb
(General)

```
Microsoft Access Class Objects
    Form_frm_Agents_Resources
    Form_frm_EditASTs
    Form_frm_EditMortgage
    Form_frm_EditNote
    Form_frm_EditProperty
    Form_frm_EditTenants
    Form_frm_EditTrans
    Form_frm_EditValuation
    Form_frm_MainMenu
    Form_frm_NewASTs
    Form_frm_NewMortgage
    Form_frm_NewNote
    Form_frm_NewPayment
    Form_frm_NewProperty
    Form_frm_NewTenants
    Form_frm_NewTrans
    Form_frm_NewValuation
Modules
    mdl_General
```

```
Option Compare Database
Option Explicit

Public Sub ExitSystem()
    Dim i As Integer
    i = MsgBox("You are about to exit the syst
            & "Do you wish to proceed?", vbQue
    If i = vbNo Then Exit Sub
    Application.Quit acQuitSaveAll
End Sub

Public Sub Highlight(oLabel As Label, i As Int
    If i = 1 Then 'main buttons
        oLabel.BorderColor = 65535
        oLabel.ForeColor = 65535
    End If
    oLabel.SpecialEffect = 1
End Sub
Public Sub ResetAll(frm As Form)
    Dim c As Control
    On Error GoTo handler
    For Each c In frm.Controls
```

8 A note on modularity: MVC

When developing any computer system, it is often an advantage to isolate the UI from the data, and the data from the execution logic (whatever the program does to the data). Intuitively, this is not hard to grasp: the UI fetches the user's intentions, and passes them on; the computation part of the program alters the data accordingly. When done, the UI is notified there has been a change to the data, and it assesses the data anew to reflect those changes.

Depending on the UI (and the overall system), the UI may show actual data, and/or disable/enable GUI elements (like buttons) to reflect the new state. In most cases, the UI can simply reassess the whole data set and redraw whatever information and interaction options it presents: from an implementation point of view, it is easier to have the UI concerned with *states*, and not the transitions in between. Conveniently, this will be transparent to the user, who will only take notice of the subset of information that has changed in response to his or her most recent action.

As for the computation logic (the algorithms to add, modify, and delete data items), it should present an interface to the UI based on procedures, whose names in plain English semantically describe how they affect the data (or rather, the information the data represents). That way, the programmer can setup the UI to hook onto those procedures with interactive GUI elements.

Last, the data should only be concerned with itself: it should specify the data types of data items, and, when appropriate, further infringe their domains. Domains may be expressed extansionally (lists with all possible values) or as constraints: for example, a signed decimal number (a real) to express temperature in Celsius may be stated invalid for values less than -273.15. Here, depending on the RDBMS (or the otherwise underlying framework), if a UI textbox is hooked to such a temperature data item, and the user attempts to set the temperature to, say, -300, this overstep can be communicated immediately to the user, with no setup whatsoever required from the programmer.

For those interested in Computer Science theory, this trinity is a brand within the modular paradigm: it is called *Model-View-Control*, or MVC for short. During my work, I can't say I adhered to this school with anything but intuitive rigour. However, I was aware of what problems might arise: in the manual part of this report, I mention whenever poor modularity is obvious (almost all such cases were hard-coded, and/or “hard-layouted” database–UI intersections). Again, those are only the obvious breaches; were anyone to extend or modify my work, I wouldn’t be astonished if they encountered unexpected problems, wholly or in part because of poor modularity.

An even worse issue (that is more plenty, and rooted) is all the **recordsets** used as intermediate representation for the database tables, in the VBA code. In the relational data model, in terms of their data, relations (or tables) are *sets* of tuples. By definition, sets are not sorted: $\{a, b\} = \{b, a\}$. But, the algorithm that I implemented is all about iteration on *sequences*. So, instead of exclusively using SQL to interact the database, I often resorted to the VBA recordset datatype, including data extraction, explicit testing, and more. Problem is, this data migration back and forth between different representations makes for an error-prone system, whose behaviour is difficult to survey without extensive testing. True, there is explicit iteration in SQL (at least, in some SQL dialects): the use of **cursors**. However, in general, I don’t consider cursors a that much better solution than my recordset workaround, and in one respect actually worse, because SQL should be declarative and data oriented, and not procedural (at least not habitually so).

9 The UI

9.1 How often will the UI be used?

The importance of a UI is to a great extent proportional to how often the application (and thus its UI) is used.

If the application is used every day, the UI should just be a bulletin board of text fields and buttons (all with keyboard shortcuts) to invoke the application's services. In such a case, it is important that no time is wasted switching back and forth between windows (computer delay to re-draw, human delay to re-orient). There is no need for any such visual and/or logical groupings: as the application is used every day, the user will be intimately familiar with it after but a few days. Instead, the GUI elements should be placed wherever they fit the best in terms of *space*, so as to be able to show as many of them as possible at the same time, without intervention.

Also, if the UI is used every day, any verbosity (such as button labels) is not only a wasted effort, but counter-productive: a button should be labeled **B** instead of **Break** (or, even better, be given a sole symbol to express its purpose), in order to free space, and reduce reading. Although the user very soon won't actually *read* "Break", the word will still be there, distracting the user from, say, a text section with messages from the application, text that the user *is* supposed to read.

9.2 Critical section

If the situation in which the UI is used may impact people's health there shouldn't be a UI at all: there should be a one-to-one correspondence between a physical device and the desired computer (or otherwise machine) response. A fighter pilot doesn't have time to "think": his body and mind must be one; and, his aircraft must respond instantly to the lightest leaning to the throttle. Also, in the flight control tower — although their most important "tool" is probably the tower's *window* — during a crisis, any desired surplus information must be readily available (at most) by the turn of a head.

9.3 The “UI UI” of MS Access

The MS Access all-in-one development environment includes a drag-and-drop, WYSIWYG UI editor. This editor is similar to the one that accompanied Microsoft’s flavour of the Basic programming language for Windows development, Visual Basic.

It works as follows: In order to add a GUI element, such as a button, the UI designer chooses that item, then drops it wherever on the form [s]he sees fit. The button is automatically given a standard, unique name; however, as this name does not reflect the button’s intended functionality, it should be changed first thing. After that, the UI designer can change the button’s visual properties, such as size, alignment, etcetera; also, the UI designer sets the button label, and associates it with a keyboard shortcut. Last, double clicking on the button makes the UI editor jump into MS Access’ code mode, where a stub procedure has automatically been added to the form’s associated code. This procedure catches the event that is considered the most intuitive way a user can interact with the GUI element (for a button, a left click). The UI designer, suddenly elevated to programmer, can now write whatever code [s]he wishes to be invoked at such an event.

Although I personally prefer to work exclusively with text, this is certainly one way to do it: it makes for rapid development, and it is consistent with everything else in Windows, both visually and in terms of workflow. But, I can see a couple of drawbacks as well.

First, such a UI is obviously not in the least portable: the UI created will exclusively be a Windows one. No doubt, this is not considered a drawback by everyone, but on the contrary an asset.

Second: A GUI item is placed in a form on the basis of absolute position (for example, in points). Consequently, it won’t look the same on different computers (with different screen resolutions, window manager settings, and so on; not to mention if the window is sizable). Although I can think of workarounds — for example, make a text field as wide as the entire form, and then center-align the text — even so, this is nowhere near the abstract window toolkits available for mature programming languages (and even in web layout, with CSS).

Also, absolute layout is inflexible: if ever an element unsuspectingly has to be added to what the UI designer thought was a fixed layout, instead of logically inserting it ("in the middle, right next to..."), it has to be put there manually, in worst case requiring moving everything already there to make room.

Last, the level of integration in MS Access sometimes works to the disadvantage of the developer. On numerous occasions, I browsed the code in search of a bug, only to later find out that some GUI element was the source of the malfunction (or, some inconsistency relating to the database, for that matter). The traditional development cycle — write code, execute (test), debug, and so on — is a blurr in MS Access, and this made for a lot of debug time, not knowing even *where* to look for a bug. For sure, MS Access development actually encourages a cocktail of GUI, execution logic, and data: a *reversal* of the MVC approach.

9.4 My thoughts when I did it

This project is part of a larger database interaction system, which was already employed when I started. By and large, there weren't any options as to what tools to use or what interface to implement. Nonetheless, MS Access isn't inflexible (apart from not being portable), and I managed to put my stamp on the interface.

The number one advantage with a text based interface is that the user won't have to move his or her arm back and forth from the keyboard to use the mouse. The number two advantage is that the user won't have to take aim at graphical elements on the screen, squeezing his or her eyes to strike a target. Not having to do that will make for a pleasant, smooth mental-physical workflow, one that will maximize productivity and minimize frustration, thus minimizing the energy drain as work progresses.

As it happens, those two advantages may be somewhat replicated in a Windows GUI by taking a few simple steps. Most important, all buttons should have keyboard shortcuts.

One shortcut type is the combination of `Ctrl` and a letter key. Such a shortcut can be setup mnemonically, as in `Ctrl-F` for *find* (perhaps equivalent to clicking the button with a magnifying glass or a pair of binoculars as its icon). If a mnemonic command can't be found, any shortcut will do. If the application is used on a regular basis, it doesn't matter what is mnemonic and what is not: those keystrokes will soon enter the muscle memory, bypassing any such considerations in the brain.

The other shortcut type offers shortcuts that relate to the *text* of, say, a button. In MS Access, this is set up by placing an ampersand (`&`) before the desired letter. The button is thereafter reachable by use of the `Alt` key. Such shortcuts have the advantage that they are always visible to the user: they are self-documenting.

However, both shortcuts variants suffer from the MS Access menu, which takes precedence, thus making it impractical to use a large subset of possible shortcuts. I suspect it is possible to workaround this with event driven procedures in the form's associated code, thus bypassing the GUI altogether. However, I didn't put any effort on this as I didn't have that many buttons anyway.

Further, there is a unique, **default** status that can be assigned a button: in that case, clicking the button is equivalent to hitting the `Return` key on the keyboard. While it is intuitive, and common in many Windows (and other) applications, hitting `Return` (with the right little finger) is actually *inferior* to hitting an `Alt` keystroke (sliding the left thumb left, holding `Alt`, then hitting a letter key with the right hand), because in that case neither hand move from their correct positions. Also, the default assignment could be a source of confusion since it is not always clear what the "default" action is.

Last: The possibility to reach GUI elements by tapping the `Tab` key (to travel in the reverse direction, hold the right `Shift` key at the same time). This is especially useful when entering data in text fields. It is beneficial if the field text gets marked whenever the text field gets focus; that way, the user may just tab to the intended field, enter whatever text, and tab away.

In order for tabbing to work, a property called *tab stop* must be enabled for any element that should be reachable this way; also, a *tab order* must be set so the cursor won't jump all over the place but instead jump between the textboxes (and buttons) in an intuitive way. MS Access is capable of creating such an order based on the locations on the form of the interactive elements; however, this order may not be what the UI designer had in mind, in what case setting it up manually can be a tedious (but worthwhile) task.

Apart from setting up a keyboard interface, there wasn't much to it. The interface of this application is not intended to be used frequently, so there wasn't any need to cramp everything together. As a rule of thumb, I made each form do one thing; everything related to that thing is accessible in that selfsame form, and there isn't anything unrelated to that one task. Also, there is some automatic jumping between forms that was set up programatically. Such a feature is annoying when malfunctioning (switching forms when not supposed to, or switching to the wrong form), but I felt confident it would only happen when intended, and correctly whenever so.

10 The forms

My involvement in this project included setting up five forms: four to setup and execute the scheduling algorithm, and one to produce fine-looking documents showing the final schedules.

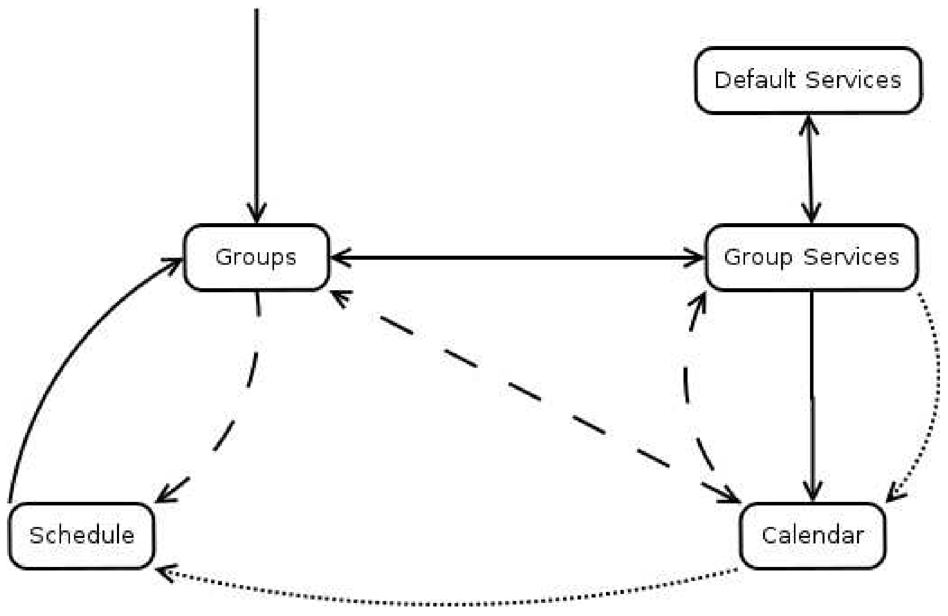
In MS Access, those documents are called *reports*; they are to be made available on course homepages, or handed out staff members, teachers, and students. In MS Access, there is a GUI editor that will both layout reports, and populate them by generating SQL-queries. So, the fifth form is actually an interface to such reports, which I also set up.

When we started this project, we believed that the algorithm presented the big challenge. But, although the algorithm indeed is the most advanced part of the project, once defined pen and paper, it did not take long to translate into code.

Instead, countless of hours were spent on the forms. The forms were not difficult in an analytic sense; rather, trouble arose making them work in a smooth fashion, all the while correcting bugs as they became apparent.

When running the application, only one form is visible (in fullscreen) at a time. All other forms are inaccessible to the user, because they are made invisible on all exit points. This is mostly to make the application foolproof, but also so that checks won't have to be made as to what is consistent in one form, with regards to data in another. It is like climbing a ladder: the steps below can't be broke, because then the climber couldn't have reached the current height.

The figure below shows user navigation between forms, implicit jumps, as well as data interaction.



balloon Forms

solid line The user can navigate between forms (typically, a button click in one form brings up the next in line)

dashed line The user can navigate between forms, under certain conditions:

1. From the **Group form**, it is possible to move to the **Calendar** *and* **Schedule forms** of a certain group *if such data exists*. If the algorithms hasn't concluded successfully for a group, that group doesn't have a schedule; as for the calendar, that is populated from the **Group services form**.

2. From the **Calendar form**, movement is triggered by the execution of the algorithm. If it was a *success*, the user is taken to the **Group form**; on *failure*, to the **Group service form**.

dotted line If changes to data are made in one form (at the arrow's base), data in another form (at the arrow's tip) could get obsolete and, if so, that data must be removed. There isn't any check to see if this actually happened: the whole range of (possibly) affected data is erased, to enforce consistency. This can cascade.

10.1 Name issue

A matter of confusion is the names of the forms and their associated tables. I nicknamed them by function, ignoring their database names.

form groups

name `F_PracticeStartsCreateIn`

table `Tv_PracticeStarts`

selection `CourseIdNo`

form group services

name `ass_services_new`

table `ass_course_specific_services`

selection `cmdShowServices`

form default services

name `ass_service_defaults`

table `ass_service_types`

selection none

form calendar

name `F_DesignPracticePositions`

table `Tp_ClinicalPassCalendar`

selection `PracticeStartNo`

form schedule

name `F_MakeClinicSchedules`

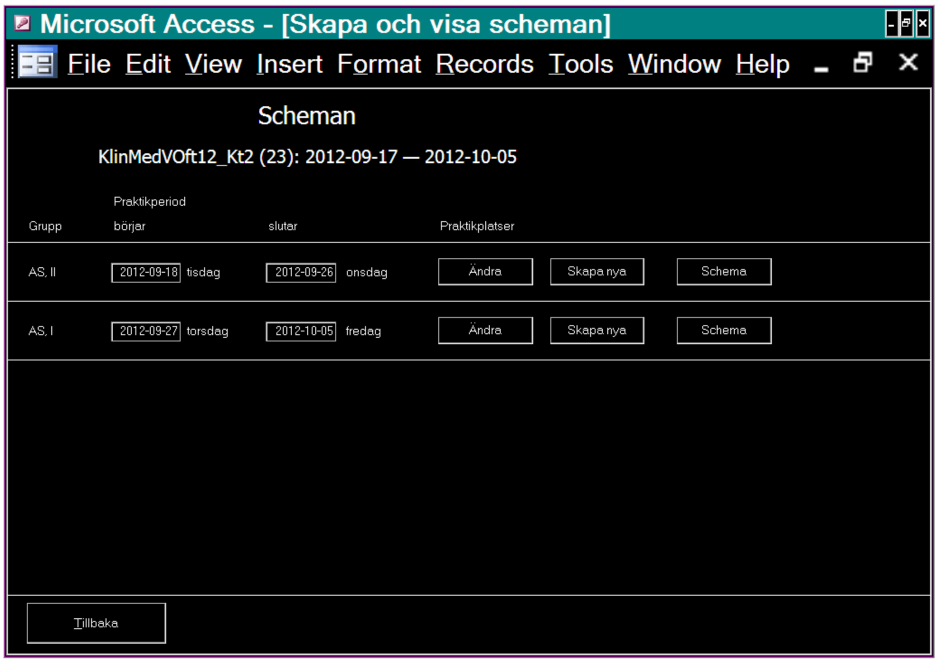
table `Tv_PracticeStarts` (read only)

selection `PracticeStartNo`

10.2 Form by form

Those are the forms:

10.2.1 Groups



Microsoft Access - [Skapa och visa scheman]
File Edit View Insert Format Records Tools Window Help

Scheman

KlinMedVOft12_Kt2 (23): 2012-09-17 — 2012-10-05

Grupp	Praktikperiod börjar		slutar		Praktikplatser		
AS, II	2012-09-18	tisdag	2012-09-26	onsdag	Ändra	Skapa nya	Schema
AS, I	2012-09-27	torsdag	2012-10-05	fredag	Ändra	Skapa nya	Schema

The first of the five forms that the user will see is the group form. Prior to displaying this form, a course has been selected by the user.

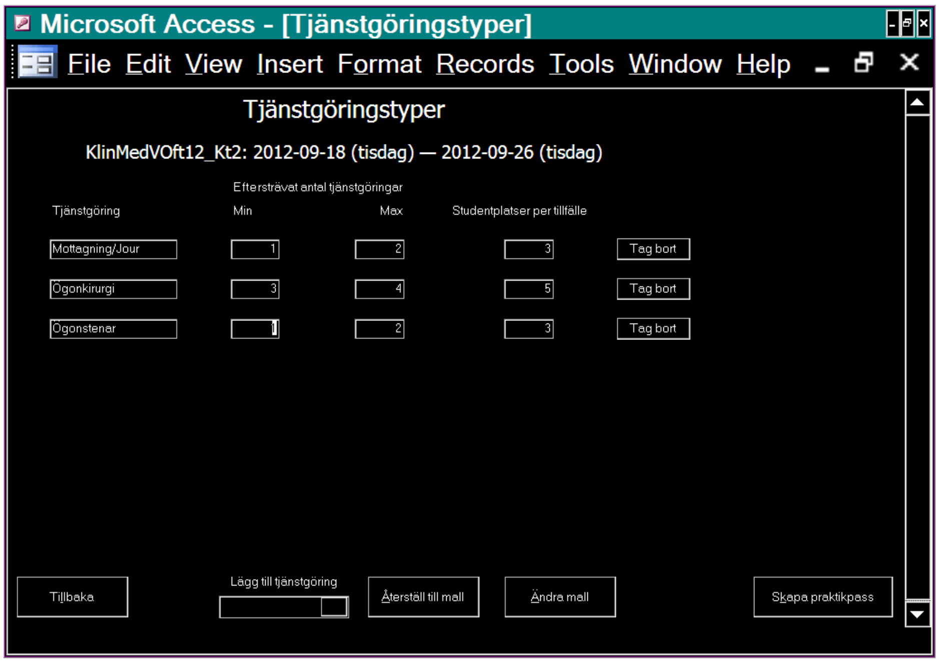
Two student groups take part in each course. The groups are distinguished by location.

Each group has distinct service *and* occasion sets. Schedules are group distinctive as well. Only, as part of the same course, the groups share the time frame of possible activity (in effect, their first and last date of their calendars).

Possible navigation paths from the groups form:

1. Edit services (if any; else, click to setup)
2. Edit calendar (if there is one; else, setup from services)
3. Show schedule (if there is one; else, run algorithm from calendar)

10.2.2 Group services


Microsoft Access - [Tjänstgöringstyper]
File Edit View Insert Format Records Tools Window Help
Tag bort
Tillbaka
Lägg till tjänstgöring
Återställ till mall
Ändra mall
Skapa praktikpass

Tjänstgöringstyper

KlinMedVOft12_Kt2: 2012-09-18 (tisdag) — 2012-09-26 (tisdag)

Tjänstgöring	Eftersträvat antal tjänstgöringar Min	Max	Studentplatser per tillfälle
Mottagning/Jour	1	2	3
Ögonkirurgi	3	4	5
Ögonstenar	1	2	3

Each student group has a service set that defines what services the students of that group are expected to perform.

Also, for each service, the minimum and maximum numbers of participations are defined. The minimum is what is required for each group student; if the algorithm cannot achieve this for one or more students, no schedule is created. The maximum limit is not that dramatic; rather, it tells the algorithm when to refrain from associating a student with a service, even if there is such a possibility.

In addition, the group set includes, for each service, a benchmark number of student slots for a typical occasion. This shorthand facilitates the creation of the calendar: the number of slots can later be changed for any (individual) occasion. Probably, the number of occasion slots of a service will not differ that much (if at all). In any case, it is preferable to first setting them all to, say, five, and then to change one or two occasions to, say, two, rather than setting the slots of each occasion one by one.

`service_set_changed` is invoked whenever there is a change to the the group set carried out by the user in this form.

If the group set is changed, the schedule and the calendar become outdated (if they exist). This subroutine checks if there is any such data: if there is, it is erased.

The invocation of `service_set_changed` must be explicit as there are not any RDMS-style trigger functionality in Microsoft Access 2003. For this reason, all changes to the group set should be carried out in the form, and *not* directly in the actual table.

`update_all`: This brings the form up-to-date visually, to reflect changes in the group set. Also, `cbAddService` is updated to list only those services of the default set currently not in the present group set.

10.2.2.1 From group services to calendar

`cmdPopulate_Calendar`: Clicking this button will take the user from the group services to the calendar. During this transition, the calendar will be populated. The time frame of the calendar is here determined by the group. The services present in the calendar (and the suggested number of slots for each occasion) are determined by the group services (that is, what was set up in the form just left behind).

Here, a check could be made not to add occasions that take place on Swedish holidays. As it is now, those are added and must be deleted manually. On the bright side, Saturdays and Sundays are excluded.

10.2.2.2 Loss of course problem

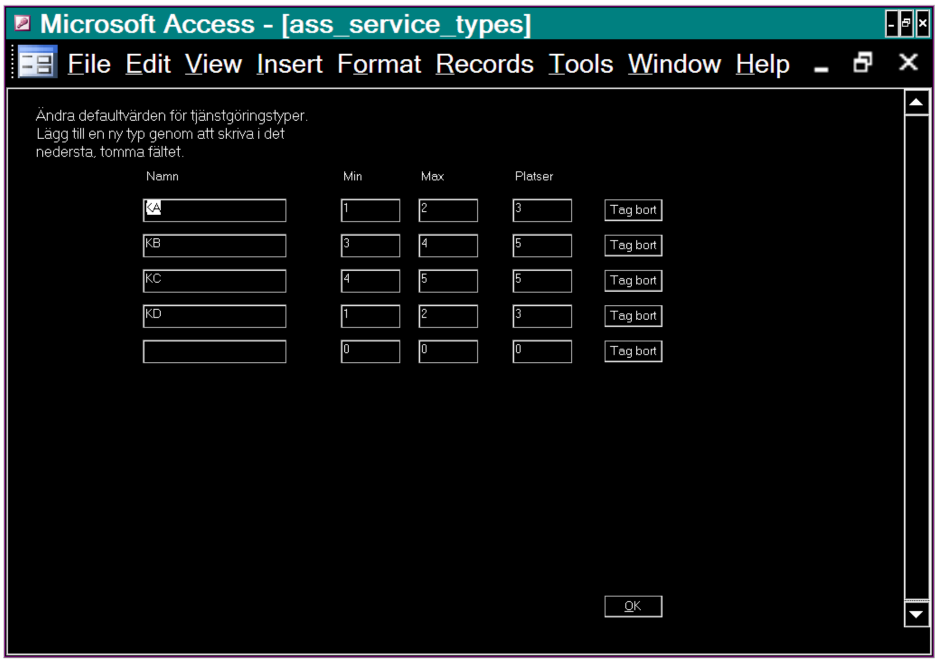
`check_set_args`: This function is a GUI hack. In order to do almost anything in the service form, the student group id must be known. That id is `PracticeStartNo`.

The form is populated from the `ass_course_specific_services` table. Each record in that table contains a `PracticeStartNo` field, and the subset of all such records with a particular number in that field constitutes the service set for that student group. However, if there is not any data as to that group service set (that is, not a single record with a corresponding `PracticeStartNo`), the `PracticeStartNo` cannot be determined that way.

To circumvent a crash in such a case, the `PracticeStartNo` is also sent as an **open argument**, whenever there is a transition to this form (by a button click) from another form. If the `PracticeStartNo` is lost, this function will use the open argument instead, destructively updating its argument to this value. However, this may *also* fail: if the service form is opened from the MS Access menu, there is no open argument!

If those two misfortunes coincide, there is nothing to do. This is communicated to the user. Also, the function returns `False` to tell whatever piece of code that invoked it, "there is no point proceeding as we cannot identify the student group".

10.2.3 Default services


Microsoft Access - [ass_service_types]
File Edit View Insert Format Records Tools Window Help
OK

Ändra defaultvärden för tjänstgöringstyper.
Lägg till en ny typ genom att skriva i det
nedersta, tomma fältet.

Namn	Min	Max	Platser	
KA	1	2	3	Tag bort
KB	3	4	5	Tag bort
KC	4	5	5	Tag bort
KD	1	2	3	Tag bort
	0	0	0	Tag bort

The default service set is not related to a specific group. As the algorithm operates on such a group, the default set is not used by the algorithm. Instead, it is a shorthand to facilitate the definition of group service sets. Typically, those do not differ that much; if they do, the default set may still be a good starting point.

10.2.3.1 `cmdDelete`

The `if` statement requires some explanation. In the default form, the way for the user to add a new default service (that is, a new record to the table) is to start typing in any of the blank textboxes at the bottom of the form.

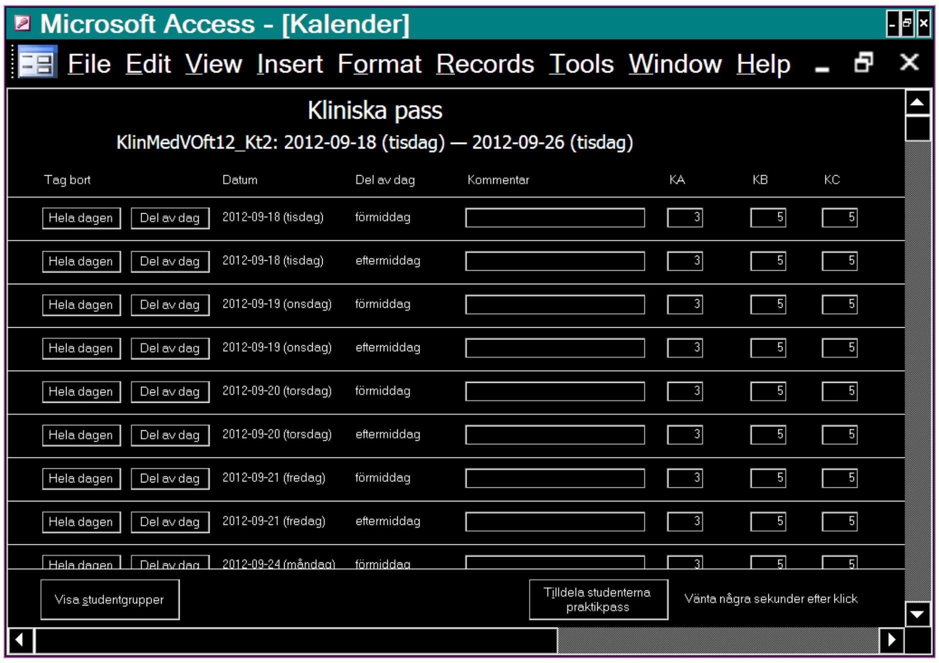
This is all well and good apart from the delete button to the right of each service row: unfortunately, that button appears next to the empty "add new record" textboxes as well. Clicking it doesn't make any sense, but if the user clicks it anyway (as a mistake or out of curiosity), the `if` statement is there to make sure nothing happens. (Obviously, this solution is a hack. Does it fully remedy the problem?)

10.2.3.2 Group/default services interface

`cbAddService`: If there are any services in the default set not in the group set, those are listed in this combobox. If selected, a service will be added to the group set along with the minimum, maximum, and slots default values, as defined in the default set for that service.

`cmdDefault`: This will clear the current (if any) group set and replace it with a set identical to the default set, if the user wishes to start anew.

10.2.4 Calendar


Microsoft Access - [Kalender]
File Edit View Insert Format Records Tools Window Help

Kliniska pass

KlinMedVOft12_Kt2: 2012-09-18 (tisdag) — 2012-09-26 (tisdag)

Tag bort		Datum	Del av dag	Kommentar	KA	KB	KC
Hela dagen	Del av dag	2012-09-18 (tisdag)	förmiddag		3	5	5
Hela dagen	Del av dag	2012-09-18 (tisdag)	eftermiddag		3	5	5
Hela dagen	Del av dag	2012-09-19 (onsdag)	förmiddag		3	5	5
Hela dagen	Del av dag	2012-09-19 (onsdag)	eftermiddag		3	5	5
Hela dagen	Del av dag	2012-09-20 (torsdag)	förmiddag		3	5	5
Hela dagen	Del av dag	2012-09-20 (torsdag)	eftermiddag		3	5	5
Hela dagen	Del av dag	2012-09-21 (fredag)	förmiddag		3	5	5
Hela dagen	Del av dag	2012-09-21 (fredag)	eftermiddag		3	5	5
Hela dagen	Del av dag	2012-09-24 (måndag)	förmiddag		3	5	5

Vänta några sekunder efter klick

The calendar is a set of occasions. It is group specific.

An occasion is located in time by date *and* day part: for example, `2011-01-05, early`. Also, on a specific occasion, a *number* of students can carry out a particular *service*.
The definition of an occasion is:

1. date (`2011-01-05`)
2. day part (`early` *or* `late`)
3. service (`eye surgery`)
4. maximum number of students that can participate (slots; e.g., `5`)

In terms of technology, the occasions are records in the calendar table.

`cmdDeleteDay`: This is typically used to remove holidays (e.g., New Year's Eve) when there isn't any activity. If there are an early *and* a late entry of that day present in the calendar, *both* will be removed if there is a click on *either* "delete day" button. (The Swedish label on this button is **Hela dagen**.)

`cmdDeleteDayPart`: This is like the "delete day" button, only it will only delete that particular day part. (**Del av dag**)

`cmdRunAssociationAlgorithm`: First thing, a check is made if there already is a schedule for the group. If so, a message box will ask the user to confirm that the old schedule should be removed, and the algorithm run anew. (If there is no schedule, the algorithm is simply executed.)

After the execution of the algorithm, `analyze_fallout` is invoked.

`show_and_hide_services`: The purpose of this is to hide and show slots boxes and service labels according to what services are intended for that group (which is determined by the group service set). Two loops accomplish this:

1. A `While` loop to make the boxes visible, and to set the labels to the service names. This loop iterates the group service set.

   Unlike many other such cases, the *order* of the set records matters. The sort is made according to the `index` field of the records. This is bad design as it mixes real-world information with a hack (the `index` field) to make the GUI possible.

2. A `For` loop that simply counts to nine and makes everything (box and label) invisible, using the iterator as the array index. It starts off where the while loop ended, when all present services have been named and made visible.

   This is also poor design: to begin with, there should not be a limit to the number of service types. As it is, there is such a limit (of ten); in addition, that limit is "hard-layouted" (the GUI elements) as well as hard-coded.

`get_slots_box`: This is a shorthand to get a reference to one of the ten textboxes in the calendar form. (This form is populated dynamically, so if in **Form view**, it may appear as there are a whole lot of textboxes. But, in **Design view**, there are only ten and they all refer to the same occasion.)

Each occasion has potentially ten such boxes visible, depending on the number of services in the group service set.

Above each box is a label showing the name of the service. The digit in the box is the number of slots of that occasion. (From the slots, look right for occasion, and up for service.)

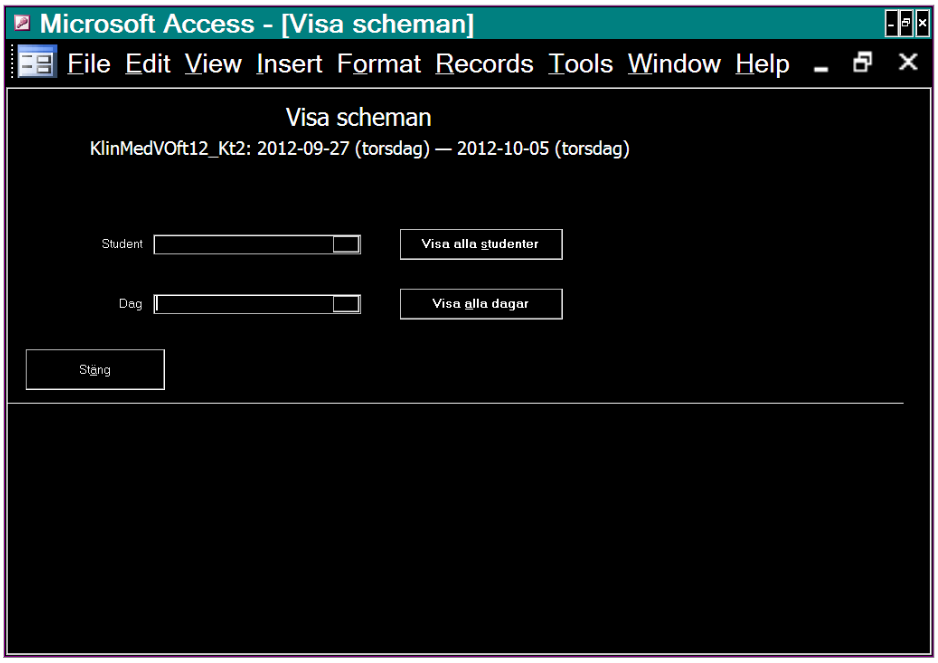
`get_service_label`: This works as `get_slots_box`, only it returns not the actual box, but the label above it, used to display the service name.

The boxes are named `PositionNSpaces`, where $1 \leq N \leq 10$. (In the lingo of this document, they should have been named *ServiceNSlots*: "service" is better than "position" as it is less ambiguous, and more descriptive.)

`occasions_changed`: If there is a change to the calendar, this subroutine must be (explicitly) called. For this reasons, changes to the calendar should be done in the calendar form, and not directly in the calendar table.

A change to the occasion set makes it likely that an algorithm execution would produce a different schedule. If there already is a schedule, it doesn't necessarily correspond to the (in part) new occasion set (actually, it is likely that it doesn't). For this reason, if there is a schedule, it is removed. Again, there is no check to see if this is indeed necessary: such a check would have been difficult and error-prone to implement, and as for execution, probably as costly as a fool-proof redo.

10.2.5 Schedule


Microsoft Access - [Visa scheman]
File Edit View Insert Format Records Tools Window Help
Student
Dag
Visa alla studenter
Visa alla dagar
Stäng

Visa scheman

KlinMedVOft12_Kt2: 2012-09-27 (torsdag) — 2012-10-05 (torsdag)

The schedule is the successful result of an algorithm execution.

If `test_fallout` (which is invoked after execution) finds that, for (at least) one student and one service, the minimum requirement could not be fulfilled — then, the algorithm result is deemed a failure and the schedule is removed. As a consequence, if there *is* a schedule, *any* schedule, it makes sense.

The schedule form is, in contrast to the other forms, not a place where anything is done. It is but an interface to the schedule for that group. It is possible to show the entire schedule, by day or by student. Also, it is possible to show all activity a certain day, as well as all activity associated with a certain student.

The schedule is presented as two reports: one by student, and one by day. So, to show the schedule of any student, is just a matter of narrowing down the complete student schedule, that which encompasses *all* (group) students. The same principle is used to show the activities of any singe day. That is, one schedule makes two reports; the interface offers two ways to show each.

The reports are compiled by SQL-queries. Those are, unfortunately, a complete mess to the human eye (even to the trained eye) as they were indeed not written by any human, but generated by the MS Access report GUI.

As for the implementation, there is one surprise to this form: the two invisible textboxes with location and course data. At some point in in the database, for some reason, that data could not be accessed directly from the form data table. I found that, by using these textboxes as middlemen, I could circumvent the problem. As for data consistency, it should not pose a problem as those textboxes, in turn, indeed fetch the data from the correct place. Of course, the only appeal of this hack is that it works.

11 Code notes

11.1 Algorithm code

`test_fallout`: This subroutine is invoked after the execution of the algorithm. At that point, a potential schedule is in place for the current student, service, and occasion sets. `test_fallout` uses the same subroutine as does the algorithm — `student_needs_service` — to determine if every student reaches the minimum goal for all services.

If this is the case for all students, the algorithm has succeeded. Otherwise the test is aborted at the first failing student/service pair. Either way, the outcome is communicated; in case of failure, the failing service is mentioned.

The interpretation of such a failure is not trivial. True by definition: "The minimum goal for some service was set too high." Depending on what is put weight on, it can be argued that either "there were too few occasions of that service", "the occasions of that service had too few slots", or simply, "there were too many students." In this, the tester doesn't offer any help.

After this, the user is moved to a form depending on the test outcome. On *success*, the user is moved to the *groups* form (so that [s]he can show the newly created schedule, or start working on another group). On *failure*, the user is moved to the the current group's *service* form (so that [s]he can modify the minimum goal and other parameters, run the algorithm anew, and hopefully this time end up with a schedule).

In `test_fallout`, there is a `max_need` boolean that is never used. It is needed for the `student_needs_service` as that uses destructive update and expects two booleans as references.

`get_next_slot`: This function iterates a schedule and, for each record, compares the data to the function arguments. If there is a match, it means a slot has been occupied by a student; a variable is then increased to reflect that (perhaps) the *next* slot is available for allocation. That is why that variable is initiated to 1 (and not 0): if there aren't any matching schedule records, the slot to work with is the first one, which is empty.

This function is called from the algorithm. That algorithm needs to count slots as not to overflow an occasion with students. (Again, looking at the end result, it doesn't matter what student was associated with what slot.)

`load_group_students`: First, all records are removed from `ass_students`. Second, all students in `Tv_Students` that belong to a certain group are inserted into `ass_students`. A group is defined by course *and* location.

`move_participations`: At the end of its execution, the algorithm populates the `ass_participations` table. This table contains all data as to when what student should carry out what service. However, there is also a similar (but less streamlined) table, namely `T_ScheduleSpaces`. This table is the basis for the schedule reports that are later generated. The task of `move_participations` is to migrate the data of `ass_participations` into `T_ScheduleSpaces`.

Four sets are used: the participations and the schedule; the calendar and the group services. This subroutine is a consequence of bad design: obviously! it searches back and forth in four (!) data sets. Thinking only computation-wise, it would have been better to get away with `ass_participations` and populate the schedule directly. However, during the implementation of the algorithm it was beneficial to strip the tables of anything unrelated, to be able to observe the algorithm execution without distraction. That being said, it looks a bit silly (inefficient and error-prone) as it is now.

`student_needs_service`: This uses `count_allocations` to find out if a student needs any more of a particular service. Note that it doesn't return a boolean. Instead, two reference booleans are destructively updated: one for the *minimum* need, and one for the *maximum*. Also, note that, although it would seem that a minimum need implies a maximum need, this is not so: for technical reasons, there is only a maximum need if there is *not* a minimum need.

`do_allocation`: This adds a data item to the `ass_participations` table. Such an item has four fields: the student, the time (date and day part), and the service. This table is the streamlined, intermediate representation of the schedule; later, its data is migrated to the schedule. As an optional debug tool, every allocation is tagged with a string that describes the algorithm mode at allocation time.

11.2 Modules

11.2.1 Text

In VBA, special characters are displayed by use of the `Chr` function. Given an integer argument, `Chr` returns the associated character. As an example, consider the (longest) hyphen character: a long dash, sometimes called an *M-dash* as it has the length of the letter M. This character is obtained with `Chr(151)`.

But, using such cryptic codes, the (VBA) code becomes less readable, not to mention the time lost looking up the codes during programming.

For this reason, I made a couple of mnemonic functions. Apart from the M-dash, those will give horizontal tabs (which do not use `Chr` but the `Space` built-in), *carriage return* (`Chr(13)`), and *line feed* (`Chr(10)`) (typically, those two are used in sequence).

11.2.2 Time

`Day_part`: This enumeration is of great importance to the algorithm part of the database, since it is part of the definition of a moment in time.

`date_to_US_style`: `FindNext`, `FindPrevious`, and a couple of other recordset methods do not work with the Swedish date representation. To circumvent that, dates are (when needed) converted to the US (searchable) representation. This way, dates are still Swedish in the tables.

11.2.3 `close_and_go`

This subroutine is used to backtrack the user's movement between forms. Typically, if the user specifies some arguments in form A, and then moves to form B, B will present a **Back** button to allow the user to go back to A and redo his or her choice, before once again moving on to B.

This subroutine accomplishes its task by making A *visible* and *closing* B. As A has been invisible during B's reign (thus inaccessible to the user), A has the same set of data it had when B became visible to the user. For this to work, A is made <u>in</u>visible at the transition from A to B and, of course, B is brought forward.

12 Summary and Conclusion

The education of to-be physicians at Akademiska sjukhuset, Uppsala, includes a couple of practical *services* that students take part in. Those students are divided into *groups*, and each group has its own goals: the goals specify (1) what services that group's students should perform, and (2) for each such service, a *minimum* number of times each student should attend that service.

Naturally, it is only possible to perform any particular service at certain *occasions*: each occasion offers a number of *slots*, to be filled by on-duty students. The occasions make up a *calendar* of possible activity.

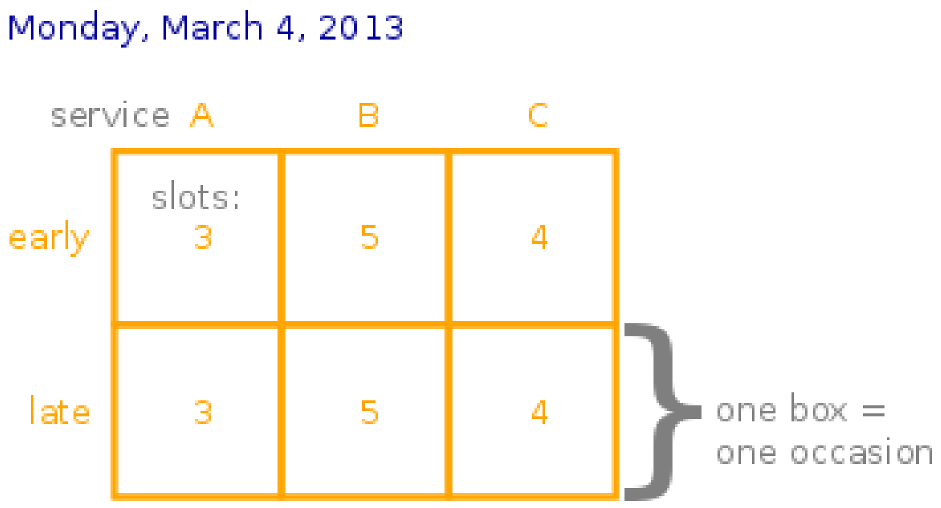
12.1 The challenge

The challenge is to distribute the students over the calendar, so that the goal is achieved for each student, for each service. No occasion should be overpopulated by students, and no student should be due to attend two (or more) occasions that collide in time.

12.2 The calendar

The calendar is simple, yet flexible. At the very most, on a single day, *all* services are possible to attend, *twice*: for each service, there is one occasion *early*, and one *late*. This full-schedule day is one of two endpoints, the other being a day with zero occasions (and thus zero activity); everything in between is just as representable. In practice, a more or less full schedule is probable.

For instance, a Monday, March 4, 2013, could look like this:



Note: In practice, slot numbers are often the same for a particular service.

Also: The calendar representation implies the definition of colliding events: the same day (date), *and* the same day part (early or late).

12.3 The algorithm to solve it

The algorithm to solve this problem sets up a table with occasions (expanded horizontally by their number of slots) as the x-axis, and dates (expanded vertically by two: the day parts) as the y-axis. Then, distribution of students is done top-down, left-right. Collision is avoided by having students only appear once per row. Overpopulation won't happen as the allocation of students is done explicitly to *slots*, not to an occasion in general.

12.4 The UI

A couple of MS Access forms make up the UI. My thoughts when I set them up was that each form should boil down to a single purpose, but include everything to fulfill that purpose (and nothing else). Also, I setup an intuitive flow of movements between forms, and I made an effort to setup mnemonic shortcuts (and tab chains) as to minimize mouse use.

13 References

[1] Paul Ammann and Jeff Offut. *Introduction to Software Testing*. 6th edition. Cambridge University Press, 2008. ISBN: 978-0-521-88038-1.

[2] Edward Angel. *Introduction to Computer Graphics. A Top-Down Approach using OpenGL*. 5th edition. Pearson/Addison-Wesley, 2009. ISBN: 0-321-54943-3.

[3] Andrew Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, 1999. ISBN: 0-521-60764-7.

[4] Krzysztof Apt. *Principles of Constraint Programming*. Cambridge University Press, 2003. ISBN: 0-521-82583-0.

[5] Bitting and Skansholm. *Vägen till C*. Tredje upplagan. Studentlitteratur, 2000. ISBN: 978-91-44-01468-5.

[6] Cameron et al. *Learning GNU Emacs*. 3rd edition. O'Reilly, 2004. ISBN: 0-596-00648-9.

[7] Edgar Codd. "A Relational Model of Data for Large Shared Data Banks". *Communications of the ACM* 13.6 (1970).

[8] Cormen et al. *Introduction to Algorithms*. 2nd edition. MIT Press, 2001. ISBN: 0-268-53196-8.

[9] Peter Dyson. *The Unix Desk Reference: The hu.man Pages*. SYBEX, 1996. ISBN: 0-7821-1658-2.

[10] George Luger. *Artificial Intelligence. Structures and Strategies for Complex Problem Solving*. Pearson/Addison-Wesley, 2009. ISBN: 0-321-54589-3.

[11] Kim Marriott and Peter Stuckey. *Programming with Constraints. An introduction*. MIT Press, 1998. ISBN: 0-262-13341-5.

[12] McFadden, Hoffer, and Prescott. *Modern Database Management*. 5th edition. Addison-Wesley, 1998. ISBN: 0-8053-6054-9.

[13] Padron-McCarthy and Risch. *Databasteknik*. Studentlitteratur, 2005. ISBN: 978-91-44-04449-1.

[14] Vijay Saraswat. *Concurrent Constraint Programming*. MIT Press, 1993. ISBN: 0-262-19297-7.