**Revised reply, written exam, DBD, January 2006**

**Some statistics**

Enrolled for the exam: 56 Passed: 48 Didn't show up: 7

Marks: 5 6 7 8 9 10 11 13

Count: 1 2 4 17 18 7 0 0

Average: 8.43 (Average S04: 8.55, A04: 8.87, S05: 8.45)

This semester the exam results were a bit worse than usual. Not so much on the average, but because we didn't give marks above 10. Usually we have several 11's. Censor and teacher wonder why, because both of us thought the questions were a bit simpler than usual. For instance there were fewer E/R boxes and virtual windows. However, this made the defects on the detail level more visible.

Some students suggested that the reason was that the questions covered the entire process from tasks and E/R to function design and usability test (usually we cover only part of the process). The students found it hard to "change mind" so many times during the four hours.

**Frequent problems in the replies**

The main problem seems to be that the students didn't read the text carefully. Censor and teacher have paid much attention to the wording, and it is important to pay attention to the details. There is very little "noise" in the text.

**Question 1.** In principle the data model was easy. But more than half of the students forgot that supporters have some special attributes: the line they work on right now, whether they are absent (e.g. in a meeting), and whether they for the time being want email warning when there is a request for them. Many of you who remembered these attributes, forgot that supporters also are ordinary employees, so we don't have to keep their name, email, etc. in a separate table.

Many students forgot that the problem causes should be a separate table, preferably with a hierarchical structure. And they forgot details such as a deadline or warning time for a request, that it was crucial to remember the supporter who had received the request by phone, etc. Some students created large models with the relationship between causes and solutions, models for the spare part store, etc. This didn't detract, but gave no points since something of that kind wasn't required in the text.

**Question 2.** Tasks were better than usual apart from two things:

a. Most students forgot the important task that dealt with changing the role of a supporter, for instance from line 1 to line 2 or absence. The system must support this well, for instance help getting the supporter's requests back to the general pool for first or second level hotline.

b. More than half of the students failed to write the existing problems in the task descriptions. If they mentioned anything at all about problems, it was about problems they anticipated in the new system.

**Question 3.** There were two basic virtual windows: An overview or search window like the one shown in the text, and a window for a specific request (possibly connected to the overview as list-detail pattern). Nearly all students figured this out. Many spent much time redrawing an overview window that didn't provide any improvement over the one in the text. For instance it would be important to show the deadline in the overview, but they didn't do it.

Only a few had figured out that a virtual window was needed to show the catalog of causes, preferably as a hierarchy. And very few had made a virtual window (a list) with the roughly 15 supporters. It would be useful when assigning a supporter to a special request. And even fewer had realized this list would be the natural place to show and change the special supporter attributes such as line and absence.

**Question 4.** Many students had a good understanding of functions and mini-specs, but only a few were able to relate problems and functions to each other. For instance, how will we ensure that the user gets a message when his request is completed? The solution might be to let the system send the user a message when the request is closed. In other words a side-effect of changing the request state. And how will we ensure that a cause is recorded? The solution might be that the system reminds the supporter when he closes the request - and the recording must be easy, for instance a combo box with lists in several levels rather than an explorer-like tree in a separate window.

**Question 5.** More than half of the students spent time considering usability factors and usability requirements. According to the text there was no need to do this. The only thing needed was some realistic test tasks, but this was hardly done at all.

Since all replies had serious defects in one or more of the key areas, it was impossible to give 11 or 13. A real pity!

However, some replies had points that were better than the teacher's solution, for instance a virtual window that showed the user's history across several requests, rather than just a single request. Or were able to bundle a lot of requests into one, for instance when 50 users reported that a server was down. In this way all of them could get a reply at the same time.

**The suggested solution.** The teacher's solution below has more comments than we expect from you at an exam. For instance all tasks are fully described and not just an annotated list. All mini-specs are there. The solution is also aimed at helping a programmer implemen­t the system in MS-Access. For instance it discusses how tricky parts of the solution might be implemented. (Remember that the designer should know the platform well - or at least cooperate with a developer who knows.)

**Question 1. Data model (estimated time: 30 minutes)**

Make an E/R model for the system. Show the attributes for each class. Omit artificial keys and foreign keys.

**Comments**

**Request** is a problem report, identified by the number *requestID*. *Subject* is the title given by the user. *Status* is the current state of the request. It always has a *sender* (a relationship), which is the user who reported the problem. Since the sender may work from a different place than usual, we may also record his current phone.

*Source* is the person who has set the request in this state. When a supporter receives the request by phone or in person, he records the request and the sender, and becomes the *source.* When a user records the request by email or on the web, he becomes sender as well as source. The request has an *owner* when a supporter is working on it. *Note* is an explanatory text provided by the person who set the request in the current state. When the request is created, *note* is the explanation given by the user. Later it may be a message sent to the user or from one supporter to another. *Priority* is the supporter's estimate of how urgent the request is. *Cause* (a relationship) is what supporters currently believe is the cause of the problem.

*StartTime* is the point in time where the user sent the request or the supporter started recording it. *ReminderTime* is the point in time where the system will set the request in *reminder* state. The idea is that hotline sets the reminder time according to when the user should expect a reply, or a parked request should be resumed.

**History**keeps track of the progress of the request. Whenever the request changes, the system copies the updated request record to a new history record (except for *startTime*, which doesn't change). This allows the users and supporters to see the progress of the request, and it also allows statistics to be made. *History* has an attribute not found in *Request*: *saveTime*, which is the point in time where the request was updated.

**Other ways to keep track of the history**. The technique explained above uses redundant data because the current data for the request is stored in *Request* as well as in the latest history record. Another technique is to store the current data only in *Request*. However, this makes the user interface more complex to program in Access. If we for instance want to show a list of the entire history, we must make a Union query of Request and History. The result looks right, but Access doesn't allow the user to edit the data in a union query.

A third technique is to abandon *History* and instead keep all the versions of the request as separate Request records. The only thing needed is to add an attribute *saveTime* to Request. The key will now be requestID, saveTime. This is a clean solution, but you need a tricky self-correlated query to show a list of the current state of all the requests. In large databases, this may give performance problems.

For these reasons, I would not recommend these two techniques, but honestly I am not quite sure.

**Employee** is the existing employee table with name, email, etc. (We show such existing data classes with a double border).

**Supporter**is a role an employee can have. The role has special attributes, for instance that he wants an email when there is a special request he should deal with. He can also specify whether he is on first or second line, or not in the hotline at all, for instance because he has other duties or is away from work.

**Question 2. Task descriptions (estimated time: 60 minutes)**

Describe the tasks for users as well as supporters as a task list (annotated). Add sub-tasks and problems where relevant. Because of question 4, it is a good idea to make space for notes about the solution. (In this and the rest of the questions, you may ignore tasks that deal with request statistics, creating/deleting employees, and editing the catalog of request causes.)

**Reply**

The descriptions below show suggested solutions. Many of the solutions correspond to mini-specs in question 4, as noted in the right-hand column. In a few places a question mark indicates that I haven't designed a solution.

**Work area:** 1. The user

|  |  |
| --- | --- |
| T1.1: Problem reporting | Example solution |
| 1 Call hotline or meet in person. |  |
| 1a Send mail to hotline. |  |
| 2 The problem may be solved on the spot. |  |
| 2p **Problem:** When can I expect a reply? | The system reminds the supporter to set *reminderTime* and helps him send a mail about it to the user. (See Request, SaveRequest, point 4 and 5.) |

|  |  |
| --- | --- |
| T1.2: Follow up Start: Impatience or mail from hotline. | Example solution |
| 1 Check whether the problem has been solved. |  |
| 1p **Problem:** Hotline forgets to tell when the request is completed. | The system helps the supporter send a mail when hotline closes the request. (See Request, SaveRequest, point 3.) |
| 2 See how far the request has come. | Lookup in the request base |
| 3 Provide more information. |  |
| 4 Maybe close the request. |  |
| 5 Maybe resume the request (open it again). |  |

**Work area: 2.** Hotline

|  |  |
| --- | --- |
| T2.1: Request to first line  Start: A user calls or sends a mail, or a reminder. | Example solution |
| 1 Record the request, particularly the user's phone or email. | The system warns if user or phone isn't specified when the request is created. |
| 1p **Problem:** Cumbersome, particularly when it is an on-the-spot solution. | Fields allowed to be blank as far as possible. (See Request, default values, and data entry, *Cause* and *Sender*) |
| 1q **Problem:** Difficult to specify a cause. | Easy to define a cause. (See Request, Data entry, *Cause*.) |
| 1a The request arrives by email through the support system. |  |
| 1b The request is a reminder. |  |
| 1c The user provides new information about an open request |  |
| 1d The user wants to reopen a closed request |  |
| 1e The new information or reopening the request is done on-line by the user. |  |
| 1f A request has been waiting in the 1st line "in-tray" for some time. |  |
| 2 Maybe solve the problem on the spot and close the request, possibly with an explanatory email. |  |
| 3 Maybe leave the request in the "in-tray". | The system warns if a reminder time hasn't been set. (See Request, SaveRequest.) |
| 4 Transfer the request to second line or a supporter with special expertise. Give it a priority and a reminder time. |  |
| 4p Hard to see who is present now. |  |

|  |  |
| --- | --- |
| T2.2: Request to second line  Start: Look at open second-line requests from time to time, or receive an email.  **Problem:** Hard to spot the important and urgent requests. | Example solution  Can restrict the list to relevant requests. Can sort according to reminder time, priority, etc. (See RequestList, Search and OrderBy.) |
| 1 If necessary, contact the user or receiver to obtain more information. |  |
| 2 Solve the problem by moving to the problem location. |  |
| 2p **Problem:** Would be nice if the supporter could handle several problems on his trip. | Mark some of them and print them? |
| 3 Work for some time on the problem. Inform others that they don't have to look at it. | Put the request in state *taken*. |
| 4 Order something from an external supplier and park the request. | The system warns if no reminder time has been set. (See Request, SaveRequest, point 4.) |
| 5 In case of a reminder, contact the supplier and set a new reminder time. |  |
| 5p **Problem:** The user doesn't know about the delay. | The system sends a mail when reminder times are changed. (See Request, SaveRequest, point 5.) |
| 6 Close the case. | The system warns if the cause hasn't been set. (See Request, SaveRequest, point 2.) |
| 6p **Problem:** The user isn't informed when the request is closed. | The system sends a mail when the request is closed. The supporter has possibility to write an explanation in the mail. (See Request, SaveRequest, point 3.) |
| 7 Maybe leave the request in the "in-basket" or transfer it to someone else. |  |

T2.1 and T2.2 might be combined into one session task because several subtasks occur in both places. But they have different triggers, and many details differ in the two situations. In general it is a bad idea to combine tasks too early, because important details may be lost.

|  |  |
| --- | --- |
| T2.3: Message from an external supplier. | Example solution |
| 1 Find the request. | ? |
| 2 (Continue as in T2.2) |  |

|  |  |
| --- | --- |
| T2.4: Change of role.  Start: The supporter has to change line, leave the hotline, etc. | Example solution |
| 1 Change own settings for line, etc. May also change other supporter's settings, for instance if they are ill. |  |
| 2 Transfer any taken requests to 2nd line. |  |
| 2p **Problem:** The supporter forgets to transfer them. | The system warns and suggests changing status for all of them. (See SetAbsence in Supporters.) |

**Question 3. Virtual windows (estimated time: 60 minutes)**

Design virtual windows in graphical form for the supporters in hotline. Assume that the system will run on a Windows-like platform, i.e. without html-look, links, etc.

**Request list**

The request list window is used for searching and for monitoring the requests. A reminder should be shown with yellow background to catch attention.

A first-line supporter will usually set the criteria to show only first-line requests. A second-line supporter will set the criteria to show second-line requests. Independent of the setting, the supporter should always see all requests taken by him (or assigned to him by another supporter). He may choose to see also requests taken by someone else.

Compared to the present screen in the exam text, the virtual window has search criteria, time left till a reminder (*Left*), and the current cause of the problem.



Request (detail)

The request window shows all the details of a single request. It shows the full name and phone for the sender (the user), the start time for the request (instead of *Age*) and the reminder time (instead of *Left*). Initially the latest note is shown in the text box. When the supporter edits the request, the text box shows a new note. The system may send the note to the user to inform him of the progress.

The request window also shows the history of the request with either increasing time or decreasing time. In the example the latest event is at the top. Two displays of the history are available: As a text with all the notes in full text, and as a table with status, owner, etc. [The user should be able to select which request attributes to be shown in the table, but we ignore it here.]

In the example, nhn called hotline because she couldn't log on. Peter on 1st line created a request, gave her another password, and instructed her to change it to one of her own choice. Then he set the state to *closed* and saved the request. The start time is when they created the request, the time in the history list is when they saved it. A few minutes later, nhn called again because it didn't work. The same supporter realized that it was more complex, found the request and assigned it to 2nd line. After some time Kevin took the request, found a solution, told the user by phone and closed the case once more. He also remembered to change the problem cause.



The supporter can change the request by means of the combo boxes. As an example, he can close the request by setting *status* to *closed*. He can set the cause by means of the hierarchical drop-down list. He can assign the request to a specific supporter by setting *owner*. Then status becomes *taken*. *Sender* and *subject* are set when the request is created and shouldn't change later. However, as an exception the supporter may change them.

When the supporter saves the request (e.g. closes the window), the system creates a new history record.

**Supporters**

Gives an overview of all the supporters and how they currently participate in the hotline. A supporter will usually only change his own settings, but occasionally he may set it for others too, for instance if a colleague has become ill. It is an advantage to see the whole overview, for instance to get an idea about who is 1st line right now.

**Drop-down lists (combo boxes)**

The various drop-down lists are virtual windows too. Only *cause* and *reminder* are shown. Drop-down for *owner*, *status* and *priority* should be obvious.

**Question 4. Functions and problem resolution (estimated time: 40 minutes)**

For each virtual window, show the semantic functions and search functions that are needed. Briefly describe how the functions and windows solve the problems that exist today in a hotline. Where needed, use a short mini-spec to explain the details of a function.

**Reply**

In the task descriptions, I have for most problems given references to the functions that solve the problem. The mini-specs, however, don't systematically give references to the problems. (Such backward tracing is less important than the forward tracing.)

Since this reply is to be the basis for programming the system, I explain all the functions below - including the navigation functions. For the same reason I show mini-specs for all of them.

Request list

Search: Find requests matching the criteria.

OrderBy: Order the list by owner, status, etc. Order increasing or decreasing.

CreateRequest: Open the request window for creating a new request.

SelectRequest: Select one of the request lines.

OpenRequest: Open the request window for the selected line.

SeeSupporters: Open the *Supporters* window.

Requests received by email

The system automatically records requests received by email. *Sender*, *source* and *subject* are taken from the email header, *note* from the email body. Status is set to *1st line*.

**Timer**

The system checks once a minute whether the reminder time has arrived for some of the requests. If so, it sets the request into *reminder* state. If the request has an owner and he wants email notification, send an email.

Request (detail)

StartEdit: [In MS-Access, this is signaled by the *Dirty* event.] Set *source* to the current supporter. Clear note and mark it as *New note*.

SaveRequest: [In MS-Access, points 1-5 are done at BeforeUpdate, points 7-8 at AfterUpdate.]

1. Check the edit buffer to see that sender (or currentPhone) and subject are defined.
2. If status becomes closed and *cause* is empty, suggest that the supporter sets a cause. [Some advanced language analyzers might be used to suggest a reasonable cause based on the subject and description.]
3. If status becomes closed and the supporter hasn't set a note to be sent to the user, set a standard note and suggest that the supporter sends it to the user.
4. If status isn't closed and no reminder is set, suggest that the supporter sets one.
5. If a reminder has been set and the supporter hasn't set a note to be sent to the user, set a standard note and suggest that the supporter sends it to the user.
6. Save the record and make a copy in *History*.
7. If *SendToUser* is on, send the note as an email to *sender*.
8. If a new owner has been set and he wants email notification, send a mail.

Cancel: Undo all changes so that the situation is as before edit.

ToggleHistoryMode: Set history display to table or text.

ToggleHistorySequence: Order the history list forwards or backwards.

SendToUser: Mark that the note is to be sent to the user.

**Data:**

Default values for a new request: *Source* is the supporter. *Status* is 1st line. *StartTime* is now. Other fields are empty.

**Non-trivial data entry functions:**

Owner: Setting *owner* also sets status to *taken*.

TakeRequest: [Shortcut data entry instead of changing the state explicitly.] Set status to *taken* and owner to myself.

Status: Setting status to 1st or 2nd, also sets *owner* to nothing. Setting status to *closed*, sets *reminder* to empty.

CloseRequest: [Shortcut data entry instead of changing the state explicitly.]

Cause: If no subject has been set, the subject is also set to *cause*. [This should encourage the supporter to set the cause in all the frequent cases where he deals with a known problem. In these cases, setting the cause spares the supporter typing the subject.]

Remind: The user may type directly into the field or select from the list.

Sender: A combo box where the user can type the first letters of the email address and then select from the list. [The list would have 1000 entries, but the first letters would reduce it to less than a screen full. Having a separate search window seems overkill.]

SenderName: Similar to Sender, but here a combo box with pattern match should be used if possible.

Phone: Retrieved from sender data, but may be overwritten if the user calls from another phone.

CopyText: Allow copying text from the history list when it is in text form. [Will be useful when composing longer notes.]



Supporters

SelectSupporter: Select one of the supporters.

SetAbsence: If some requests are taken by the selected supporter, warn and offer to transfer them to 1st or 2nd line.

SetLine: If there are no supporters left on 1st line, warn the user.

SetEmailNotify: Simple check box.

CloseWindow: Save changes (also done when focus moves from line to line).

[Undo hasn't been considered, but may not be trivial. What should we for instance do if the supporter by mistake closed the request, and a message was sent to the user?]

Function presentation

Since this reply is to be the basis for programming the system, I also suggest a function presentation.

|  |  |  |
| --- | --- | --- |
| Request list | Syntax | Comments |
| Search | Live search, button, shortcut | Intuitive, live search feasible time-wise |
| OrderBy | Select column from a dropdown list | Access has a built-in order-by through right-clicking the column. It is not as intuitive as other programs. |
| CreateRequest | Button, shortcut |  |
| SelectRequest | Std Access |  |
| OpenRequest | Enter, double click | Probably intuitive to IT-people |
| SeeSupporters | Button, shortcut |  |
| Goto field | Shortcut on label. F6 to toggle between subform and main form | Std Access, although F6 has to be programmed. |
| **Total buttons** | 3 | Okay |

|  |  |  |
| --- | --- | --- |
| Request (detail) | Syntax | Comments |
| StartEdit | Any change of field (*Dirty* event) | Maybe button too? |
| SaveRequest | Close window, Button, Shift+Enter | Shift+Enter is built-in Access, but probably not intuitive. |
| Cancel | Esc (std Access), Button |  |
| ToggleHistoryMode | Check box, shortcut |  |
| ToggleHistory­Sequence | Check box, shortcut |  |
| SendToUser | Check box, shortcut |  |
| Goto field | Shortcut on label. F6 to toggle between subform and main form. |  |
| **Data:** |  |  |
| Owner | Combo box |  |
| TakeRequest | Button, shortcut |  |
| Status | Combo box |  |
| CloseRequest | Button, shortcut |  |
| Cause | Hierarchical combo box? | Does it exist in Access? |
| Remind | Combo box |  |
| Sender | Combo box |  |
| SenderName | Combo box |  |
| **Total buttons** | 4 (5 ?) | Okay |

|  |  |  |
| --- | --- | --- |
| Supporters | Syntax | Comments |
| SelectSupporter | Std Access |  |
| SetAbsence | Check box |  |
| SetLine | Combo box |  |
| SetEmailNotify | Check box, shortcut |  |
| CloseWindow | Std Access |  |

**Question 5. Usability test (estimated time: 20 minutes)**

During development, usability tests must be carried out. Specify two test tasks that would be suited as the first tasks during the test.

Test subject: A potential supporter. The test subject has got a short introduction (two minutes) to the purpose of the system, but hasn't seen any screens. The system is started up and shows the request list. The subject is asked to think aloud.

Test task 1: Investigate the latest problem report. What would you do about it? (The request is about replacing toner in a printer. It should be handled by 2nd-line.)

Test task 2: The phone rings and Lisa Nelson explains that she has forgotten her password. What would you do?

Some real good test tasks from student replies:

Test task 3: Assume you are supporter on 2nd line. Which problem report would you start working on? (There is a request for changing toner, and there is no more toner on store.)

Test task 4: A user calls with a problem you cannot solve, but you know that IMF can do. What would you do?