User Interface Design

A Software Engineering Perspective

Soren Lauesen

Harlow, England • London • New York • Boston • San Francisco • Toronto
Sydney • Tokyo • Singapore • Hong Kong • Seoul • Taipei • New Delhi
Cape Town • Madrid • Mexico City • Amsterdam • Munich • Paris • Milan

Contents

Preface

When you design the user interface to a computer system, you decide which screens the system will show, what exactly will be in each screen and how it will look. You also decide what the user can click on and what happens when he does so, plus all the other details of the user interface. It is the designer's responsibility that the system has adequate *usability* – it can do what is needed and is easy to use. It is the programmer's responsibility that the computer actually behaves as the designer prescribed.

If you are involved in designing user interfaces, this book is for you. You don't just learn about design and why it is important, you actually learn how to do it on a real-life scale.

Designing the user interface is only a small part of developing a computer system. Analysing and specifying the requirements to the system, programming the software and testing and installing the system usually require much more effort, and we will barely touch on it in this book.

Making the user interface

How do you design the user interface to a computer system? Ask a programmer and he may say:

> *The user interface? Oh, it is so boring. We add it when the important parts of the program have been made.*

Ask a specialist in human–computer interaction (HCI) and he might say:

> *The user interface? Oh, you have to study the users and their tasks. To do this you must know a lot about psychology, ergonomics and sociology. Designing it? Well, you have to come up with a prototype of the user interface and review it with the users. Programming? Oh, that is what the programmers do when the user interface has been designed.*

Is there a communication gap here? Yes, for sure. The truth is that it is easy to make a user interface – exactly as the programmer says – but it is hard to make a *good* user interface – and that is what the HCI specialist tries to do.

There are thousands of books on programming. Common to all those I have seen is that the user interface is rather unimportant – it is just a matter of input to and output from the program. Programming *is* very difficult and there is nothing wrong with dedicating thousands of books to it. The only problem is that the programmers

don't learn to design the user interface, although typically more than half of a program deals with the user interface.

There are a few dozen books on HCI. They tell a lot about human psychology, how to study users and their tasks, how to test a prototype or a finished system for usability and many other good things. Amazingly, they say very little about the actual design of real-life user interfaces: how many screens are needed, what should they contain, how should we present data, how can we make sure that our first prototype is close to a good solution. To be fair, I have seen a few books that explain about design: Redmond-Pyle and Moore (1995), and to some extent Cox and Walker (1993) and Constantine and Lockwood (1999). However, even here most of the screen design pops out of the air.

Many programmers have looked into the books about HCI. They read and read, but find little they can use right away. And they don't get an answer on how to design a user interface. The books somehow assume that it is a trial and error process, but programmers don't trust this. They have learnt that trial and error doesn't work for programming. Why should it work for user interface design?

Bridging the two worlds

This book tries to bridge the gap. A crucial part of the book is how to design the user screens in a systematic way so that they are easy to understand and support the user efficiently. We call this approach the *virtual window method*. The virtual windows are an early graphical realization of the data presentation. Task analysis comes before the virtual windows and dialogue design after.

I have developed the virtual window method over many years with colleagues and students, and it has become a routine way to develop a user interface. We don't claim that the result is free of usability problems right from the beginning, just as a good program isn't free of bugs right from the beginning. But the large-scale structure of the interface becomes right and the problems can be corrected much better than trial and error.

The cover picture illustrates this. It was painted by the Danish artist Otto Frello in 1995. Imagine that it is the road to good user interfaces. It is a firm road, but it is not straight. However, there is support at every corner (notice the lady who appears repeatedly along the road).

In order to design in a systematic way, we have to draw on the programmer's world as well as the HCI specialist's world, but we pick only what it necessary and adapt it for our specific purpose. Here are some of the major things we draw on:

- How to measure usability
- Usability testing and heuristic evaluation
- Different kinds of prototypes

- Data presentation
- Psychological laws for how we perceive screens
- Task analysis and use cases
- Data modelling (programmer's world)
- State diagrams (programmer's world)
- Checking the design and keeping track of the problems (programmer's world)

To get from these classical issues to the systematic design, we add some novelties:

- Mental models for how users understand what they don't see
- Virtual windows: Presenting data in few screens that cover many tasks efficiently
- Function design of the interface: Adding buttons, menus, etc., to the screens in a structured and consistent fashion.

Programming the user interface

The book is about design of user interfaces down to the detail. Programming the user interface is closely related, so we need teaching material on how to do this too. Here is a pedagogical and even a political problem: which platform to use for the programming. There are many choices, Microsoft Windows, UNIX, HTML, Swing, etc. Ideally, we should have teaching material for each of these.

I have chosen to release a free companion to this book: a booklet on user interface programming with Microsoft Access. The main reason for this choice is that Microsoft Access is readily available to most students as part of Microsoft Office; the connection to a real database is easy; you can quite quickly systems that look real, and you can gradually add the full functionality. Access also exhibits the techniques found on other platforms, such as GUI objects, embedded SQL and event handling. My dream is to have additional booklets that cover other platforms.

The Access booklet is free for download from www.booksites.net/lauesen. It uses examples from this book and shows how to turn them into a functional system.

Design cases covered

The book illustrates most of the design techniques with a system for supporting a hotel reception. This is sufficiently complex to illustrate what happens in larger projects. However, it makes some people believe that this is the only kind of system you can design with the virtual window method. Fortunately, this is not true.

The book shows how to design different kinds of systems too with the same method. Chapter 10 shows how to design a Web system for course evaluation and

management. Chapter 11 shows how to design an advanced e-mail system. In section 3.7 we show part of a complex planning system designed with the method. The design exercises in Chapter 17 deal with other kinds of systems, for instance a Web-based knowledge management system, a photocopier, a system for supporting small building contractors and an IT project management system.

Uncovered issues

In the book we pay little attention to aesthetics – how to make the screens look pretty and attractive – or to the entertainment value of the system. These issues are very important in some systems, but in this book we mainly look at systems that people use to carry out some tasks, and here aesthetics is not quite as important. And to be honest, I have not yet figured out how to deal with aesthetics in a systematic way.

The book focuses on systems with a visual interface where a lot of data has to be shown in some way. As a result, we don't discuss verbal interfaces (e.g. phone response systems) or user interfaces for the blind.

We also pay little attention to low-level aspects of the user interface, for instance how to display a button so that it appears in or out of the surface, how a scroll bar works, whether to use a mouse or a touch screen. We take these things for granted and determined by the platform our system has to run on. These issues are important in some projects, and they are excellently covered by, for instance, Preece *et al.* (1994, 2002), Cooper (1995) and Dix *et al.* (1998).

Why doesn't the author mention object orientation? Doesn't he know about it? Yes, he does, very well indeed, but the traditional object-oriented approaches don't really help us designing a user interface in a systematic way. However, we can describe the design results as objects, and this gives an interesting perspective on what the design produces and how it relates to traditional object-oriented approaches (see section 9.2.3).

How the book is organized

The book is organized as three parts that may be combined in various ways for different kinds of courses.

Part A: Best of the classics (Chapters 1–4)

Some classical usability topics: defining and measuring usability; using prototypes, usability tests and iterative design; data presentation and how users perceive what they see on the screen.

Part B: Systematic interface design (Chapters 5–9)

How to design the prototype in a systematic way so that it is close to being right early on. This includes task analysis, designing the virtual windows, defining the system functions and their graphical representation. Chapter 9

reflects on the virtual window approach and compares it with other design approaches.

Part C: Supplementary design issues (Chapters 10–16)
Optional topics that may be included depending on the audience. Examples are systems development in general, data modelling, user documentation and support, design cases for different kinds of systems.

Throughout parts A and B we present the topics in a learn-and-apply sequence: After each chapter you can do something that would be useful in a real development project. You then try it out in a design exercise. Here is a summary:

Chapter 1: Usability. Here you learn to identify the usability problems of an existing system through a usability test. You also learn how to specify and measure usability. In an exercise you specify and measure usability for an existing system, for instance an existing Web site.

Chapter 2: Prototyping and iterative design. You learn the classical design approach for usability: Make a prototype, test it for usability, revise the design, test again and so on. In an exercise you design a prototype of a small system with a few screens, and test it for usability.

Chapter 3: Data presentation. You learn about the many ways to present data on the screen, and the psychology behind them. You apply the rules on existing designs, and you try to design non-trivial data presentations.

Chapter 4: Mental models and interface design. You learn to use the psychological law of object permanence to explain why users often misunderstand what the system does, and how you can avoid it with a systematic way of designing the user interface. You also learn about the two extreme interface architectures: the database oriented and the step-by-step oriented.

Chapter 5: Analysis, visions and domain description. You learn how to model user tasks and data in a way suitable for user interface design. You apply it in a larger design project that continues over the next chapters. (Parts of task analysis and data modelling are explained in the supplementary part, since they are well-known topics needed only for some audiences.)

Chapter 6: Virtual windows design. You learn how to design *virtual windows*: the large-scale structure of the user interface – how many screens, which data they show, how they show it, how you make sure that they support user tasks efficiently and how you check the results. You try all of this in a larger design project.

Chapter 7: Function design. You learn how to convert the virtual windows into computer screens, and how to add system functions to the screens based on the task descriptions and state diagrams. You also learn how to decide

whether a function is to be shown as a button, a menu, drag-and-drop, etc. Again you apply it in the large design project.

Chapter 8: Prototypes and defect correction. Based on the previous work products, it is rather easy to make a prototype – ready for usability testing. This chapter has a case study of how this worked in a real project, which problems were found, how they were removed and what happened when the final system was tested for usability. Your design exercise is to make a prototype for your design project, test it for usability and suggest ways to correct the problems.

Chapter 9: Reflections on user interface design. This chapter is theoretical. We review the virtual window method and compare it with other systematic design methods, for instance object-oriented design.

Courses

The book is based on material I have developed over many years and used at courses for many kinds of audiences.

IT beginner's course. The first semester for computer science and software engineering students is hard to fill in. Programming is mandatory in the first semester, and a prerequisite for most other IT courses. So what else can you teach? An introductory course in general computer technology and a database course are often the attempt. We have good experience with replacing either of these with user interface design based on this book. The course has a strong industrial flavour and students feel they learn to design real systems. And in fact, the course is readily useful. It is also a bit difficult, but in another way than programming.

Parts A and B of the book are aimed at this audience. Depending on what else they learn at the same time, we include data modelling from part C and constructing the user interface (the Access booklet). For some audiences, the full course is more suited for two semesters, and will then be an introductory systems development course at the same time.

When we include data modelling, the course replaces a large part of the traditional database course, and prepares students for an advanced database course. On our courses we often have students who have already followed a traditional database course, and to our surprise they follow our data model part enthusiastically. *Now we learn how to use it in practice*, they say.

We take the data model part concurrently with the soft parts about data presentation and mental models (part A). In this way we combine hard and soft topics and keep everyone on board.

We usually finish the course with a 4-hour written exam. This is not a multiple-choice or repeat-the-book exam. The students are for instance asked to specify

usability requirements, make a data model and design screens for a real-life case. They may bring whatever books they like to the exam. Chapter 17 contains sample exam questions.

IT-convert course. In recent years we have had many students that come from humanities or social sciences and want to become IT professionals. This book has worked amazingly well with this audience. The students are mature and learn fast. They appreciate very much the real-life attitude of the whole thing, and they feel motivated to enter also the hard technical part of the course (the Access booklet).

Courses for mature IT-students. The book makes excellent courses for software engineering or information systems students who know about programming and the technical aspects of design. Parts A and B of the book are suited for such courses. If the students have followed a traditional HCI course already, we treat part A lightly.

In these courses the data model part need not be included, and the Access booklet might be replaced by programming the user interface for any other platform the students might know.

Professional courses. We have used parts A and B of the book for professional courses taking 2–3 days. The participants are systems developers and expert users participating in development. Data modelling is not essential here; the professional developers usually know it already, and the expert users don't have to learn it when they cooperate with developers. Actually, we invented the novel parts of the book through such courses. We observed that some professional design teams produced excellent user interfaces, while others ended up with a messy interface that users found hard to use and understand. By comparing their work, we learned the cause of these differences and used it to improve the systematic design approach.

Course material

Overheads corresponding to all the figures of the book, and solutions to most of the exercises, are available for teachers. See www.booksites.net/lauesen or e-mail the author at slauesen@itu.dk.

The Access booklet is free for download from www.booksites.net/lauesen. The package includes a handy reference card for Access and Visual Basic, plus the hotel system application in various stages of completeness corresponding to the exercises in the Access booklet.

The author's background

At the age of 19, I started to work in the software industry for the Danish computer manufacturer *Regnecentralen*. At that time user interfaces were extremely primitive. We fed data into the system through punched cards or paper tape, and output was printed reports. When we later began to talk about on-line computing, input/output

was through typewriter-like equipment. Renting a computer for an hour cost the same as paying someone to work for 30 hours; and computers were 5000 times slower than they are today.

Nobody thought of usability. The customer paid until he had a program that printed results he could use with some effort. Everything to do with computers was a specialist's job.

My first project in 1962 was to develop a program that made the computer play a game invented by the Danish poet and designer Piet Hein. Neither he nor anybody else knew how to win the game. I had never programmed before, so I had to learn that too. And of course I didn't know that it was difficult to make such a program, so it took me a couple of weeks to solve the problem. Piet Hein wanted to have people play the game against the computer at exhibitions, but how could the user interact with the computer? Our hardware developers found a way to connect a bunch of buttons and lamps directly to the multiplier register of the CPU, and I made the program without multiplying anything. We then designed a nice user interface that people could use immediately. I didn't realize that this was the only good user interface I would make until 1973.

In the period until 1973, I developed many kinds of systems, for instance computation of molecule shapes based on X-ray diffraction, administration of pension contributions, optimization of tram schedules and rosters. Later I moved to another department in the company, where we developed compilers and operating systems. These systems became extremely fast, compact and reliable technical wonders. It took me many years to realize that we had made these miracles without understanding that the customers had other interests than speed and reliability. They also wanted usability – efficient task support – which we didn't provide and didn't understand.

In 1973, I moved to Brown Boveri, now part of ABB (Asea Brown Boveri). We were a new department and our first project was to develop a new line of process control systems with colour screens and special-purpose keyboards that could be operated by users with big, insulating gloves. Our first delivery was power distribution control for a part of Denmark. We knew how to make reliable, fast and compact code, but this was the first time I realized that ease of use was also important. If our system failed technically, 300,000 people wouldn't have power. But if the user couldn't figure out how to operate the system, the consequences might be the same. I had never been involved in anything as serious before. We made it, and usability became very good because we were inspired by the way users had controlled the power before.

In 1974–1975, I took temporary leave from Brown Boveri and worked for ILO in Ghana, helping with management consultancy in IT issues. This was the most fascinating year of my life. I learned how different other cultures and value systems could be, and that our own society had gained much in economic welfare and security, but lost a lot in other aspects of life quality. I also learned that I didn't know

anything about management and that such knowledge was important. Returning home, I took a business diploma as fast as possible.

In 1979–1984, I moved to a new software division established by NCR in Copenhagen. For some reason I became a manager, but kept being involved in software development – even on the programming level. I soon got two management responsibilities: (1) developing experimental technology for the next generation of products, and (2) assuring that the rest of the division delivered adequate quality in the present products. My main lesson in these years was that there was a very, very long way from proving a new technology to having a multinational company accept it for new products.

Now, where did I learn about usability? Not in industry! In 1984, I became a full professor at the Copenhagen Business School. They had decided to establish a new degree program – Business and Software Engineering. I found it an excellent idea since I had seen so many bright IT people who didn't understand business; and I had seen so many excellent managers who didn't understand technology. There was a communication gap, and educating young people in both areas at the same time seemed a great idea. I was willing to take responsibility for the IT side of the new education.

Working here without the daily pressure of finishing new products gave me time to wonder why the users and customers of all the technical wonders we had made over the years didn't like our systems as much as we did ourselves. I started to study usability and wondered whether users tried to understand what the computer did, or whether they just thought of carrying out their tasks, as most researchers assumed. I ran several experiments and concluded that if the system has a certain complexity, users form a mental model of how it works. They do so unconsciously, and often their model doesn't match with what the system actually does. This is the source of many usability problems. (Norman, 1988, found similar results in the same period.)

At that time, user manuals was a big issue, and I used the mental-model insights to develop ways to write manuals that combined two things at the same time: learning how to carry out the tasks, and understanding – at a non-technical level – what happens in the system. The approach was quite successful, and I served as a consultant for many technical writers and gave courses on writing user documentation.

Later I became more interested in designing good user interfaces – where manuals weren't necessary. During a long cooperation with Morten Borup Harning and varying master's students, we developed the approach that is central to this book.

I had expected that I would return to industry after about five years, but found that the combination of IT and business was fascinating, and that working at a business school helped me open the doors to a wide range of companies. Gradually I became a researcher who worked closely with industry.

In 1999, I moved to the new IT University established in Copenhagen. Now my role seemed to be reversed. At the business school, I had been regarded as the technical guy who didn't quite understand that business was the most important thing, while technology had minor importance. Now my computer science colleagues regarded me as the business guy who thought that business and usability were more important than technology.

This taught me one more thing: balancing between the extremes is very hard but also very important. I have tried to strike such a balance in this book.

Acknowledgements

The ideas behind virtual windows (Chapter 6) were developed by Morten Borup Harning and I, with some input from Carsten Grønning.

I have also learned a lot from cooperation with many of my students, in particular Susanne Salbo and Ann Thomsen who planned and carried out a hit-rate comparison of mock-ups against the real system. I want to thank William Wong for reviewing part of the book and encouraging me to publish it, Jean-Guy Schneider and Lorraine Johnston for helping me develop terminology in some of the areas, Klaus Jul Jeppesen for many insights in the user support area, and Flemming Hedegaard and Jacob Winther Jespersen for trying the entire course material in their own courses and giving me much valuable feedback.

Finally, my colleagues Peter Carstensen, Jan C. Clausen, Anker Helms Jørgensen and Rolf Molich have for many years been my excellent sparring partners in the HCI area, and I am most grateful for the insights I have got from them in many matters.

6

Virtual windows design

Highlights

- Virtual windows: Detailed screens for presenting data and for searching. No system functions yet.
- Few screens to ease learning and support tasks efficiently.
- Graphical design now to prevent disasters later.
- Check the screens and keep track of the defects.
- Key exercise: Design virtual windows for the design project.

Definition

A virtual window is a user-oriented presentation of persistent data. Early during design, the virtual windows may be on paper. They show data, but have no buttons, menus or other functions. Later we allocate functions to them, and finally they become computer screens, Web pages, etc. A complex application will need several virtual windows.

The basic idea when composing a set of virtual windows is this: Create as few virtual windows as possible while ensuring that (1) all data is visible somewhere and (2) important tasks need only a few windows. First we make a plan for what should be in each window (section 6.1), and next we make a detailed graphical design of the windows (section 6.2). Finally we check with users that they understand the windows, and we check against the task descriptions and the data model that everything is covered (sections 6.4 to 6.6). Usually there is a great deal of iterative design at this stage.

The way we design the virtual windows will help us balance between the two extremes: the database-oriented and the step-by-step-oriented user interfaces.

Although we present the design below in a step-by-step fashion, design is not an automatic procedure where you start in one end and end up with a good result. A good design always includes some magic points. The step-by-step procedure is good support for the magic – not a replacement for it.

6.2 Virtual windows, graphical version

The virtual windows plan can look quite convincing, and developers may conclude that they can come up with the graphical presentation when they design the final screens. When they later do so and try to fill in some realistic data, they realize that the outline doesn't work. There may not be sufficient space for realistic data, the user doesn't understand what the windows show, or the user doesn't get the necessary overview. Consequence: go back and redesign a lot – or ignore usability. For this reason it is important to make a careful graphical design in connection with the plan.

Design procedure

a) For each virtual window in the plan, make a detailed graphical design. Design only the data presentation. (Don't add buttons, menus or other functions. It is too early to fight against these complexities.)

b) Fill in the windows with realistic data. (They may need more space than you expect.)

c) Fill in the windows with extreme, but realistic data. (Extreme data must also be easy to see.)

d) If the windows don't look right, you may have to change the plan for the windows.

Figure 6.2 shows the result for the hotel system. It is not the designer's first version, but number two or three (sometimes hard to tell how many you have made). To give an overview of all the virtual windows in one figure, we have omitted some trivial fields, e.g. phone number, and we have shortened many others.

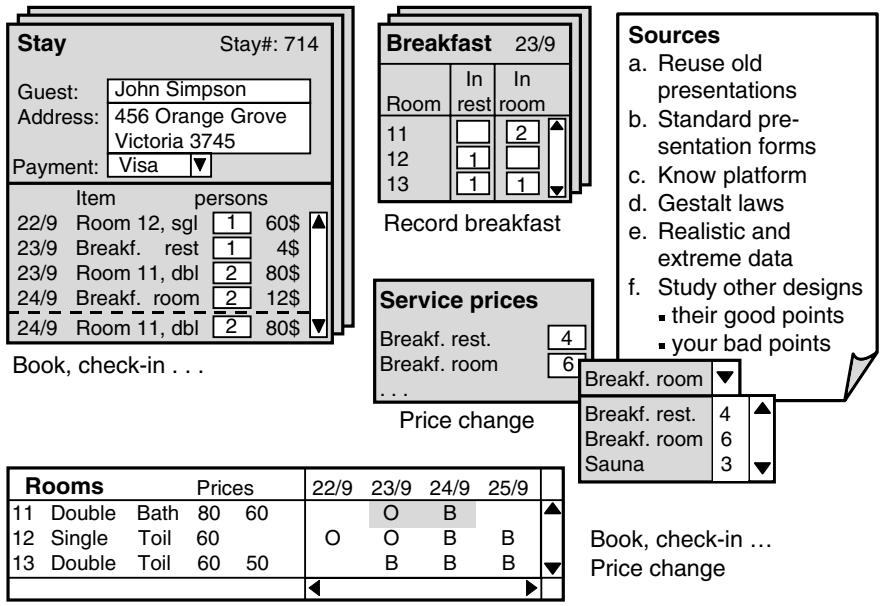
In the figure we have indicated the tasks that use each window. This is often convenient during design to check that all tasks are adequately covered and that some windows have multiple purposes. Let us give a few comments on the design.

Stay window. The Stay window has a top section about the guest and a bottom section with the rooms he has used and the services he has received. The dotted line separates the past from the future. Above the dotted line we have the nights he has stayed and the services he has received. Below the line we have the bookings – the nights he hasn't stayed yet.

We have indicated which fields the user can modify with a light border, resembling what the fields would look like on the final screen. We have also indicated that the user can choose the pay method with a combo box (drop-down list). Finally, we have used a scroll bar to indicate that the item list may be much longer.

This virtual window may appear almost as it is in the final computer screens. The window may also guide what the invoice should look like, making it easy for the receptionist to compare the two.

Fig 6.2 Virtual windows, graphical version



Breakfast. This window shows the breakfast servings for a single day. It has a line for each room and space for two kinds of servings: breakfasts in the restaurant and breakfasts served in the room. In practice, there might be some more columns for other frequent services. The window may appear in almost the same form on the computer screen, and it may also appear as a paper form filled in by the waiters.

Service prices. This window is a simple list of the different services including their current price. It will typically appear in two forms on the computer: as a separate window for changing prices and as a drop-down list when entering or editing service lines on the Stay window (see Figure 6.2). Conceptually it is the same virtual window, but it will look a bit different in the two forms.

Rooms. The rooms window uses a matrix form (like a spreadsheet). There is a line for each room and a column for each date. The letters O and B stand for Occupied and Booked. There may be a long list of room numbers and a long list of dates covering the past as well as the future. Scroll bars indicate this.

Try realistic and extreme data

It is important to test the design by filling in the windows with realistic data, as shown in Figure 6.2. (For space reasons, we have shown address and name fields that are much too short; also the Rooms window is much too small.) We have also shown a complex, but slightly unusual situation: a guest checks into a single room and gets

6.3 Virtual windows in development

Work product. A work product is the developer term for an intermediate result during system development. Figure 6.3 shows four important work products in user interface design:

a) The data model and the associated data description (data dictionary).

b) The task descriptions (in one of the many forms).

c) The virtual windows.

d) A list of design defects or things to find out.

They document different views of the final system, and they duplicate information to some extent.

Design sequence. In which sequence should we make these work products? Until now we have pretended that we first make data models and task descriptions, and then virtual windows. In many cases this sequence is okay, but some designers do it differently. They make virtual windows before the data model. This is because they can better imagine data when they see them as screens. They make the data model later to get a consistent overview of all the data.

Concurrent design. Personally I develop data model, tasks and virtual windows concurrently. Each of these work products tells me something about the other ones. When they are all okay, I feel that the design can move on to the next stage.

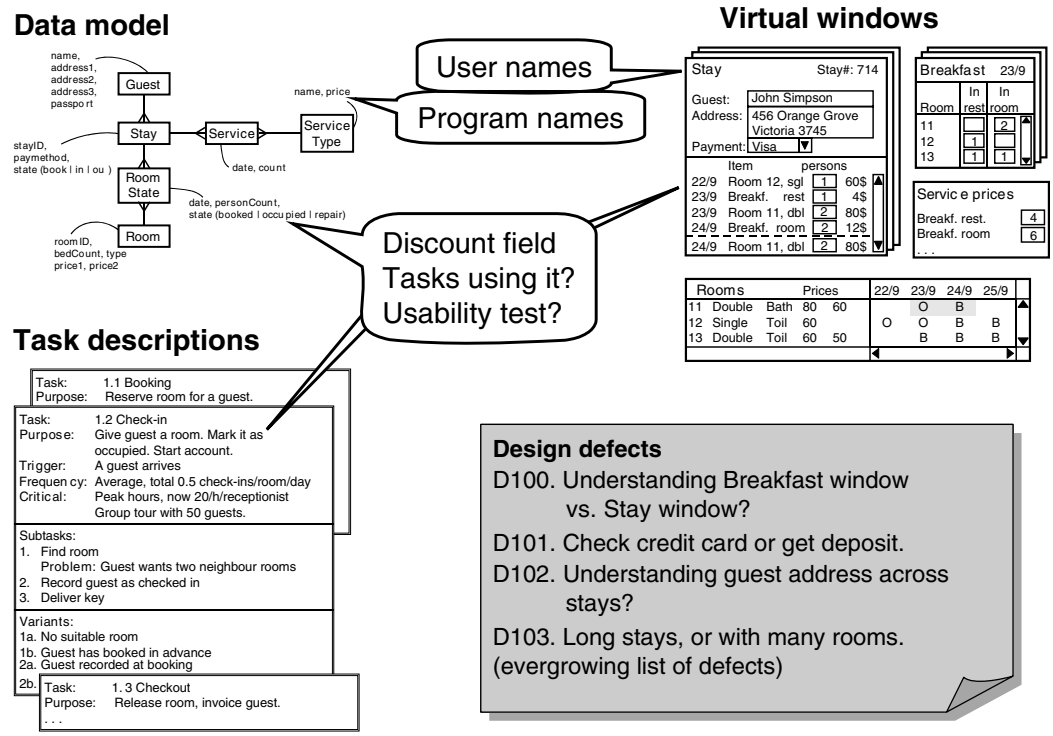
Need for early graphical design

In early versions of the technique, we didn't split virtual window design into a planning step and a detail step. We observed that some design teams produced excellent user interfaces, which scored high during usability tests, while other teams produced bad designs. Furthermore, *excellent designs were produced much faster than bad designs*. Why was that?

Gradually we realized that the main difference between good and bad teams was the amount of detail they put into the virtual windows. Both groups could quite fast produce the outline version. The bad teams then continued with dialogue design, but when designing the final screens, everything collapsed. The outline could not become useful screens; fields could not contain what they were supposed to; it was impossible to get an overview of data, etc. The teams had to redesign everything, resulting in a mess.

The good teams took some more effort designing the graphical details of the virtual windows, filling them with realistic data, etc. As part of that, they often went back to modify the window plan, grouping data in other ways. These changes were easy to handle at that time. From then on, things went smoothly. Dialogue functions were

Fig 6.3 Work products to maintain



added, and the screen design was largely a matter of cutting and pasting parts of the virtual windows. These interfaces also scored high in usability tests.

For this reason, we have since put much emphasis on a very early graphical design of the data presentation.

Typical problems

Adding functions too early. We have observed that designers tend to add some buttons to the virtual windows from the very beginning. This is against the idea of dealing with only data at this stage, delaying functions to later steps. The designers cannot, however, resist the temptation to put that *Check-in* button on the Stay window.

Why is it bad to add buttons at this stage? Because it easily makes you focus on the windows as function-oriented, for instance one window for data entry, another for data editing, a third for deletion. The result is too many windows, which makes the system harder to understand and also more costly to program. Focusing on the data presentation aspect helps you avoid this.

Another reason it is bad to add buttons at this stage is that the proper planning of buttons requires a good overview of the windows. There may be several choices for which window to put a button on, and you cannot make a good decision until you are sure which windows are available. It turns out that you also need to know how many buttons each window needs, in order to figure out whether there is space for the buttons at all or whether you need to put the functions in menus. Chapter 7 explains how to deal with all of this.

On the other hand, when you review the virtual windows with users, a few functions may help the users understand the window. This is particularly true for data entry and scrolling functions. For instance, we have noticed that users better understand that a field is for data entry if it is a box or a drop-down list. Users also better understand that something is a long list if it has a scroll bar. We have used these tricks in the virtual hotel windows.

Forgetting the virtual windows. We have observed many cases where designers made excellent virtual windows, only to forget them when designing the final screens. The physical design switched to being driven by available GUI controls and beliefs that traditional windows were adequate. The concern for understandability and efficient task support disappeared, and the final user interface became a disaster.

Two things can help in overcoming this: (1) Make sure that the virtual-window designers know the GUI platform to be used, in order that they don't propose unrealistic designs. (2) Ensure that quality assurance includes tracing from the virtual windows to the final screens.

Forgetting to update work products. The work products shown in Figure 6.3 give different views of the final system, and they duplicate information to some extent. As an example, the stay number field may be mentioned in all these documents.

This duplication of information is in one way a disadvantage because things may become inconsistent. We may, for instance, change the concept *stay number* to *booking number*, but change it only in some places – to the confusion of other developers. In other ways the different views are an advantage, because they help us find design defects. In the forthcoming sections we will see several ways to check things against each other, revealing serious design defects in the hotel system.

However, we often see that designers forget to keep the design results updated during the design process. The result is problems later. As an example, we may find out that the user interface needs a discount field when offering some guests a discount. We add it to the appropriate virtual window, but forget to put it into the data model, so the database will not have this field. Further, there should also be a task or task variant that deals with it, but we forget to update the task descriptions. What are the results? During programming a lot of time is wasted changing the database structure to include the discount stuff. And since the task description isn't changed, we forgot to test whether the user can figure out how to use the discount field.

List of design defects. We also see that designers observe a problem, but later forget about it. For this reason it is important to maintain a list of *design defects* (or design problems). Good developers do this no matter how they design the system. In the hotel case, we noted that at debriefing after usability tests we should check whether the user understands how the breakfast window cooperates with the stay window. We might ask questions such as:

> *When you have recorded breakfasts with this window, what would you expect that this Stay window should show?*
>
> *When you have recorded breakfast directly in the Stay window, what would you expect that the breakfast window should show?*

Here is a summary of the problems we have noted earlier during the hotel system design.

D1–D35: Problems noted during the first usability test of a hand-drawn mock-up (see Figure 2.3B).

D100: Do users understand that data in the Breakfast list window automatically end up in the stay windows – and vice versa? (from section 6.1, Record services).

D101: Receptionist should check credit card or get deposit before check-in (from section 5.3.2, Review with expert users).

D102: Do users understand that guest name and address is repeated in all the guest's stay windows – while pay method is individual for each stay? (from section 6.1, Checking against design rules).

D103: How to show long stays or stays with many rooms? (from section 6.2, Try extreme data).

Some developers use more space to record and describe each defect, also leaving space for describing possible solutions. They may even use a full paper page per problem. It may be based on a template where the developers also record time found, way found, evidence for the problem, seriousness of the problem, possible solutions, etc. Other developers record the problems in a database, which allows both the full-page printout, and various summaries and overviews.

User names and program names. Names of data fields, etc., will appear in several places in the final system, for instance in screens, prints, help texts, and maybe course material. To avoid confusing users, we should ensure consistency, so it is important that we early on define the names to be used in the user's language.

A good place to do it is in the virtual windows. This means that we have to carefully choose field names and other names, as they will become the standard on the user interface. (Our example in Figure 6.2 is not good, because we have abbreviated names in order to give an overview.) We can review the virtual windows with users and in that way check that the names make sense to them.

7.1 Semantic functions and searching

In order to identify the system functions, we take the tasks one by one and manually carry them out by means of the virtual windows. During this, we note down which functions we would need in each virtual window. Some usability specialists call this a *cognitive walk-through* of the tasks. Here is the procedure.

Design procedure

a) Place the virtual windows on a desk. Imagine that they are on a huge computer screen. Virtual windows with many instances (e.g. the stay windows) are put in a pile so that a search function is needed to find the right one.

b) Look at an important and frequent task. You might place the task description in the corner of the desk.

c) Look at the first subtask and imagine how you would carry it out. Which virtual windows would you have to look at, and which fields would you have to fill in? Which system functions (buttons) would be needed to make the system do something, for instance search, create items, save data, delete items, print or send something? You may also need 'buttons' that change data in special ways – other than simple data entry.

The system functions you identify in this way are semantic functions, search functions and non-trivial data entry functions. Don't care about navigation functions that take you from one virtual window to another. The 'screen' is so large that it has space for at least one copy of each virtual window. At this stage you can navigate between virtual windows simply by pointing at them.

d) When you have identified a 'button', give it a name that might be written on the button and write it down beside the virtual window where it was needed. Often the button has to do something complex. Write a short description of what it should do (a *mini-spec*).

Don't worry that we call the functions 'buttons'. Later they may become menu items, drag-and-drop or real buttons.

e) Look at the next subtask and imagine how you would carry it out. Again you identify system functions (buttons), but try to reuse the buttons you have defined already. When you reuse a button, it may happen that it should work a bit differently than before. Amend the mini-spec to explain about it.

f) When you are through all subtasks and variants, the system should be able to support the entire task. Review what you have defined. Check to see that you can carry out the task in other reasonable ways, for instance performing the subtasks in other sequences. Sometimes you may have to make minor changes to buttons and mini-specs to achieve this.

g) Now look at the remaining tasks one by one in the same way. Try to reuse the earlier buttons as far as possible. You will normally experience that tasks need fewer and fewer new buttons as design progresses.

h) Review the entire set of functions. Check whether it is possible for the user to switch between two or more tasks at the same time. Check that standard functions are provided, such as Undo, Print and Data exchange.

Is it necessary to make some kind of CREDO check to see that all records in the database can be created, deleted and updated? In principle not, because if we made the CREDO check for tasks against data model, all the necessary tasks and subtasks should be there. So if the system can support all the tasks, it can also create, delete and update what is needed. However, systems development is so complex that designers easily overlook things. So checking once more might be a good idea.

Booking task

Below we will show how the design procedure worked for the hotel system. We start with the booking task.

Figure 7.1A shows our desk with some of the virtual windows. At the bottom right we have placed the task description for the booking task.

Subtask 1, Find rooms. How do we carry out the first subtask? Well, we need the virtual window *Rooms* with the search criteria (vwRooms). We fill in the search criteria for the booking: the type of room the guest wants, and the period of the stay.

Then we somehow activate a button *FindRooms*, and the system shows a list of the appropriate rooms. We have now identified the first 'button' – a search function. We write this function besides vwRooms as shown. Most likely it will become a button or a menu point in that window, but we delay the decision. (Maybe it will be the button we have already put on the virtual window.) For now it suffices to know that it is a function associated with this virtual window.

If one of the rooms suits the guest, we select this room. This gives us another function, *ChooseRoom*. To illustrate the approach, we have written the subtask number besides the function to show when we use the function. Both functions were used for subtask 1.

Subtask 2, Record guest. We continue with the next subtask. It has a variant *Regular guest* for guests that already are in our files. Often guests don't know whether they are in our files, and the standard hotel procedure is to always check whether they are. So it is a good idea to use the window vwFindGuest. We enter part of the guest name and activate a new function *FindGuest*, which shows a list of the guests that match. If one of the guests is the right one, we select the guest (*SelectLine*) and activate *NewStay*. Up comes a window for a new stay (*vwStay*), and the system pre-fills the guest data. We can now check with the guest that his data is still correct, and edit it as needed.

If many guests have names that match, how do we know which of them is the right one? As the virtual search window is designed, we may not have enough data, but if we somehow show the details for each guest as we point down the list, we can determine the right person. We will leave this as a note for SelectLine.

This was variant 2a. If it is not a guest in our files, we don't select anything but just use *NewGuest*. Up comes an empty stay window where we enter the guest data. Whether we have a new guest or retrieved a regular guest, we now have the necessary guest data.

Could we avoid *NewGuest* and reuse *NewStay* instead? If we point at a guest, *NewStay* should show this guest for the new stay, but if we don't point at anything, *NewStay* should show blank guest data – just what we need. However, the problem is whether the user can tell the subtle difference between pointing at nothing and pointing at an incorrect guest. We want to be sure, so we define a separate function for a new guest.

In order to carry out step 2 or 2a we thus needed four functions on vwFindGuest, as shown in the figure. For the sake of completeness, we have also indicated that we enter persistent data in the Stay window (EditData). This is a trivial data entry function.

Special data entry. Most of the data entry functions are trivial, and we have implicitly defined them as part of the virtual window design. However, there are a few special data entry functions in the virtual windows. For instance, the user should be able to fill some of the fields by choosing from a list rather than typing – using a combo box or something similar. In vwRooms, this includes the fields *Type*, *FreeFrom* and *Departure*.

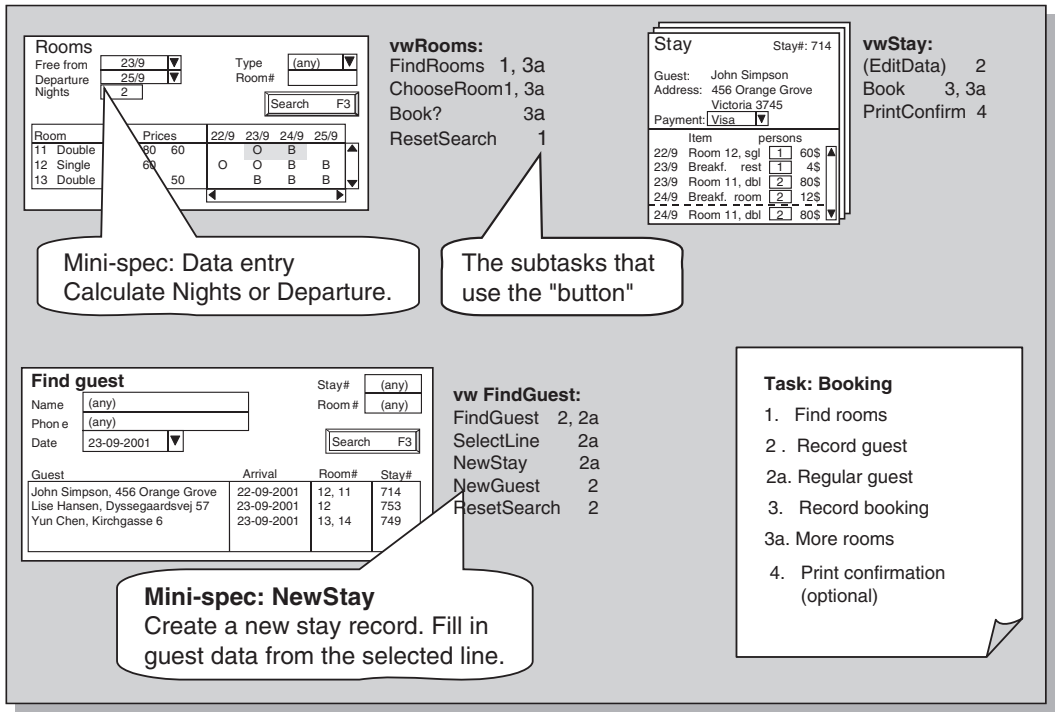
There is a tricky thing with the fields *Departure* and *Nights*. Sometimes the receptionist prefers to specify the departure date, and at other times the number of nights stayed. And he likes to check the two things against each other. We want the system to calculate the field he hasn't specified. To keep track of these details we write them in a mini-spec, as explained below.

We have added a special data entry function in vwRooms: *ResetSearch*. It sets the search criteria to their default value: *(any)* for most fields and the current date for the FreeFrom field. This function is, strictly speaking, unnecessary because the user can set the fields one by one, but it is annoying to do. We have added a similar function to vwFindGuest.

Subtask 3, RecordBooking. At this stage we have a Stay window with guest data and we have selected a room. What we need to do is somehow activate a Book function. It seems natural to activate it through vwStay, so this is where we put the Book function. The function will store the guest and stay data into the database.

We have a variant of subtask 3. The guest may want more than one room, maybe of another kind and for another period. We can again handle this through vwRooms by means of FindRooms and ChooseRoom, as shown in the figure. When we have selected the room, we use Book once more. This time it may be convenient to have the Book function on vwRooms, and we have shown it there with a question mark.

Fig 7.1 A Booking: semantic functions and search

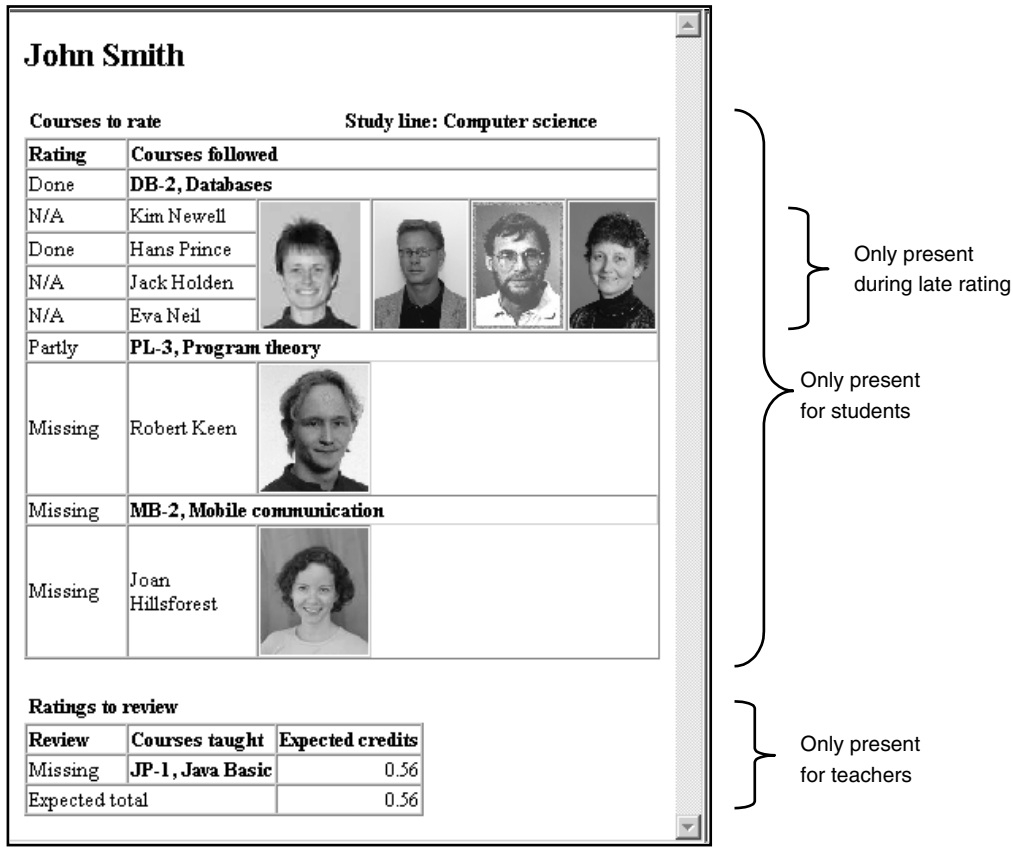


Can we have the Book function in more than one window? Yes, why not. The Book function requires two things (two *parameters* or *preconditions*) : rooms selected and guest data properly filled out. This suggests that the function could be on either vwRooms or vwStay, or in both places. Actually, we cannot make the decision now. It depends on the platform and the navigation functions, which we haven't looked at yet.

Subtask 4, PrintConfirmation. Finally, we may have to confirm the booking by printing a confirmation and sending it to the guest. This is easy. We just need a PrintConfirm function. A convenient place would be at the Stay window, since it is this stay we confirm.

Task sequence. We have earlier stressed that the user should be allowed to vary the sequence of subtasks (sections 4.6 and 5.3.2). Above we have only looked at one particular sequence. Could we choose another sequence? Yes, we could very well do subtask 2 first, finding or recording the guest. Next we could do subtask 1, finding a suitable room. The only restriction is that we cannot use Book until we have done both subtasks 1 and 2. This is not a restriction caused by the user interface, but a rule in the application area (a *business rule*).

Fig 10.4 D Virtual window, Person (student and/or teacher)

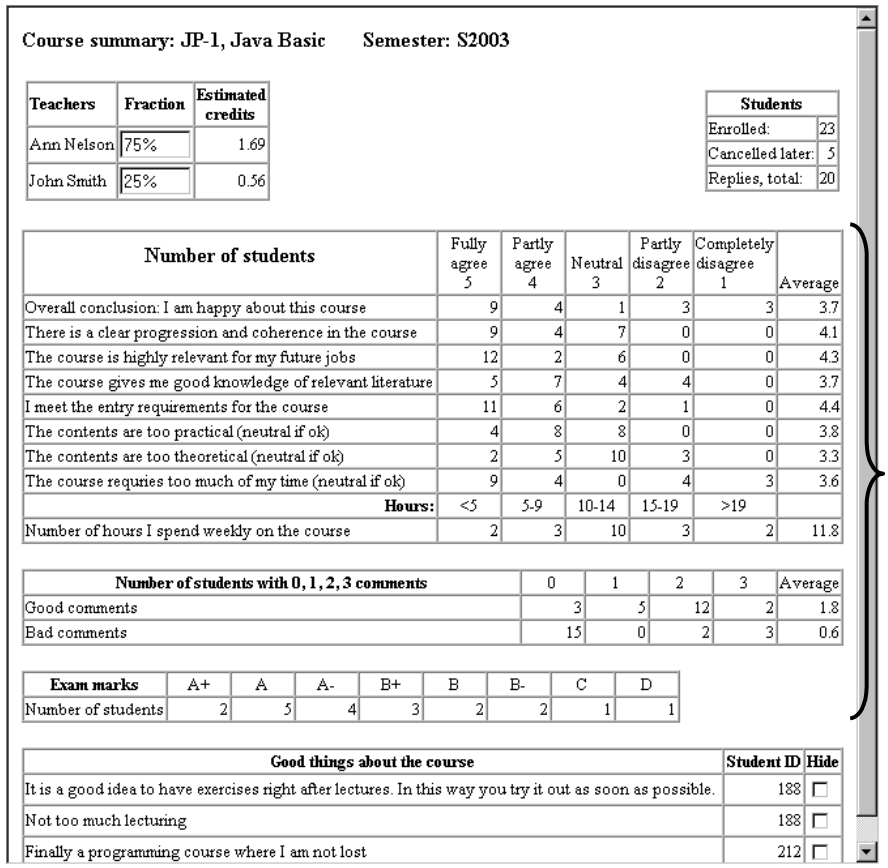


similar to the Course summary). In principle, we don't need additional virtual windows for this. The teacher could go through all the students' rating windows (made anonymous, of course). However, this would violate the design rule about few window instances for important tasks, and it also gives a bad overview of data.

After the late rating, the Course summary window shows the numerical ratings in the same form as the student's rating window, but with totals in each field. The average is also shown. (This is very similar to the way course summaries were shown in the old system.)

After early and late rating, the window shows the student's remarks as a long list. The first part of the list shows the good points, the last part the bad points. The teacher can hide remarks from publication by means of a check box. This is very similar to the old system. For courses with many students, the list may be very long, but it is hard to get an overview. To help a bit, I have added a few things in the new system:

Fig 10.4 E Virtual window, Course summary



Course summary: JP-1, Java Basic **Semester: S2003**

Teachers	Fraction	Estimated credits
Ann Nelson	75%	1.69
John Smith	25%	0.56

Students	
Enrolled:	23
Cancelled later:	5
Replies, total:	20

Number of students	Fully agree 5	Partly agree 4	Neutral 3	Partly disagree 2	Completely disagree 1	Average
Overall conclusion: I am happy about this course	9	4	1	3	3	3.7
There is a clear progression and coherence in the course	9	4	7	0	0	4.1
The course is highly relevant for my future jobs	12	2	6	0	0	4.3
The course gives me good knowledge of relevant literature	5	7	4	4	0	3.7
I meet the entry requirements for the course	11	6	2	1	0	4.4
The contents are too practical (neutral if ok)	4	8	8	0	0	3.8
The contents are too theoretical (neutral if ok)	2	5	10	3	0	3.3
The course requries too much of my time (neutral if ok)	9	4	0	4	3	3.6
Hours:	<5	5-9	10-14	15-19	>19	
Number of hours I spend weekly on the course	2	3	10	3	2	11.8

Number of students with 0, 1, 2, 3 comments	0	1	2	3	Average
Good comments	3	5	12	2	1.8
Bad comments	15	0	2	3	0.6

Exam marks	A+	A	A-	B+	B	B-	C	D
Number of students	2	5	4	3	2	2	1	1

Good things about the course	Student ID	Hide
It is a good idea to have exercises right after lectures. In this way you try it out as soon as possible.	188	☐
Not too much lecturing	188	☐
Finally a programming course where I am not lost	212	☐

Only present during Late Rating

- The remarks are ordered according to their author. To preserve confidentiality, each student gets an anonymous number for this course, and the number is shown besides the remark. This allows the teacher to get an impression of whether a lot of students made this kind of remark or only a few who repeated it in various ways. (With the old system, teachers complained about not having this information). The name behind the number is kept secret, of course, unless the student wants to be known. In the latter case, the student number is not shown, but the student *name* is shown in the remark box. There is no need to store the anonymous number in the database. The system can generate the number when it generates the screen picture.
- A three-line section of the window gives an overview of how many students made good comments and bad comments. This also helps the teacher get a feeling for how big the problems are.