Last changed by: Soren Lauesen
18-04-2009

VisTool for Unified Data Visualization

1. Introduction

Users often find IT systems hard to use. The user interface is difficult to learn and cumbersome to use. In many cases the users need overview of a lot of data, but the system can only show simple screens with text fields, or screens with lists of data, for instance as in accounting systems. Overview of a different kind is possible with advanced data visualizations (using graphics rather than text), but this is rarely integrated into the user interface.

There are several reasons for this situation:

- Methods for designing user-friendly interfaces are not widely used. The best approach is to iteratively develop prototypes and test the usability of them with typical users. For economic reasons the prototypes are mockups without real functionality. Even this is often considered expensive and unnecessary.
- User interfaces need changes as new demands and possibilities are discovered during use. Today this requires programmer assistance, even for fairly simple changes.
- An advanced visualization shows data as screen position, size, shape and color - often combined with traditional presentation in text form. This is hard to program - even for experienced programmers.

To solve these problems, we develop **VisTool**. The tool will make it possible for **designers** to fully construct traditional as well as advanced user interfaces without real programming. Designers don't need programming knowledge, but they need IT knowledge corresponding to making spreadsheets. They might for instance be local super-users in a department. The tool will make it possible to produce fully functional prototypes faster than mockups can be produced today. When a prototype has been tested for usability as well as functionality, it can be deployed for the end-users right away.

Figure 1 shows two examples that combine traditional user interfaces with advanced visualization. Both examples might have been produced by designers from simple components such as text boxes, simple boxes and arrows. The designers would connect the components to a database so that real data is shown. One of the tricks in VisTool is to allow a component to repeat itself according to records in a database. Another trick is to allow components to have data-dependent positions and colors.

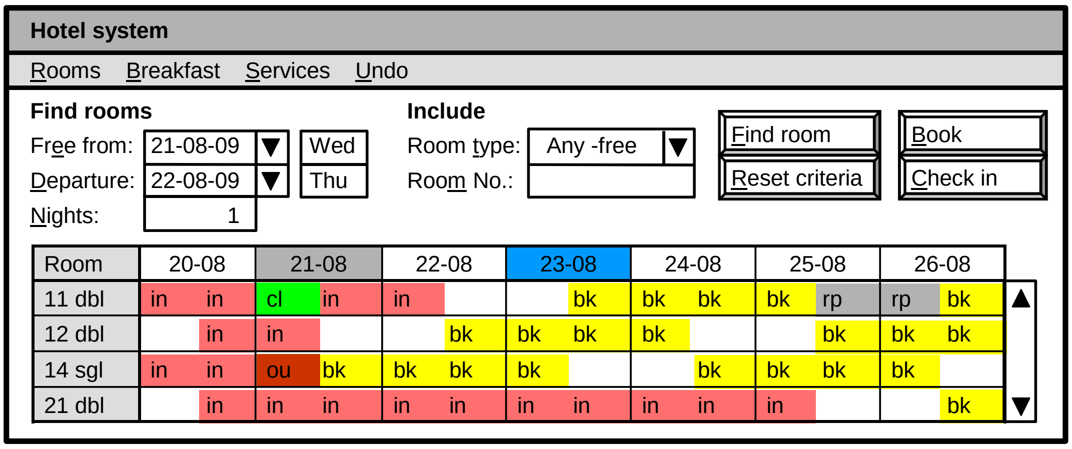
The first example is part of a hotel reception system. It gives the receptionist an overview of the hotel rooms and their present and planned state. The receptionist can use search criteria to see only some of the rooms and a specific range of dates. At first sight the grid of rooms and dates looks like a traditional cross tabulation, but each cell has two parts: the state of the room in the morning and the state in the afternoon. The two parts are two text boxes that repeat themselves. Colors show whether the room is occupied, booked, etc. The grid heading uses colors too to show Sundays, etc.

The second example is a screen from an Electronic Health Record system (EHR). It gives an overview of medical information for a patient. It shows diagnoses, doctor's notes, medication, lab results, etc. in such a way that various relationships between these things are easily visible. The end-user may click on for instance a note icon to see note details or on a medication box to see medication details.

We show details of these examples later. The same components may be used to construct also traditional visualizations such as Gantt charts and business graphics.

Our plan is to test VisTool by developing user interfaces for Electronic Health Records, primarily in hospitals. This area is known to be very demanding. For this reason, we use many health record examples below.

Fig. 1. Examples: Hotel system and Electronic Health Records



Hotel system

Rooms Breakfast Services Undo

Find rooms

Free from: 21-08-09 Wed

Departure: 22-08-09 Thu

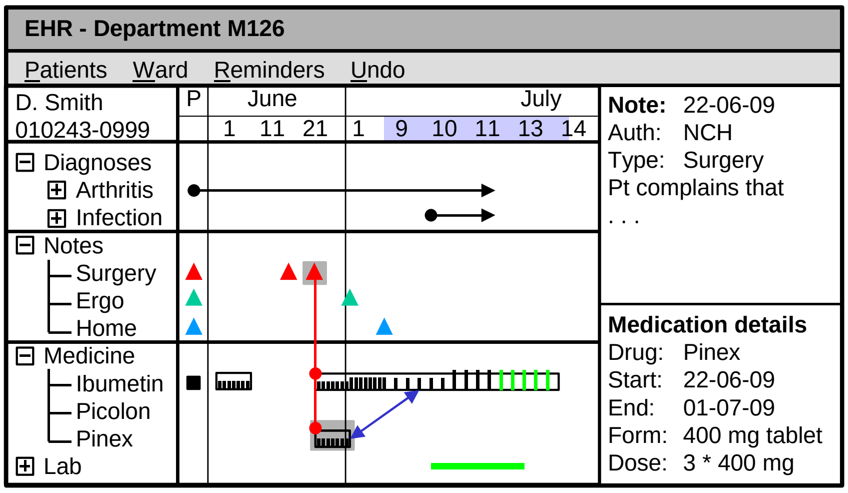
Nights: 1

Include

Room type: Any -free

Room No.:

Room	20-08		21-08		22-08		23-08		24-08		25-08		26-08	
11 dbl	in	in	cl	in	in			bk	bk	bk	bk	rp	rp	bk
12 dbl		in	in			bk	bk	bk	bk			bk	bk	bk
14 sgl	in	in	ou	bk	bk	bk	bk			bk	bk	bk	bk	
21 dbl		in	in	in	in	in	in	in	in	in	in			bk



EHR - Department M126

Patients Ward Reminders Undo

D. Smith
010243-0999

Note: 22-06-09
Auth: NCH
Type: Surgery
Pt complains that
. . .

Medication details
Drug: Pinex
Start: 22-06-09
End: 01-07-09
Form: 400 mg tablet
Dose: 3 * 400 mg

2. State of the art in user interface construction

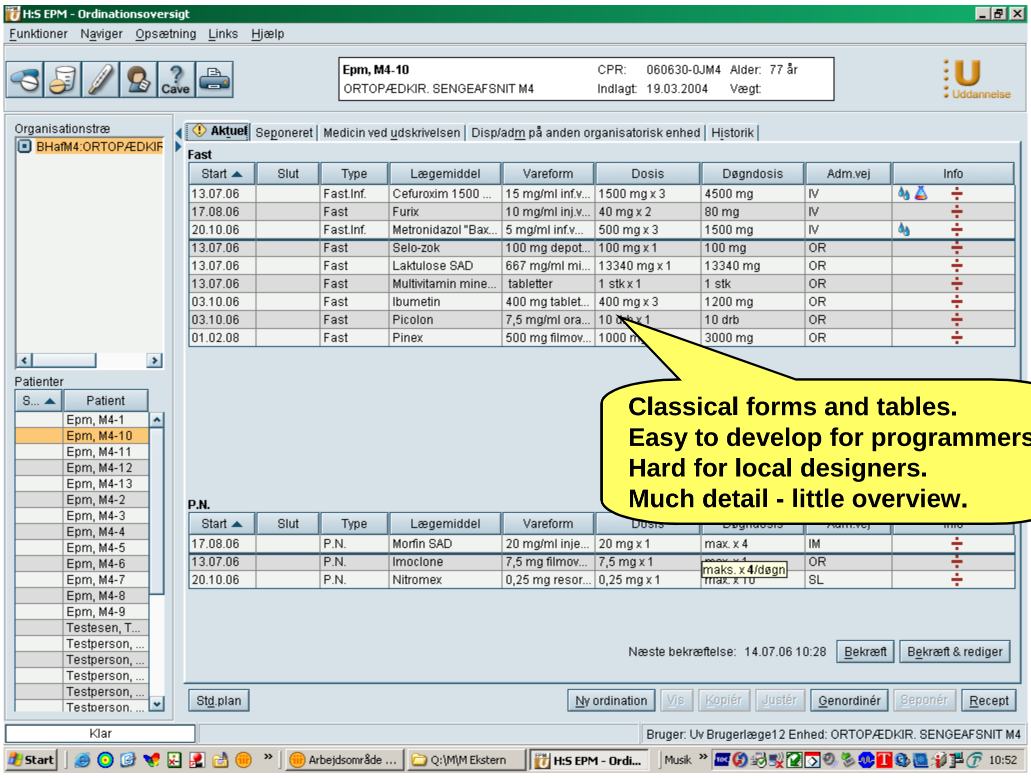
Figure 2 is an example of a classical computer screen, similar to screens found in business applications. The example is from a system for ordering and recording of patient medication (developed by Acure and deployed at Bispebjerg Hospital). Data is presented as text fields, often arranged as lists of data records (*datagrids*).

Such screens provide a lot of detailed information, but give a poor overview. In order to see the kinds of medicine and which kinds are given at the same time, the user has to scrutinize several pieces of text.

These screens are fairly easy to develop for a programmer. Good tools have existed for this for several years. Early tools such as Microsoft's Access are actually suited for designers too, and small Access applications still proliferate, also at hospitals. However, these applications are unstable, particularly in a multi-user environment. The application often crashes or the database becomes inconsistent. Integration with other systems and databases is not easy. All of these tools have very limited ways of presenting data. As an example, it is hard or impossible to give fields a color that depends on data. This would be useful if we for instance wanted to show urgency of a medication with a color.

The trend in recent years seems to be to split the work of user interface construction. The graphical designer develops the visual appearance of the screen, for instance for a web page. Next comes the programmer and provides binding to the database and other functionality. The trend is that this programming becomes more and more complex. The approach is not suited for the designers who are our audience. They need to see real data in the screens while they design them. And they don't use colors and pictures to entertain or sell, but to provide efficient support of users.

Fig. 2. A traditional screen (Acure's EPM, Bispebjerg Hospital)


H:S EPM - Ordinationsoversigt
Funktioner Naviger Opsætning Links Hjælp
Cave
Uddannelse
Organisationstræ
BHafM4:ORTOPÆDKIF
Aktuel | Seponeret | Medicin ved udskrivelsen | Disp/adm på anden organisatorisk enhed | Historik
Std.plan
Ny ordination
Vis
Kopiér
Justér
Genordinér
Seponér
Recept
Bekræft
Bekræft & rediger
Klar
Start
Arbejdsområde ...
Q:\M\M Ekstern
H:S EPM - Ordi...
Musik
10:52
Classical forms and tables.
Easy to develop for programmers.
Hard for local designers.
Much detail - little overview.

Epm, M4-10 CPR: 060630-0JM4 Alder: 77 år
ORTOPÆDKIR. SENGEAFSNIT M4 Indlagt: 19.03.2004 Vægt:

Fast

Start	Slut	Type	Lægemiddel	Vareform	Dosis	Døgndosis	Adm.vej	Info
13.07.06		Fast.Inf.	Cefuroxim 1500 ...	15 mg/ml inf.v...	1500 mg x 3	4500 mg	IV	
17.08.06		Fast	Furix	10 mg/ml inj.v...	40 mg x 2	80 mg	IV	
20.10.06		Fast.Inf.	Metronidazol "Bax...	5 mg/ml inf.v...	500 mg x 3	1500 mg	IV	
13.07.06		Fast	Selo-zok	100 mg depot...	100 mg x 1	100 mg	OR	
13.07.06		Fast	Laktulose SAD	667 mg/ml mi...	13340 mg x 1	13340 mg	OR	
13.07.06		Fast	Multivitamin mine...	tabletter	1 stk x 1	1 stk	OR	
03.10.06		Fast	Ibumetin	400 mg tablet...	400 mg x 3	1200 mg	OR	
03.10.06		Fast	Picolon	7,5 mg/ml ora...	10 drb x 1	10 drb	OR	
01.02.08		Fast	Pinex	500 mg filmov...	1000 m	3000 mg	OR	

P.N.

Start	Slut	Type	Lægemiddel	Vareform	Dosis	Døgndosis	Adm.vej	Info
17.08.06		P.N.	Morfin SAD	20 mg/ml inje...	20 mg x 1	max. x 4	IM	
13.07.06		P.N.	Imoclone	7,5 mg filmov...	7,5 mg x 1	maks. x 4/døgn	OR	
20.10.06		P.N.	Nitromex	0,25 mg resor...	0,25 mg x 1	max. x 10	SL	

Næste bekræftelse: 14.07.06 10:28

Patienter

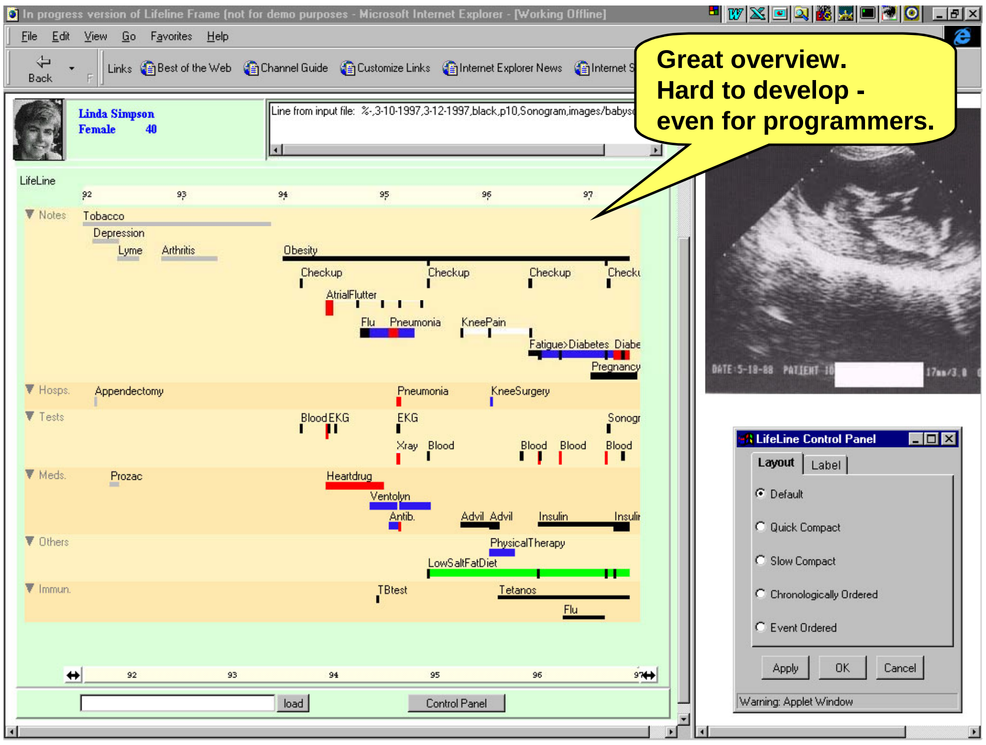
S...	Patient
	Epm, M4-1
	Epm, M4-10
	Epm, M4-11
	Epm, M4-12
	Epm, M4-13
	Epm, M4-2
	Epm, M4-3
	Epm, M4-4
	Epm, M4-5
	Epm, M4-6
	Epm, M4-7
	Epm, M4-8
	Epm, M4-9
	Testesen, T...
	Testperson, ...
	Testperson, ...
	Testperson, ...
	Testperson, ...
	Testperson, ...

Bruger: Uv Brugerlæge12 Enhed: ORTOPÆDKIR. SENGEAFSNIT M4

Figure 3 is strikingly different. It shows diagnoses as well as medication and other treatments on a single screen with very few texts. The time when something happens is shown on a horizontal time scale. It is immediately visible what goes on at the same time. This is an example of advanced visualization. It was developed by Shneiderman and his team. They called it a LifeLine presentation.

Such screens are hard to develop with present tools, even for a good programmer. If you want to show things a bit differently, you need the source code and a programmer who knows the code.

Fig. 3. Novel overview: LifeLine (Shneiderman)



3. VisTool example

Figure 4 is an example of what our tool should be able to do. The central part of the figure is a *Form* (a screen) similar to Shneiderman's LifeLine. Where he uses horizontal bars to show everything, our example uses different symbols for different things. For medication it shows a medicine order as a box extending over the period where the medicine is given. It also shows each intake of the medicine as a small vertical line. The height of the line shows the amount given. Black lines are intakes that have been given. Green ones are scheduled intakes.

The essential point is that a designer might have composed this advanced visualization from simple building blocks. The building blocks may be combined in many other ways to produce curve graphs, Gantt diagrams, traditional fields and tables, and novel visualizations.

The main building blocks used in the example are:

1. A Horizontal Scale at the top, set up to show the calendar. Different parts of the scale have different zooms. The left part (P for *Past*) is zoomed out to show the entire past of the patient.
2. A Data Row at the bottom left, set up to show medicine types in an expand-collapse hierarchy. It will automatically produce a number of rows according to the contents of a database.
3. A Box set up to show medicine orders for a specific patient. It will automatically produce one box for each of the patient's medicine orders as they are recorded in a database. The boxes will position themselves according to time and medicine type.
4. Another Box set up to show an intake of medicine as a small line. It will automatically produce one line for each intake.
5. The Form consists in this case of a single Grid with 2 * 4 cells. The horizontal scale occupies the top right cell. The DataRow occupies the lower left cell.

Fig. 4. VisTool example: LifeLine composed of simple controls

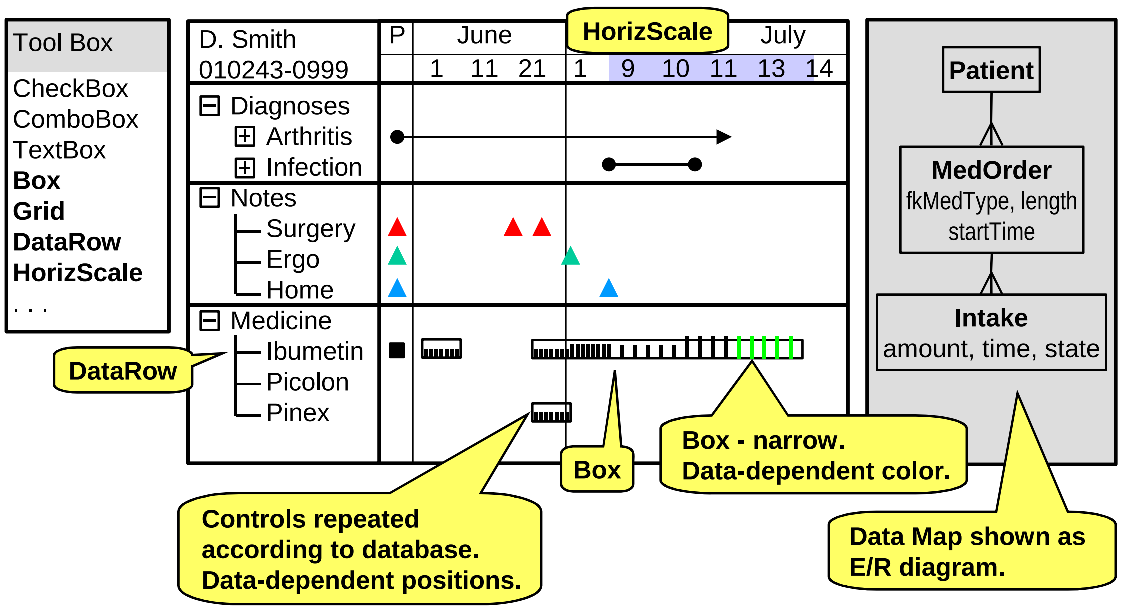


Figure 4 is also a snapshot of the designer's workbench when constructing this screen. It looks much like existing development tools such as Eclipse or Visual Studio. The toolbox at the left shows all the building blocks. In order to make this Form, the designer chose the various blocks from the toolbox and dragged them to the Form.

In the world of user interface design, the building blocks are called **controls** or **widgets**. We will use the term *control* below. A control has a number of *properties* that specify its position on the screen, its size, color, etc. With present tools, most of the properties have a fixed value defined by the graphical designer, so that for instance a specific textbox always will be blue and positioned at the top left. The text shown inside the textbox is the only property that changes according to data, and often some programming is needed to make it do this.

Present tools provide simple controls such as textboxes and combo-boxes. The visualization tool will provide these too, but give them additional functionality that allows them to produce several copies of themselves, set their color according to data, etc. In addition the visualization tool will provide a few new types of controls. The Horizontal Scale and the Data Row mentioned above are examples.

The data map at the right shows the data tables the designer can address. For each control he can specify that its height depends on a specific field in the database, its color on another field, etc. As an example, he can specify that the color of the *intake* line depends on the state field in the Intake database. He can also specify that the control has to repeat itself for each record in a selected part of the database. All of this is done by writing formulas for the properties instead of fixed values. The formulas are much like those in a spreadsheet. We show detailed examples in section 5.

4. The tool architecture

How does such a tool interact with the surroundings? Figure 5 shows this as a context diagram. The large box is the visualization tool. The tool consists primarily of two kinds of components:

1. **Application screens (Forms)**, e.g. one set for orthopedics and another for intensive care.
2. **Data classes**, e.g. admin data from an external Admin database and medication data (EPM) from an external EPM database. The application screens show data from these databases.

In order to allow the building blocks to work on all kinds of data in existing, external systems, the data must be mapped into a standardized format in the data classes. Depending on many factors, the mapping may be a view of the external data or a copy. It may allow reading and/or writing of data.

The tool interfaces to four user groups:

3. **PC users** work in the application domain, for instance as clinicians (nurses and surgeons).
4. **Mobile users** also work in the application domain, but move around and use mobile phones or PDA's. PC user and mobile user are two roles. Clinicians, for instance, change role frequently. We will use the term *end user* to denote a PC user or a mobile user.
5. **Local designers** design and construct the screens of the user interface. They have IT skills corresponding to people who construct spreadsheets. They might for instance be clinicians with an IT interest.

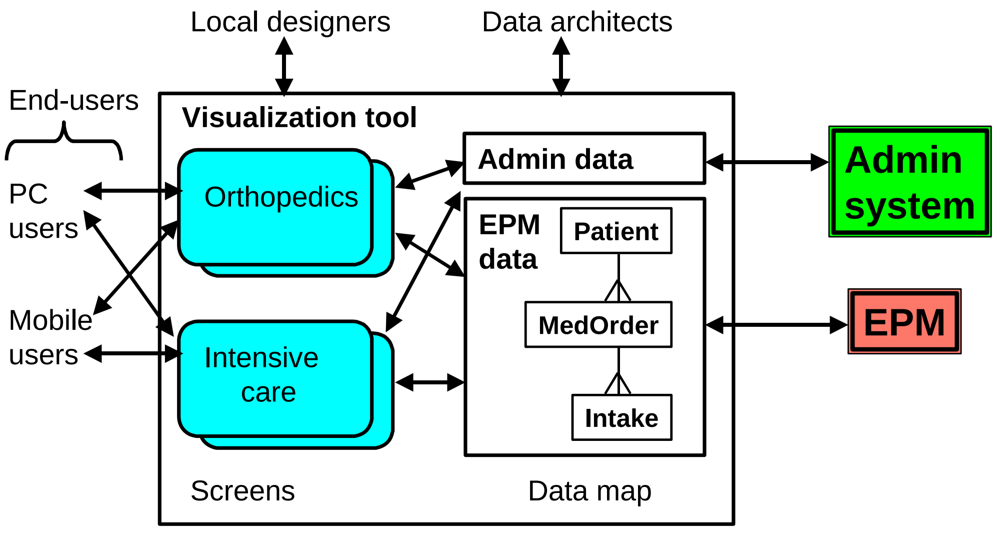
6. **Data architects** construct mappings of external data to the data classes used by the screens. Many aspects of security are also handled by this mapping. As an example, hospital data would only comprise patients currently admitted to the end-user's department. Data architects have wider IT skills, know about databases, XML and security.

The standardized data format might be defined in many ways. It might for instance be based on relational database tables, XML, or container classes. Since the designers have to connect the controls to data, they must specify data addresses in some way. It is very important that these addresses are easy to understand and convenient to read and write.

The example in Figure 5 shows external data from two different systems:

7. **Admin system** is an old mainframe system that holds patient names, addresses and hospital departments where the patient has been admitted. We have not shown the details of this.
8. **EPM** is a modern electronic medication system. For each patient it records the medicine ordered for the patient. And for each order it records each instance where the patient got this medicine (the intakes). We assume that the same patient ID can be used for Admin data as well as EPM data.

Fig. 5. VisTool: Context diagram



5. Details of the LifeLine example

This section explains more about the controls and their properties. Figure 6 shows some of the properties of the Horizontal Scale, the Data Row, and the Box that shows a medicine order.

5.1. The data map

The data map at the right shows the data the designer can access. It now shows one more database table: *MedType*. This table has a record for each kind of medicine available. It holds the name of the medicine, e.g. "Ibumetin" and the standard daily dose for an adult. The crow's foot to MedOrder shows that each MedType may refer to several MedOrders, while each MedOrder may refer to only one MedType.

The data panel also shows **dialog data** that only exist during the end-user's dialog with the system. The important dialog data in this example is the ID of the patient that the end-user currently works with (curPt). It is shown as a crow's foot that refers to a record in the Patient table. Other dialog data will show the ID of the current end-user, the control currently selected (for instance when the user clicks one of the medicine bars), windows currently open, etc.

5.2. The main Grid

The LifeLine screen consists of a grid with 2 * 4 large cells. The cells are numbered in the spreadsheet manner as A1, B1, A2, B2, etc. The end-user may move the boundary between the cells as in a spreadsheet. The properties of the grid are trivial and we haven't shown them. One of the properties is the name of the control. The designer has specified *PatientGrid* as the name.

5.3. The Horizontal time scale

Below the LifeLine Screen we show the property box for the Horizontal Scale. The designer has given it the name *TimeScale* to distinguish it from other horizontal scales he might set up.

Parent. A control is positioned relative to a parent cell. The time scale is positioned relative to PatientGrid, cell B1. The parent property also specifies that the time scale manages positions for controls in cells B2, B3 and B4. We have shown this specification in gray to indicate that the tool may set it automatically. This happens when the designer chooses the HorizScale tool and draws a rectangle that encloses cells B1 to B4.

Managing the positions means that the scale can translate domain-specific values such as the time of something into screen coordinates. There might be a horizontal scroll bar somewhere in the managed cells, and in this case the scale will synchronize scrolling of the relevant cells.

DataSource. The time scale shows data from a Calendar, which is a built-in dataset. The designer may add a *where* clause to specify the period of time to be handled.

Zoom1. The designer has specified that the first zoom area of the time scale is the leftmost 10 pixels. They should show all data older than a week ago (now-7 days). This is a simple example of a property formula. It specifies the default zoom when the Form opens. At run time the end-user may drag the zoom borders. There will be more zoom areas specified in the same way, but we haven't shown them.

5.4. The DataRow control

The designer has named the data row *MedRow* since it shows the rows for medicine.

Parent. The parent is PatientGrid, cell A4. The data row manages positions in cell A4 as well as B4. When the designer draws the DataRow as a narrow rectangle across the cells, the tool will set this property automatically.

DataSource. The data source is the MedType table. This means that the control will generate an instance of itself for each record in MedType. The result is that cells A4 to B4 are covered with a list of rows - one row for each type of medicine. The designer may add a *where* clause to restrict the list, for instance to all medicine types given to the patient.

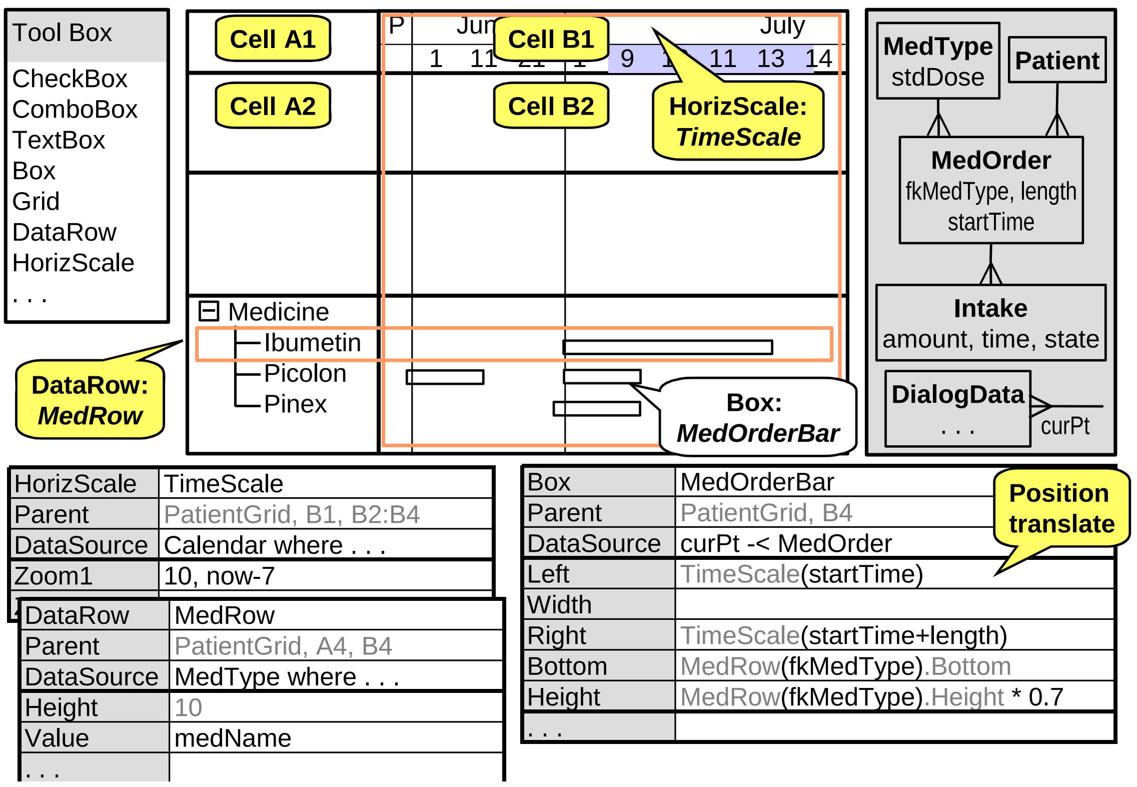
Each instance of the data row will have its own local data consisting of its own screen position (relative to the parent cell) and the data from its source record in MedType.

Height. The designer has specified that each row will be 10 pixels high. The tool will set this property automatically according to how the designer draws the row.

Value. The text to be shown in the left-hand side of the row. In this case it is *medName* from the source record in MedType. (There isn't space to show this database field in the data panel - sorry.)

Other properties. As in present user interface tools, there will be many other properties, for instance the background color of the row and whether there will be a border between the rows. All of these may depend on the local data in the data row instance.

Fig. 6. Patient Grid: Position translation



HorizScale	TimeScale
Parent	PatientGrid, B1, B2:B4
DataSource	Calendar where . . .
Zoom1	10, now-7

DataRow	MedRow
Parent	PatientGrid, A4, B4
DataSource	MedType where . . .
Height	10
Value	medName
. . .	

Box	MedOrderBar
Parent	PatientGrid, B4
DataSource	curPt -< MedOrder
Left	TimeScale(startTime)
Width	
Right	TimeScale(startTime+length)
Bottom	MedRow(fkMedType).Bottom
Height	MedRow(fkMedType).Height * 0.7
. . .	

5.5. MedOrderBar

The real interesting control is the box that makes up the medicine orders. The designer selected the Box tool and drew a rectangle in cell B4. The tool knows that the MedRow control is managing vertical positions in cell B4, and that the TimeScale control is managing horizontal positions in cell B4. It therefore suggests that these controls compute the position of the box. Here are the details:

Parent. The parent is PatientGrid, cell B4. The positions are relative to this cell.

DataSource. The data source is the MedOrders for the current patient. This means that the control must generate an instance of itself for each of these records in the database. Each MedOrderBar instance will have this record as part of its local data. It is specified with this formula:

```
DataSource: curPt -< MedOrder
```

It works this way: The dialog variable *curPt* refers to a record in the Patient table. From this record the system accesses MedOrder records according to the crow's foot to MedOrder. The -< operator symbolizes a walk along a crow's foot from one record to many. The result is the patient's medicine orders. This is a very compact notation. If the designer had to express it in database terms as an SQL statement, it would look like this:

```
select startTime, length, fkMedType
from Patient inner join MedOrder on Patient.ptID = MedOrder.ptID
where Patient.ptID = '1012087';
```

Left. This is the left screen position of the rectangle, measured in pixels relative to the surrounding cell. Since TimeScale manages horizontal positions in this cell, the tool has suggested this formula to compute the left position:

```
Left: TimeScale( )
```

TimeScale takes a point in time and translates it into a screen position, taking into consideration the various zoom areas. The designer has simply specified the point in time as the startTime of the medicine order:

```
Left: TimeScale(startTime )
```

Width. This is the width of the box in pixels. Nothing is specified, which means that the system decides the width, in this case as Right - Left. It is tempting to let TimeScale compute the width based on *length* of the medicine order, but due to the multiple zoom areas, TimeScale cannot do this without knowing the left position.

Right. This is the right-hand screen position in pixels. It corresponds to a point in time and is easy to compute:

```
Right: TimeScale(startTime + length )
```

Bottom. This is the bottom position of the box in pixels. Since MedRow manages vertical positions in this cell, the tool suggests this position:

```
Bottom: MedRow( ).Bottom
```

The designer filled in the domain-specific value that determines the row:

```
Bottom: MedRow(fkMedType).Bottom
```

The result is that MedRow looks up fkMedType to see which row instance the medicine type belongs to. The result of the formula is the bottom position of the row. Notice that several medicine orders may end up in the same row.

Height. This is the height of the box in pixels. While a scale cannot translate height or width, a data row can because it has a height of its own. The designer specifies the domain value that determines the row, and we get this formula:

Height: MedRow(fkMedType).Height

The result would be that the box occupied the full height of the row. However, in this case the designer wants a box that is lower in order that intakes larger than the standard dose can be shown. So he simply reduces the height, for instance to 70% of the full row height:

Height: MedRow(fkMedType).Height * 0.7

This example of combining domain-specific positions with pixel stuff is quite common in real interfaces. One example is text annotation to a graph. We want to position the text a certain number of pixels from the graph.

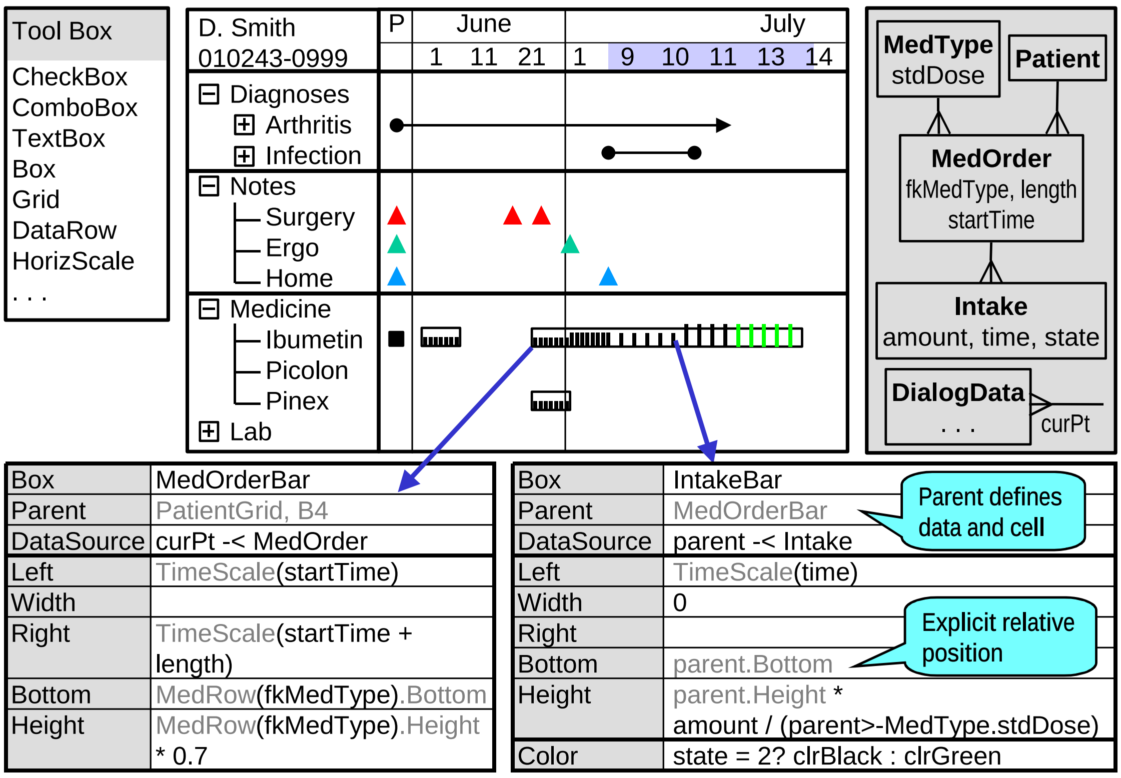
5.6. IntakeBar

Another interesting control is the small lines that show medicine intakes. They are narrow boxes (*IntakeBars*) that show data from the Intake table. Figure 7 shows the properties of the MedOrderBar next to the IntakeBar for easy comparison.

Parent. The parent of an IntakeBar is a MedOrderBar. The designer drew a tiny box inside one of the MedOrderBars and the tool suggested that MedOrderBar was the parent. The Parent property has two effects in this case:

1. Since the parent is positioned relatively to cell B4, the IntakeBar is also positioned relatively to cell B4.
2. Since each instance of the MedOrderBar has a MedOrder as its data source, the IntakeBar can address this MedOrder too. As an example, the formula *parent.startTime* used in an Intake property, would mean the start time of the medicine order.

Fig. 7. MedOrder and Intake as two levels



Box	MedOrderBar
Parent	PatientGrid, B4
DataSource	curPt -< MedOrder
Left	TimeScale(startTime)
Width	
Right	TimeScale(startTime + length)
Bottom	MedRow(fkMedType).Bottom
Height	MedRow(fkMedType).Height * 0.7

Box	IntakeBar
Parent	MedOrderBar
DataSource	parent -< Intake
Left	TimeScale(time)
Width	0
Right	
Bottom	parent.Bottom
Height	parent.Height * amount / (parent>-MedType.stdDose)
Color	state = 2? clrBlack : clrGreen

DataSource. The data source is *parent -< Intake.* This means that the IntakeBar must generate an instance of itself for each of the Intakes that the parent MedOrder refers to. The result should be all the IntakeBars that relate to the MedOrderBar.

Left. This is similar to the position of MedOrderBar. The tool suggested that TimeScale will translate the horizontal position, and the designer specified the domain value to be translated into pixels:

```
TimeScale(time)
```

This formula implies that if some Intake by mistake is recorded with a time that is outside the MedOrder range, it will be shown outside the MedOrderBar.

Width and Right. The designer specified the width as 0, to show the intake as a vertical line. Nothing is specified as the right-hand position, and the system automatically calculates it from Left and Width.

Bottom. The bottom of the line should be the bottom of the parent MedOrderBar:

```
Bottom: parent.Bottom
```

Height. The height should reflect the amount given to the patient in this intake. The tool may suggest this:

```
Height: parent.Height
```

This would generate an intake line as long as the standard adult dose. The designer should adjust it in this way:

```
Height: parent.Height * amount / (parent >- MedType.stdDose)
```

Notice the >- operator. It walks along a crow's foot in the database from one record to another. In this case it starts with the parent's MedOrder and walks to the related MedType record. The dot will then select the standard dose in this record.

Color. The color should depend on the state of the intake. The designer has specified that it should be black when the state is *done*, green otherwise:

```
Color: state = done? black : green
```

6. What is new?

Many things in the visualization tool are similar to other tools, but some are new. Figure 8 summarizes the important new things.

Multiple instances of controls

When the designer specifies a control, he can let it generate multiple copies of itself. There are three levels of abstraction involved in this:

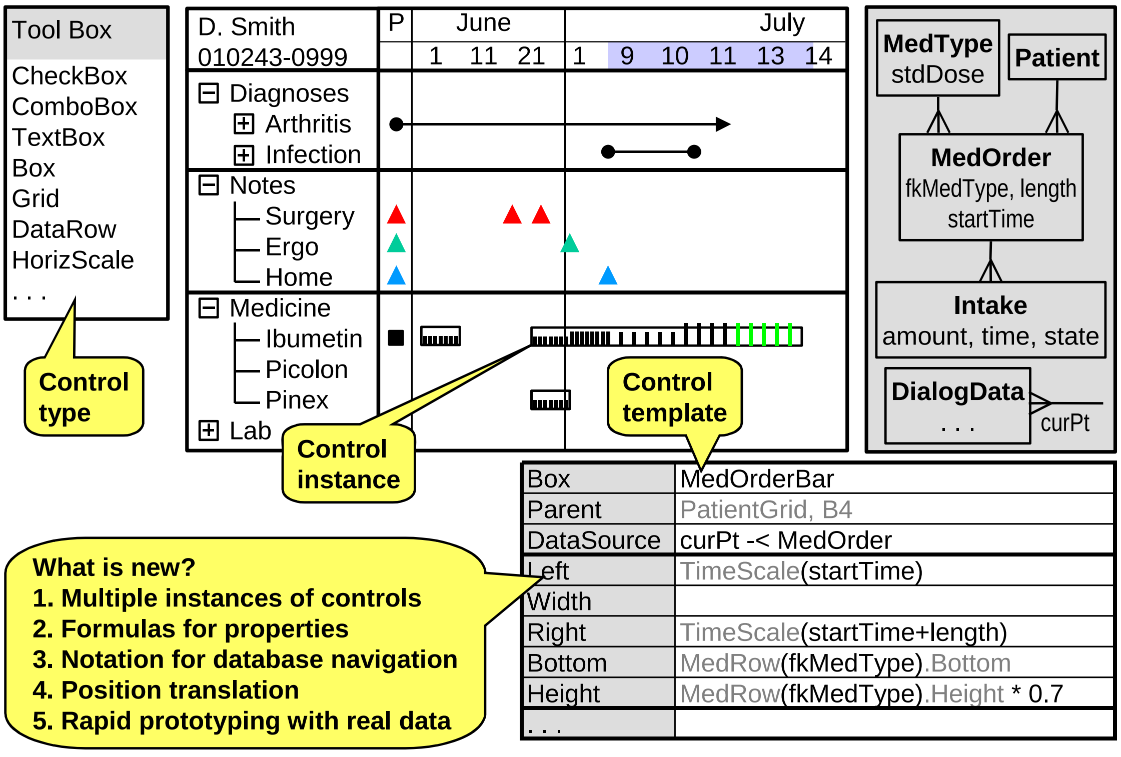
1. **Control types** found in the toolbox. The designer is not supposed to add new control types. They are made by programmers and installed as new program modules.
2. **Control templates**. The designer adds a control template to a Form by dragging a control type to the Form. The designer specifies the template properties in the property box.
3. **Control instances**. Each control template can generate one or more instances of itself. The end-user sees only the instances and is not aware of the templates. An instance has its own property values, for instance screen position (Left, Width, Bottom, etc.) and a reference to a database record.

Property formulas and screen updates

Most template properties can be formulas. The formula specifies how to compute the property value for a control instance. A formula can refer to data in the database, dialog-data valid for the current user session, data in the control itself, and data in other controls. The formulas are evaluated for each control instance and in this way the controls get their own property values. Formulas allow colors, positions, etc. to depend on data in the database.

The formulas also allow the system to keep track of what has to be updated on the screen when something changes. When a field changes in the database, the system can find the few control instances that depend on this field and update them. Similarly, if the end-user changes

Fig. 8. What is new?



Box	MedOrderBar
Parent	PatientGrid, B4
DataSource	curPt -< MedOrder
Left	TimeScale(startTime)
Width	
Right	TimeScale(startTime+length)
Bottom	MedRow(fkMedType).Bottom
Height	MedRow(fkMedType).Height * 0.7
. . .	

the zoom factor for an area of the screen, the system can find the controls that depend on it. This principle has been used for decades in spreadsheets, and the user is hardly aware of it. In user interface tools programming is needed to manage the updates. Simple systems just redraw the entire screen, but for complex screens this can give disturbing delays.

Notation for database navigation
The formulas can refer to database records and specify a walk from one database record to one or more related ones. We use two special symbols for this:

-< walks along a relation (a crow's foot) from one record to several.
>- walks along a relation from one record to another one.

In principle we could just use a dot for this as in Person.name, but experiments showed that separate symbols improved understandability.

Traditionally, database access is specified as SQL-statements. Walks along a relation are expressed as joins, but it is awkward to combine this with controls and properties. However, we will keep other parts of SQL as elements of the formulas, for instance *where* and *order by*.

Position translation
Data can be shown as screen positions, for instance as curves and data grids. In order to do this, the domain value stored in the database must be translated into pixel postions on the screen. We provide a few controls that can make this translation: A vertical and a horizontal scale; a data row generator and a data column generator.

The LifeLine example showed the use of a horizontal scale combined with a data row generator. Section 7 shows overview of rooms in a hotel reception. Here we combine a data row generator with a data column generator and put two text boxes with data-dependent colors in each cell. We have combined position translators in many other ways to make for instance curves with many scales in the same area; traditional data grids but with data-dependant colors; Gantt diagrams.

Prototyping with real data
One of the main goals of the project is to support iterative development of user interfaces, since this is crucial for high usability. Programming the user interface is very expensive today, so each iteration is made as a mockup without functionality. As long as we talk about simple applications, the designer can come up with realistic data to show in the mockup, but as complexity grows this becomes harder and harder.

The visualization tool will allow the designer to work with real data (or rather a representative copy of the data). He can immediately see the real-life results of a design - or a design change. The key factor here is the easy binding of controls to databases.

7. Hotel example

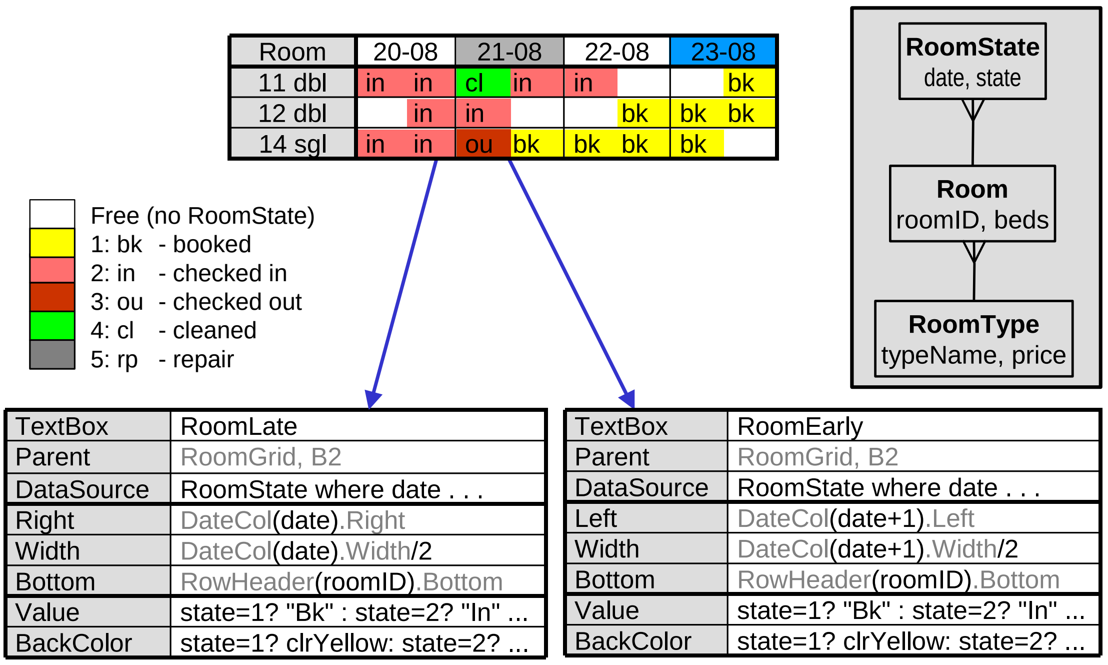
Figure 9 is an example from a different application area - a system for supporting a hotel reception. The central part of the figure is a room grid that gives an overview of the hotel rooms for a period of days. In the example, today is August 21 (shown with a gray background). August 23 is a Sunday (shown with a blue background).

The column for today shows the morning room states at the left and the evening room states at the right. For room 11 (a double room), the guest has checked out and the room has been cleaned (green color). A new guest has checked in already (red color). For room 14, the guest has checked out, but the room has not yet been cleaned. The room is booked for a new guest in the afternoon. This overview is very important for managing room cleaning and checking new guests in.

The data map shows that a room has an ID (the room number on the door) and a number of beds. The room is connected to a room type that holds the type name (e.g. *dbl* and *sgl*) and the price for this type of room. The room also has a collection of room states - one room state for each date where the room is used or booked. The possible room states correspond to the color codes shown in the figure.

In a hotel database, the room state refers to a single date, but in the real world it describes a period from the afternoon of this date until noon the next day. When designing the user interface, there is nothing we can do about this database structure. Some hotel systems show the room state only for the first date. As an example, a guest who checked in August 20 and checked out the next morning will have a checkout indication for August 20. This is very confusing to a novice receptionist who expects the checkout to appear for August 21. For this reason we want to show the room state as in the figure - using the right-hand side of August 20 and the left-hand side of August 21.

Fig. 9. Hotel example: Overview of room states



Room	20-08		21-08		22-08		23-08	
11 dbl	in	in	cl	in	in			bk
12 dbl		in	in			bk	bk	bk
14 sgl	in	in	ou	bk	bk	bk	bk	

⬜	Free (no RoomState)	
🟨	1: bk	- booked
🟥	2: in	- checked in
🟫	3: ou	- checked out
🟩	4: cl	- cleaned
⬛	5: rp	- repair

TextBox	RoomLate
Parent	RoomGrid, B2
DataSource	RoomState where date . . .
Right	DateCol(date).Right
Width	DateCol(date).Width/2
Bottom	RowHeader(roomID).Bottom
Value	state=1? "Bk" : state=2? "In" ...
BackColor	state=1? clrYellow: state=2? ...

TextBox	RoomEarly
Parent	RoomGrid, B2
DataSource	RoomState where date . . .
Left	DateCol(date+1).Left
Width	DateCol(date+1).Width/2
Bottom	RowHeader(roomID).Bottom
Value	state=1? "Bk" : state=2? "In" ...
BackColor	state=1? clrYellow: state=2? ...

Figure 9 shows how this is accomplished. The list of rooms at the left is handled by a data row control that repeats itself for each room. (We used the same control type for the list of medicines on the LifeLine.) A similar control type - a data column control, handles the list of dates at the top. The figure doesn't show the details of these.

The interesting controls are two ordinary text boxes - RoomLate and RoomEarly. Both of them have part of the RoomState table as their data source and thus repeat themselves for each room state. RoomLate positions itself in the date column corresponding to RoomState.date. It aligns to the right border and uses half of the width. RoomEarly positions itself in the next column and aligns to the left border. Their BackColor is a formula that finds the proper color. There is a slight difference between the two, because RoomEarly shows the special colors for *checked out* and *cleaned*, while RoomLate shows both of them as *checked in*. To support color-blind people the text boxes also show an abbreviated state name.

8. Challenges

There are many challenges in developing the tool. Below we describe some of them.

Tool for the data architect

We have only vague plans for the data-architect's tool. Most existing databases use table names and field names that are very cryptic to a designer. One of the data architect's jobs is to define more friendly names.

He must also define the crow's feet by means of the foreign keys in the database. Further he must provide a data map that shows it all, and provide ways for the formula translator to get these meta-data and access the database tables. Finally he may have to add code that checks security issues at run-time.

How to do all of this, depends on the software platform. Microsoft's .NET has many features that can do part of the work and make different kinds of databases look the same.

Optimizing database access

A simple implementation of the tool would translate each formula into SQL queries and run these queries whenever an instance of a control is to be shown. This would give a poor performance since the overhead to run a query is high. However, it seems possible to automatically analyze the property formulas and generate a few SQL queries for the entire screen. In the LifeLine example, one query would suffice to generate all the MedOrderBars and another to generate all the IntakeBars.

Under MS Windows, an alternative might be to address the databases through LINQ, which probably can do the kind of buffering we look for, so that we don't have to make the advanced analysis of the formulas. However, this might restrict the range of databases we can access.

Platform choice

Which platform should we use for the tool and on which platform should the finished user interface run? Although we want the designer to see real data during design, the final user interface may run on another platform. This is for instance the standard approach when developing software for mobile devices. And to what extent can we develop the tool so that large parts of it can be ported to another platform.

Some platform choices for the tool are: Eclipse (Java-based, open source), Microsoft Forms combined with .NET, Microsoft WPF combined with .NET

Some platforms for the finished user interface are: Web browsers, e.g. with Flash, Windows .NET, Windows Silverlight, Axapta and other ERP systems, Mac, Mobile devices of several kinds.

All of this interferes with each other and the kind of user interfaces we want to provide.

Integration with existing user interfaces

As we have described the tool above, it interfaces to existing databases and delivers a stand-alone user interface. In practice it will be necessary to integrate the tool into an application that already has a user interface, but needs additional screens with the kind of overview we can provide. We have only vague ideas about how this can be done.

Dialog functionality

The examples have only explained the data presentation. Data entry and other functionality must also be provided.

Present user interfaces provide functionality through program pieces called *event handlers*. Each control has a set of event handlers that are activated when the user clicks or types something. As an example, when a user clicks a button, the system activates the click-event handler of the button control. When the user passes the mouse over the button, the mouse-move-event handler is activated. An event handler can do anything. The click-event handler may for instance save the data the user has entered, or open a new screen, or send a message to someone.

We believe our tool needs the event concept too, although it somehow violates the principle of "spreadsheet level". We may reduce the problem in several ways, for instance by providing means to address data in the event handler in the same way as in a property formula. The addressing should deal with getting data as well as updating it. We should also provide simple means for common functions such as opening and closing windows, starting a database update, and committing it.

A great deal of functionality can be provided by letting an event handler set dialog data that causes recalculation of a property formula. As an example, the designer may want to show details about a medicine order when the end-user clicks the corresponding MedOrderBar. To do this, the designer adds a dialog variable called *selectedMedOrder*, which is a relation to the MedOrder in the same way as curPt is a relation to a patient. He also adds a large box to the LifeLine screen where medicine details can be shown (see Figure 1). He sets the data source of the large box to this:

Data Source: selectedMedOrder

He puts text boxes inside the large box and let them show details of the MedOrder.

Finally he defines a click-event handler for the MedOrderBar and lets it set *selectedMedOrder* to the proper MedOrder record. The result: When the end-user clicks a MedOrderBar, the system activates the event handler, which sets the dialog variable, which in turn causes the box with medicine details to update itself.

However, whatever we provide, at some point the designer will need more functionality. For this reason it is an advantage that the visualization tool uses the existing event handlers. When needed, the designer - or a programmer - can add arbitrary code to the event handlers.