Emerging User Interfaces through First-Class Viewers

Rick Mugridge, Mitra Nataraj, Darryl Singh
Department of Computer Science,
University of Auckland,
New Zealand
r.mugridge@auckland.ac.nz, {mnat010,dsin038}@ec.auckland.ac.nz

ABSTRACT
User interfaces for most applications are fixed, so that users have little individual control over how they can view and manipulate information. Our aim is to provide a general framework in which sophisticated users can tailor their user interfaces to suit their evolving needs.

This work is layered on the Naked Objects framework [11], which generates expressive user interfaces automatically from augmented JavaBean objects. We have added viewers as first-class objects, to allow an end-user or a developer to select from a range of general-purpose viewers and compose them to create tailored user interfaces. This empowers the user to organise their interactions in ways that most suit their current tasks, allowing for emergent user interface design.

Keywords
User interfaces, model-based UI, expressive systems

INTRODUCTION
Pawson has argued persuasively for the development of expressive systems, which empower users through exposing the underlying objects and their methods [10]. He argues that many user interfaces script the interactions with the users, forcing them to be "process followers" and preventing them from carrying out tasks which have not been explicitly allowed for in the user interface design.

We go a step further and argue that sophisticated users need to be able to control the form of their interaction with a system as well. While many users don't wish to change their user interfaces, others will do so if it is straightforward. There are several reasons for tailoring interaction:

- Shrink-wrapped software can become bloated as it is extended to suit a wide range of use. Users find that they have to traverse through multiple tabs and dialogs in order to carry out simple tasks. The "user model" becomes one of finding a way through the confusion. Tailoring the interface to avoid the unneeded detail would be helpful.
- Once a stable software system has been used for some time, changes in the tasks being carried out will evolve, making the interface less convenient to use.
- With the Naked Objects framework, it is possible to mix several applications together. For example, it would be convenient to allow references to email messages to be included in to-do lists. New user interface elements need to be composed from existing ones.

In traditional approaches to software development, requirements, user and task definition, software design, coding and testing are carried out at different times by specialists who attempt to communicate through various forms of documentation. Extreme Programming advocates a very different approach, with the iterative development of software, driven by the needs of users [1]. The various phases of traditional software development are carried out most of the time in Extreme Programming, with testing playing a much earlier role [2, 9]. However, the role of HCI specialists has been unclear in XP.

We argue that the appropriate time to design the user interface is once an essential system has evolved through one or more XP iterations. As some of the objects of the system begin to stabilise, the focus can shift to user interaction with those objects. As users on the XP team trial the current system through a simple interface, the need for more sophisticated presentations and interactions will become clearer and they can evolve.

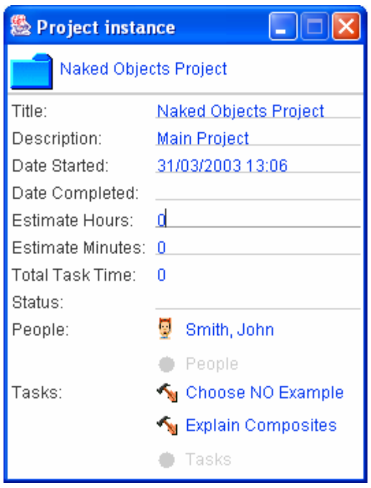
However, there is no reason to stop considering the interaction needs of users once the system is complete. Some users may wish to continue to tailor their interaction as their use of the system evolves, and to share their tailored interfaces with others.

The remainder of the paper is organised as follows. We introduce the Naked Objects approach in the next Section. Sections 3 and 4 introduces first-class viewers and show how they can be used. We follow with a general discussion of viewers and our future work. Related work is covered in Section 6. The final section concludes.

NAKED OBJECTS
The Naked Objects framework makes it easy to build an application without early concern for the user interface. There is a strong focus on the domain objects of interest to the users and the actions that they wish to carry out.

The framework defines how JavaBean classes in Java can be augmented so that a user interface can be created automatically from naked objects. Provision is made for a range of UI. A basic GUI is provided with the framework that allows the user to create, view, change and call methods on the (naked) objects of domain-level classes that are made visible to the user. A web-based interface to the framework is also under development.

We illustrate the use of the basic GUI for Naked Objects through *TimeKeeper*, an example application that we developed to help drive the evolution of our general-purpose viewers. For example, Fig. 1 shows a *Project* object viewed through the basic GUI. It shows several textual and numeric fields which can be edited and two collections of objects (*People* and *Tasks*).



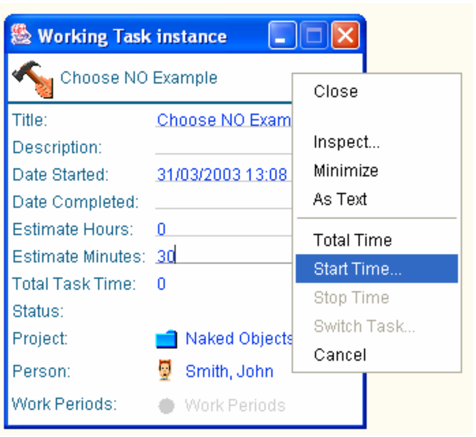
Title:	Naked Objects Project
Description:	Main Project
Date Started:	31/03/2003 13:06
Date Completed:	
Estimate Hours:	0
Estimate Minutes:	0
Total Task Time:	0
Status:	
People:	Smith, John
	People
Tasks:	Choose NO Example
	Explain Composites
	Tasks

Figure 1. A *Project* object shown by the Basic GUI

The first collection is a list of the *People* that are associated with the *Project*. Other *People* objects may be added to the collection by drag and dropping them onto the dot at the bottom of the collection. Double-clicking on an embedded object shows the details of that object in place. Dragging the icon of an embedded object on to the desktop creates a new window that displays a view for that naked object. For example, the first *Task* in the second collection from Fig. 1 is shown in Fig. 2.

Fig. 2 also shows that actions of a naked object may be carried out through a popup menu provided by the basic GUI. For example, the *Start Time...* menu item calls a method that creates a *WorkPeriod* instance which records the elapsed time until the user stops the timer.

The Naked Objects framework extends the conventions of JavaBeans with information that aids the auto-generation of a user interface. A domain class that is to be visible to the user implements the Java interface *NakedObject*; it may do this by extending the abstract class *AbstractNakedObject*. An instance of a naked object may be shown in a window in the basic GUI.



Title:	Choose NO Exam
Description:	
Date Started:	31/03/2003 13:08
Date Completed:	
Estimate Hours:	0
Estimate Minutes:	30
Total Task Time:	0
Status:	
Project:	Naked Objects
Person:	Smith, John
Work Periods:	Work Periods

Figure 2. Popup Menu in the Basic GUI

A Naked Object *field* is a JavaBean property that contains a reference to another *NakedObject*, to a collection of a specific subtype of *NakedObject* or to a *NakedValue*. Drag and drop may be used by the user to associate naked objects. There are several subtypes of *NakedValue* for *TextString*, *Date*, *Money*, etc. These manage display and input validation, and may be read-only.

A Naked Object *action* is a method in a class *N* whose identifier begins with 'action' and which take either zero or one arguments:

- An action with zero arguments is accessible to the user through the popup menu. It is called when the user selects the corresponding menu item.
- An action with a single *NakedObject* argument is accessible to the user through drag and drop. When an object of the argument type *A* is dropped on to an object of class *N*, the action method of the *N* object is called with the *A* object as argument.

An action method may be void or may return a *NakedObject* as value. In the latter case, the returned object is displayed by the basic GUI in a window.

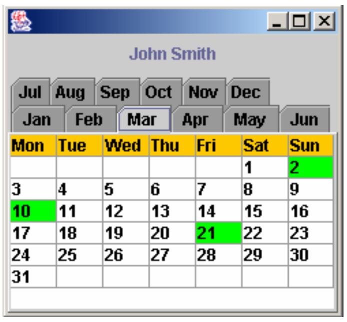
The framework manages bi-directional associations between Naked Objects. For example, when a *Task* is dropped onto the *Tasks* collection field of a *Project*, the *Project* field of the *WorkingTask* is updated to refer to that project.

Automatic persistence of naked objects is provided in various forms, including in relational databases, XML, EJBs and Prevayler, an in-memory object database [13].

While the basic GUI is likely to be fine for many tasks, it is often useful to have specialised ways of viewing data and acting on it. For example, in *TimeKeeper* tasks may be scheduled. A calendar provides a convenient way of viewing a subset of information about the tasks for a particular time period, far superior to looking at a list of tasks.

VIEWERS

We started by building a few specialised viewers for *TimeKeeper*. A calendar viewer is shown in Fig. 3.



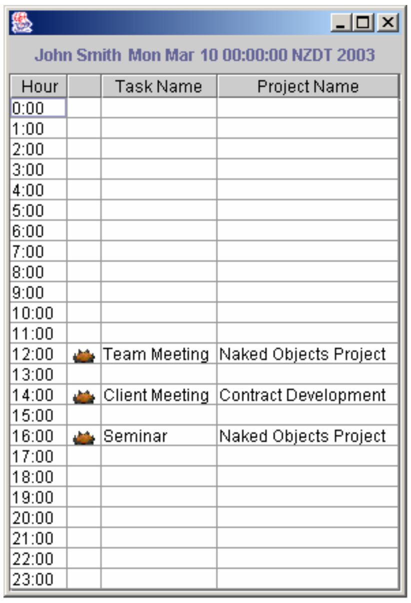
John Smith

Jul Aug Sep Oct Nov Dec
Jan Feb Mar Apr May Jun

Mon	Tue	Wed	Thu	Fri	Sat	Sun
					1	2
3	4	5	6	7	8	9
10	11	12	13	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28	29	30
31						

Figure 3. Calendar viewer

This viewer takes a *Person* object and highlights those days where one or more tasks are scheduled. On dragging any day on to the desk top, a tabular viewer shows the tasks scheduled in each hour, as shown in Fig 4. New tasks may be added by entering them into this table or through the basic Naked Objects GUI.



John Smith Mon Mar 10 00:00:00 NZDT 2003

Hour		Task Name	Project Name
0:00			
1:00			
2:00			
3:00			
4:00			
5:00			
6:00			
7:00			
8:00			
9:00			
10:00			
11:00			
12:00		Team Meeting	Naked Objects Project
13:00			
14:00		Client Meeting	Contract Development
15:00			
16:00		Seminar	Naked Objects Project
17:00			
18:00			
19:00			
20:00			
21:00			
22:00			
23:00			

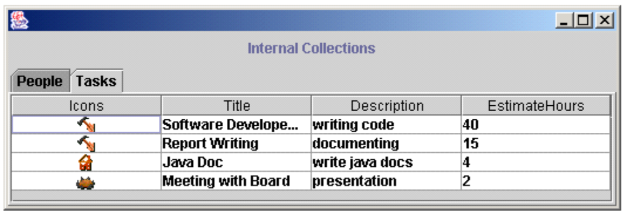
Figure 4. Daily Viewer

GENERAL-PURPOSE VIEWERS

After exploring various specialised viewers for *TimeKeeper*, we generalised some of them into viewers that could be applied to a wider range of naked objects.

Tables

A *Table* viewer was defined as a generalisation of the daily view in *TimeKeeper*. This shows the collections of a naked object in a tabular form. For example, a tabular viewer for a subset of the fields of a *Project* is shown in Fig. 5.



Internal Collections

People Tasks

Icons	Title	Description	EstimateHours
	Software Develope...	writing code	40
	Report Writing	documenting	15
	Java Doc	write java docs	4
	Meeting with Board	presentation	2

Figure 5. Table Viewer of a Project

When a naked object from the basic GUI is dropped onto a table viewer, the user is able to configure the viewer. They choose the information to be shown in the table by selecting the subset of fields of the elements of the collection that are to be shown in the columns.

Changes to the field data may be made through the resulting table viewer and are reflected in the basic GUI (through the underlying naked objects) and vice versa. An element of the table may be dragged out from the table and on to the desk top; it is then shown using the basic GUI.

All of the collections of a naked object are shown in the table window, with a tab for each. This is because the basic GUI does not permit collections to be dragged around independently.

Configuration of the table is carried out through a naked object. After developing other viewers, we found that it was more convenient for the user to be able to also configure the viewer directly, rather than having to go through the basic GUI of a separate configuration object.

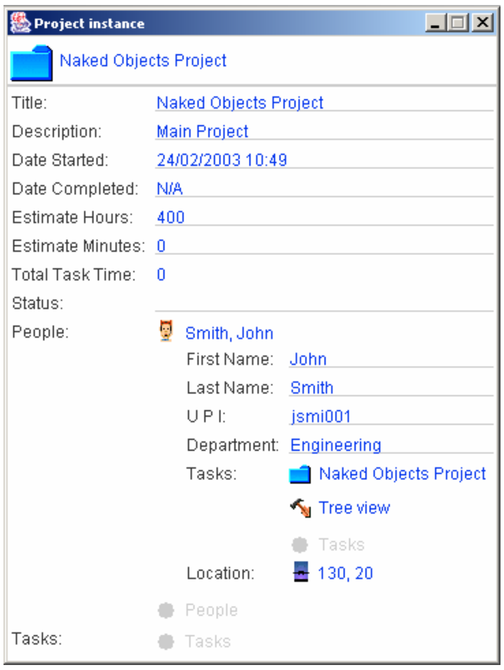
While this table viewer shows the value of our approach, it clearly needs to be developed further to make it more useful. Sorting the table on different columns would be convenient. One issue is how to show objects in a collection which are of various subclasses, with different extra fields. One approach would be to show the union of all fields; when a row was selected, just the fields of that row could be shown. Another approach is to provide another means of seeing the extra fields for any row.

Trees

As shown in Figure 6, an embedded object inside the basic GUI view may be expanded to show further detail. However, often it is useful to see just those associations between objects that are relevant to the user's task.

We introduced tree viewers to permit this. For example, a tree viewer for a *Project* is shown in Fig. 7, in which a subset of all the information is provided.

After exploring various approaches to configuration of a tree, we found that it was best to permit confiuration once the tree was presented to the user. It is awkward to choose beforehand what information is to be shown. This is especially because of references to objects that are of subclasses of the classes of interest. These can only be known dynamically, along with the further objects that they may reference.


Project instance

Naked Objects Project

Title:	Naked Objects Project
Description:	Main Project
Date Started:	24/02/2003 10:49
Date Completed:	N/A
Estimate Hours:	400
Estimate Minutes:	0
Total Task Time:	0
Status:	
People:	Smith, John

First Name:	John
Last Name:	Smith
U P I:	jsmi001
Department:	Engineering
Tasks:	Naked Objects Project
	Tree view
	Tasks
Location:	130, 20

People

Tasks: Tasks

Figure 6. Expanding in the Basic GUI

While a naked object is still used to represent the configuration information (so that it is also persistent), we allow the user to configure the tree through the use of a popup menu.

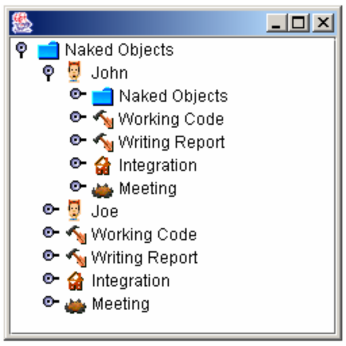

```
Naked Objects
    John
        Naked Objects
        Working Code
        Writing Report
        Integration
        Meeting
    Joe
    Working Code
    Writing Report
    Integration
    Meeting
```

Figure 7. Tree Viewer of Project

Charts

A rather specialised viewer for charts was created which makes use of *jFreeChart* [4], as illustrated in Fig. 8. This viewer is to be generalised so that the user can more easily select the data in a collection that is to be charted.

Maps

Pawson shows how a map can be used effectively in an expressive system [10]. An example of the use of our map viewer is shown in Fig. 9. Icons for *People* naked objects in a collection are located on a map (the background is specified as a GIF), according to a *Location* field with *x* and *y* values. The user may drag the icons to move them on the map and to change their underlying location. As with other viewers, an icon may be dragged from the map on to the desktop, where it is shown by the basic GUI in a window.

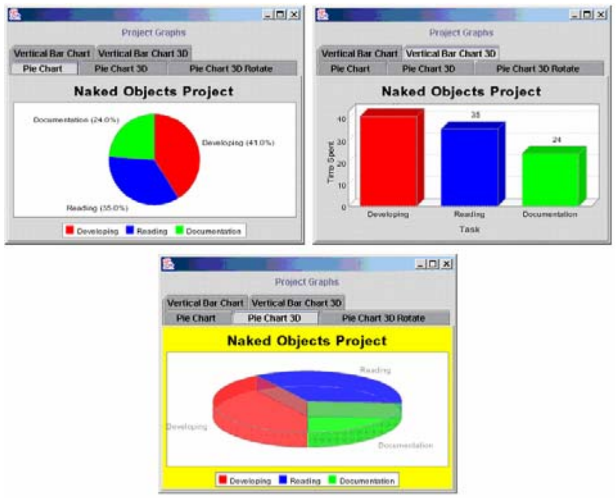


Figure 8. Chart Viewers

This is to be generalised in various ways. The location could also be defined through a method, so that arbitrary information can be mapped to a map location (this would prevent the user from changing the location on the map unless this could also be mapped back to the underlying data).

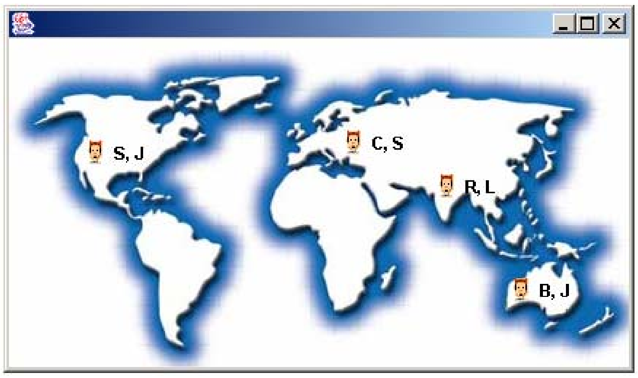


Figure 9. Map viewer

There are circumstances where the area of the naked object on a map may have meaning, such as when showing furniture of various sizes in a room. The viewer could allow the sizes of the furniture to be changed, as well as their location.

Containers

A user may be viewing multiple objects that are relevant to some decision that's to be made. If there is some delay before this can be completed, the context of the decision can be easily lost. We developed a simple Container viewer which locates naked objects in a window. For example, a container holding a few objects is shown in Figure 10.

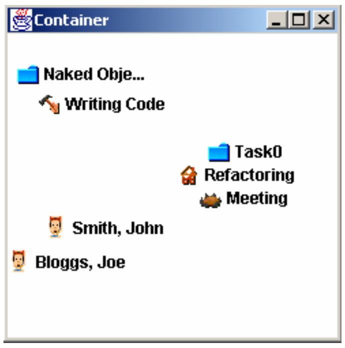
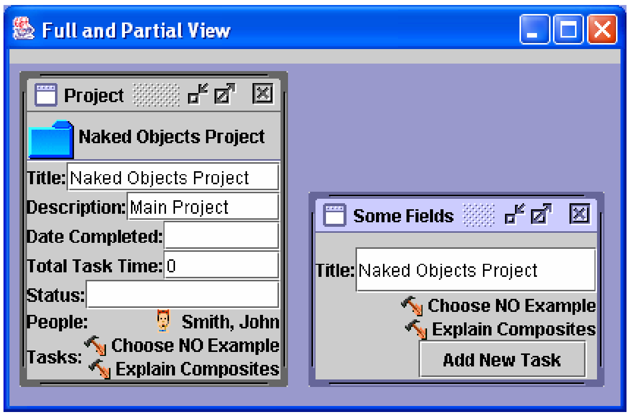


Figure 10. Container Viewer

Any naked object can be dropped into the container and dragged about within it. Unlike the map viewer, the location of an object is not a function of the object itself. Separate naked objects are used to represent the container, each of the objects within it and their location; these are then automatically persistent.

Composites

It can be useful to compose viewers from other ones. We have experimented with a range of simple viewers and with means of composing them. Fig. 11 shows a Desktop viewer which composes viewers by showing them as Java Swing *JInternalFrame*s.



Title: Naked Objects Project
Description: Main Project
Date Completed:
Total Task Time: 0
Status:
People: Smith, John
Tasks: Choose NO Example
Explain Composites

Title: Naked Objects Project
Choose NO Example
Explain Composites

Figure 11. Composite viewers

Each of the frames in turn are composed of viewers. The left frame is similar to a basic GUI window. It is constructed automatically by adding a header viewer and viewers for each of the fields of the objects to a Composite viewer. Collections are composed using header viewers with small icons. The header viewer displays the icon and title for the object and provides a way to invoke methods on the naked object through a popup. The user may use a popup on any field name to get a copy of the viewer displaying that field (with or without the field name).

The right frame in Fig. 11 provides a specialised view into the same *Project* as that shown on the right side. It is composed of a subset of the viewers of the left frame (the *Title* and the collection of the *Tasks* field, without the field name). In addition, it includes a button-based viewer for an *action* of the *Project* object.

We plan to rebuild some of the previous viewers to make use of such composites; for example, to allow viewers in Table cells.

DISCUSSION AND FUTURE WORK

Viewers are also naked objects; they are first-class. Hence we are able to apply viewers to viewers themselves. This can be useful with the configuration of more complex viewers. It also means that viewers are automatically persistent in the Naked Objects framework.

We are currently working on ways to reuse composed viewers in a general way, by being (visually) explicit about parameters. We have built some method viewers, including buttons, that provide alternatives to the popup menu for invoking methods on an object. We are now exploring ways of composing method viewers in a general way.

We aim to provide viewers that span the range from user interfaces to visual languages. A graph viewer is in progress, which will allow visual notations such as UML and circuits to be easily defined. We have defined Emperor, a meta-model for Naked Objects classes; this can be used to define classes by creating naked objects and associating them. It generates the appropriate Java code. It would be convenient to use this with the graph viewer to provide something similar to UML class diagrams.

Finally, we are reconsidering the use of space in user interfaces. The approaches of basic interfaces appear to arise as a compromise due to a lack of screen space. For example, a common approach is illustrated with an email client. The user clicks on a Folder to select it and to show the headers of the contained email messages inside a table. Selecting a message in the table shows the full message elsewhere. The menus are shared to avoid using space. They are modal, reflecting the state of the selected items and operating on them. The organisation of the menus can be confusing to the user, as there is usually not a direct mapping on to the objects of interest.

We are developing a zoomable interface using *Jazz* [3], which will allow for viewers to be scaled and embedded to arbitrary degrees. As the user zooms in on a viewer, finer detail will become apparent. This eliminates the need to conserve screen space, and allows the user to follow the metaphor of going closer to see more detail. There are several crucial issues in the development of this interface:

- How does the user move through the space without getting lost?
- How does the user compose and scale viewers in this space in a simple manner?

RELATED WORK

Related work has been carried out in the general area of model-based user interface development [6]. Our work differs from other work in this area by breaking some usual assumptions (eg, upfront design of interaction to handle tasks) and bringing several ideas and technologies together to provide a more sophisticated and general approach.

VIKI is a hypertext system which allows users to evolve the structure as required from semi-structured data [7]. As the information is better understood, it can be reorganised (refactored) by the users, with the user of multiple views. VIKI differs from our work in that text is the elementary data type, augmented by simple graphics. In our approach, we permit the full range of OO modelling capability of Java and permit methods to be represented visually.

Perlin and Meyer show that zoomable user interfaces can provide a way of managing the nesting of complex data [12]. User interface components may be scaled and nested. The work that we have underway differs in that we allow viewers to be composed dynamically.

Jazz provides a general-purpose zoomable interface [3]. It permits Swing components to be embedded in the 2D screen graph. Changes to the nodes in the scene graph are automatically reflected to the user through one or more cameras looking into the scene.

BuildByWire is a toolkit for defining visual language notations and their editors through direct manipulation [8,5]. As well as using Java layout managers, this makes use of constraints to define the geometric relationships in the notation. Once a visual notation has been developed, it is connected to an underlying object model, which manages the semantics of the visual notation. However, developing the view and the model separately is awkward in comparison to the approach we have used with naked objects, where the viewers grow out of the underlying model.

CONCLUSIONS

First-class viewers augment the power of the Naked Objects framework. They allow the user to select appropriate ways of viewing the data of interest at a particular moment.

We have developed several general-purpose viewers that may be used for a range of applications. Some need to be generalised further to make them more generally useful. Other viewers are needed to provide a more complete set. Specialised viewers for particular applications will still be required, these can easily be added.

We consider that there is a continuum from a graphical user interface through visual notations to visual languages. Our longer-term aim is to provide first-class viewers to cover this range.

REFERENCES

1. Beck, K. *eXtreme Programming Explained*, Addison Wesley, 2000.
2. Beck, K. *Test Driven Development: By Example*, Addison Wesley, 2002.
3. Bederson, B. B. , Meyer, J., Good, L. "Jazz: an extensible zoomable user interface graphics toolkit in Java", *Procs of the 13th annual ACM symposium on User interface Software and Technology*, p.171-180, November 6-8, 2000, San Diego.
4. http://www.object-refinery.com/jfreechart/
5. J.C. Grundy, W.B. Mugridge, and J.G. Hosking, "Visual Specification of Multi-View Visual Environments", *procs. VL'98*, IEEE CS Press, 1999.
6. Keränen, H. and Plomp, J. "Adaptive runtime layout of hierarchical UI components", *Procs of the Second Nordic Conference on Human-computer Interaction*, Aarhus, Denmark, ACM, 2002, pp251-254.
7. Marshall, C. C., Shipman III, F. M. and Coombs, J. H. "VIKI: spatial hypertext supporting emergent structure", *Proc. of ECHT'94*.
8. Mugridge, W.B., Hosking, J.G. and Grundy, J.C. "Vixels, CreateThroughs, DragThroughs and Attachment Regions in BuildByWire", *OzCHI'98*, pp320-327, 1998.
9. Mugridge, R., "Test Driven Development and the Scientific Method", procs. *Agile Development Conference*, Salt Lake City, June 2003.
10. Pawson, R. and V. Wade. "Agile Development with Naked Objects", *4th Int. Conf. on Extreme Programming and Agile Methodologies in Software Engineering* (XP2003), 2003. Lecture Notes in Computer Science, Springer-Verlag.
11. Pawson, R. and Matthews, R. *Naked Objects*, John Wiley and Sons, 2002.
12. Perlin, K. and Meyer, J. "Nested User Interface Components", *Proceedings of the 12th Annual ACM Symposium on User Interface Software and Technology*, ACM, 1999, pp11-18.
13. K. Wuestefeld, "Prevayler", http://www.prevayler.org.