

---

JADE Development Centre

---

# User Interface Considerations

---

In this paper . . .

## USER INTERFACE CONSIDERATIONS

INTRODUCTION	1
LARGE FORMS	2
DYNAMIC FORMS	4
KNOW YOUR EVENTS	6
KNOW YOUR DATA	7
CONCURRENCY ISSUES	10

---

## Introduction

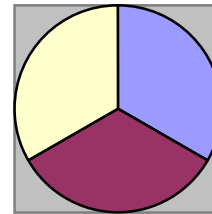
---

When developing a user interface, there are two areas that must be considered - appearance and behaviour.

There is a lot of general literature available that covers designing user interfaces, and a large part of this is to do with appearance.

This paper discusses some of the issues associated with building appropriate behaviour into your user interface with particular reference to JADE applications.

In most cases when addressing these issues, the trade-off between functionality provided, speed of development and runtime performance will influence decision-making.



■ Performance	■ Development Speed
■ Functionality	

These three factors are so closely linked that it is unlikely that a gain in one area would ever be made without some impact on another factor. This paper considers some common problems seen in user interface behaviour and considers solutions in the light of these three factors.



# J A D E™

---

Jade Software Corporation Limited believes that the information furnished herein is accurate and reliable and has been prepared with care. However, no responsibility, financial or otherwise, can be accepted for any consequences arising out of the use of this material, including loss of profit, indirect, special or consequential losses or damages.

# Large Forms

A common cause of poor performance within applications is forms that simply have too many controls on them. This is especially a problem when using folders with many sheets.

Firstly, with a complex form containing lots of data and many options, you run the risk of overwhelming the user.

Secondly, the overhead of loading a form with hundreds of controls is much greater than that for a smaller form. This overhead increases, again significantly, if the form is deployed as part of a Web application.

If you find large forms are performing poorly then there are a number of options to consider. The most obvious of these is to split the data across several forms. The screen shots below are from two similar applications and illustrate the different approaches:

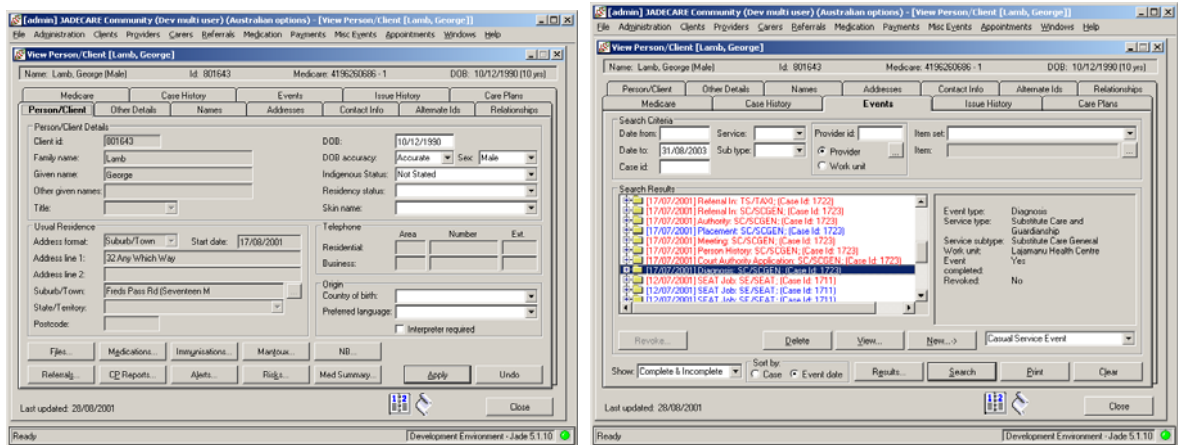
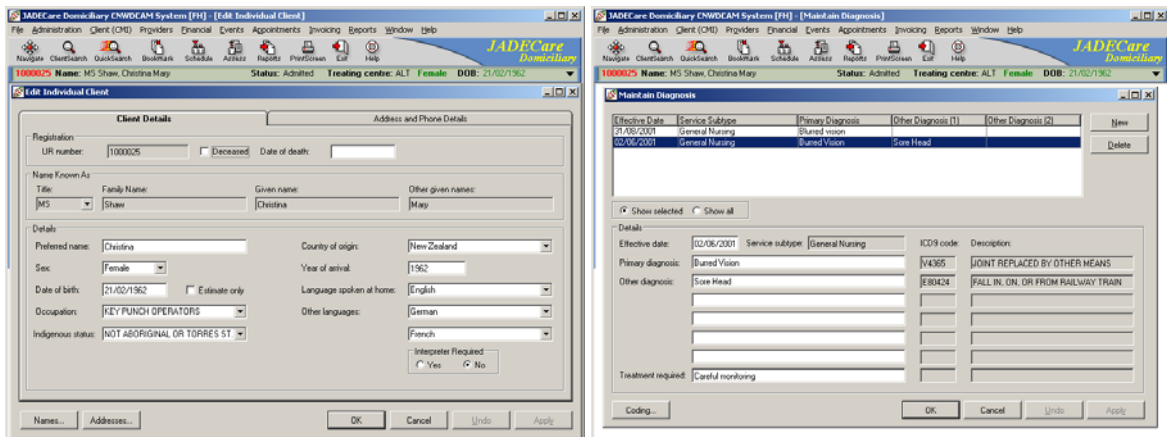


Figure 1 - Data displayed on single form

In the above example, all data is displayed on a single form using a tabbed folder.





The above form uses tables to display header information as well as details of an item selected from the list. Therefore instead of 20 textbox or label controls, only two tables are required.

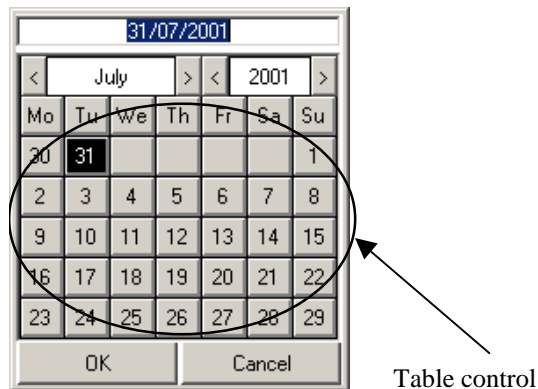


Figure 4 - Sample calendar control

In the above calendar control, instead of using buttons for the days, the developer used a table with the **fixedRows** property set to the number of rows and the fixed columns property set to the number of columns. Therefore instead of 42 buttons, the form uses a single table.

## Dynamic Forms

---

Most forms are designed using the JADE Painter but JADE also provides facilities to create controls dynamically at run time.

This is useful when:

- The number and type of controls cannot be predicted at design time; for example a highly graphical application such as a container terminal tracking system.
- There are a large number of controls that are all expected to respond to the same set of events; for example a multi-line data entry form.
- A user-defined control consisting of a container and a number of child controls must be built at run time.

There are two ways of adding controls to a form at run time:

1. The **loadControl** method clones a persistent control associated with the form. The cloned controls respond to the same events defined for the original control.

#	URNO	Name/episode	Visit time	Visit cat	Visit code	Charge code	Travel duration	Visit duration	Activities
1									
2									
3									
4									
5									

Figure 5 - Cloned controls for data input

In the above example, all controls on the first line are cloned four times to allow a user to enter quickly five transactions at a time. This approach can reduce the amount of code that must be written, as all controls share the same event methods. Of course, since the controls are created at runtime the entire form image cannot be cached on the presentation client so this form would take longer to load initially, than if all controls had been designed using the painter.

2. The **addControl** method can be used to create a new control for a form at run time. It differs from the **loadControl** method, as the new control does not need to be based on an existing control on the form. The control responds to event methods defined on the parent form based on the control name; for example **newControl\_click**. The **setEventMapping** method may be used to define another method to be invoked for an event.

In the calendar control shown above, all child controls, table, buttons, and labels are created at run time in this way.

---

**Note:** Dynamically creating controls allows you to write more generic code but there is a measurable performance overhead associated with creating the controls in the first place. Furthermore, the use of the **setEventMapping** method is quite expensive and should be used sparingly.

---

## Know Your Events

---

The performance characteristics of an application vary greatly, depending on whether the application is run in single user mode, as a thin client, or using the standard fat client. There is a danger that if development takes place in one mode and the system is deployed in another, performance issues may arise.

One significant point of difference between running in single user or fat client mode, as opposed to thin client mode, is the overhead associated with the handling of events. If an event method has been implemented for a particular event, a request must be sent from the presentation client to the application server to process the event method.

The events that are likely to have most impact when running in thin client mode are those that have the potential to happen most frequently. These are:

- Events that occur in response to keystrokes; e.g. **keyDown**, **keyUp**, **keyPress**, **change**.
- Events that occur as a result of moving the mouse, e.g. **mouseMove**, **dragOver**.
- **Paint** events.

The first question to ask is whether the application really needs to respond to particular events. While most users will have experienced highly responsive Windows applications that they run, when asked whether they are prepared to sacrifice performance in a multi-user business application, the answer is usually no. Further, if there is a requirement to handle specific events, consider whether this can be confined to a limited area of the application.

Given there is a need to handle frequently occurring events, consider using some of the features JADE provides to reduce the frequency of these events.

In most cases, when handling keystroke events, you are only interested in trapping a subset of keys but by default all keystrokes cause a request to be sent from the presentation client to the application server. The **Form::registerFormKeys** and **Control::registerKeys** methods allow you to define a subset of keys for which keystroke events are handled for a form and a control, thus reducing the number of requests sent between presentation client and application server. If there is a requirement to handle navigation within a table using the keyboard, consider using the **Table::tabKey** property instead of handling keystroke events on the table.

By default in JADE thin client mode, **mouseMove** and **dragOver** events are discarded when moving the mouse within the same window if the time since the execution of the last move event is less than the mouse move time defined for the current application, unless the mouse comes to rest. The default value for mouse move time is 200 milliseconds. The **Application::setMouseMoveTime** method allows you to change this default if it is appropriate for your application. The **MouseMoveTime** setting in the JADE initialization file can also be used for this purpose.

Many text-based applications contain code to specifically detect whether a user has changed anything on a form in order to prevent unnecessary processing. For

example, customer details are displayed and when the **OK** button is clicked, if the user has made changes, details are updated. The **TextBox::firstChange** event occurs the first time text is changed whereas the **TextBox::change** event that occurs every time the text is changed.

Generally speaking, when a window needs to be repainted on a presentation client, this requires no interaction with the application server, unless:

- The developer has implemented a paint event for the form or control
- The control is a subclass of a JADE control and the developer has implemented a **paint** method for the control.

In the second case where the **paint** method implements code to draw the control, it may be preferable to use the **Window::autoRedraw** property instead. When this property is set, no paint events are processed and instead, when the form or control is repainted, draw commands are replayed on the presentation client.

Finally, the **Window::enableEvent** method can be used to enable or disable handling of specific events. For example, you may choose to disable all keystroke events for any client running in thin client mode.

## Know Your Data

---

A common problem in designing a user interface is the inappropriate use of controls. Often what works well during development with a small amount of data does not scale up when the system is deployed in production with many more users.

It is important that when the decision is made to use a particular type of control that consideration is given to the volume and volatility of the data associated with the control. Volume-related problems are most common when using list boxes, combo boxes, and tables. If the number of entries that could be displayed is likely to be large, you should consider a strategy for reducing the amount of data that will be loaded.

The **listCollection** method can be useful when loading entries from a collection into a **ComboBox** or **ListBox**. The advantage of this method is that the collection will be loaded incrementally; that is entries are loaded as the user scrolls through the list as opposed to being loaded in one hit. This feature works well as long as the number of entries in the collection is in the region that a user would feel comfortable scrolling through. It is also most useful when the order of the collection is the order in which the entries are to be displayed, as setting the **sorted** property for the list box or combo box causes all the entries to be loaded.

Where it is not practical to load all collection entries into a list box or combo box, consider implementing some kind of browse facility.

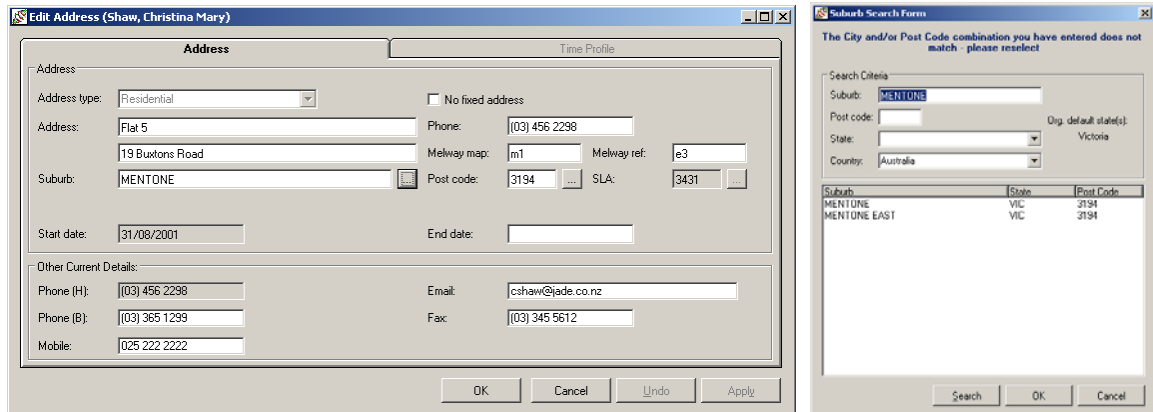


Figure 6 - Browse example

In the example above, instead of loading all of the valid suburbs into a combo box, the developer has implemented a browse button to display a suburb search form.

Both list boxes and combo boxes provide a facility to organise data display into a tree or hierarchy structure. Where the volume of data to be loaded is low, it may be appropriate to load a tree with all branches expanded. However, loading only the first level of the hierarchy can result in significant performance improvements where the volume of data is large. Loading of lower-level branches can be deferred until a particular node is expanded. In this case if a node is never expanded, the data need never be loaded.

The **displayCollection** method can be useful when loading data from a collection into a table, list box or combo box. The advantage of this method is that only the number of entries required to fill the control are loaded and as the user scrolls down the table entries that are no longer visible are discarded. JADE handles the loading and discarding of entries automatically.

Another approach to loading a table is to load the table entries incrementally. This works well for searches where a user may enter some key values and the entries they are looking for are likely to be at the beginning of the list.

The screenshot shows a 'Person/Client Search' window with the following sections:

- Search Criteria:** Fields for Family name (smith), Given name, Client id, Indigenous Status, Usual clinic, Medicare, Alternate id type, Alternate id no, Sex, and Skin name.
- Search Name:** Radio buttons for Standard (selected), Phonetic, and Indigenous phonetic.
- Search In:** Checkboxes for Clients only, Include deceased, and Any Name.
- Birth / Age details:** Radio buttons for DOB (selected) and Age, with options for Exact and Approximate, and a Date field.
- Search Results:** A table with columns: Client id, AKA, Family name, Given name, DOB, Sex.
 

Client id	AKA	Family name	Given name	DOB	Sex
397129		Smith	Dave	20/02/1960	Male
800599		Smith	Fred	13/02/1955	Male
800189		Smith	Fredy	21/02/1998	Male
800883		Smith	George	11/07/1955	Male
374146		Smith	Guy	24/02/1965	Male
801022		Smith	Jack	01/01/1992	Male
800644		Smith	Jill	27/07/1927	Female
372085		Smith	John	24/05/1969	Male
800617		Smith	John	01/01/1948	Male
- Details:** Information for the selected client (Client id 800617):
  - Currently known as: Smith, Adele Nelley
  - Usual Residence address: 19 Buxtons Rtd, Some Where North, East Of Nowhe, Barrow Creek, NT, Postcode: 0870
- Actions:** Buttons for Files..., Alerts..., New..., View..., Delete, Med Summary..., Waitlist..., Referrals..., CP Reports..., Cases..., Events..., New Event...->, Search, Print, Clear, Close.

Figure 7 - Incremental load example

In the above example, search results are loaded in batches. Clicking **More results...** or the search button displays the next page of search results.

In both of the above approaches, it is not practical to provide sorting from within the table as the table generally does not contain all of the search results at a specific time.

Where a large amount of data is being loaded into a form, there is often an opportunity to speed up the process by treating the load as a read-only transaction using the **beginLoad/endLoad** instructions. Loading data from collections can result in a large number of locks and unlocks of collections that can be reduced by using **beginLoad/endLoad**. Generally, holding the locks for a longer period of time is not a problem unless the collections being read are volatile, in which case there could be increased queuing of updating transactions waiting for read-only transactions to complete. In that case, it may be appropriate instead to create a transient collection containing data to be browsed and load the table from the transient collection.

For forms with many controls, the use of **folder** controls allows you to organise information and provide an intuitive means of navigating the information. Because all data is not visible at one time, there is an opportunity to rationalise loading of data in folders so that a sheet is populated only when the user views the sheet for the first time.

In the case where the user never views a sheet, the load is never done. For example:

```
folder_sheetChg(folder :Folder) updating;
begin
  if not self.detailsSheetLoaded then
    self.loadDetailsSheet;
    self.detailsSheetLoaded := true;
  elseif not self.historySheetLoaded then
    self.loadHistorySheet;
    self.historySheetLoaded := true;
  endif;
end;
```

## Concurrency Issues

---

An important part of the design of a user interface involves putting in place a strategy to handle the situation when a user updates data, being viewed or maintained by another user. The approach taken will depend to a large extent whether users tend to operate on their own working set of data or a number of users maintain a common set of data.

Generally as a bare minimum, the application should at least detect the situation where two users are simultaneously updating the same data. Generally this is done by saving the edition of the object(s) being updated at the time data is presented to the user on a form. When the method to update the data is called the saved edition is compared with the latest edition and if these are not the same updating does not proceed. Generally the user interface is responsible for saving the original edition with the model responsible for checking the edition has not changed. In most cases this checking is adequate where the chances of conflict are relatively low.

In some cases this concept is extended so that instead of waiting until updating takes place, the user interface registers for notification of updates or deletes of any object being displayed. In the event notification of an update is received, the application could:

- Display a message to the user indicating that the data they are viewing has changed and prompt them as to whether wish to view the changed data
- Provide some visual cue that something has changed, for example an icon becomes visible. The user may then choose whether they wish to refresh the display to see the changes.

In the event that a delete notification is received the user would be notified by way of a message and the form would be closed.

Occasionally there are situations where it does not make sense to allow more than one user to even start editing the same object and a strategy for preventing this must be put in place. The JADE editor and JADE painter are examples of this kind of approach. In the case of the editor, as soon as a user starts keying in changes to a method, the method object is locked so that other users are prevented from making changes. In the case of the painter, the form object is locked at the time it is loaded in painter.

A refresh strategy should also be considered in the context of search or list forms that display data in read-only form. In most cases it is sufficient to ignore changes made to data by other users after it has been displayed and this is by far the most efficient solution performance wise.

At the next level it may be desirable to ensure changes made by the current user are reflected in data displayed.

In some cases it may be necessary to ensure that changes made by any user of the system should be reflected in read only data displayed. This may be done using a visual cue, e.g. an icon, and a refresh button or it could be done by automatically refreshing data as it changes. The JADE **Table, ListBox, ComboBox displayCollection** method and the **listCollection** method for the **ComboBox, and ListBox** controls allow data from persistent collections to be refreshed automatically.

When refreshing data manually a common approach taken in response to a notification is to re-do the search that resulted in the data being displayed in the first place. While this is very simple from a development point of view it is preferable to make some attempt to only refresh those items that have actually changed unless you know for certain performance is not going to be an issue.