Limited View

Alias:

Blinders
Child Proofing
Invisible Road Blocks
Hiding the cookie jars
Early Authorization

Motivation:

When a congresswoman visits a base in her district, she is given an escorted tour. Before arrival a tour would be set up. Some areas of the base, such as the target range would be designed as unsafe and others might be designated top secrete or need-to-know. The tour would be pre-designed to give her a limited view of the base relevant to her security clearance.

Graphical applications often provide many ways to view data. Users can dynamically choose which view on which data they want. When an application has these multiple views, the developer must always be concerned with which operations are legal given the current state of the application and the privileges of the user. The conditional code for determining whether an operation is legal can be very complicated and difficult to test. By limiting the view to what the user has access to, conditional code can be avoided.

Problem:

Users should not be allowed to perform illegal operations.

Forces:

- Users may be confused when some options are either not present or disabled.
- If options pop in and out depending upon *Roles*, the user may get confused on what is available.
- Users should not be able to see operations they cannot do.
- Users should not view data they do not have permissions for.
- Users do not like being told what they cannot do.
- Users get annoyed with security and access violation messages.
- User validation can be easier when you limit the user to see only what they can access.

Solution:

Only let the users see what they have access to. Only give them selections and menus to options that their current access-privileges permit. When the application starts up, the user will have some *Role* or the application will default to some view. Based upon this *Role*, the system will only allow the user to select items that the current *Role* allows. If the user can not edit the data, then do not present the user with these options through menus or buttons.

A *Limited View* is controlled in several ways. First, a *Limited View* configures which selection choices are possible for the user based upon the current set of roles. This makes it so that the user only selects data they are allowed to see. Second, a *Limited View* takes the current *Session* with the user's *Roles*, applies the current state of the application, and dynamically builds a GUI that limits the view based upon these attributes. *Null Objects* [Woolf 97] can be used in the GUI for values the user does not have permission to view.

The first approach allows users to see different lists and data values depending upon their current *Role*. This approach is primarily used when a user is presented with a selection list for choosing items to view. The GUI presented to the user is static, however the values listed on the GUI changes according to the current *Role* of the user. An individual user may have many *Roles* and may have to choose a *Role* while running the application. Whenever a user changes their *Role(s)*, the *Limited View* will change.

For example, consider a financial application in which a manager has access to a limited set of products. When making a product selection inside a *Limited View*, the application will only present products that the manager is allowed to see. Thus, when the manager goes to select the desired products available, he or she will not get an "access denied" error.

When using the *Limited View* inside a *Session* with *Role* information, the view based upon the current state of the application is also limited. The actual GUI that the user sees on the screen is dynamically created. For example, a *Limited View* might add buttons or menus for editing, if the user's *Role* allows for editing. Alternatively, edit options might always be disabled, but they could be dynamically enabled depending upon the *Role* of the user and the current state of the application. The *Strategy* pattern could be used here to plug in different GUIs depending upon the desired results.

By limiting users to only viewing the data to which they have access and to only showing them the options that are available, they know what options are currently legal. For example, in Microsoft Word, when there are no documents open, the file menu does not show the option of saving a file since there are no files to save. Similarly, when previewing an Internet document, the user does not have any options available to edit the document.

Limited View can be implemented many ways. GUIs can be dynamically created through *Composites* and *Builders* [GHJV 95]. The *Type-Object* [Johnson & Woolf 97], *Properties*, and *Metadata* [Foote & Yoder 98] are also very useful in creating these dynamic GUIs. Alternatively, the *State* pattern can be used to represent different views with different classes. A *Strategy* can be used for choosing the appropriate *State*.

A *Limited View* maximizes security and usability. Unfortunately, it can be difficult to implement. You really want to consider using a *Limited View* when many privileges vary between users and you don't want to frustrate your users with many error messages.

Example:

One example of a *Limited View* can be seen in the Selection Box example in Figure 5. Here, the user is only provided with a list of the products they can view or edit. While other products are available in the system, those products are not shown because this user does not have access rights to them. For example, if a GUI provides a list of detailed transactions, a *Limited View* on this would only show a list of transactions for beans, corn, and hay for this user since that is all he or she is allowed to access.

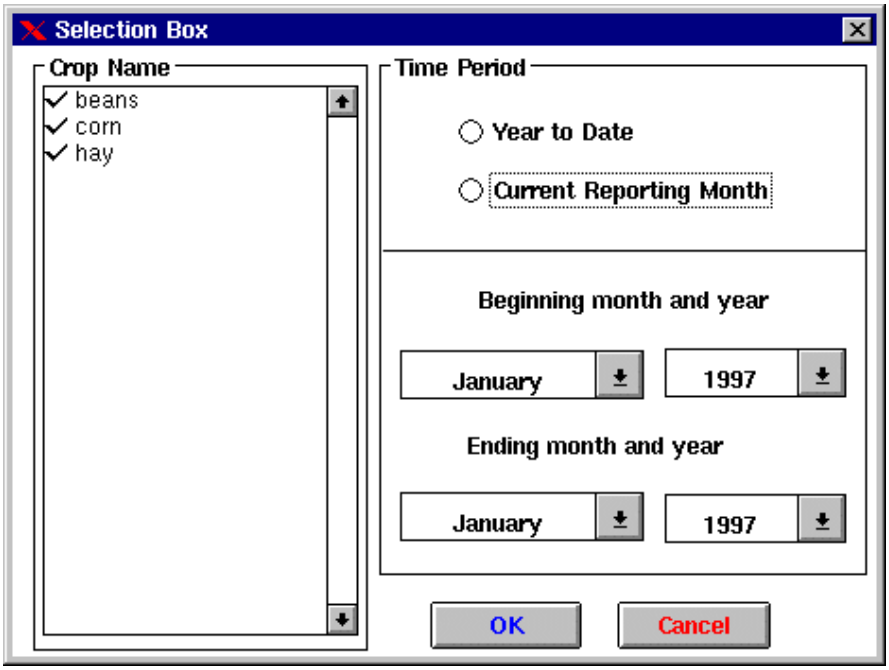

Figure 5 - Limited View on Selection

Another example can be seen in Figure 6 and Figure 7. Note that Figure 6 does not show a button for changing the values in the database, while Figure 7 has that option. The view limits the ability for editing in Figure 6 since the user does not have the authority for editing the detailed income. In Figure 7, the view is limited by disabling some of the buttons. The buttons are disabled because the user has not changed any transactions and there are no values to accept or commit to the database.

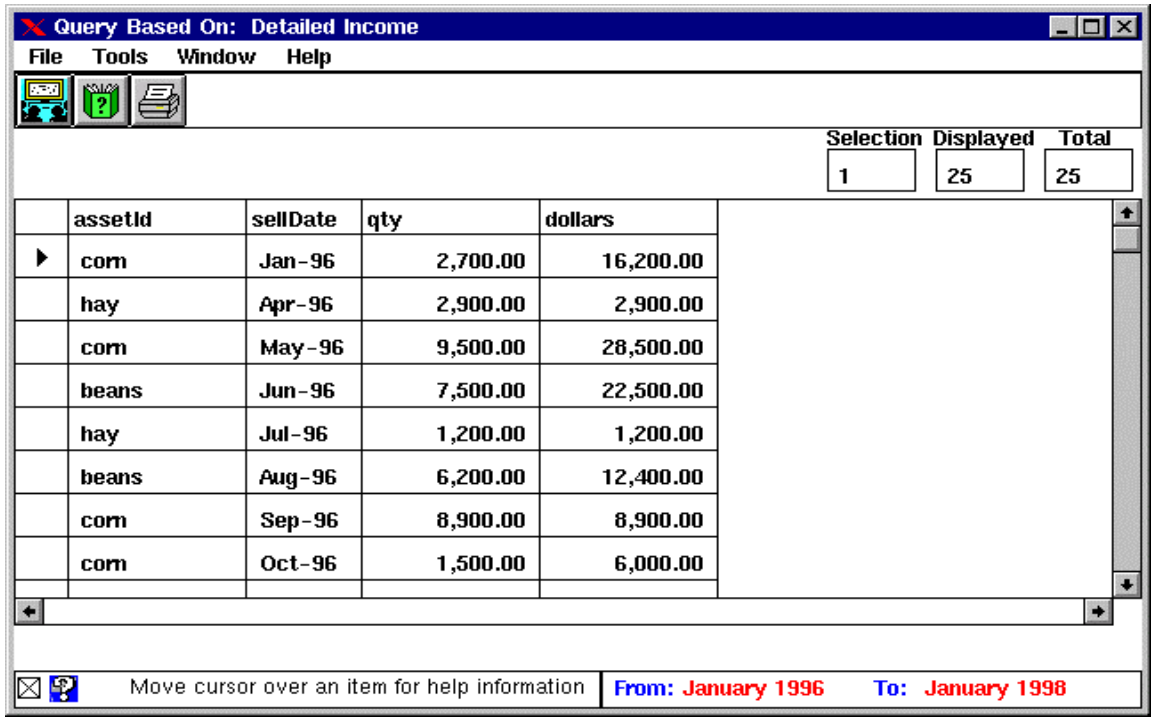

Figure 6 - Limited View on non-editable transactions

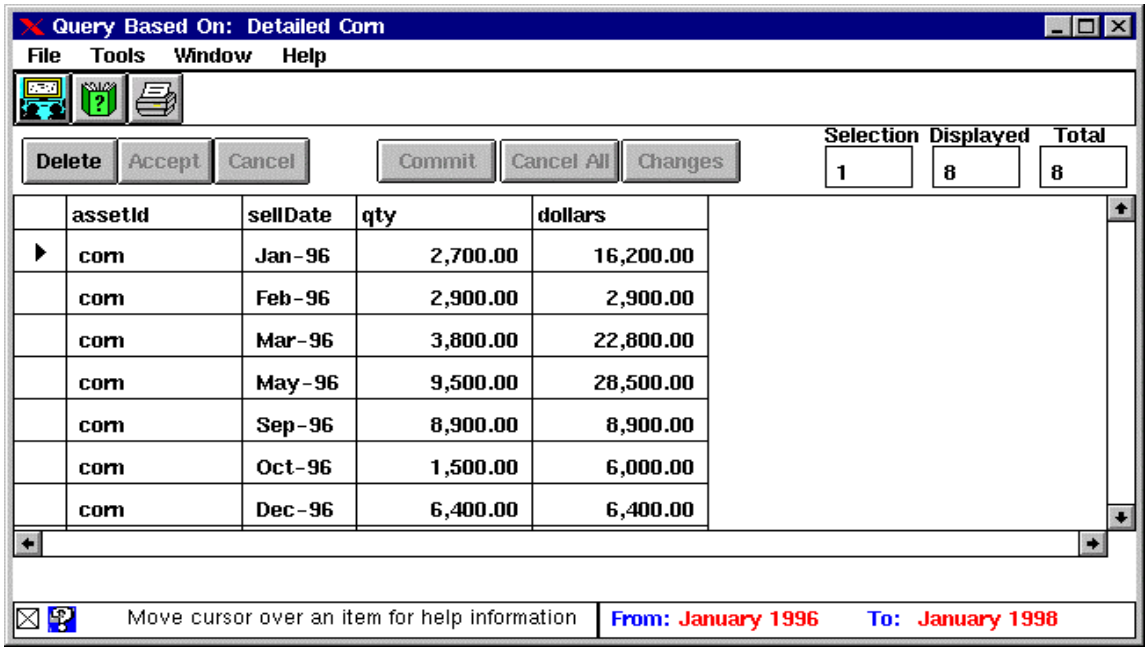

Figure 7 - Limited View on editable transactions

Both the disable and the hide approaches to limiting a view have tradeoffs that designers make depending upon the overall needs of the application. If the primary user will not be able to edit values, the *Limited View* will probably want to hide editing buttons. Whereas if the primary user will have editing functions, the *Limited View* will probably want to simply disable the buttons and menus as needed.

Consequences:

- ✔ By only allowing the user to see and edit what he or she can access, the developer doesn't have to worry about verifying each operation after a user selects it. The user will only be permitted to select legal items. Similarly, if the user can edit values, the editing menus and buttons will only be presented when the user has editing capabilities on the presented data.
- ✔ Security checks can be simplified by performing all of them up front.
- ✔ Users will not get frustrated with error dialogs popping up all the time telling them what they can not do. Users will also not get frustrated by constantly seeing options they do not have access to.
- ✘ Users can become frustrated when options appear and disappear on the screen. For example, if when viewing one set of data, the editing button is there and when viewing another set of data, it disappears, the user may wonder if something is wrong with the application or why the data isn't available.
- ✘ Training materials for an application must be customized for each set of users because menu operations will disappear and reappear and GUIs will change based on the *Limited View*.
- ✘ Retrofitting a *Limited View* into an existing system can be difficult because the data for the *Limited View*, as well as the code for selecting it, could be spread throughout the system.

Related Patterns:

- *Full View With Errors* is a competitor to this pattern. If limiting the view completely is not possible, error messages can fill in the holes. The "Putting It All Together" section at the end of this paper discuss when to use each pattern.
- A *Session* may have a *Limited View* of data that it distributes throughout the application.
- *Roles* are sometimes used to configure a *Limited View*.
- *State* with *Strategy* can be used to implement a *Limited View*.
- *Composites* and *Builders* can be used to implement a *Limited View*.
- *Null Objects* can be used in places where a view as been limited.
- *MetaData* can be used to configure what parts of a view need to be limited.
- *Checks* [Cunningham 95] describes many details on implementing GUI's and where to put the error notifications.

Known Uses:

- The Caterpillar/NCSA Financial Model Framework [Yoder] prestens a *Limited View* on the data to the user. This framework also provides *Limited View* in user interfaces by changing editing view screens based upon the *Roles* of the user.
- Firewalls provide *Limited Views* on data by filtering network data and making it available only to some systems.
- Web servers provide a Limited View by only allowing users to view directories in the root web directory and in user's public_html directories.
- Most operating systems provide hidden files and directories, which are a form of a *Limited View*.
- Microsoft's Windows NT provides *Limited Views* based upon a user's role. Users are only allowed to see files they have permissions to and they get customized menus based upon those roles.
- The PLoP '98 registration program [Yoder & Manolescu 98] provides *Limited Views* by having a view for the administrator and a view for someone registering for PLoP. People registering for PLoP get a *Limited View* that allows them to view and edit only their information.

Non-security Known Uses:

- Netscape Communicator, Microsoft Office, and many other applications change their user interface depending upon what the user may be editing or viewing. This is commonly done through the use of context sensitive menus and enabling buttons. For example, the menus available while editing a chart in Excel is quite different from those provided for editing a spreadsheet. Also, if no documents are opened, the "Save" and "Save As" menu items are not available in the "File" menu.
- Windows 95's right-click menu is context-sensitive, presenting only the options legal at the mouse pointer's target.
- Hidden files and directories, provided by most operating systems, are forms of *Limited Views*.
- Unlike Microsoft Word and vi, Netscape Composer does not let the user try to save over a read-only file. The "Save" option in the file menu is grayed-out for read-only files. "Save as…" is available as part of the *Limited View*.
- All modern GUIs really provide some sort of *Limited View* by enabling and disabling buttons and menus on the fly.