

## 9.6 Limited Access

Designing the user interface for a system in which different users are granted different access rights can be challenging. This pattern guides a developer in presenting only the currently-available functions to a user, while hiding everything for which they lack permission.

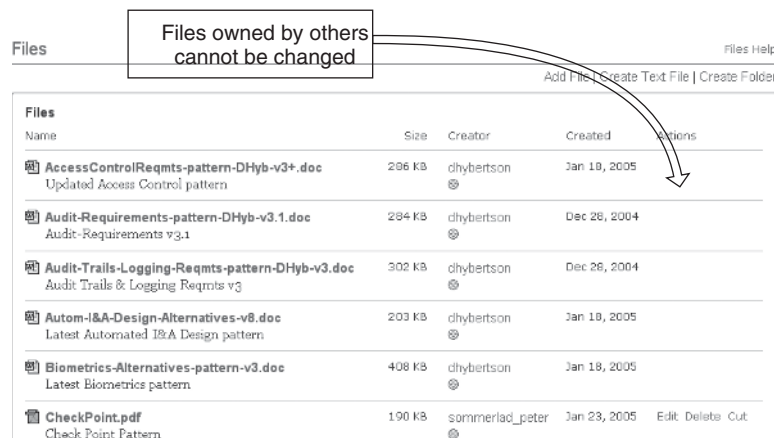
### Also Known As

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

### Example

Extending the Web site example from FULL ACCESS WITH ERRORS (305): registered users are able to upload files to your Web site. After uploading they also need a means of deleting or changing the uploaded files. However, an individual user should only be able to change their own files. Only a user with administrative rights should be allowed to maintain files by all users.

Yahoo! groups provides a similar means of uploading files to groups for its group members. However, only the uploading member is able to delete or edit the file. The group owner can delete files uploaded by all users. If no permission is given, one can see the 'File' menu, corresponding to FULL ACCESS WITH ERRORS (305), but the file folder itself cannot be extended by adding files.



The screenshot shows a web-based file management interface. At the top, there's a header bar with 'Files' on the left and 'Files Help' on the right. Below the header, there's a sub-header with 'Add File | Create Text File | Create Folder'. A callout box with a black border and the text 'Files owned by others cannot be changed' has two arrows pointing to the 'AccessControlReqmts-pattern-DHyb-v3+.doc' and 'CheckPoint.pdf' rows in the table below. The table has five columns: 'Name', 'Size', 'Creator', 'Created', and 'Actions'. The first four rows have a lock icon in the 'Actions' column, while the last row has 'Edit', 'Delete', and 'Cut' options.

Name	Size	Creator	Created	Actions
AccessControlReqmts-pattern-DHyb-v3+.doc Updated Access Control pattern	296 KB	dhybertson	Jan 18, 2005	
Audit-Requirements-pattern-DHyb-v3.1.doc Audit-Requirements v3.1	284 KB	dhybertson	Dec 26, 2004	
Audit-Trails-Logging-Reqmts-pattern-DHyb-v3.doc Audit Trails & Logging Reqmts v3	302 KB	dhybertson	Dec 26, 2004	
Autom-I&A-Design-Alternatives-v8.doc Latest Automated I&A Design pattern	203 KB	dhybertson	Jan 18, 2005	
Biometrics-Alternatives-pattern-v3.doc Latest Biometrics pattern	408 KB	dhybertson	Jan 18, 2005	
CheckPoint.pdf Check Point Pattern	190 KB	sommerlad_peter	Jan 23, 2005	Edit Delete Cut

## **Context**

You are designing the user interface of a system in which access restrictions such as user authorization apply to parts of the interface. While most of the applications of this pattern are within the domain of graphical user interfaces (GUIs), it can also apply to other interface types as well.

## **Problem**

Presenting all potentially-available functionality to users not privileged to use it can represent a security problem. You might want users that don't have access to a functionality to not even be aware that it exists.

A system utilized by people with varying skills and access rights can be very hard to use if every possible option is presented to every user, as is proposed by FULL ACCESS WITH ERRORS (305). It is much more user friendly if a user can only see or select the options actually available.

How can you present a system's functionality and ensure that users can only access those parts or data of a system to which they are entitled?

The solution to this problem must resolve the following forces:

- Users should not be able to see or activate operations they cannot perform.
- Users should not view data for which they have no permissions.
- Users do not like being told what they cannot do and become annoyed by access violation messages.
- Input validation can be easier when you limit users to see and operate only what they can access.
- If options pop in and out dynamically because of changes to access rights or roles, users can become confused and the GUI's usability decreases.

## **Solution**

Only let users see what they have access to. In a GUI, show them only the selections and menus that their current access privileges permit.

For example, if a user is not allowed to edit some data, do not present an edit button and use a read-only field to show that data.

When a user starts the system an I&A mechanism authenticates them and associates a SECURITY SESSION (297), typically within a CHECK POINT (287) architecture. The SECURITY SESSION (297) object caches the current privileges of the user that can then be used by the GUI implementation to decide what functions and data are permissible and may be presented to the user, independently of the function's implementation. In contrast to checking the access rights of a user after a request is issued, as would be done

in FULL ACCESS WITH ERRORS (305), the system checks the access rights before presenting the user interface. Only functionality that is available to the user is rendered by the interface builder.

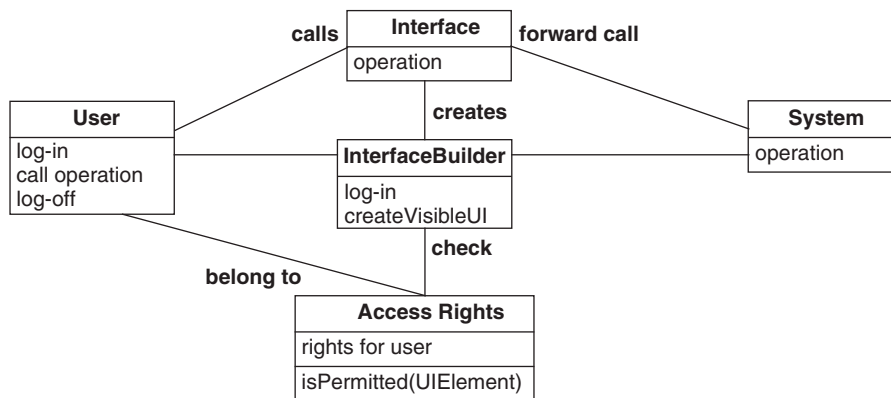
Details of GUI implementation are beyond the scope of this pattern, but patterns such as NULL OBJECT can be used to represent unavailable items or disable active GUI elements. The same mechanism that is used for hiding non-available GUI elements can be used to disable GUI elements, depending on the application's state. For example, if no document has been opened by a text editor, there is no active 'save the document' button—only buttons to open an existing document or create a new one might be available in that state of the application. If the user lacks the permission to create new documents, even the latter might be missing.

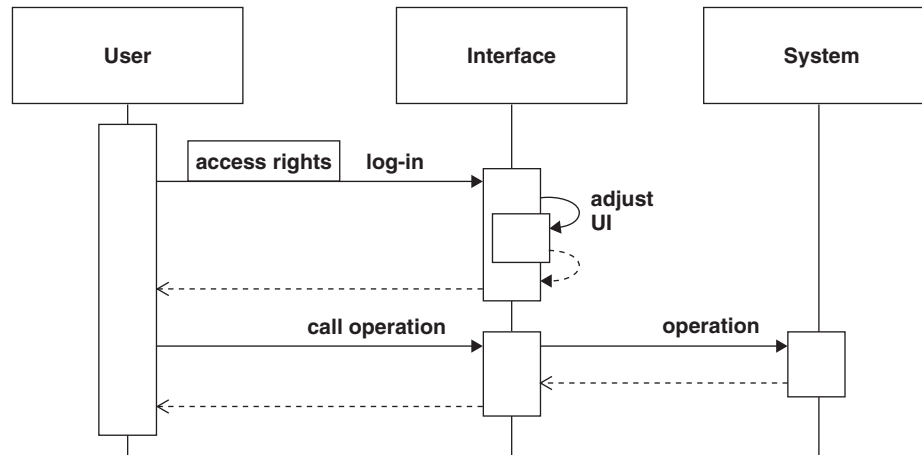
### Structure

In contrast to FULL ACCESS WITH ERRORS (305), the interface is responsible for checking a user's access rights, even before the user interface is presented to the user. The protected system itself does not in general need to check rights, but may still rely on access rights for additional responsibilities such as logging and so on.

### Dynamics

This scenario shows that when the user logs in or otherwise changes their access rights, the interface adjusts its appearance according to the current set of access rights. Later on when the user selects an operation, only valid operations can be accessed, so further checks are unnecessary and the operation is passed on directly to the system. See figure on page 315.





## Implementation

To implement LIMITED ACCESS (312), several aspects need to be considered:

1. *Implement the association of access rights with users.* CHECK POINT (287) and SECURITY SESSION (297) are typical means of providing a user log-in and attaching their access rights. In the case of CHECK POINT (287) the interface can rely on the Check Point object to provide access to the set of enabled user interface elements. For further details about user identification and authentication, see Chapter 7, *Identification and Authentication (I&A)*. For the modeling and management of access rights, see Chapter 8, *Access Control Models*.
2. *Design the user interface and define the mapping of access rights to interface elements.* The issues surrounding the design of a good (graphical) user interface are far beyond the scope of this pattern. However, you should model your individual access rights close to the available user interface elements, so that management and checking of access rights for your application is straightforward. If your application requires complex rules to decide whether an option is valid for a user, evaluating these rules every time the interface is (re-) drawn can be too costly, so you should either use FULL ACCESS WITH ERRORS (305) in that case, or cache the results of such an evaluation, for example within the user's SECURITY SESSION (297). The latter approach is viable as long the user's rights will not change during a session.

A third way to optimize a LIMITED ACCESS (312) user interface is to provide an individual design for each role in ROLE-BASED ACCESS CONTROL (249). This works if your system only needs to support a few defined, stable and clearly-distinguished user roles. You can opt to design separate visual user interface

layouts for each user role. However, you should rely on common layouts for functionality that is available across the different roles.

Regardless of the approach chosen, the interface builder is the component to implement the mapping of access rights to the UI.

3. *Decide how to present and implement accessible versus inaccessible interface elements.* Most UI toolkits already provide the Boolean attributes ‘enabled/disabled’ or ‘show/hide.’ Either of these can be used to control the appearance of user interface elements that depend on the access rights of a user. Hiding is more appropriate for data elements that a user should not access, whereas disabling is better for operational elements such as buttons, menus, or menu items. If possible, you should refrain from creating your own visual appearance mechanism for enabling and disabling options, instead follow the mechanisms suggested by your GUI platform’s guidelines, to avoid confusing your users.

For Web applications, rendering of interface elements mainly consists of string concatenation, so disabling an element often means rendering an empty string or an alternative to the active representation. Since such rendering is mainly sequential, the checks for permissions can be done on the fly during that sequential process.

4. *Take care of security issues in a distributed environment.* If the interface and the protected system are separate, for example as is the case with a Web application, then LIMITED ACCESS (312) can fall short in protecting the system. An attacker in the middle might trick the system into thinking that an access comes from the interface, where it already would have been checked, but instead is actually crafted by the attacker to force access without proper permission.

Even on a single machine on which your system consists of individual components, for example Windows COM components, just relying on the user interface to check every access is dangerous, because compromised parts of the system might access the components without encountering the user interface checks.

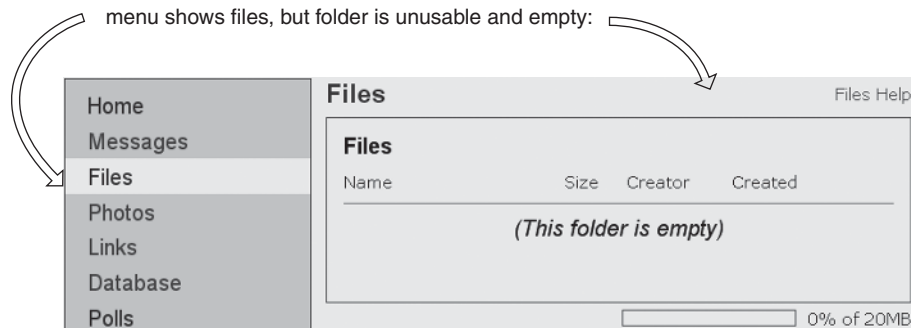
In those situations it is wise to combine LIMITED ACCESS (312) with the checks close to the functionality, as proposed by FULL ACCESS WITH ERRORS (305). For Web applications especially, it is crucial to re-check every request sent to the system for validity and permission before it is handled, even when the user interface already checked for parameters.

## **Variants**

The opposite strategy to LIMITED ACCESS (312) is FULL ACCESS WITH ERRORS (305). While both approaches are proven and practical, it is often the case that in a concrete system you need to combine them. This may be either because both sets of

forces apply, or because you need to re-check access rights within a called operation, for example when you cannot trust your limited access interface to ensure no invalid call is made.

The Yahoo! groups example shows both patterns applied together. Using FULL ACCESS WITH ERRORS (305), you see the complete menu of group features. However, a group without access rights for its regular members doesn't allow use of the File folder after it is selected.



### Known Uses

Most current operating systems' and applications' GUIs provide LIMITED ACCESS (312) with user- and context-specific menus and buttons that are enabled when usable and disabled when their use is impossible in the current context. For example, most word processors don't allow a file to be re-saved if it hasn't changed since the last save.

Firewalls implement LIMITED ACCESS (312), by restricting network traffic to the point at which only allowed network connections can be seen from the outside, all other network packets being silently dropped by the firewall.

Eclipse JDT, the Java development, environment shows only those refactorings, in a pop-up menu, that are applicable to the currently-selected source code portion. While this allows efficient use by experienced programmers, it makes it hard for novices to learn about all available automatic refactorings without studying Eclipse's documentation.

Unix' restricted shell `/usr/lib/rsh`<sup>1</sup> provides a user interface with limited access. Using the `chroot` command with a carefully-crafted restricted Unix environment can provide an even more restricted LIMITED ACCESS (312). The simplest, but insecure, variation of LIMITED ACCESS (312) in Unix is the hiding of files by starting their name

<sup>1</sup> Do not confuse with today's more popular remote shell, `/usr/bin/rsh`.

with a dot, making them invisible to the casual user through not being listed in `ls` commands without specific options.

### **Consequences**

The following benefits may be expected from applying this pattern:

- LIMITED ACCESS (312) disables access to restricted operations and data by providing no means by which the user can even try to access it.
- Since the user can only access permitted data and operations, developers don't have to worry about verifying rights for every access. Note that in the case of a decoupled user interface such as a Web browser, this is careless, and re-checking of parameters and access rights is necessary on the server side.
- Security checks can be simplified, as only the interface depends on it. The rest of the system can neglect further checks.
- Users are guided within their work with the system and won't become confused or annoyed by unavailable options, or by the system barking error messages at them.

The following potential liabilities may arise from applying this pattern:

- Users can become confused and frustrated when options appear and disappear. For example, if when viewing one set of data, an editing button is available, while when viewing another set of data, it disappears. A user might not be aware that a security issue is triggering the hiding of the option, but might assume the application is broken, or that there is something wrong with the data.
- A graphical user interface built on the principle of LIMITED ACCESS (312) can look ugly and weird if it blanks out unavailable options and data without changing its layout. On the other hand, it might be completely confusing if the arrangement of UI elements changes whenever operations appear or disappear. To achieve stability in such cases, often icons, options and buttons are still displayed, but disabled and given a visual hint that they are unusable by reducing their contrast or coloring (graying them out).
- Training and documentation must be tailored for different user groups, since the mode of operation and available options differ with their permissions. References to non-existent UI elements should be avoided, while all accessible ones need to be explained.

- Retrofitting LIMITED ACCESS (312) into an existing system can be difficult, because data for limiting and enabling access, as well as code for doing so, could be spread throughout the system.
- Relying solely on LIMITED ACCESS (312) for checking access rights at the front end of a system carries the danger of someone in the middle tricking a back end into performing operations without the front end having checked permissions. The security principle of Defence in Depth (see Section 15.1, *Security Principles and Security Patterns*) should be applied when an architecture (such as Web applications) is vulnerable in such a situation.

### **See Also**

CHECK POINT (287) and SECURITY SESSION (297) should be considered for designing and implementing the user I&A and association of access rights.

ROLE-BASED ACCESS CONTROL (249) can be used to provide several pre-built incarnations of the user interface for the different user roles. This allows you to avoid the dynamic rendering of available options, with all its problems of layout automation.