9.5 Full Access with Errors

Designing the user interface for a system in which different users are granted different access rights can be challenging. At one end of the spectrum is the approach taken by this pattern, which provides a view of the maximum functionality of the system, but issues the user with an error when they attempt to use a function for which they are not authorized.

Also Known As

Full Access with Exceptions, Full View with Errors, Reveal All and Handle Exceptions, Notified View

Example

Consider you are developing an Internet site. The site should present your company on the World-Wide Web as well as provide downloads for brochures, user manuals, and demo software. However, to be able to track who downloaded such material, Internet surfers are required to provide their name and address before they can start a download. However, to avoid irritating returning users, they are granted privileges by the site via a cookie, and thus do not need to register again. See figure on page 306.

For example Yahoo! groups show a group's features to anonymous users, without letting them access the 'members only' menu. Once logged in and registered as a member of a group, the 'members only' menu is accessible.

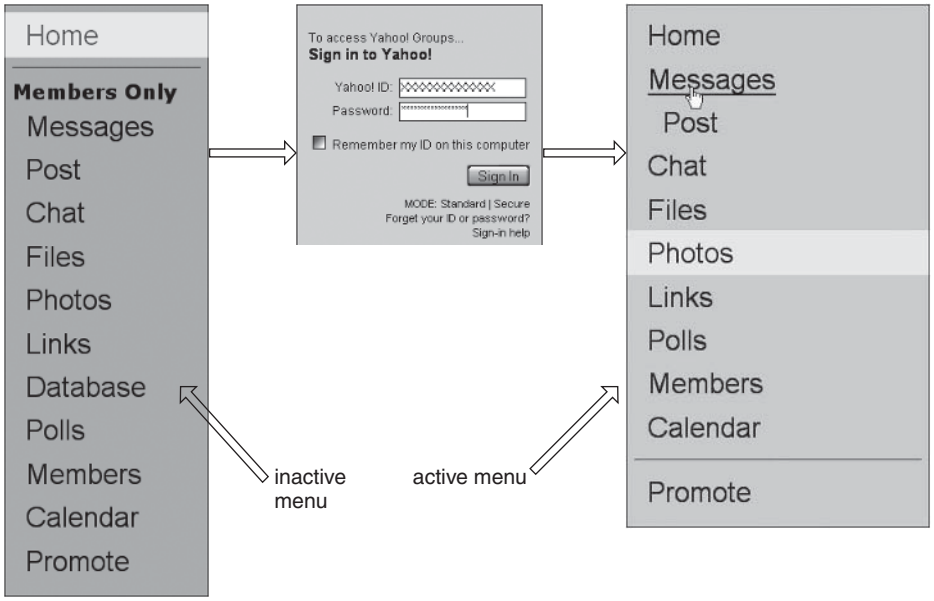
How do you design the Web site so that it shows the possibility of downloads while still restricting access to registered users v?

Context

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

Problem

When designing the user interface for a system with partial access restrictions, you face the challenge of whether to present functionality that a user might not be able to access within their current role or set of access rights, and how to do so. To complicate



the issue, you might not know in advance what possible combinations of access rights will be used.

This problem generalizes to any interface you design whenever there are multiple modes of use, such as different access rights.

How do you present available functionality that might be partially inaccessible?

The solution to this problem must resolve the following forces:

- Users should not be able to view data or perform operations for which they have no permissions.
- Hiding an available and accessible function is inappropriate, because users must be able to see what they can do.
- The visual appeal and usability of a graphical user interface (GUI) can be degraded by varying layouts depending on the (current) access rights of a user. For example, blank space might appear for some users where others see options they can access, or sequence and number of menu items might differ, depending on the current user's rights, and thus 'blind' operation of the menu by an experienced user is no longer possible.
- Showing currently unavailable functions can tease users to into upgrading their access rights, for example by paying for the access or buying a license after using a demo version.
- Trial and error are ineffective means of learning which functions are accessible. Invoking an operation only to learn that it doesn't work with your access rights is confusing.

- The privilege grouping of the typical user community might not be known at the design time of the GUI, and it might change over time, for example through organizational or business process changes, so that providing a few special modes of the GUI depending on the corresponding user roles is inappropriate.
- Checking whether a function is allowed by a user is most efficient, robust and secure, if done by the function itself—at least the code performing the checks is then closely related to the code performing the subsequent operation afterwards.

Solution

Design the system so that every available functionality is visible on its interface. When an operation or data is accessed by someone, the system first checks the access permission. If the access is allowed, the function is performed or data is displayed correspondingly. On lack of permission, an error notification is generated and presented to the user.

Often such an error notification gives the user a chance to upgrade or change access permission, or to provide further authorization for performing an otherwise failing operation. For example, a text editor might fail to save a file over an existing one with the same name. It will give the user an error message with the option of canceling the operation and thus accept the failure, or explicitly authorize the text editor to overwrite the file, thus raising the user's privileges for the current operation.

FULL ACCESS WITH ERRORS (305) works best when denial of an operation is sporadic and users are well aware of what they can and can not do.

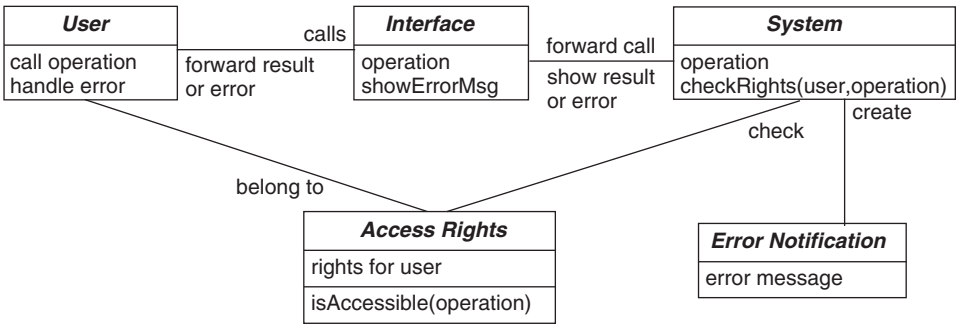
The implementation of access right checks that are closely related to the function to be performed allows implementation of more sophisticated authorization schemas that depend on more than just a flag for an access right, such as data values used in the current transaction. For example, a bank might withhold the account statements of their bosses or of very rich clients from regular clerks, who could otherwise look at almost all accounts within their range of duty.

In this example the check not only involves the regular flag-based permissions of the current user, but also the organizational status of the account holder, as well as the current balance of the account. Such complex business logic for access rights is almost impossible to implement using a generic access rights management system. The benefit of implementing FULL ACCESS WITH ERRORS (305) is that there is a consistent error-handling mechanism in place that is readily available for developers to use and is also well known to the system's users.

Structure

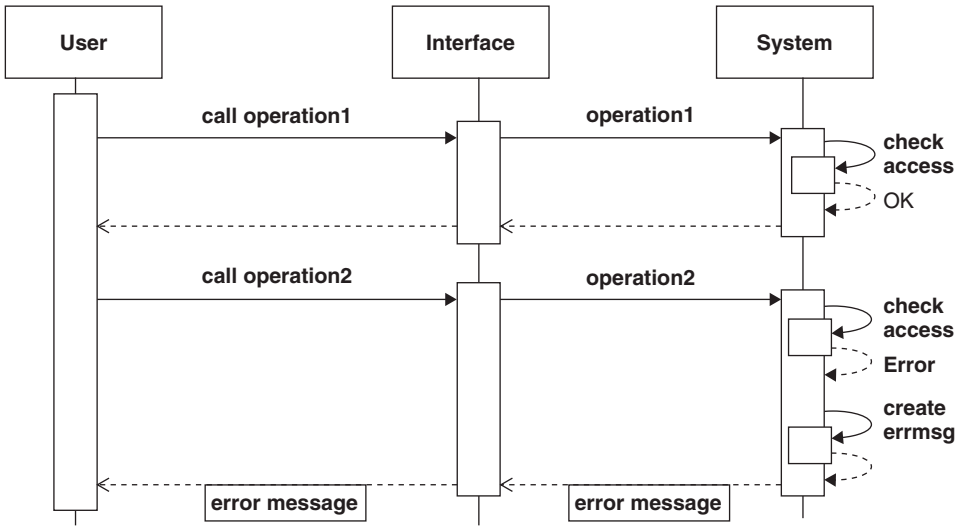
The diagram shows the implementation of FULL ACCESS WITH ERRORS (305) in which the system code performing an operation checks the access rights before each operation, rather than the interface code. Usually such checking code is similar for

each operation, and should be factored into a common routine, or, with modern programming approaches such as aspect-oriented programming, added as an aspect to each operation of the system.



Dynamics

The sequence diagram shows two cases relevant for FULL ACCESS WITH ERRORS (305). The call of the first operation is granted to the user and thus after the check of the access rights is done, it returns successfully. The second case calling operation 2 shows the situation in which the user has insufficient rights and an error message is returned instead.



Implementation

To implement FULL ACCESS WITH ERRORS (305) several tasks need to be done:

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 FULL ACCESS WITH ERRORS (305) the modeling of access rights should follow closely the underlying operations of the system, so that is easy to associate them with the rights. Further details on I&A are given in Chapter 7, *Identification and Authentication (I&A)*. Modeling and management of access rights are discussed in Chapter 8, *Access Control Models*.
2. *Design the interface representing the full set of the system's functionality.* You should provide visual hints for novice users to recognize features that might be unavailable to them and show them how to achieve the corresponding access rights. Visual grouping of normally-available features versus those requiring authorization can be helpful.
3. *Design error notification for the user.* In simple cases this often allows the user to register or authenticate themselves, such as with Yahoo groups. In more complex settings with finer-grained access rights management, it might give a process description of how to obtain the corresponding access right. At the other end of the spectrum, an unauthorized access can be simply ignored. An additional means of security is to audit users, especially for unauthorized access. This can provide hints on potential for misuse, or on the lack of access rights for a given user role.
4. *Provide access rights checks for system components*. Before the execution of each system function, a user's right to do so must be checked. A combination of CHECK POINT (287) and SECURITY SESSION (297) provide a means of implementing the place where this checking is performed. Further details about how to ensure that such checks are carried out is beyond the scope of this pattern, but, for example, a COMMAND PROCESSOR [POSA1] can check access rights for issued commands centrally. Integrate checks and error notification mechanisms with the system.

Example Resolved

Following FULL ACCESS WITH ERRORS (305), the Web site initially contains all possibly available links. The download links are only available for registered users, so they point to a CGI script that first checks whether the user has registered already by checking whether the corresponding cookie is set in the request. In this case the Web site can also deal with bookmarks, links pointing to protected material issued by unauthenticated or unauthorized users.

Variants

LIMITED ACCESS (312) demonstrates the opposite strategy to FULL ACCESS WITH ERRORS (305) by presenting the user only the permitted functionality, thus avoiding surprising error messages. In many concrete cases neither of the extremes shown by these two patterns are used, but concrete designs lie somewhere in the middle. However, it is difficult to define 'somewhere in the middle.' As a starting point, you should chose a pattern according to the more visible forces in your context, and adapt it to suit the application domain.

A middle ground between FULL ACCESS WITH ERRORS (305) and LIMITED ACCESS (312) is to show all an application's features, but to 'gray out' menu items or buttons not available in the current situation. This allows a user to see what is available, but still provides instant feedback on what is really usable at the moment. Often this approach combines best of both worlds, as long as there aren't too many features to clutter the UI.

Known Uses

The Internet book-seller Amazon, as well as many other Internet sites, provide casual surfers with a view of almost all their functionality. For example, the links to a shopping cart and 'view my account' are active for everyone. When you open these, you can proceed. However, before checking out, or before you can actually see your orders, you have to either sign in or register yourself. This way, everybody sees what functionality is available, but only logged-in users can access their own data.

Under the Unix shell you can activate almost any program on any file in the file system. However, when your access rights are insufficient, accessing files or programs fails, often with a message saying 'permission denied.' Only the dedicate super user 'root' is unprotected from carelessly calling programs or overwriting files, giving access to everything and overriding all access rights set.

Oracle's SQLPlus interactive database access language allows you to execute any syntactically-valid SQL statement, displaying an appropriate error message if illegal access to data is attempted.

Most word processors and text editors, including Microsoft Word and vi, let the user try to save over a read-only file. The program displays an error message after the save has been attempted and has failed.

Consequences

The following benefits may be expected from applying this pattern:

- A system can be effectively secured, because a user's permissions for each individual operation are checked before the operation is executed.

- All possible functions are visible to a user, not only providing a consistent interface, but also demonstrating all available features, even when the user is not (yet) privileged to use them.
- It is easy to change access rights and groups for such a system without influencing the concrete implementation of the system or its interface.
- Retro-fitting this pattern into an existing system is straight forward: write an interface that will handle all possible functions, and whenever a problem happens with an operation, simply abort the operation and display an error message.
- Documentation and training material for an application can be consistent for each type of user.
- FULL ACCESS WITH ERRORS (305) fits well in situations in which users can upgrade their privileges for a otherwise unavailable operation on the fly, for example by confirming a dialogue, without breaking their flow of work.
- For Web applications applying the pattern allows stable URLs and links to a download area, even in the case in which a user must register first. A pre-registered user will be able to download directly using the same URL.

The following potential liabilities may arise from applying this pattern:

- Users can become confused and frustrated when they are forced to apply trial and error to learn and use a system that presents many things but then often just replies with an error message.
- Every operation needs to check permissions. This may cause complexity, duplicated code or—if omitted—lack of security.
- The user interface design becomes bloated when a system is serving disparate user roles. If you show everything to everybody, regardless of their interest in the system, it is very hard to find your way through the task. Too many features either remain hidden in deeply-nested menus or dialogs, or clutter available screen space.

See Also

CHECK POINT (287) and SECURITY SESSION (297) can be used to implement FULL ACCESS WITH ERRORS (305).