Architectural Patterns for Enabling Application Security

Joseph Yoder

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801
j-yoder@uiuc.edu

Jeffrey Barcalow

Reuters Information Technology
1400 Kensington
Oak Brook, IL 60523
barcalow@xnet.com

Abstract

Making an application secure is much harder than just adding a password protected login screen. This paper contains a collection of patterns to be used when dealing with application security. *Secure Access Layer* provides an interface for applications to use the security of the systems on which they are built. *Single Access Point* limits entry into the application through one single point. *Check Point* gives the developer a way to handle an unknown or changing security policy. Groups of users have different *Roles* that define what they can and cannot do. The global information about the user is distributed throughout the application with a *Session*. Finally, users are presented with either a *Limited View* of legal options or are given a *Full View With Errors*. These seven patterns work together to provide a security framework for building applications.

This paper was submitted, accepted and workshopped at PLoP '97

Introduction

Systems are often developed without security in mind. This omission is primarily because the application programmer is focusing more on trying to learn the domain rather than worrying about how to protect the system. The developer is building prototypes and learning what is needed to satisfy the needs of the users. In these cases, security is usually the last thing he or she needs or wants to worry about. When the time arrives to deploy these systems, it quickly becomes apparent that adding security is much harder than just adding a password protected login screen. This paper describes how to design a system so that details of security can be implemented late in the development.

In corporate environments where security is a priority, detailed security documents are written describing physical, operating system, network, and application security. These security documents deal with issues such as user privileges, how secure passwords have to be and how often they might need to be changed, if data needs to be encrypted, and how secure the communication layer needs to be.

Often security is initially ignored because either the security policy is not generally available, or it just seems easier to postpone security concerns. Ignoring the security issues is dangerous because it can be difficult to retrofit security in an application. While an application's design could initially be more complicated by incorporating security from the start, the design will be cleaner than the result of integrating security late in the development cycle. Also, huge code rewrites can be avoided because the corporate security policy can be integrated at any stage of the development cycle. A well thought out design that includes security considerations will make it simpler to adapt to changing security requirements.

The seven patterns presented in this paper can be applied when developing security for an application. They are not meant to be a complete set of security patterns. Rather, they are meant to be the start of a collection of patterns that will help developers address security issues when developing applications. Both authors have had the experience of refactoring a system to make it meet corporate security requirements *after* the system was developed (The Caterpillar/NCSA Financial Model Framework [Yoder]). From this experience, we found the following patterns that can be applied while developing secure applications. In general, it can be very beneficial to structure the architecture of an application with these patterns in mind, even if you are not going to need security at a later point.

Secure applications should not allow users to get through a back door that allows them to view or edit sensitive data. *Single Access Point* helps solve this problem by limiting application entry to one single point. Also, an application is only as secure as its components and its interactions with them. Therefore, any application should have a *Secure Access Layer* for communicating with external systems securely and all components of an application should provide a secure way to interact with them. This layer helps keep the application's code independent of the external interfaces.

It is important to provide a place to validate users and make appropriate decisions when dealing with security breaches. *Check Point* encapsulates the strategies dealing with different types of security breaches while making the punishment appropriate for the security violation.

When there are many users for a system, privileges for these users can usually be categorized by the users' jobs instead of their names. In these cases, users can be assigned to certain *Roles* that describe appropriate actions for that type of user. Groups of users will have different *Roles* that define what they can and can not do. For example, an application might provide *Roles* for administrating the application, creating and maintaining data in an application, and viewing the data. All users of this application will be assigned a set of *Roles* that describe what privileges they have.

Common data that will be used throughout a secure system, such as who the user is, what privileges they have, and what state the system is in. For example, to access a database, the application needs the username and database location. Multiple users might be logged onto the system at the same time and this information needs to be maintained separately for each user. Global information about the user is distributed throughout the application via a *Session*.

A user's view of the system will depend upon the current roles he or she has. These views can be presented as a *Limited View* of legal options by only showing the user what they have permission to see or run.

Another option is to present a *Full View With Errors* that allow the user to see everything. Options that the user does not have current permission to perform can be disabled, or exceptions can be fired whenever the user performs an illegal operation.

The pattern catalog in Table 1 outlines the application security patterns discussed in this paper. It lists each pattern's name with the problem that the pattern solves. These patterns collaborate to provide the necessary security within an application. They are tied together in the "Putting It All Together" section presented at the end of this paper.

Pattern Name	Intent	Page
Single Access Point	Providing a security module and a way to log into the system.	4
Check Point	Organizing security checks and their repercussions.	7
Roles	Organizing users with similar security privileges.	11
Session	Localizing global information in a multi-user environment.	14
Full View With Errors	Provide a full view to users, showing exceptions when needed.	17
Limited View	Allowing users to only see what they have access to.	19
Secure Access Layer	Integrating application security with low level security.	24

Table 1 - Pattern Catalog

This paper does not discuss patterns or issues dealing with low-level security on which our *Secure Access Layer* pattern is built. These low-level security patterns deal with issues such as encryption, firewalls, Kerberos, and AFS. Many good sources of low-level security issues and techniques are available. The *International Cryptographic Software Web Pages* [ICSP] and *Applied Cryptography* [Schneier 95] are very good references for more details on these issues.

Single Access Point

Alias:

Login Window
One Way In
Guard Door
Validation Screen

Motivation:

A military base provides a prime example of a secure location. Military personnel must be allowed in while spies, saboteurs, and reporters must be kept out. If the base has many entrances, it will be much more difficult and expensive to guard each of them. Security is easier to guarantee when everyone must pass through a single guard station.

It is hard to provide security for an application that communicates with networking, operating systems, databases, and other infrastructure systems. The application will need a way to log a user into the system, to set up what the user can and can not do, and to integrate with other security modules from systems that it will be interacting with. Sometimes a user may need to be authenticated on several systems. Additionally, some of the user-supplied information may need to be kept for later processing. *Single Access Point* solves this by providing a secure place to validate users and collect global information needed about users who need to start using an application.

Problem:

A security model is difficult to validate when it has multiple "front doors," "back doors," and "side doors" for entering the application.

Forces:

- Having multiple ways to open an application makes it easier for it to be used in different environments.
- An application may be a composite of several applications that all need to be secure.
- Different login windows or procedures could have duplicate code.
- A single entry point may need to collect all of the user information that is needed for the entire application.
- Multiple entry points to an application can be customized to collect only the information needed at that entry point. This way, a user does not have to enter unnecessary information.

Solution:

Set up only one way to get into the system, and if necessary, create a mechanism for deciding which sub-applications to launch.

The typical solution is to create a login screen for collecting basic information about the user, such as username, password, and possibly some configuration settings. This information is passed through some sort of *Check Point* that verifies the information. A *Session* is then created based upon the configuration settings and the user's access privileges. This *Session* is used to keep track of the global information that deals with the user's current interaction with the application. When opening up any sub-application, requests are forwarded through the *Check Point* for handling any problems and validations.

UNIX is an example of a system with multiple access points. Each network port can provide access to a separate service. These multiple access points make it difficult to secure a UNIX system as a hole. Individual applications running on a UNIX system can still be secured, however. For example if an application has web, email, and programming interfaces, steps must be taken to ensure that each of these interfaces are forced through a shared *Single Access Point* before accessing the rest of the application.

The "Putting It All Together" section describes the initialization process in more detail. The important idea here is that *Single Access Point* provides a convenient place to encapsulate the initialization process, making it easier to validate that the initial security steps are performed correctly.

Example:

There are many examples of *Single Access Point*. In order to access an NT workstation, there is a single login screen which all users must go through to access the system. This *Single Access Point* validates the user and insures that only valid users access the system and also provides *Roles* for only allowing users to see and do what they have permissions to do. Most UNIX systems also have *a Single Access Point* for getting a console shell. Oracle applications also have many applications such as SQLPlus and the like that provide a *Single Access Point* as the only means for running those applications.

Consequences:

✔ A *Single Access Point* provides a place where everything within the application can be setup properly. This single location can help ensure all values are initialized correctly, application setup is performed correctly, and the application does not reach an invalid state.

✔ Control flow is simpler since everything must go through a single point of responsibility in order for access to be allowed. Note, *Single Access Point* is only as secure as the steps leading up to it.

✘ The application cannot have multiple entry points to make entering an application easier and more flexible.

Related Patterns:

- *Single Access Point* validates the user's login information through a *Check Point* and uses that information to initialize the user's *Roles* and *Session.*
- A *Singleton* [GHJV 95] could be used for the login class especially if you only allow the user to have one login session started or only log into the system once. A Singleton could also be used to keep track of all Sessions and a key could be used to know which session to use

Known Uses:

- UNIX telnet and Windows NT login applications use *Single Access Point* for logging into the system. These systems also create the necessary *Roles* for the current *Session.*
- Most application login screens are a *Single Access Point* into programs because they are the only way to startup and run the given application.
- The Caterpillar/NCSA Financial Model Framework [Yoder] has a FMLogin class, which provides both *Single Access Point* and *Check Point.*
- The PLoP '98 registration program [Yoder & Manolescu 98] provides a *Single Access Point* for logging into the system and entering in credit card information when users registered for PLoP '98.
- Secure web servers, such as Java Developer's Connection appear to have multiple access points for each URL. However, the web server forces each user through a login window before letting them download early access software.

Non-security Known Uses:

- Any application that launches only one way, ensuring a correct initial state.
- Windows 95, also uses a login window which is a *Single Access Point*, but it is not secure because it allows any user to override the login screen.
- Single creational methods provide for only one way to create a class. For example, Points in VisualWorks Smalltalk [OS 95] guides you to creating valid points by providing a couple of creational methods that ensure the Object is initialized correctly. Kent Beck's describes *Constructor Methods* as a single way to create well-formed instances of objects [Beck 97]. These are put into a single "instance creation" protocol. This becomes the *Single Access Point* to create new objects.

- *Constructor Parameter Method* [Beck 97] initializes all instance variables through a single method, which is really a *Single Access Point* for that class to initialize its instance variables.
- Concurrent programs can encapsulate non-concurrent objects inside an object designed for concurrency. Synchronization is enforced through this *Single Access Point*. *Pass-Through Host* design [Lea 97] deals with synchronization by forwarding all appropriate methods to the Helper using unsynchronized methods. This works because the methods are stateless with respect to the Host class.

Check Point

Alias:

Access Verification
Authentication and Authorization
Holding off hackers
Validation and Penalization
Make the Punishment Fit the Crime

Motivation:

Military personnel working at a base have a security badge which is checked by the guard front gate. They also have keys which will allow them to enter only areas in which they are authorized. No one is allowed in without a badge. Anyone who tried to enter a restricted area without clearance is dealt with severely.

The goal of application security is to keep unwanted perpetrators from gaining access to application areas where they can find confidential information or can corrupt data. *Single Access Point* can help solve this problem by making a single point to enter an application. At the *Single Access Point*, checks must be made to verify that a user is permissible. Unfortunately, these checks could make getting into the application more difficult for legitimate users. For example, a common mistake is entering the wrong password. While users may often mistype their passwords, frequent, consecutive failures without success could indicate that someone is trying to guess a password and break into the application. Application security must allow occasional mistakes while doing its best to keep a hacker out. A developer could design many checks to determine if a user is trying to break into the system or is just making common mistakes. These checks could become complicated and spread out throughout the application, making it difficult to manage and maintain the applications. *Check Point* helps out with this by organizing these checks.

Problem:

An application needs to be secure from break-in attempts, and appropriate actions should be taken when such attempts occur. Different organizations have different security policies and there needs to be a way to incorporate these policies into the system independently from the design of the application.

Forces:

- Having a way to authenticate users and provide validation on what they can do is important.
- Users make mistakes and should not be punished too harshly for mistakes.
- If too many mistakes are made, some type of action needs to be taken.
- Different actions need to be taken depending on the severity of the mistake.
- Lots of error checking code all your application can make it difficult to debug and maintain.

Solution:

Create an object that encapsulates the algorithm for the company's security policy. This object usually keeps track of how many exceptions happen and decides what action needs to be taken based on the severity of the violation. Any security check can be part of the *Check Point* algorithm, from password checks to time-outs to spoofing.

All organizations will have some sort of security policy that the applications developer needs to implement. The security policy may not be available when an application is designed, however. Also the policy might change over the life of the application. The principal goal here to design the application with a place to encapsulate the security policy. Early in prototyping and development, a developer might just create a dummy object that allows any user to get into the system. This dummy *Check Point* makes it easier to add security at a later point.

The implementation of *Check Point* combines several patterns. *Single Access Point* is used to ensure that security checks are performed correctly and that no initial security checks are skipped. A key feature of *Check Point* is that the security policy can be changed in a single place. In this way, the *Check Point* algorithm is a *Strategy* [GHJV 95]. This *Strategy* knows what security checks are performed, the order in which checks should be performed, and how to handle failures. Different *Strategy* objects could be plugged in depending on the desired level of security. The authors have found the *Strategy* quality of *Check Point* to be useful because security can be turned off in a development situation and can be turned back on for testing and deployment.

In some systems, this pattern is separated into two conceptual parts: authentication and authorization. Authentication verifies who and where a user is. Authorization involves checking the privileges of an authenticated user. This distinction can be useful in a distributed system where applications may run at different locations. Users can be authenticated from a central login application so they do not have to re-enter a password for each application. Authorization can be performed on operations as needed for each application. *Check Point* utilizes the user's *Role* to provide the proper authorization to the system.

It is also useful to break this pattern into the two parts since usually authentication is done when a user starts using the system and authorization is needed whenever a user tries to use a secured operation. Therefore, whenever a security check is needed, it can be forwarded to authentication part of the *Check Point*, which in turns provides proper access according to the user's *Role*.

A *Check Point* design can vary greatly because of the security policy. If a company's security policy is not yet defined or if it might change, then the *Check Point* algorithm should be designed carefully. Three factors should be considered in designed a *Check Point* component when the security policy is uncertain: failure actions, repeatability, and deferred checks.

Depending on the security violation or error, different types of failure actions may be taken. Failure actions can be broken down by level of severity. These types of failures and actions are contingent upon the security policy you are implementing. For example, the simplest action is to return a warning or error message to the user. If the error is non-critical, the security algorithm could treat it as a warning and continue. A second level of failure could force the user to start over. The next level of severity could force an abort of the login process or quit the program. The highest level of severity could lockout a machine or username. In this case, an administrator might have to reset the username and/or machine access. Unfortunately this could cause problems when a legitimate user tries to login later, so if the violation is not extremely critical, the username and machine disabled flags could be timestamped and automatically re-enabled after an hour or so. All security failures could also be logged.

Sometimes, the level of severity of a security violation depends on how many times the violation is repeated. A user who types a password incorrectly once or twice should not be punished too harshly. Three or four consecutive failures could indicate a hacker is trying to guess a password. To handle this situation, the *Strategy* can include counters to keep track of the frequency of security violations and parameterize the algorithm.

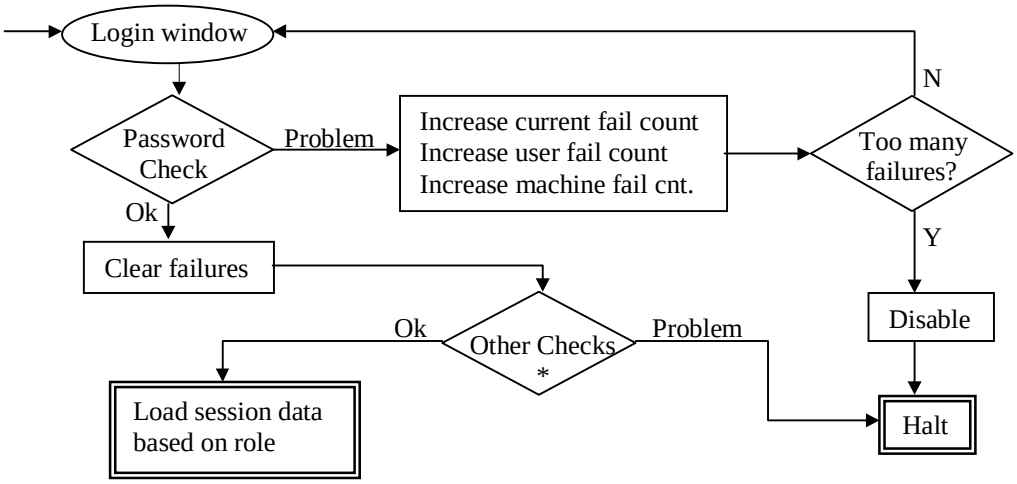
If some security checks cannot be performed in the *Check Point* as the user enters the application, they must be deferred until later. For these cases, the *Check Point* could have a secondary interface for application components to use, or a separate authorization component is needed. This secondary interface is the third factor to consider when designing a *Check Point*.

Because application security can be a tricky point in development, it may be desirable to try to make a reusable security module for use in several applications. That goal is difficult when security requirements vary between the applications. All application teams will need to be involved to ensure the framework will satisfy each application's requirements.

One approach for reusing security code is to create a library of pluggable security components and a framework for incorporating these components into applications. However, the algorithm for putting together components will almost always be overridden, making the framework difficult to generalize. Another approach is to create configuration options that allow small parts of the algorithm to be turned on and off. The *Check Point* can be an Achilles' Heel of an application's security, so every possible branch in the logic must be carefully checked.

Example:

Figure 1 summarizes the *Check Point* algorithm that was used for the Caterpillar/NCSA Financial Model. The process starts with a *Single Access Point* at the login window. The password check and failure loop has three counters. One keeps track of the number of failures since the application has been loaded. The other two counters keep track of consecutive failures by that user and by someone at that terminal, which could be spread out over several application startups. When too many consecutive failures occur, the user and/or terminal is disabled for a fixed duration of time or until the administrator resets the account or terminal. These decisions were part of the Caterpillar policy and could change. Password verification is performed by the database through the *Secure Access Layer*. Once a correct password has been entered, several other checks are applied. This is because while a login might be allowed to access the database, the login might not be legal for the application. Password expiration is also implemented after these checks. Notice that while the consecutive failure error disables the account and/or machine, refusing to enter a new password simply aborts the application. When *Check Point* is complete, it configures the *Session* based on a *Role*. The *Check Point* can enter Login successes and failures into a log file.



* These other checks could include: Is the machine legal? Is the machine disabled? Is user's account disabled? Does user have valid role? Has the user's password expired? These other checks are related to the companies security policy.

Figure 1 – An Example of a Check Point Algorithm

Consequences:

✔ *Check Point* is a critical security location where security must be absolutely enforced. *Check Point* localizes the security model that needs to be certified.

✔ *Check Point* can be a complex algorithm. While this complexity may be unavoidable, at least it is isolated in one location, making the security algorithm easier to change.

✘ Some security checks cannot be performed at startup, so *Check Point* must have a secondary interface for parts of the application which need those checks. Some information needed for these security checks must be kept until needed later. This information, which could include username, password, and *Roles*, can be stored in a *Session*.

Related Patterns:

- The *Check Point* algorithm uses a *Strategy* for application security.
- *Single Access Point* is used to insure that *Check Point* gets initialized correctly and that none of the security checks are skipped.
- *Roles* are often used for *Check Point's* security checks and could be loaded by *Check Point*.
- *Check Point* usually configures a *Session* and stores the necessary security information in it. It can also interact with the *Session* to get the user's *Role* during the authorization process.

Known Uses:

- The login process for an ftp server uses *Check Point*. Depending on the server's configuration files, anonymous logins may or may not be allowed. For anonymous logins, a valid email is sometimes required. This is similar for telnet.
- The Caterpillar/NCSA Financial Model Framework [Yoder] uses *Check Point* to check passwords, *Roles*, and machines. The example above summarizes its implementation.
- Xauth uses a cookie to provide a *Check Point* that X-windows applications can use for securely communicating between clients and servers.
- Java applets that need to write to a user's hard drive use the authentication/authorization process in a different way. The applet is authenticated with a certificate authority to verify who created it. Then the web user manually authorizes to run outside the Java "sandbox" on the user's machine.
- The PLoP '98 registration program [Yoder & Manolescu 98] has a *Check Point*, which only allows authorized users to log onto the system and change their user information.

Roles

Alias:

Actors
Groups
Projects
Profiles
Jobs
User Types

Motivation:

Every military base has a commander. This base commander gives the order if the base is to be shut down or brought to full alert status. These commands are privileges of the office of base commander, not of the particular person who holds that position. When the base commander is transferred or retires, a new person holds the position of base commander. "Base commander" is that person's role.

Security can be more complicated in multi-user applications. Users have different areas of the application that they can see, can change, and "own." When the number of users is large, the security permissions for users often fall into several categories. These categories could correspond to a user's job titles, experience, or division. Meanwhile, the administrator must struggle with managing the security profiles for a large number of users. The administrator needs an easier way to manage permissions.

Problem:

Users have different security profiles, and some profiles are similar. If the user base is large enough or the security profiles are complex enough, then managing user-privilege relationships can become difficult.

Forces:

- With a large number of users it is hard to customize security for each person.
- Groups of users usually share similar security profiles.
- A user may need to have an individual security profile.
- Security profiles may overlap.
- A user's security profile may change over time.

Solution:

Create one or more role objects that define the permissions and access rights that groups of users have.

When a user logs in, he is assigned a set of privileges that specify what data is accessible and which parts of the application can be activated. From an administrator's standpoint this user-privilege relationship [Figure 2] is M-to-N, making it difficult to manage.

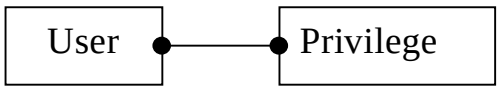


Figure 2 - User-Privilege Relationship

This pattern introduces a level of indirection (the *Role*). This level of indirection splits the user-privilege relationship into user-role and role-privilege relationships [Figure 3]. While these two new relationships are still M-to N, selecting appropriate *Roles* can reduce the total number of relationships. The primary benefit is that privileges can usually be grouped together into common categories.

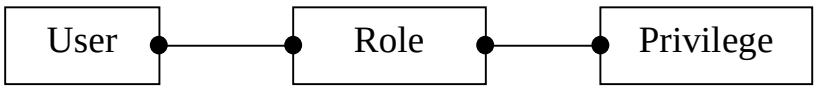


Figure 3 - User-Role-Privilege Relationship

Introducing *Roles* creates two new relationships that must be managed: user-role and role-privilege. These new relationships can help make managing security easier. When the privileges of a job title change, that role-privilege relationship can be edited directly. When a user gets a promotion, the user-role relationship can be changed instead of checking each user-privilege relationship for accuracy.

Sometimes, a subset of the original user-privilege relationship must also be maintained to allow each user to have private privileges. The easiest way to do this is to give each user an independent role, which happens to be the same as their username. *Roles* should only be used when the extra level of indirection provides a conceptual or manageability advantage over the direct user-privilege relationship.

For the role-privilege relationship, role objects could know what privileges for which they are authorized. The converse implementation would require every privilege call to check its roles before returning a value or performing an operation. This option spreads security code throughout the application, so it should be avoided. The preferred implementation, defining privileges within the role objects, is an example of the *Limited View* pattern.

While assigning each user a single *Role* may simplify organization, in reality a user could be both an accountant and a manager. An "Accountant Manager" *Role* could be created but that is not a generalized solution and could make it so *Roles* are created every possibility. A better solution is to make the user object's *Role* variable store a set of *Roles*. While this approach makes it easier to map a user to the appropriate set of *Roles* from an administrative standpoint, it makes privilege lookups more complicated inside the application. The role-privilege relationship must be checked for each of the user's *Roles*, meaning simple comparisons must be replaced by for-loop comparisons.

This pattern presents a simplified way of designing with *Roles*. A variation of this pattern is to allow a user to have several types of *Roles*. For example, a user object could have a set of *Roles* describing its editing privileges and another set of *Roles* describing its viewing privileges. If *Roles* overlap heavily, it may make sense to build a hierarchy of role types and subtypes. You can think of organizing roles in such a way that a role can contain other *Roles*. Thus, you could also create *Composite Roles* for maintaining *Roles*. The role of supervisor has the role of an employee and the role for accessing detailed employee information for those employees that the supervisor is responsible for. Martin Fowler has written a detailed paper [Fowler 97-2], which contains a good description of *Roles* and their implementations. While this pattern discusses *Roles* only in the context of security, *Roles* can be useful throughout the design of any application.

Example:

Roles can be used with security in *many* different ways. The Caterpillar/NCSA Financial Model uses a relatively straightforward approach. After a user passes authentication and other security checks successfully (from the *Check Point*), the application looks up the user's roles in a protected table. Then, the user selects a single role (such as accountant, manager, or developer) from his or her possible roles. The application uses the selected role as the role for its database session (the database's *Secure Access Layer*). Since database tables can be permissioned by role, setting the database session's role automatically creates part of a *Limited View*. The user's login name is used for further permissioning and limiting of the view.

Consequences:

- ✔ Instead of managing user-privilege relationships, the administrator will manage the user-role and role-privilege relationships.
- ✔ *Roles* can be a convenient organizational technique for administrators.
- ✔ *Roles* are a good way to group together common privileges.

✔ Administrative tasks can be simplified by using *Roles*. For example, all new employees could be allowed to view and edit a training database, but only view the real database. A "training" *Role* could be created for these permissions. Then, any new employee account will only have to be given a training *Role* instead of a potentially large set of permission options.

✘ *Roles* add an extra layer of complexity for developers.

✘ Even if *Roles* are used, each user will need a private *Role* to maintain private privileges and preferences.

Related Patterns:

- *Dealing with Roles* [Fowler 97-2] provides a whole pattern language discussing roles with more specific implementation details.
- *Check Point* is used when a user tries to perform an operation without having a *Role* with the proper permissions.
- A user's *Role* is needed throughout the application. The *Role* information can be stored in a *Session* object for access whenever needed.
- *Roles* could be used to determine the scope of a *Limited View.*
- When an application should behave differently depending on a user's job, the user's *Roles* could be used to select a *Strategy* for the application.

Known Uses:

- UNIX uses three classifications for secure access to files and directories. The middle classification is "group," which is an example of *Roles*. The user-role relationship is stored in /etc/group and is sorted by *Roles*. The file system stores the role-privilege relationship.
- Some web servers use .htaccess and .htgroups files which define groups of users (*Roles*) that can access certain areas of a web site.
- Oracle has a *Roles* feature for defining security privileges. User-role and role-privilege relationships are stored in tables.
- In the GemStone OODB data is stored in a segment. GemStone treats segments analogously to the way UNIX treats files. Users in GemStone can have one or more groups (*Roles*), and each segment has read and write privileges defined for all users, a set of groups, and the owner. Since a segment can have a set of groups, it is a little more powerful than UNIX with respect to groups.
- The PLoP '98 registration program [Yoder & Manolescu 98] has two *Roles*; attendee and administrator.

Non-security Known Uses:

- Office 97's Help system is an example of using roles for a non-security issue. The animated paperclip help character can be configured to give from no help (expert) to frequently unrequested help (beginner). This configuration is the current user's role. Windows '95 profile is also a role.

Session

Alias:

User's Environment
Namespace
Threaded-based Singleton
Localized Globals

Motivation:

Military personnel's activities are tracked while they are in a high-security military installation. Their entry and exit are logged. Their badges must be worn at all times to show they are only where they are supposed to be. Guards inside of the base can assume personnel with a badge have been checked thoroughly at the base entrance. Therefore they only have to perform minimal checks before allowing them into a restricted area. Many people are working in a base at the same time. Each security badge uniquely identifies who that person is and what they can do. It also tracks what the carrier of the badge has been doing.

Secure applications need to keep track of global information used throughout the application such as username, roles, and their respective privileges. When an application needs to keep one copy of some information around, it often uses the *Singleton* pattern [GHJV 95]. The *Singleton* is usually stored in a single global location, such as a class variable. Unfortunately, a *Singleton* can be difficult to use when an application is multi-threaded, multi-user, or distributed. In these situations, each thread or each distributed process can be viewed as an independent application, each needing its own private *Singleton*. But when the applications share a common global address space, the single global *Singleton* cannot be shared. A mechanism is needed to allow multiple "Singletons," one for each application.

Problem:

Many objects need access to shared values, but the values are not unique throughout the system.

Forces:

- Referencing global variables can keep code clean and straightforward.
- Each object may only need access to some of the shared values.
- Values that are shared could change over time.
- Multiple applications that run simultaneously might not share the same values.
- Passing many shared objects throughout the application make APIs more complicated.
- While an object may not need certain values, it may later change to need those values.

Solution:

Create a Session object, which holds all of the variables that need to be shared by many objects. Each *Session* object defines a namespace, and each variable in a single *Session* shares the same namespace. The *Session* object is passed around to objects which need any of its values.

Certain user information is used throughout a system. Some of this information is security related, such as the user's role and privileges. A *Session* object is a good way for sharing this global information. This object can be passed around and used as needed.

Depending on the structure of the class hierarchy, an instance variable for the *Session* could be added to a superclass common to every class that needs the *Session*. Many times, especially when extending and building on existing frameworks, the common superclass approach will not work, unless of course you want to extend object which is usually not considered a good design. Thus, usually an instance variable needs to be added to every class that needs access to the *Session*.

All of the objects that share the same Session have a common scope. This scope is like the environments used by a compiler to perform variable lookups. The principle differences are that the

Session's scope was created by the application and that lookups are performed at runtime by the application.

Since many objects hold a reference to the *Session*, it is a great place to put the current *State* [GHJV 95] of the application. The *State* pattern does not have to be implemented inside of the *Session* for general security purposes, however. *Limited View* data and *Roles* can also be cached in a *Session*. It is important to note that the user should not be allowed to access any security data that may be held within a *Session* such as passwords and privileges. It can be a good idea to structure any application with a *Session* object. This object holds onto any shared information that is needed while a user is interacting with the application.

Example:

A *Session* can be used to store many different kinds of information in addition to security data. The Caterpillar/NCSA Financial Model Framework has a FMState class [Yoder]. An FMState object serves as a *Session*. It provides a single location for application components to access a *Limited View* of the data, the current products that can be selected, the user's *Role*, and the state of the system. Most of the classes in the Financial Model keep a reference to an FMState. A true *Singleton* could not be used because a user can open multiple sessions with different selection criteria, each yielding a different *Limited View*.

Figure 4 Shows FMState from the Financial Model. Security info includes username and role. The security info and selection criteria define the limited views. Each ReportView and ReportModel has a reference back to the FMState so it can access other data.

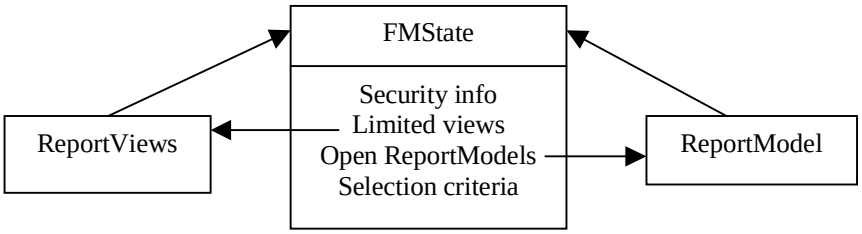


Figure 4 - The Financial Model's FMState

Consequences:

- ✔ The *Session* object provides a common interface for all components to access important variables.
- ✔ Instead of passing many values around the application separately, a single *Session* object can be passed around.
- ✔ Whenever a new shared variable or object is needed, it can be put in the *Session* object and then all components that have access to the object will have access to it.
- ✔ Change propagation is simplified because each object in a thread or process is dependent on only a single, shared *Session* object.
- ✘ While an object may not need a *Session*, it may later create an object that needs the *Session*. When this is the case, the first object must still keep a reference to the *Session* so it can pass it to the new object. Sometimes, it may seem as if every object has a *Session*. The proliferation of *Session* instance variables throughout the design is an unfortunate, but necessary, consequence of the *Session* pattern.

- ✗ Adding *Session* late in the development process can be difficult. Every reference to a *Singleton* must be changed. The authors have experience retrofitting *Session* in place of *Singleton* and can attest that this can very tedious when *Singletons* are spread among several classes. This is also true when trying to consolidate many global variables that were being passed around as parameters into a *Session*.
- ✗ When many values are stored in the *Session*, it will need some organizational structure. While some organization may make it possible to breakdown a *Session* to reduce coupling, splitting the session requires a detailed analysis of which components need which subsets of values.

Related Patterns:

- *Session* is an alternative to a *Singleton* [GHJV 95] in a multi-threaded, multi-user, or distributed environment.
- *Single Access Point* validates a user through *Check Point*. It gets a *Session* in return if the user validation is acceptable.
- A *Session* is a convenient place to implement the *State* pattern because the state is needed throughout the application.
- A Session can keep track of the users *Role* and possibly cache *Limited View* data.
- Lea's *Sessions* [Lea 95] discusses the beginning, middle, and end actions performed during resource management. This paper's *Session* pattern, on the other hand, focuses on separating data that cannot go in a *Singleton* because of a shared environment.
- The Thread Specific Storage Pattern [] allows multiple threads to use one logically global access point to retrieve thread-specific data without incurring locking overhead for each access

Known Use:

- For VisualWorks, the Lens framework for Oracle and GemBuilder for GemStone have OracleSession and GbsSession classes respectively. Each keeps information such as the transaction state and the database connection. The *Sessions* are then referenced by any object within the same database context.
- The Caterpillar/NCSA Financial Model Framework has a FMState class [Yoder]. An FMState object serves as a *Session*, while keeping a *Limited View* of the data, the current product/family selection, and the state of the system. Most of the classes in the Financial Model keep a reference to an FMState.
- The PLoP '98 registration program [Yoder & Manolescu 98] has a *Session* object that keeps track of the user's global information as they are accessing the application.
- Most databases use a *Session* for keeping track of user information.
- VisualWave [OS 95] has a *Session* for its httpd services, which keeps track of any web requests made to it.
- UNIX ftp and telnet services use a *Session* for keeping track of requests and restricting user actions.

Non-Security Known Uses:

- VisualWorks has projects that can be used to separate two or more change sets. While information about window placement is also stored in each project, image code is shared among all of the projects. So, projects could be considered non-secured *Session*.

Full View With Errors

Alias:

Full View With Exceptions
Reveal All and Handle Exceptions
Notified View

Motivation:

Once an officer is allowed on a military base, he or she could go to any building on the base. In effect, the officer has a full view of the buildings on the base. If the officer tries to enter a restricted area without proper clearance, either someone would stop and check them noting that they are not allowed in the restricted area, or alarms would sound and guards would show up to arrest the officer.

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 giving the user a complete view to what all users have access to can make teaching how to use the system easier and can make for more generic GUIs.

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 are not allowed to 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 errors, permission denied, and illegal operation messages.

Solution:

Design the application so users see everything that they might have access to. When a user tries to perform an operation, check if it is valid. Notify them with an error message when they perform illegal operations.

This pattern is very useful when a user can perform almost any operation. It is easier to show the user everything and just provide an error message when an illegal operation is attempted.

The solution for this pattern is simple when only a few error message need to be displayed. Just display the error message to standard error or in a dialog box. If many error messages are spread throughout the application, a separate error reporting mechanism may be useful. This mechanism could also be used for error logging.

Typically, an error-reporting framework would have two principal components. The log event object has a message describing the error condition and a severity level indicating if the event is a warning, an error, or just user information. When a log event is created it can automatically register itself with the logger. The logger is a *Singleton* that automatically receives and processes log events. The logger can be configured to display dialogs or write to a file depending on the severity of the event.

Example:

One example is an Oracle database. When you are using SQLPlus to access the data, you can execute any command. However, if you try to access data you don't have permission to see, an appropriate error message will be displayed.

Consequences:

- ✔ Training materials for the application are consistent for each type of user.
- ✔ Retrofitting this pattern into an existing system is straightforward. Just write a GUI that will handle all options and whenever a problem happens with an operation, simply exit the operation and open an error dialog.
- ✔ It is easier to dynamically change privileges on an operation because authorization is performed when the operation is attempted.
- ✔ It is easier to implement since you don't have to have multiple views on the data.
- ✘ Users may get confused with a constant barrage of error dialogs.
- ✘ Operation validation can be more difficult when users can perform any operation.
- ✘ Users will get frustrated when they see options that they cannot perform.

Related Patterns:

- *Limted View* is a competitor to this pattern. If limiting the view completely is not possible, this pattern can fill in the holes. The "Putting It All Together" section at the end of this paper compares these two patterns and describes when to use each.
- *Checks* [Cunningham 95] describes many details on implementing GUI's and where to put the error notifications.
- *Roles* will be used for the error notification or validating what the user can and can not do.

Known Uses:

- Login windows inform users when they enter incorrect passwords.
- SQLPlus, used to access an Oracle database, allows you to execute any and displays an appropriate error message if illegal access 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.
- Reuters SSL Developers Kit has a framework for reporting error, warning, and information events. It can be configured to report errors to standard error, a file, or a system dependent location such as a dialog.

Non-security Known Uses:

- Assertions [Meyer 92] are a form of this pattern used within a programming interface. Since a class' interface cannot change, preconditions ensure that each method is used within a legal context.
- The Java programming model of throwing exceptions follows this pattern. Instead of designing methods to only be used in the proper context, exceptions are thrown forcing the rest of the application to deal with the error.

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.

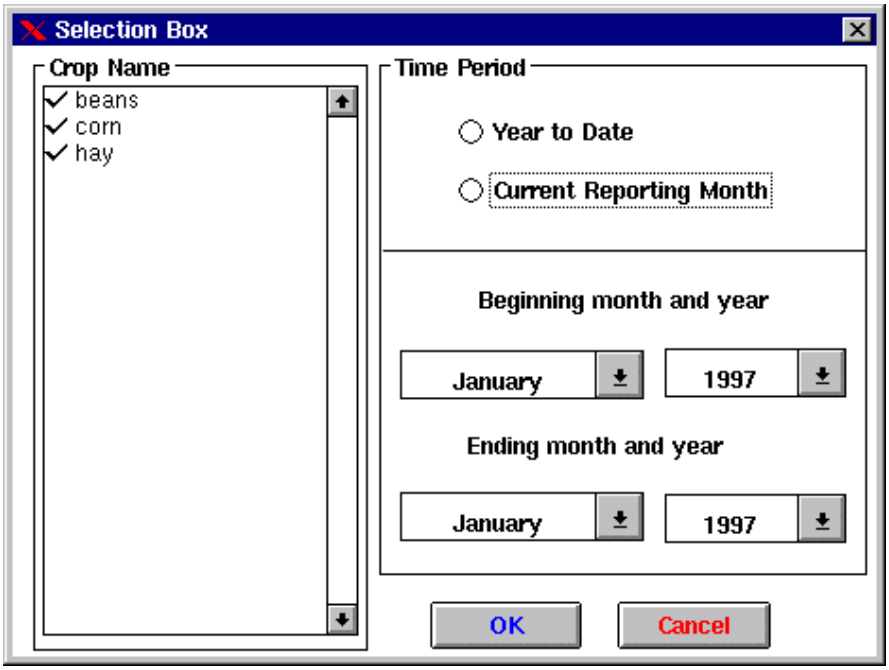


Figure 5 - Limited View on Selection

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.

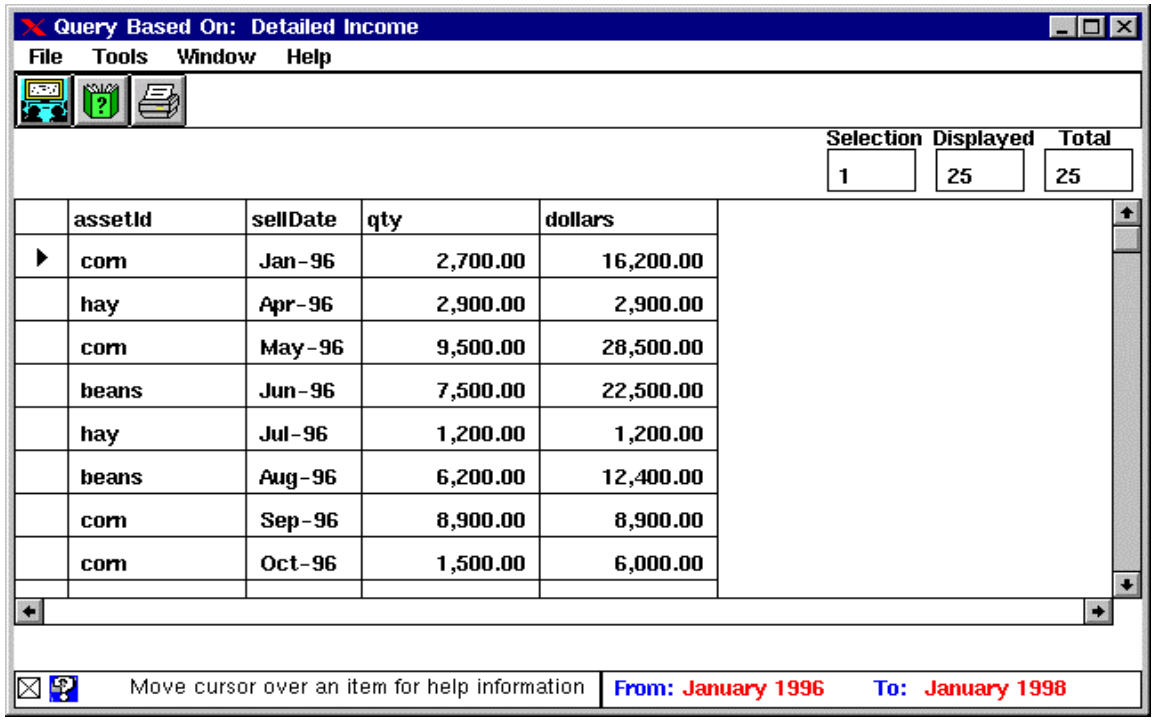


Figure 6 - Limited View on non-editable transactions

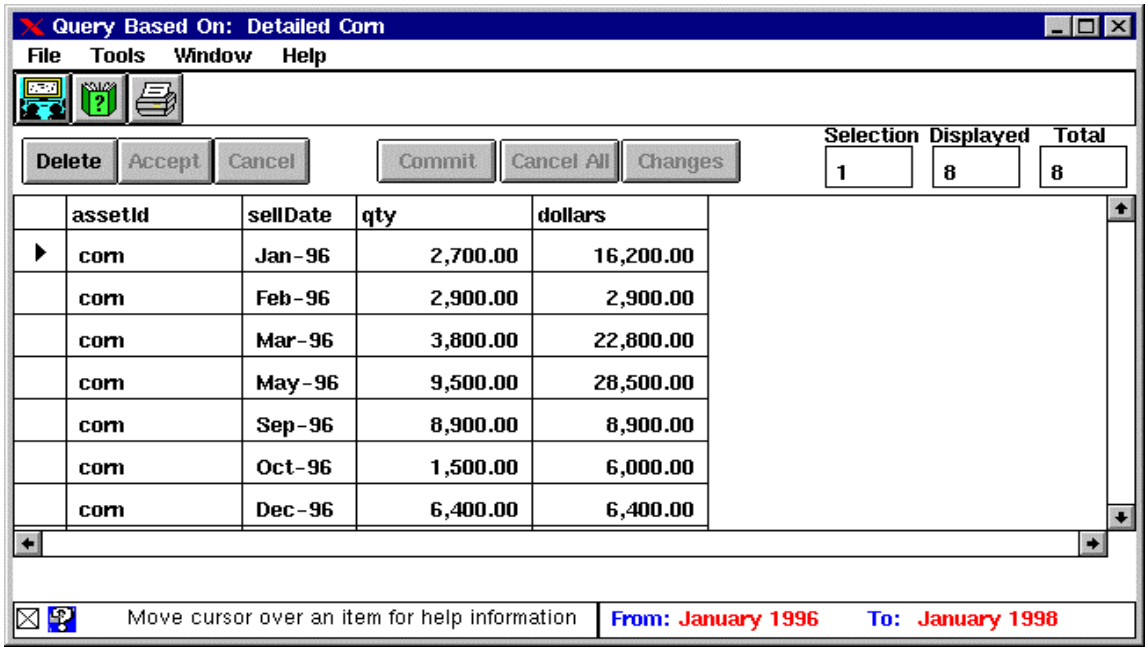


Figure 7 - Limited View on editable transactions

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.

Secure Access Layer

Alias:

Using Low-level security
Using Non-application security
Only as strong as the weakest link

Motivation:

When secure documents are transferred from one secure area to another in the military base, it is important that the security of the documents is not violated during the transfer. If the document is being transferred via a computer disk, the data could be encrypted and then locked in a briefcase and handcuffed to the arm of the courier during transfer. This will provide an isolation layer to protect the secure information during the transfer.

Most applications tend to be integrated with many other systems. The places where system integration occurs can be the weakest security points and the most susceptible to break-ins. If the developer is forced to put checks into the application wherever the applications communicates with these systems, then the code will be very convoluted and abstraction will be difficult. An application that is built on an insecure foundation will be insecure. In other words, it doesn't do any good to bar your windows when you leave your back door is wide open.

Problem:

Application security will be insecure if it is not properly integrated with the security of the external systems it uses.

Forces:

- Application development should not have to be developed with operating system, networking, and database specifics in mind. These can change over the life of an application.
- Putting low-level security code throughout the whole application makes it difficult to debug, modify, and port to other systems.
- Even if the application is secure, a good hacker could find a way to intercept messages or go under the hood to access sensitive data.
- Interfacing with external security systems is sometimes difficult.
- An external system may not have sufficient security, and implementing the needed security may not be possible or feasible.

Solution:

Build your application security around existing operating system, networking, and database security mechanisms. If they do not exist, then build your own lower-level security mechanism. On top of the lower-level security, build a secure access layer for communicating in and out of the program.

Usually an application communicates with many pre-existing systems. For example, a financial application on a Windows NT client might use an Oracle database on a remote server. Given that most systems already provide a security interface, develop a layer in your application that encapsulates the interfaces for securely accessing these external systems. All communication between the application and the outside world will be routed through this secure layer.

The important point to this pattern is to build a layer to isolate the developer from change. This layer may have many different protocols depending upon the types of communications that need to be done. For example, this layer might have a protocol for accessing secure data in an Oracle database and another protocol for communicating securely with Netscape server through the Secure Sockets Layer (SSL) [Netscape]. The crux of this pattern is to componentize each of these external protocols so they can be more easily secured. The architecture for different *Secure Access Layers* could vary greatly. However, the components' organization and integration is beyond the scope of this pattern.

By creating a *Secure Access Layer* with a standard set of protocols for communicating with the outside world, an application developer can localize these external interfaces and focus primarily on applications development. Communicate in and out of the application will pass through the protocols provided by this layer.

This pattern assumes a convenient abstraction is possible. For example, VisualWorks' LensSession does not support Microsoft Access, so QueryDataManager cannot be used with a Microsoft Access database. *Secure Access Layer*, however, provides a location for a more general database abstraction. Third party drivers have been developed for ODBC that can communicate with Microsoft Access. By using the *Secure Access Layer*, it is easy to extend your application to use the ODBC protocol, thus allowing your application to communicate with any database that supports ODBC.

Example:

The PLoP registration program uses a *Secure Access Layer*. A layer was created where all communications is processed for registering through the web. This communications layer is positioned on top of Apache's Secure Socket Layer. This prevents any information from being sniffed during the entry of data such as credit card numbers. Also, a layer on the database side was also created to provide additional security by encrypting the credit card information in the database. The secure layer uses a key for encrypting and decrypting the data when needed. Thus, even if someone was able to access the database through some back door, the credit card data is still protected.

Consequences:

✔ A Secure Access Layer can help isolate where an application communicates with external security systems. Isolating secure access points make it easier to integrate new security components and upgrade existing ones.

✔ A *Secure Access Layer* can make an application more portable. If the application later needs to communicate with Sybase rather than Oracle, then the access to the database is localized and only needs to be changed in one place. *QueryObjects* [Brant & Yoder 96] uses this approach by having all accesses to the database go through the QueryDataManager, which is built on top of the LensSession [OS 95]. The LensSession can map to either Oracle or Sybase. Therefore the application developer does not need to be concerned with either choice or future changes.

✘ Different systems that your application may need to integrate with use different security protocols and schemes for accessing them. This can make it difficult to develop a *Secure Access Layer* that works for all integrated systems, and it also may cause the developer to keep track of information that many systems do not need.

✘ It can be very hard to retrofit a *Secure Access Layer* into an application which already has security access code spread throughout.

Related Patterns:

- *Secure Access Layer* is part of a layered architecture. *Layers* [BMRSS 96] discusses the details of building layered architectures.
- *Layered Architecture for Information Systems* [Fowlers 97-1] discusses implementation details that can be applied when developing layered systems.

Known Uses:

- Secure Shell [SSH] includes secure protocols for communicating in X11 sessions and can use RSA encryption through TCP/IP connections.
- SSL (Netscape Server) provides a *Secure Access Layer* that web clients can use for insuring secure communication.
- Oracle provides its own *Secure Access Layer* that applications can use for communicating with it.
- CORBA Security Services [OMG] specifies how to authenticate, administer, audit and maintain security throughout a CORBA distributed object system. Any CORBA application's *Secure Access Layer* would communicate with CORBA's Security Service.

- The Caterpillar/NCSA Financial Model Framework [Yoder] uses a *Secure Access Layer* provided by the LensSession in ParcPlace's VisualWorks Smalltalk [OS 95].
- The PLoP '98 registration program [Yoder & Manolescu 98] goes through a *Secure Layer* for access to the system. First the application runs on top of SSL from the Apache Server. Also, all credit card information is stored encrypted when users registered for PLoP '98.

Putting It All Together

Problems with Retrofitting

An important point to note is that it can be very hard to retrofit most of these patterns into an existing system. Specifically, *Secure Access Layer*, *Session*, and *Limited View*, can be very difficult to retrofit into a system that was developed without security in mind. If a *Single Access Point* is created up front, it is fairly straightforward to add *Check Point* later. Since *Roles* are used to define a *Session* and set up during *Check Point*, additional *Roles* can easily be added later. A dummy *Check Point* can start out as just a placeholder and the details of the corporate security policy can be added later. Also, if all outside requests are forwarded through some form of a *Secure Access Layer*, it will be easy to enhance and abstract the *Secure Access Layer* at a later point. Because of these problems, application developers should develop the primary architecture for implementing security from the start. It is also useful to get the security requirements as early in the design process as possible so pitfalls can be avoided.

Limited View verses Full View With Errors

Limited View and *Full View With Errors* are competing patterns. *Limited View* can simplify security checks by performing them at login. The rest of the application has a simpler design because it does not have to handle security exceptions throughout the code. It is also more user friendly because users do not get barraged with error dialogs. *Full View With Errors* is easier to implement. It is as simple as popping open an error dialog or printing to standard error. It also can be used when a security check cannot be performed up front because it is time-consuming or all necessary information is not yet known.

If an application has a small number of security checks to perform on user operations, *Full View With Errors*, provides the easiest solution. *Full View With Errors* is also good when most options are the same independent of the user's roles. If security exception handling is spread throughout the application, however, *Limited View* should be used. If some checks cannot be performed when creating the *Limited View*, they can still be handled individually with error messages. So while *Limited View* and *Full View With Errors* are competing patterns for individual security checks, they can both be used by an application to provide a "Limited View With Minimal Errors."

Working Together

Now that you have seen all of the patterns, you might be asking, "how do I fit it all together?" All of these patterns collaborate as an application security module and provide a mechanism for communicating with the outside world. Figure 8 is a map that shows how these patterns work together.

When a user logs into the system, *Single Access Point* takes the user's information. *Single Access Point* uses *Check Point,* which in turn interacts with the *Secure Access Layer*, to validate the user's information. After validating the user's information, *Check Point* looks up the users *Roles* and creates a *Session*. This *Session* has a reference to the *Role* for future use and can define the *Limited View* of the data along with any privileges the user has.

Check Point will be used by other application components when they need secure operations that can not be performed at startup. *Session* will be referenced by any part of the system that needs global information such as *Roles*, *State*, or a *Limited View* of the data. User interfaces, *Full View With Errors* or *Limited Views*, will be created using the *Session*.

Security checks can be performed during regular program operation by calling back to the *Check Point*, which will use the *Secure Access Layer,* if necessary.

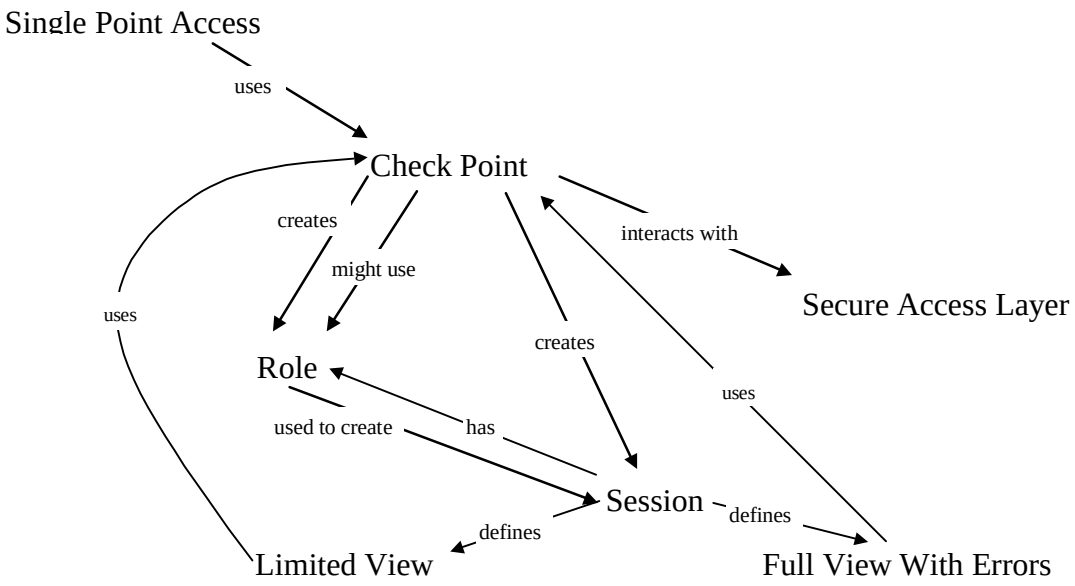


Figure 8 - Pattern Interaction Diagram

Figure 9 shows the steps for a typical scenario that are taken when a user successfully logs into the system. A login screen could be the *Single Access Point*. It initializes the system by going through *Check Point*. *Check Point* validates the user and creates a set of *Roles* for the user. User validation is done through the *Secure Access Layer*. *Roles* are then used to create the current *Session* that will be used by the application. This *Session* initializes itself with a *Limited View* based on the *Roles* and username passed in by *Check Point*. Any future requests will be forwarded through the *Limited View* and *Check Point* to the *Secure Access Layer*, which will return values to the *Limited View*, ensuring that a *Limited View* of the data is maintained. *Full View With Errors* will be used as need to compliment the Limited View.

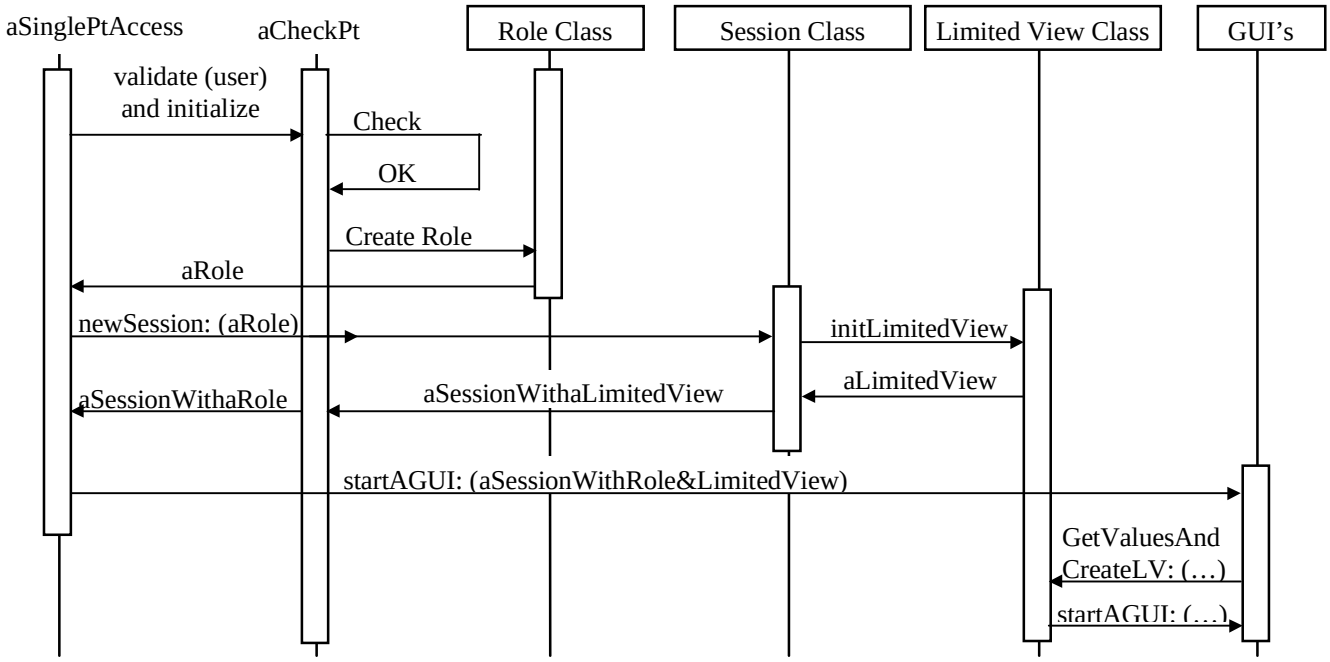


Figure 9 - Class Collaboration Diagram

Acknowledgments

We are grateful to the members of Professor Johnson's Patterns seminar: Federico Balaguer, John Brant, Ian Chai, Brian Foote, Alejandra Garrido, John (Zhijiang) Han, Peter Hatch, Ralph Johnson, Ral Lu, Lewis Muir, Dragos Manolescu, Eiji Nabika, James Overturf, Ed Peters, Chieko Shirai, Rick Kirchgesner, and Imad Zorob. Dirk Riehle and our shepherd, Eugene Wallingford, reviewed earlier drafts and provided valuable feedback. Federico Balaguer, Eric Evans, Martin Fowler, Robert Haugen, Joshua Kerievsky, Eiji Nabika, Tim Ottinger, Dirk Riehle, and Wolf Siberski, provided oustanding suggestions in our PLoP 97 writers' workshop. We are also grateful to our employers and coworkers at Caterpillar, Illinois Department of Public Health, National Computational Science Alliance (NCSA), Reuters Information Technology, and the University of Illinois for providing us the support and experience to write about and develop such systems.

References

[Barcalow 97] Jeffrey Barcalow. *Strategic Planning Support for a Financial Model Framework*, M.S. Thesis, University of Illinois at Urbana-Champaign, Department of Computer Science, 1997. URL: http://www.uiuc.edu/ph/www/j-yoder/papers/thesis/barcalow.html

[Beck 97] Kent Beck. *SMALLTALK Best Practice Patterns*, Prentice Hall PTR, Upper Saddle River, NJ, 1997.

[Brant & Yoder 96] John Brant and Joseph Yoder. "Reports," *Collected papers from the PLoP '96 and EuroPLoP '96 Conference*, Technical Report #wucs-97-07, Dept. of Computer Science, Washington University Department of Computer Science, February 1997. URL: http://www.cs.wustl.edu/~schmidt/PLoP-96/yoder.ps.gz.

[BMRSS 96] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal. *Pattern-Oriented Software Architecture: A System of Patterns*, John Wiley and Sons Ltd., Chichester, UK, 1996.

[Cunningham 95] Ward Cunningham. "Checks," *Pattern Languages of Program Design*, James O. Coplien and Douglas C. Schmidt, eds., Addison-Wesley, 1995

[Foote & Yoder 96] Brian Foote and Joseph Yoder. "Evolution, Architecture, and Metamorphosis," *Pattern Languages of Program Design 2*, John M. Vlissides, James O. Coplien, and Norman L. Kerth, eds., Addison-Wesley, Reading, MA., 1996.

[Foote & Yoder 98] Brian Foote and Joseph Yoder. "Metadata and Active Object-Models *Collected papers from the PLoP '98 and EuroPLoP '98 Conference*, Technical Report #wucs-98-25, Dept. of Computer Science, Washington University Department of Computer Science, September 1998. URL: http://jerry.cs.uiuc.edu/~plop/plop98/final_submissions/

[Fowler 97-1] Martin Fowler. *Analysis Patterns: Reusable Object Models*, Addison Wesley, 1997.

[Fowler 97-2] Martin Fowler. "Dealing with Roles" *Collected papers from the PLoP '97 and EuroPLoP '97 Conference*, Technical Report #wucs-97-34, Dept. of Computer Science, Washington University Department of Computer Science, September 1997. URL: http://www2.awl.com/cseng/titles/0-201-89542-0/apsupp/roles2-1.html.

[GHJV 95] Eric Gamma, Richard Helm, Ralph Johnson, John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, MA, 1995.

[GemStone 96] Gemstone Systems, Inc. *GemBuilder for VisualWorks, Version 5*. July 1996. URL: http://www.gemstone.com/.

[ICSP] *International Cryptographic Software Pages*. URL: http://www.cs.hut.fi/ssh/crypto/.

[Johnson & Woolf 97] Ralph Johnson and Bobby Woolf. "Type Object," *Pattern Languages of Program Design 3*, Robert Martin, Dirk Riehle, and Frank Buschmann, eds., Addison-Wesley, Reading, MA., 1997.

[Lea 95] Doug Lea. ``Sessions: A design pattern", ECOOP Workshop on Patterns. August 1995.

[Lea 97] Doug Lea. *Concurrent Programming in Java*, Addison-Wesley, Reading, MA, 1997.

[Meyer 92] Bertrand Meyer. ``Applying `Design by Contract'," *IEEE Computer*. October 1992. pp 40-51.

[NCSA] NCSA HTTPd Development Team. *Mosaic User Authentication Tutorial*. URL: http://hoohoo.ncsa.uiuc.edu/docs-1.5/tutorials/user.html.

[OMG] Object Management Group. *Security, Transactions, … and More*. URL: http://www.omg.org/corba/sectrans.htm#sec.

[OS 95] ObjectShare, Inc. *VisualWorks User's Guide*. 1995. URL: http://www.objectshare.com/doc/vw/Default.htm.

[Schmidt, Price, & Harrison 97] Douglas C. Schmidt, Nat Pryce, and Timothy H. Harrison. "Thread-Specific Storage for C/C++ - An Object Behavioral Pattern for Accessing per-Thread State Efficiently" *Collected papers from the PLoP '97 and EuroPLoP '97 Conference*, Technical Report #wucs-97-34, Dept. of Computer Science, Washington University Department of Computer Science, September 1997. URL: http://www.cs.wustl.edu/~schmidt/TSS-pattern.ps.gz.

[Schneier 95] Bruce Schneier. *Applied Cryptography, 2nd edition.* John Wiley & Sons, 1995.

[SSH] *SSH (Secure Shell) Home Page.* URL: http://www.cs.hut.fi/ssh/.

[SSL] *The SSL Protocol*. Netscape Communications, Inc., URL: http://home.netscape.com/newsref/std/SSL.html.

[Woolf 97] Bobby Woolf. "Null Object," *Pattern Languages of Program Design 3*, Robert Martin, Dirk Riehle, and Frank Buschmann, eds., Addison-Wesley, Reading, MA., 1997.

[Yoder] Joseph Yoder. *A Framework to Build Financial Models*. URL: http://www.uiuc.edu/ph/www/j-yoder/financial_framework.

[Yoder & Manolescu 98] Joseph Yoder and Dragos Manolescu. *The PLoP Registration Framework*, 1998. URL: http://www.uiuc.edu/ph/www/j-yoder/Research/PLoP.