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
jeff.barcalow@reuters.com

Abstract

When the time arrives to deploy an application that needs security, it becomes apparent that adding security is much harder than just adding a password protected login screen. This paper contains a collection of patterns for application security. Six patterns are presented in this paper: 1) *Secure Access Layer*, 2) *Single Access Point,* 3) *Check Point*, 4) *Roles*, 5) *Session*, and 6) *Limited View*. *Secure Access Layer* provides a communication interface for developers and provides a means for applications to use the security of the systems on which they are built. *Single Access Point* permits entry into the application through one single point. *Check Point* gives the developer a way to handle different types of security breaches without being too harsh on users who are just making mistakes. Groups of users have different *Roles* that define what they can and can not do. The global information about the user is distributed throughout the application via a *Session*. Finally, users are only presented with legal options through a *Limited View*. These six patterns work together to provide a security framework for building applications.

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. In these cases, security is usually the last thing an application programmer is worried 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. People have been writing patterns for quite a few years now and to the surprise of the authors, we could not find anything written about the patterns that arise when addressing security issues.

In corporate environments where security is a priority, detailed security documents are written describing physical, operating system, network, and application security. The six patterns presented in this paper can be applied when developing security for application component of the security equation. 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 basically developed (The Caterpillar/NCSA Financial Model Framework [Yoder]). From this experience, we found many patterns that are used in developing security systems and outline them here.

Users should not be allowed 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. *Check Point* handles different types of security breaches while making the punishment appropriate for the security violation. Groups of users have different *Roles* that define what they can and can not do. The global information about the user is distributed throughout the application via a *Session*. Users are only presented with legal options through a *Limited View*. Finally, any application should have a *Secure Access Layer* for communicating with external systems securely. This layer helps keep the application's code independent of the external interfaces.

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	Problem
Single Access Point	Providing a security module and a way to log into the system.
Check Point	Organizing security checks and their repercussions.
Roles	Organizing users with similar security privileges.
Session	Localizing global information in a multi-user environment.
Limited View	Allowing users to only see what they have access to.
Secure Access Layer	Integrating application security with low level security.

Table 1 - Pattern Catalog

This paper does not discuss the patterns or issues dealing with low-level security on which our *Secure Access Layer* pattern is built. The low-level security patterns not discussed would 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 the Applied Cryptography book [Schneier] are very good references for more details on these issues.

Single Access Point

Alias:

Login Window
One Way In

Motivation:

Providing security for an application that communicates with networking, operating systems, databases, and other infrastructure systems can be very complex. The application will need a way to log a user into the system and a means for integrating with all of the available security modules from these systems. 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.

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 to use in different environments.
- An application may actually be a composite of several applications that all need to be secure.
- Different login windows or procedures could have duplicate code.
- Multiple entry points to an application can be customized to only collect information needed at that entry point.
- A single entry point may need to collect all of the user information that is needed for the entire application. This information will have to be kept in a global location.

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 a login screen for collecting basic information about the user, such as username, password, and possibly some configuration settings. *Check Point* is then used for verifying this information, and a *Session* is then created based upon the configuration settings and the user's access privileges. When opening up any sub-applications the requests are forwarded through the *Check Point* for handling any problems.

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.

Consequences:

One advantage of *Single Access Point* is that it 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. *Single Access Point* is only as secure as any steps leading to it.

The application cannot have multiple entry points.

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.*

Known Uses:

- UNIX and Windows NT use *Single Access Point* for logging into the system. These systems also create the necessary *Roles* for the current *Session*. Windows 95, however, does not use *Single Access Point*, because it allows any user to override the login screen and have access to all files on the system.
- 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 an FMLogin class which provides both *Single Access Point* and *Check Point*.
- *Single Access Point* has many non-security parallels:
  - Applications that launch only one way, ensuring a correct initial state.
  - Single creational methods provide for only one way to create a class. For example, Points in VisualWorks Smalltalk 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.
  - *Constructor Parameter Method* [Beck 97] initializes all instance variables through a single method.
  - 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
Holding off hackers
Three Strikes and You're Out
Proposition 184 – California's Three Strikes Rule
Above the Legal Limit
Validation and Retribution
Authentication and Penalization
Make the Punishment Fit the Crime

Motivation:

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*, the checks must be made to verify that a user is permissible. Unfortunately, these could make getting into the application more difficult for legitimate users. A user's most 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 easily become complicated and difficult to manage, so they need to be organized

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:

- 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.
- Having lots of error checking code all over 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 also decides the severity of the violation and what action needs to be taken when this violation happens.

The implementation of *Check Point* combines several patterns. *Single Access Point* is used to ensure that security checks are performed correctly and that no security checks are skipped. 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.

A *Check Point* implementation typically has three parts. The first part is the algorithm. It performs security checks one at a time, only proceeding to the next check when the current check has passed.

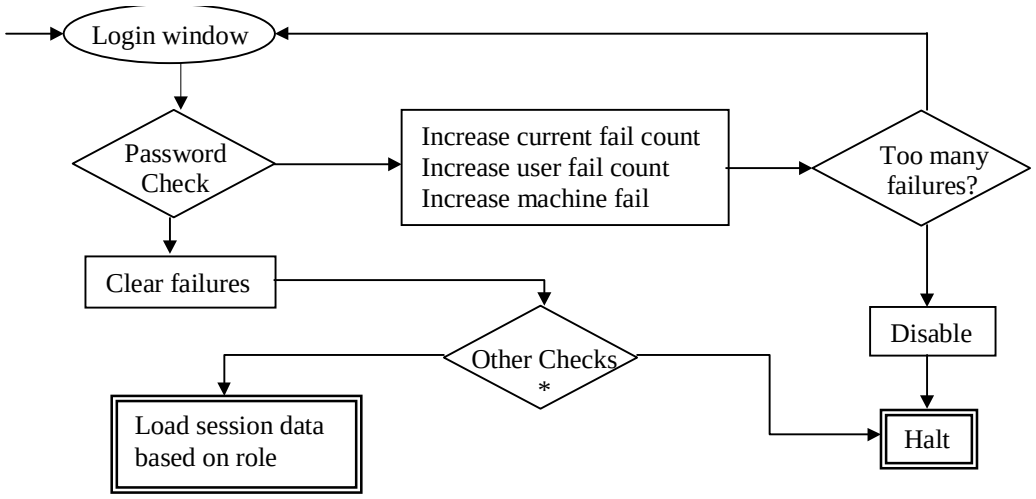
The second part includes the failure actions. Depending on the security violation or error, different types of actions may be taken. So, failure actions can be broken down by level of severity. The simplest action is to return a warning or error message to the user. If the error is non-critical, security checking could be continued or could be restarted. The second 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. This disabling feature could be implemented by having a secured database table or file that indicates if each machine and username is active. Normally, an administrator would 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 so they can automatically be turned back on 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 occurs in a row. 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. The counters can be used to parameterize the algorithm.

While it may be desirable to try to make a reusable security module, that goal is difficult when security requirements vary between implementations within a framework. It might be possible to create a library of pluggable security components and a framework for incorporating some of the basic security requirements into your application. However, the algorithm itself will almost always be overridden and it will need to be so flexible that it is hard for the authors to see a way to generalize this. The best approach might be to create configuration options that allow small parts of the algorithm to be turned on and off.

Example:

Figure 1 summarizes the *Check Point* algorithm 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,



* 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?

Figure 1 – An Example of a Check Point Algorithm

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. Password verification is performed by the database through the *Secure Access Layer*. Once a correct password has been entered, several other checks are applied because while a login might be allowed for 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.

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 more pluggable.

Some security checks cannot be performed at startup, so *Check Point* must be accessible from those 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.

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.
- The Caterpillar/NCSA Financial Model Framework [Yoder] uses *Check Point* to check passwords, *Roles*, and machines. The example above summarizes its implementation.
- Xauth provides *Check Point* that X-windows applications can use for securely communicating clients to servers. This is done through the use of a cookie.

Roles

Alias:

Groups
Projects
Profiles
Jobs
User Types

Motivation:

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, but some profiles are similar. The user base is large enough or the security profiles are complex enough that managing user-privilege relationships has 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 is n-n, making it difficult to manage.

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. While these two new relationships are still n-n, selecting appropriate *Roles* can reduce the total number of relationships.

Sometimes, a subset of the original user-privilege relationship also must be maintained to allow each user to have private privileges. *Roles* should only be used when the extra level of indirection provides a conceptual or manageability advantage over the direct user-privilege relationship.

Introducing *Roles* creates two new relationships that must be managed: user-role and role-privilege. Sometimes, a subset of the original user-privilege relationship must also be maintained to allow each user to have private privileges. *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 to which they have access. 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 avoidable. The preferred implementation, defining privileges within the role objects, is an example of the *Limited View* pattern (described latter in this paper).

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 that many *Roles* are created for all of the possibilities. 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.

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 sub-role types. With a role hierarchy, role-security checks are almost identical to type checking interfaces. A user would be the variable to be type-checked. The user's roles would be the interfaces for the object. The security privileges are analogous to the methods and instances variable accesses which have to be type-checked.

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, but they add an extra layer of complexity for developers.

Even if *Roles* are used, a username-privilege relationship is still needed to make information private to one user.

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.

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.
- *Roles* could be used to determine the scope of a *Limited View.*
- *Roles* could be used to select a *Strategy*.

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.
- 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 frequent, unrequested help (beginner). This configuration is the current user's role.

Session

Alias:

Localized Globals
Stuff that everyone should know

Motivation:

When an application needs to keep one copy of some information around, it could use 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 some situations a true *Singleton* may be needed. In other 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 environment, the single global location 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 keeps code clean and straightforward.
- Each object may only need access to some of the shared values.
- Which values are shared could change over time.
- If multiple applications are run simultaneously, they should not share the same values.
- Object creational interfaces are complicated when each object requires a set of shared values.
- While an object may not need certain values, it may later create an object that needs those values.

Solution:

Create a Session object, which holds all of the variables that need to be shared by many objects. 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. Then, 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. It is important to note that the user should not be allowed access to any secure data that may be held within a *Session* such as passwords and protections.

Consequence

The *Session* object provides a common interface for all components to access important variables.

One problem with this solution is that even though an object may not need a *Session*, the object 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 several *Singletons* are spread among several classes.

When many values are stored in the S*ession*, 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.
- Two unrelated patterns are *Type-Safe Session* [Pryce 97] and *Sessions* [Lea]. *Type-Safe Session* concentrates on client/server communication channels while Lea's *Sessions* discusses the beginning, middle, and end actions performed on resources. This paper's *Session* pattern, on the other hand, focuses on separating data that cannot go in a *Singleton* because of a shared environment.

Known Uses:

- 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.
- 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 *sessions*.

Limited View

Alias:

Blinders
Invisible Road Blocks
Hiding the cookie jars

Motivation:

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, conditional code can be ignored.

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 two ways. First, a *Limited View* configures which selections 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 information including the *Roles*, applies this with the current state of the application, and dynamically builds a GUI that limits the view based upon these attributes.

The first approach allows users to see different lists and data values depending upon their current *role*. This 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 limit 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 user goes to select the desired products available, the manager cannot get an "access denied" error.

When using the *Limited View* inside a *Session* with *Role* information, it also limits the view based upon the current state of the application. 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 disable, but they could be dynamically disabled 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, the user is knows of 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.

There are many possible implementations to Limited Views. GUIs can be dynamically created through *Composites* and *Builders* [GHJV 95]. Alternatively, the *State* pattern can be used by creating different classes that represent the different Limited Views. A *Strategy* can be used for choosing the appropriate *State*.

Example:

One example of a *Limited View* can be seen in the Selection Box example in Figure 2. Here, the user is only provided 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 they are allowed to use.

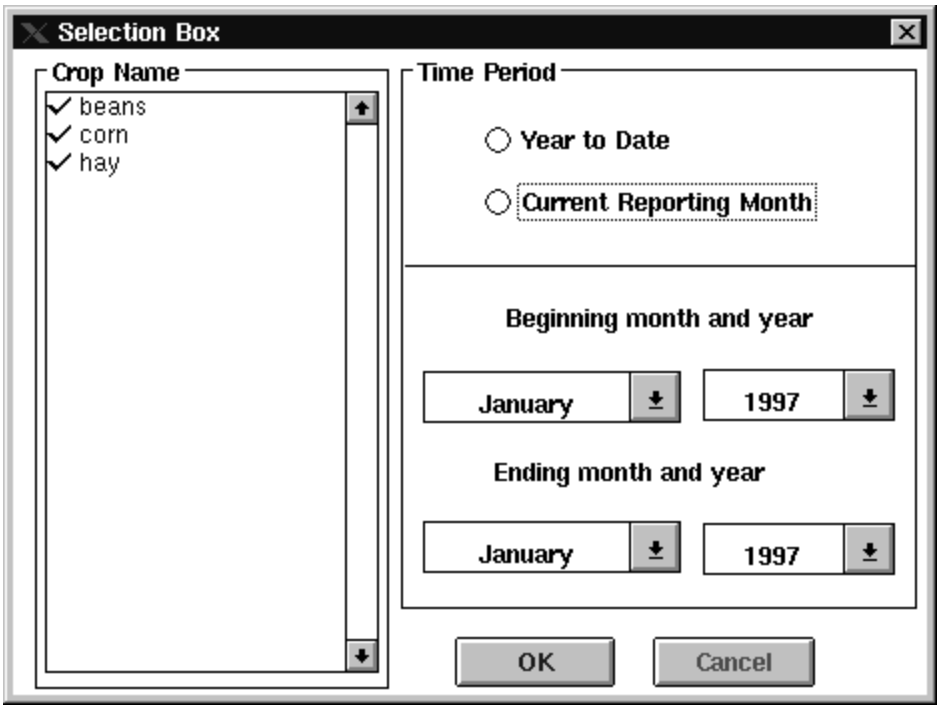


Figure 2 - Limited View on Selection

Another example can be seen in Figure 3 and Figure 4. Note that Figure 3 does not show a button for changing the values in the database, while in Figure 4 has that option. The view limits the ability for editing in Figure 3 since the user does not have the authority for editing the detailed income. In Figure 4, 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.

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.

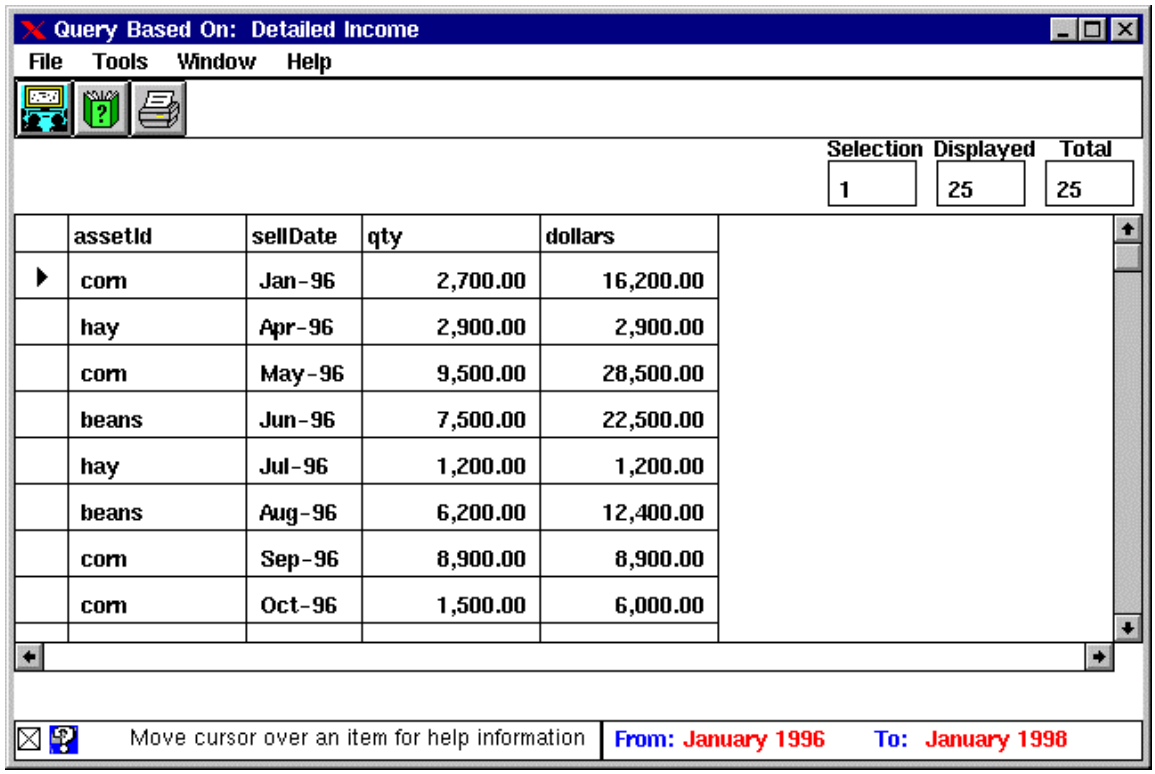


Figure 3 - Limited View on non-editable transactions

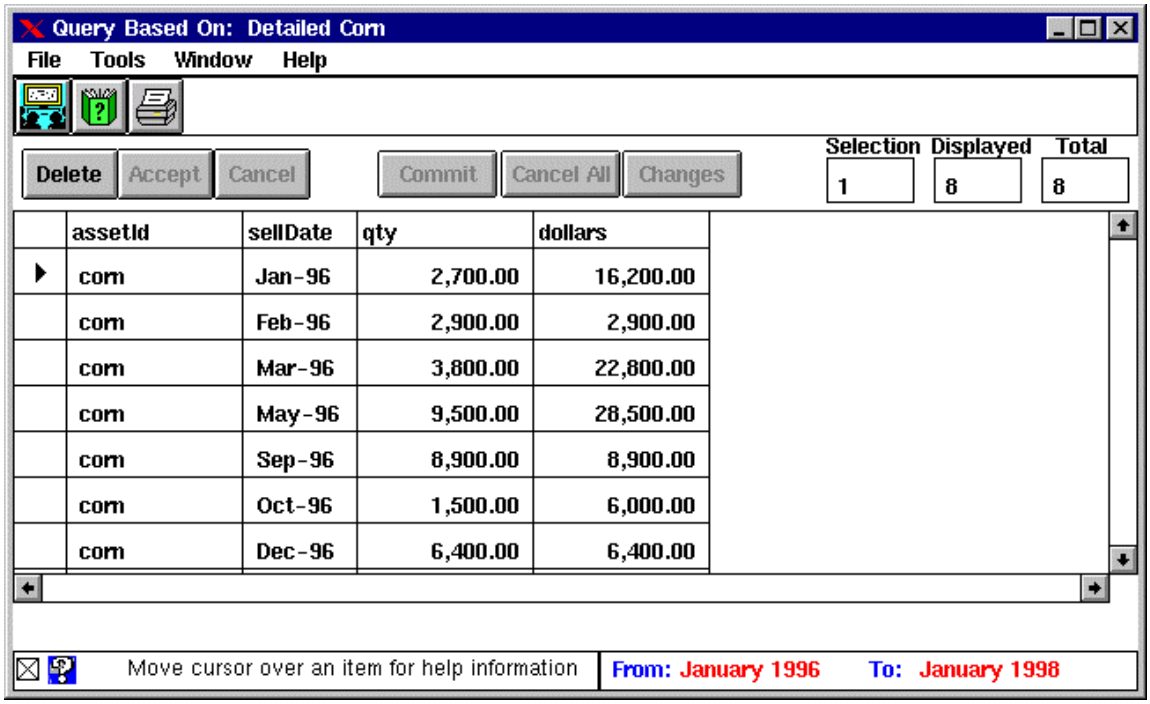


Figure 4 - Limited View on editable transactions

Consequences:

By only allowing the user to see and edit what they can access, the developer doesn't have to worry about verifying the legitimacy of the data the user is attempting to access. The user will only be permitted to select items that they can view. 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.

One problem with this pattern is that it can be frustrating for users to see 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 will be wondering if something is wrong with the application or why the data isn't available.

Similarly, if the user can view two products of data but only edit one, she or he may wonder why the editing options are not available when viewing both products at the same time.

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:

- 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*.

Known Uses:

- The Caterpillar/NCSA Financial Model Framework [Yoder] has a *Limited View* on the data by only allowing the user to see products for which they have access rights. 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.
- Hidden files and directories, which provided by most operating systems, are a form of a *Limited View*.
- Limited View has many non-security parallels. Netscape Communicator and Microsoft Office 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 or enabling buttons. For example, the menus available while editing charts 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.

Secure Access Layer

Alias:

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

Motivation:

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.
- 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 other 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.

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.

Consequences:

One advantage for using a *Secure Access Layer* is portability. 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 [PP 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.

On the other hand, this assumes a convenient abstraction is possible. 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. There have been third party drivers 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 to allow your application to communicate with any database that supports ODBC.

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] goes through a *Secure Access Layer* provided by the LensSession in ParcPlace's VisualWorks Smalltalk [PP 95].

Putting It All 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 5 is a map that shows how these patterns work together.

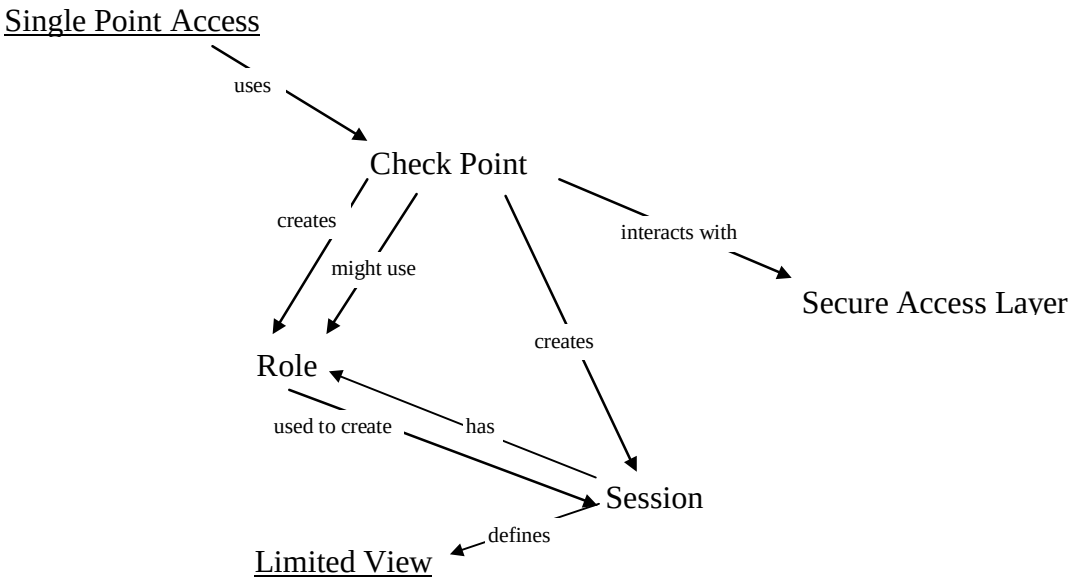


Figure 5 - Pattern Interaction Diagram

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 defines the *Limited View* of the data.

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 which needs *Roles*, state, or a *Limited View* of the data. User Interfaces with *Limited Views* will be created using the *Session*.

The example in Figure 6 shows the steps that are taken when a user successfully logs into the system. The login screen is where the *Single Access Point* usually happens. This single point of control initializes the system by going through *Check Point*. *Check Point* validates the user and creates a set of *Roles* for the user. The validation of the user is done through the *Secure Access Layer*. *Roles* are then passed to a *Session* class 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* to the *Secure Access Layer*, which will return values to the *Limited View* , ensuring that a *Limited View* of the data is maintained.

An important point to note is that it can be very hard to retrofit most of these patterns into an already developed 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, 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. 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 get the security requirements early in the design process so they can avoid the pitfalls.

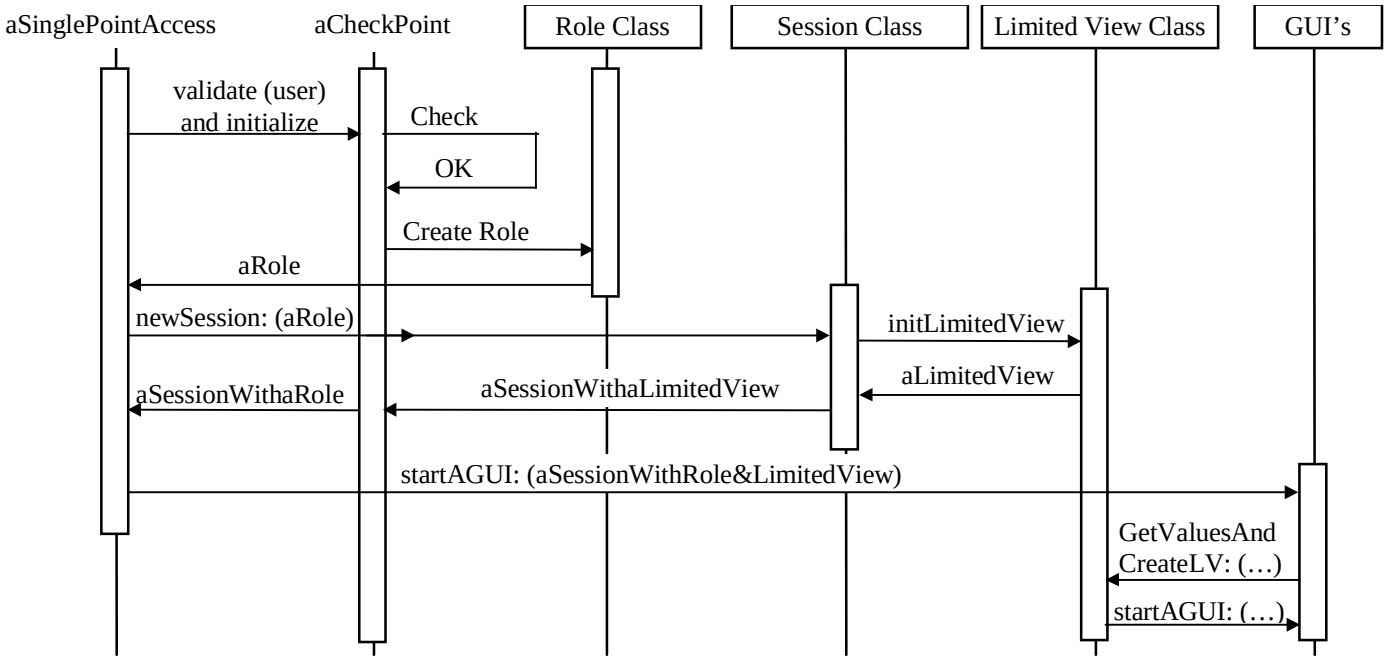


Figure 6 - Class Collaboration Diagram

Acknowledgments

We are grateful to the members of Professor Johnson's Patterns seminar: John Brant, Ian Chai, Brian Foote, Ralph Johnson, Lewis Muir, Dragos Manolescu, Eiji Nabika, and Ed Peters; Dirk Riehle and our shepherd, Eugene Wallingford, who reviewed earlier drafts and provided valuable feedback. We are also grateful to our employers, Caterpillar / NCSA and Reuturs Information Technology 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-cat.ncsa.uiuc.edu/~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.

[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.

[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.

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

[Fowler 97-2] Martin Fowler. *Dealing with Roles*, Submitted to PLoP' 97. URL: http://www.aw.com/cp/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/Products/gbs.htm.

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

[Lea] Doug Lea. *Sessions.* URL: http://gee.cs.oswego.edu/dl/pats/session.html.

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

[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.

[PP 95] ParcPlace Systems, Inc. *VisualWorks User's Guide*. 1995. URL: http://www.parcplace.com/products/vworks/info/vw25.htm.

[Pryce 97] Nat Pryce. *Type-Safe Session*, Submitted to EuroPLoP' 97. URL: http://siesta.cs.wustl.edu/~schmidt/europlop-97/workshops.html#ww2p1.

[Schneier] 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.

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