Given information about the current client stored in `currClientInfo`, the method requested by the client stored in `currMethodRequest`, a client method request would then be checked by the application as follows:

```
bool allowed = firstMethodChecker->accessAllowed(currClientInfo,
                                                 currMethodRequest,
                                                 reason,
                                                 reasonCode);
```

If `allowed` is true the client is allowed to execute the given method call on the XML-RPC server. If `allowed` is false the client is not allowed to execute the method and the reason why they are not allowed to execute the method is stored in `reason` and `reasonCode`.

3.4.9 Known Uses

Secure XML-RPC Server Library

3.5 Secure State Machine

3.5.1 Intent

The intent of the Secure State Machine pattern is to allow a clear separation between security mechanisms and user-level functionality by implementing the security and user-level functionality as two separate state machines.

3.5.2 Also Known As

Secure State

3.5.3 Motivation

Intermixing security functionality and typical user-level functionality in the implementation of a secure system can increase the complexity of both. The increased complexity makes it more difficult to test, review, and verify the security properties of the implementation, increasing the likelihood of introducing a vulnerability.

Also, a tight coupling between the security functionality and the user-level functionality makes it difficult to change and modify the system's security mechanisms.

3.5.4 Applicability

This pattern is applicable if

- the user-level functionality lends itself to implementation using the Gang of Four State pattern [Gamma 1995]; that is, the user-level functionality can be cleanly represented as a finite state machine
- the access control model for the state transition operations in the user-level functionality state machine can also be represented as a state machine. Note that in a degenerate case the access control model could be represented by a state machine with a single state.

3.5.5 Structure

Figure 12 depicts the structure of the Secure State Machine pattern.

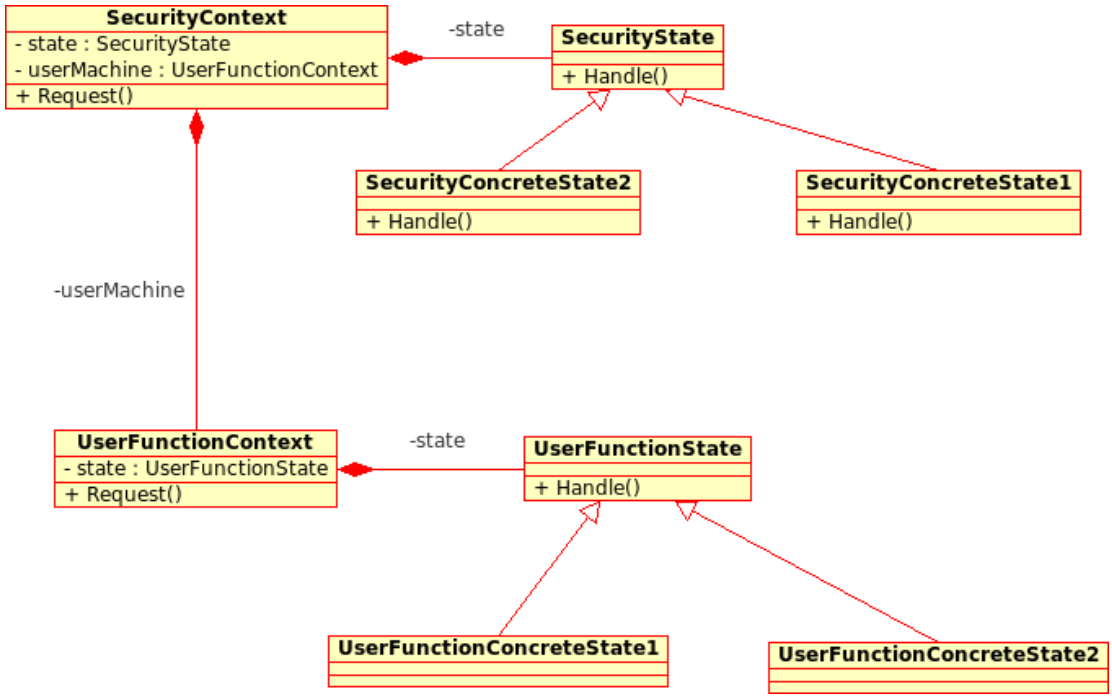

Figure 12: Secure State Machine Pattern Structure

3.5.6 Participants

These are the participants in the Secure State Machine pattern. (The class in the code presented in the Sample Code section corresponding to the listed participant appears in parentheses after the participant.)

- **SecurityContext** (ExampleSystem)
  - Defines the interface of interest to clients. All client operations are initially handled by an instance of SecurityContext.
  - As with the original Gang of Four State pattern [Gamma 1995], SecurityContext maintains an instance of a Security-State subclass that defines the current state from a security perspective.
  - Maintains an instance of UserFunctionContext; that is, the state machine implementing the non-security, user-level functionality.
  - Acts as a proxy for the instance of UserFunctionContext.
- **SecurityState** (SecurityState)
  - Defines an interface representing the possible operations handled by the security state machine. Note that UserFunctionState must share the same interface; that is, it must handle the same possible operations as SecurityState.
- **SecurityConcreteState** (NotLoggedIn, LoggedInAdmin, LoggedInClerk, Locked)
  - Each subclass of SecurityState implements the security state-dependent behavior for each operation.

The components of the user-level functionality state machine are exactly the same as those in the Gang of Four State pattern.

- **UserFunctionContext** (UserFunctionsMachine)
  - Defines all of the same operations as SecurityContext so that components of the security state machine can forward operation requests to the user-level functionality state machine when appropriate.
  - Has a private constructor to prevent outside access to the functionality of the user-level state machine. Only a SecurityContext can create a new UserFunctionContext.
- **UserFunctionState** (UserFunctionState)
  - Has the same interface as SecurityState.
- **UserFunctionConcreteState** (UserFunctionConcreteState1, UserFunctionConcreteState2)
  - In a manner similar to SecurityConcreteState, each subclass of SecurityState implements the user-level state-dependent behavior for each operation.

3.5.7 Consequences

In addition to the set of consequences associated with the general State pattern, the Secure State Machine pattern has these additional consequences:

- *It clearly separates security mechanisms from user-level functionality.* The use of this pattern requires that the security mechanisms be explicitly implemented in the security state machine and the user functionality of the system be explicitly implemented in the user-level functionality state machine. This makes it easy to
  - test and verify the security mechanisms separately from the user-level functionality. Because the security functionality is implemented separately from the user-level functionality, more rigorous testing and verification techniques can be applied to the security state machine than to the user-level functionality state machine.
  - change or replace the security mechanism. Because the security functionality is separate from the user-level functionality, a new security implementation could be implemented with less effort than would be required if the existing security mechanisms were interleaved with the user-level functionality.
- *It prevents programmatic access to the user-level functionality that avoids security.* Because only the security state machine can create an instance of the user-level functionality state machine, all interaction with the user-level functionality state machine must first pass through the security state machine, consequently defeating one class of programmatic attack.

3.5.8 Implementation

In addition to the implementation considerations associated with the Gang of Four State pattern [Gamma 1995], the Secure State Machine pattern has the following implementation consideration.

Who forwards operations on to the user-level state machine? The operations handled by the security state machine can be forwarded on to the user-level state machine by either the SecurityContext instance or the SecurityConcreteState instance.

- SecurityContext instance. The forwarding of operations to the user-level functionality state machine can be handled in the SecurityContext instance by defining the operation methods in SecurityState to return a boolean value indicating whether the operation should be forwarded. The corresponding operation methods in SecurityContext would then use this return

value to determine whether to forward the operation. This method is used in the example on this page. It is recommended over performing the forwarding in the SecurityConcreteState instance because it allows the user functionality state machine to be completely hidden within the SecurityContext instance.

- SecurityConcreteState instance. If the SecurityContext provides a method by which a SecurityConcreteState can access the user-level functionality state machine, the SecurityConcreteState can forward the operation directly.

3.5.9 Sample Code

This example of using the Secure State Machine pattern provides a skeleton of the code for implementing a system with the following behavior:

- A user must log in before using the system.
- If there are five failed login attempts, the user's account will be locked.
- Each user will be handled by a separate state machine. The allocation of users to state machines will be handled by some other portion of the system.
- The user-level functionality is abstractly represented as op1, op2, op3, login, and log-out.
- For security reasons, op3 may be performed only 50 times in a session. If op3 is performed more than 50 times, the user will be automatically logged out.
- Performing op2 requires that the user have the role of administrator. Everyone else has the role of clerk.

A collaboration diagram describing the basic behavior of the example code is shown in Figure 13.

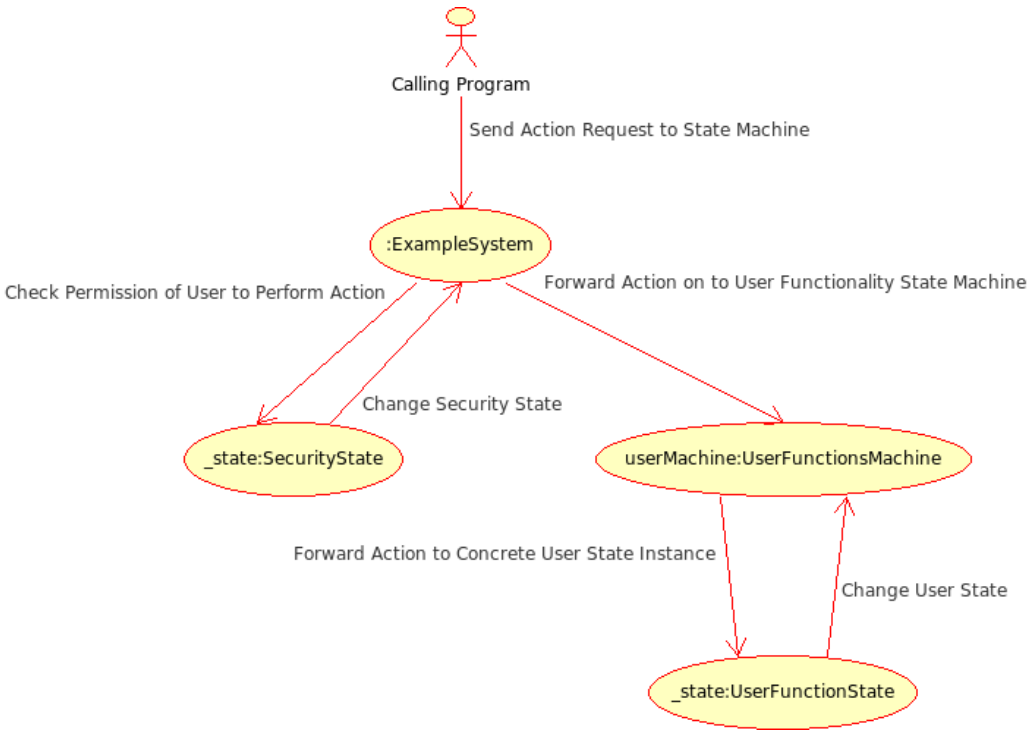

Figure 13: Secure State Machine Example Code Collaboration Diagram

This class represents the credentials of the user associated with a state machine. This class will not be sketched out in this example.

```
class UserCredentials;
```

This class represents a string that has been encrypted. This class will not be sketched out in this example.

```
class EncryptedString;
```

This is the forward declaration abstract class representing the states of the security state machine. This class will be sketched out in the example.

```
class SecurityState;
```

This class implements the security state machine for the system and acts as a proxy for the state machine that actually implements the user-level functionality.

```
class ExampleSystem {

public:

  // Create a new example system state machine for the given user.
  ExampleSystem(UserCredentials user);

  // Someone is trying to log onto the system as the current user.
  void login(EncryptedString password);

  // The user is logging out.
  void logout();

  // The user is attempting to perform one of the three user-level
  // system operations.
  void op1();
  void op2();
  void op3();

private:

  // Track the current state of the security state machine.
  SecurityState* _state;

  // Change the current state in the controller.
  void changeState(SecurityState*);

  // Let the security state machine change the current state in the
  // controller.
  friend class SecurityState;

  // Track the user associated with the security state machine.
  UserCredentials _user;

  // Get the user associated with the security state machine.
  const UserCredentials getUser();
```

```
  // Track the state machine that actually implements the user
  // functionality. The security state machine acts as a proxy for
  // the user functionality state machine.
  UserFunctionsMachine userMachine;
};
```

The methods of the SecurityContext are defined as follows.

```
ExampleSystem::ExampleSystem(UserCredentials user) {

  // Initially the user is logged out.
  _state = NotLoggedIn::instance(user);

  // We need to save the user we are dealing with.
  _user = user;

  // Create the user-level state machine for which we are a proxy.
  userSystem = UserFunctionsMachine(user);
}

void ExampleSystem::login(EncryptedString password) {
  // Forward the operation if appropriate.
  if (_state->login(this, password)) {
    userMachine->login(password);
  }
}

void ExampleSystem::logout() {
  // Forward the operation if appropriate.
  if (_state->logout(this)) {
    userMachine->logout();
  }
}

void ExampleSystem::op1() {
  // Forward the operation if appropriate.
  if (_state->op1(this)) {
    userMachine->op1();
  }
}

void ExampleSystem::op2() {
  // Forward the operation if appropriate.
  if (_state->op2(this)) {
    userMachine->op2();
  }
}

void ExampleSystem::op3() {
  // Forward the operation if appropriate.
  if (_state->op3(this)) {
    userMachine->op3();
  }
```

```
}
```

This is the declaration of the abstract class defining the interface for the state classes defining the states of the security state machine.

```
class SecurityState {

public:

  virtual bool login(ExampleSystem* controller, EncryptedString password);
  virtual bool logout(ExampleSystem* controller);
  virtual bool op1(ExampleSystem* controller);
  virtual bool op2(ExampleSystem* controller);
  virtual bool op3(ExampleSystem* controller);

protected:

  void changeState(ExampleSystem* controller, SecurityState* newState);
};
```

The security model for this example has four states:

- **NotLoggedIn**. The user is not logged in.
- **LoggedInAdmin**. The user is logged in as an administrator.
- **LoggedInClerk**. The user is logged in as a clerk.
- **Locked**. The user's account has been locked.

The default implementation of the security state methods is as follows. These statements should be redefined by the concrete state classes. In the default implementation, the operation is never forwarded on to the user-level functionality state machine.

```
bool SecurityState::login(ExampleSystem* controller,
                          EncryptedString password) { return false; }
bool SecurityState::logout(ExampleSystem* controller) { return false; }
bool SecurityState::op1(ExampleSystem* controller) { return false; }
bool SecurityState::op2(ExampleSystem* controller) { return false; }
bool SecurityState::op3(ExampleSystem* controller) { return false; }
```

`changeState()` is common to all concrete state classes.

```
void SecurityState::changeState(ExampleSystem* controller,
                                SecurityState* newState) {
  controller->changeState(newState);
}
```

Here is the definition of the concrete `NotLoggedIn` state class.

```
class NotLoggedIn : public SecurityState {

public:

  // Get an instance of this state for the current user. Each user
  // will have a single instance of each security state associated
  // with them. This ensures that each user will be associated with
```

```
  // one and only one security state machine.
  static SecurityState* instance(UserCredentials user);

  // When the user is not logged in, all they can do is try to log
  // in.
  virtual bool login(ExampleSystem* controller, EncryptedString password);

private:

  // This state will track the number of failed login attempts.
  unsigned int numFailedLogins;
};
```

Here are the method bodies of the `NotLoggedIn` state class.

Create a `NotLoggedIn` state. This initializes the number of failed login attempts.

```
NotLoggedIn::NotLoggedIn() {
  numFailedLogins = 0;
}
```

Handle a user login.

```
bool NotLoggedIn::login(ExampleSystem* controller, EncryptedString password)
{

  // Try to validate the user with the password.
  if (controller->getUser().validate(password)) {

    // The current user correctly entered their password.

    // Clear the bad password count.
    numFailedLogins = 0;

    // They user is now logged in. Choose the proper login state based
    // on the user's role.
    if (controller->getUser().isAdministrator()) {
      changeState(controller, LoggedInAdmin::instance());
    }
    else {
      changeState(controller, LoggedInClerk::instance());
    }

    // The user has now logged in. Handle the user functionality
    // associated with a login by passing the login operation on to
    // the user functionality machine.
    return true;
  }

  else {
    // The current user incorrectly entered their password.

    // Track the failed login.
    numFailedLogins++;
```

```
    // Has the user failed their login too many times.
    if (numFailedLogins >= 5) {

      // Reset the # of failed logins.
      numFailedLogins = 0;

      // Lock the user’s account.
      changeState(controller, Locked::instance());

      // Note that because the security state machine determined that
      // the security requirements were not met, the login operation
      // is not passed on to the user functionality machine.
      return false;
    }
  }
}
```

Here is the definition of the concrete `Locked` state class.

```
class Locked : public SecurityState {

public:

  static SecurityState* instance(UserCredentials user);

  // For this simple example, once a user's account is locked it
  // cannot be unlocked. Once the user's account is locked, they
  // cannot do anything. No operations are forwarded to the user
  // functionality machine.
};
```

Here is the definition of the concrete `LoggedInAdmin` state class

```
class LoggedInAdmin : public SecurityState {

public:

  static SecurityState* instance(UserCredentials user);
  LoggedInAdmin();

  // A logged-in administrator can perform all operations other than
  // logging in again.
  bool logout(ExampleSystem* controller);
  bool op1(ExampleSystem* controller);
  bool op2(ExampleSystem* controller);
  bool op3(ExampleSystem* controller);

private:

  // Keep track of the number of times the user has performed
  // op3.
  unsigned int op3Count;
};
```

Here are the method bodies of the `LoggedInAdmin` state class.

```
// Create a LoggedInAdmin state. This initializes the count of the
// number of times op3 was performed.
LoggedInAdmin::LoggedInAdmin() {
  op3Count = 0;
}

bool LoggedInAdmin::logout(ExampleSystem* controller) {

  // Just move to the logged out state.
  changeState(controller, NotLoggedIn::instance());

  // Handle user functionality actions for the logout operation.
  return true;
}

bool LoggedInAdmin::op1(ExampleSystem* controller) {
  // Based on the current state of the security machine we know that
  // this operation is valid. Forward it on to the user functionality
  // machine.
  return true;
}

bool LoggedInAdmin::op2(ExampleSystem* controller) {
  // Based on the current state of the security machine we know that
  // this operation is valid. Forward it on to the user functionality
  // machine.
  return true;
}

bool LoggedInAdmin::op3(ExampleSystem* controller) {

  // The user has done op3 one more time. Track it.
  op3Count++;

  // Has the user exceeded their quota of # of times they can do
  // op3?
  if (op3Count > 50) {

    // Reset the count of # of times they performed op3 during this
    // login session.
    op3Count = 0;

    // Log out the user. Note that this calls the controller's logout
    // method, which will result in both the security machine and the
    // user-level functionality machine handling the logout
    // operation.
    controller->logout();

    // Stop processing the op3 operation.
    return false;
  }
```

```
  // If we get here the security criteria for op3 have been
  // met. Forward op3 on to the user functionality machine.
  return true;
}
```

Here is the definition of the concrete `LoggedInClerk` state class.

```
class LoggedInClerk : public SecurityState {

public:

  static SecurityState* instance(UserCredentials user);
  LoggedInClerk();

  // A logged in clerk can perform all operations other than
  // logging in again and op2.
  bool logout(ExampleSystem* controller);
  bool op1(ExampleSystem* controller);
  bool op3(ExampleSystem* controller);

private:

  // Keep track of the number of times the user has performed
  // op3.
  unsigned int op3Count;
};
```

Here are the method bodies of the `LoggedInClerk` state class.

```
// Create a LoggedInClerk state. This initializes the count of the
// number of times op3 was performed.
LoggedInClerk::LoggedInClerk() {
  op3Count = 0;
}

bool LoggedInClerk::logout(ExampleSystem* controller) {

  // Just move to the logged out state.
  changeState(controller, NotLoggedIn::instance());

  // Handle user functionality actions for the logout operation.
  return true;
}

bool LoggedInClerk::op1(ExampleSystem* controller) {
  // Based on the current state of the security machine we know that
  // this operation is valid. Forward it on to the user functionality
  // machine.
  return true;
}

bool LoggedInClerk::op3(ExampleSystem* controller) {

  // The user has done op3 one more time. Track it.
```

```
  op3Count++;

  // Has the user exceeded their quota of # of times they can do
  // op3?
  if (op3Count > 50) {

    // Reset the count of # of times they performed op3 during this
    // login session.
    op3Count = 0;

    // Log out the user. Note that this calls the controller's logout
    // method, which will result in both the security machine and the
    // user-level functionality machine handling the logout
    // operation.
    controller->logout();

    // Stop processing the op3 operation.
    return false;
  }

  // If we get here the security criteria for op3 have been
  // met. Forward op3 on to the user functionality machine.
  return true;
}
```

This is the controller for the user-level functionality state machine. Note that only the security state machine can create an instance of the user-level functionality state machine.

```
class UserFunctionsMachine {

public:

  // Someone is trying to log onto the system as the current user.
  void login(EncryptedString password);

  // The user is logging out.
  void logout();

  // The user is attempting to perform one of the three user-level
  // system operations.
  void op1();
  void op2();
  void op3();

private:

  // Only the security state machine can create an instance of the
  // user-level functionality machine. This helps prevent direct
  // access to the user-level functionality machine.
  friend class ExampleSystem;

  // Create a new user functionality state machine for the given user.
  UserFunctionsMachine(UserCredentials user);
};
```

3.5.10 Known Uses

"Method and apparatus for secure context switching in a system including a processor and cached virtual memory" (United States Patent Application 20070260838).

3.6 Secure Visitor

3.6.1 Intent

Secure systems may need to perform various operations on hierarchically structured data where each node in the data hierarchy may have different access restrictions; that is, access to data in different nodes may be dependent on the role/credentials of the user accessing the data. The Secure Visitor pattern allows nodes to *lock* themselves against being read by a visitor unless the visitor supplies the proper credentials to *unlock* the node. The Secure Visitor is defined so that the only way to access a locked node is with a visitor, helping to prevent unauthorized access to nodes in the data structure.

3.6.2 Motivation

As with the Secure State Machine pattern, the primary motivation of the Secure Visitor pattern is to provide a clean separation between security considerations and user-level functionality. The Secure Visitor pattern allocates all of the security considerations to the nodes in the data hierarchy, leaving developers free to write visitors that only concern themselves with user-level functionality.

Making the nodes in the data hierarchy solely responsible for security functionality makes it more feasible to test and verify the security functionality more rigorously than the user-level functionality. It also frees the user functionality developers from having to reimplement security functionality each time a new visitor is developed, thereby avoiding the creation of new security holes.

3.6.3 Applicability

This pattern is applicable if

- The system possesses hierarchical data that can be processed using the original Gang of Four Visitor pattern [Gamma 1995].
- Various nodes in the hierarchical data have different access privileges.

3.6.4 Structure

Figure 14 shows the structure of the Secure Visitor pattern.