```
  try {

    // Get the appropriate query builder for the current user.
    QueryBuilder *builder =
      QueryBuilderFactory::getInstance.getBuilder(currClient);

    // Get a query director to build the desired query.
    QueryDirector direct(builder);

    // Build the query.
    string query = direct.makeQuery1();

    // Execute the query.
    // .
    // .
    // .
  }
  catch (string e) {

    // Handle any errors relating to building the query.
    displayError(e);

    // Any additional error handling...
  }
}
```

3.3.9 Known Uses

None

3.4 Secure Chain of Responsibility

3.4.1 Intent

The intent of the Secure Chain of Responsibility pattern is to decouple the logic that determines user/environment-trust dependent functionality from the portion of the application requesting the functionality, simplify the logic that determines user/environment-trust dependent functionality, and make it relatively easy to dynamically change the user/environment-trust dependent functionality.

3.4.2 Motivation (Forces)

In an application using a role-based access control mechanism, the behavior of various system functions depends on the role of the current user. The role-based specific behavior for a general system function can range from simply allowing or disallowing a user to access the functionality based on their role to offering the user various, potentially degraded, levels of specific behavior for a general system function. Rather than implementing the role-based selection of the appropriate specific behavior for a general system function in a monolithic manner, the Secure Chain of Responsibility secure design pattern makes use of the more general Chain of Responsibility pattern [Gamma 1995] to break up the role-based specific behavior and the logic for selecting the correct behavior into a chain of handlers.

For example, consider the implementation of a report generation system. In this system users have the capability of generating reports containing varying amounts and type of information based on their roles. Suppose that these are the roles defined by the report generation system:

- **Manager** – A manager may generate reports containing all available information.
- **Sales Analyst Manager** – A sales analyst manager may only generate reports containing sales data. Their reports contain all sales data.
- **Sales Analyst** – A sales analyst may generate reports containing a subset of the sales data.
- **Sales Intern** – A sales intern may only generate reports containing very general information based on the sales data.

Given these defined roles, the Secure Chain of Responsibility secure design pattern may be used as shown in Figure 10 to implement the report generation functionality of this hypothetical system:

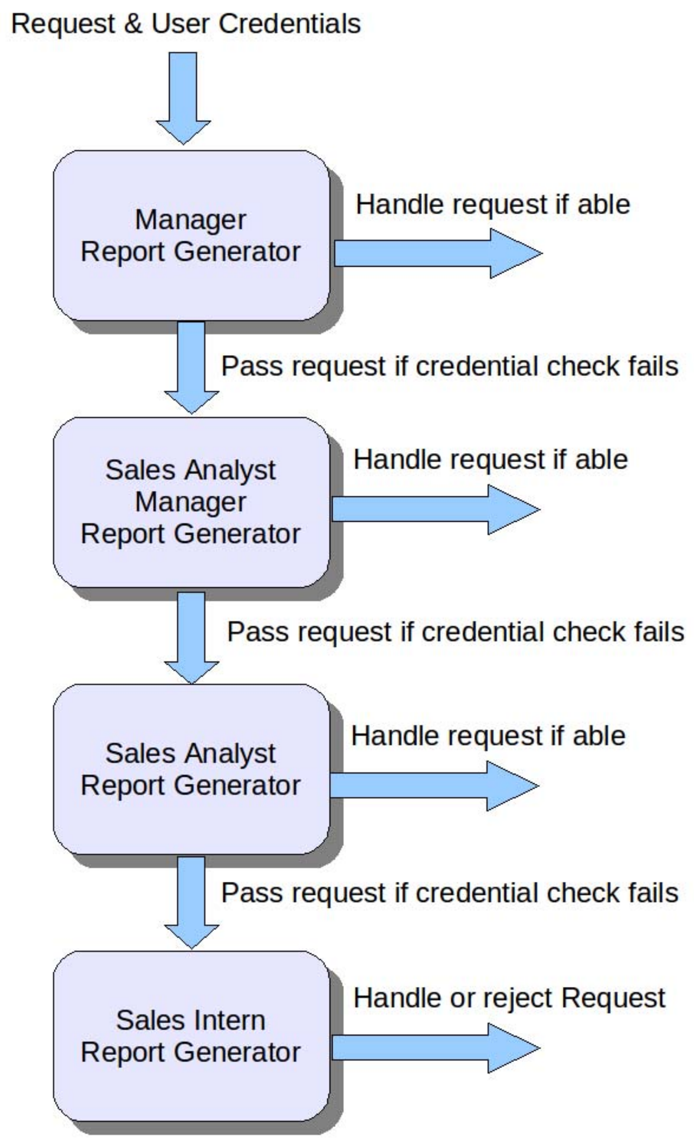
Request & User Credentials
Manager Report Generator
Handle request if able
Pass request if credential check fails
Sales Analyst Manager Report Generator
Handle request if able
Pass request if credential check fails
Sales Analyst Report Generator
Handle request if able
Pass request if credential check fails
Sales Intern Report Generator
Handle or reject Request

Figure 10: Secure Chain of Responsibility Pattern Example

Using the above chain of report generators, the application initially asks the report generator at the start of the chain of responsibility, the manager report generator, to generate the report. As part of the report generation request, the application will provide some form of security credentials for the user making the request. These credentials will be used by a report generator to determine whether the user making the request has the proper security credentials, or in this example, the proper role, for the report generator to handle the request. If the user does not have the proper security credentials for the current report generator to handle the request, the report generator will pass the request on to the next report generator in the chain. Note that since the chain of responsibility is ordered based on the required level of trust the application places in a user, from highest trust level to lowest, a report generator will always pass the request on to a report generator that requires a lower level of trust to handle the request.

3.4.3 Applicability

The Secure Chain of Responsibility pattern is applicable if

- The system has varying security-credential based specific behavior for a general system function.
- The selection of the appropriate specific behavior for a general system function is performed based on using a user's security credentials to determine the level of trust in the user.
- The functionality available to a user with a specific trust level is a superset of the functionality available to users with a lower trust level.
- The selection of the appropriate specific behavior for a general system function can be performed given only a user's security credentials. In other words, the security credentials contain all of the information needed for a request handler object in the chain of responsibility to recognize that it should handle the given request and not pass it down the chain.

3.4.4 Structure

The structure of the Secure Chain of Responsibility pattern is as follows:

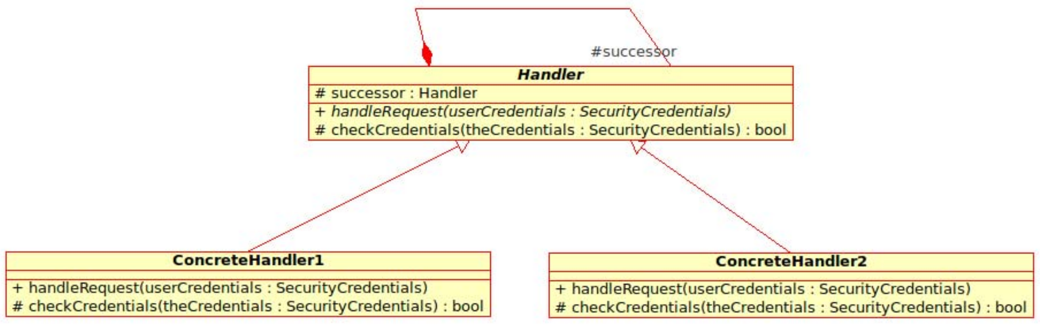
#successor
Handler
# successor : Handler
+ handleRequest(userCredentials : SecurityCredentials)
# checkCredentials(theCredentials : SecurityCredentials) : bool
ConcreteHandler1
+ handleRequest(userCredentials : SecurityCredentials)
# checkCredentials(theCredentials : SecurityCredentials) : bool
ConcreteHandler2
+ handleRequest(userCredentials : SecurityCredentials)
# checkCredentials(theCredentials : SecurityCredentials) : bool

Figure 11: Secure Chain of Responsibility Pattern Structure

The structure of the Secure Chain of Responsibility secure design pattern is similar to the structure of the basic Chain of Responsibility pattern. These are the only structural differences:

- The `handleRequest()` method of the handler classes takes the user's security credentials as an argument.

- To provide a clear separation between the checking of security credentials and the actual function performed by a handler, each handler class may implement a `checkCredentials()` method to determine whether the class has permission to handle a request. Note that if the security credential check is trivial, it may not be necessary to implement the `checkCredentials()` method.

3.4.5 Participants

- **Client** – The client initiates the request, usually with the first handler in the chain.
- **Handler** – The abstract Handler class defines the interface used for making a request. It also defines the `successor` link to the next handler in the chain.
- **SecurityCredentials** – The SecurityCredentials class provides a representation of the security credentials of a user and/or operating environment.
- **ConcreteHandlerN** – A concrete implementation of the Handler class implements the `handleRequest()` method. The implementation of the `handleRequest()` method is responsible for
  - checking the given security credentials to see if the user has a sufficient level of trust for the handler to handle the request. The `checkCredentials()` method (if implemented) may be used to check the given security credentials.
  - checking to see if the handler is actually capable of handling the given request
  - passing the request on to the next handler in the chain if the current handler is unable to handle the request

3.4.6 Consequences

- The security-credential dependent selection of the appropriate specific behavior for a general system function logic is hidden from the portions of the system that make use of the general system function. The application is not aware of which handler finally services the request.
- The black box nature of the Secure Chain of Responsibility secure design pattern makes it easy to change the security credential dependent behavior of the system. Changes to the handlers will require little or no changes to the code making use of the handlers.
- The behavior of the secure chain of responsibility can be dynamically changed during system application by modifying the successor links in the chain.

3.4.7 Implementation

The general process of implementing the Secure Chain of Responsibility pattern is as follows:

1. Identify a general system function whose specific behavior depends on the level of trust associated with a user or operating environment. Use this to define the interface to be implemented by the concrete handler classes.
2. Implement the concrete handler classes that implement the various specific versions of the general system function. This involves implementing the appropriate handling of the request in each handler and implementing the logic used by the handler to decide when to pass the request down the secure chain of responsibility.

3. Create the secure chain of responsibility. This is done by instantiating one object of each concrete handler class and then setting the successor of each handler to the handler responsible for the next lower trust level.
4. The application will then service requests by calling the `handleRequest()` method of the first handler in the chain.

An example of an instantiated secure chain of responsibility is provided in the Motivation section.

3.4.8 Sample Code

The sample code provided in this section is C++ code showing one way of implementing the Secure Chain of Responsibility secure design pattern. The sample code in this section has been taken from an implementation of a Secure XML-RPC server. In the Secure XML-RPC server a client may access a method hosted on the XML-RPC server only if the client is sufficiently trusted. The XML-RPC server is configured with the minimum client trust level required to access each method hosted by the server. The tracking of the minimum trust level required to access a method and the checking of a particular client's method request is implemented using the Secure Chain of Responsibility pattern.

In this case, due to the similarity of behavior among the handlers it was possible to implement the checking of a client's method request using only two handler classes. The general client request permission handler class is defined as follows:

```
/**
 * The MethodChecker class will implement the Secure Chain of
 * Responsibility pattern to check whether the method requested by
 * the client exists, if the client has sufficient trust to execute
 * the requested method, and if the actual parameters given in the
 * request match the method signature.
 */
class MethodChecker {

protected:

  /**
   * The minimum trust level required to execute methods handled by
   * this MethodChecker.
   */
  TrustLevel level;

  /**
   * The methods handled by this MethodChecker. This is a map from
   * method names to method definitions.
   */
  map<string, MethodDefinition> methods;

  /**
   * The next handler to try when given a method not handled by
   * this checker.
   */
  MethodChecker *nextHandler;

  /**
   * Check to see if the given credentials allow this handler to
   * handle the request.
   *
   * @param l The trust level of the credentials to check.
```

```
     *
     * @return true if the handler is allowed to handle the request.
     */
     Bool checkCredentials(TrustLevel l) {return l < level;}

public:

  //! A virtual destructor so destroying derived classes will work nicely.
  virtual ~MethodChecker();

  /**
   * Create a new method checker for methods requiring a given trust
   * level.
   *
   * @param newLevel The minimum trust level required to execute the
   * methods checked by this checker.
   *
   * @param newNextHandler The next handler to try when given a
   * method not handled by this checker.
   */
  MethodChecker(const TrustLevel newLevel, MethodChecker *newNextHandler);

  /**
   * Add the given method to the set of methods accessible to
   * clients with the current trust level or greater. Polymorphic
   * methods are not supported, that is, all method names must be
   * unique.
   *
   * @param method The method to be checked by this checker.
   *
   * @return If the name of the given method is not unique, the
   * method will not be added and false will be returned. Otherwise
   * true will be returned.
   */
  bool addMethod(const MethodDefinition method);

  /**
   * Check to see if the client is allowed to execute the given
   * request and that the request is valid.
   *
   * @param client Information about the client making the request.
   *
   * @param request The client method request.
   *
   * @param reason (OUT) The reason the client request was
   * denied. If the client is not allowed to execute the given
   * method, reason will be set to "Insufficient permission to
   * execute METHOD_NAME". If the method does not exist reason will
   * be set to "Method METHOD_NAME does not exist". If the method is
   * not called with the correct parameters, reason will be set to
   * "Method METHOD_NAME called with incorrect parameters. The
   * expected method signature is METHOD_SIGNATURE".
   *
   * @param reasonCode (OUT) The XML-RPC fault code for the reason the
   * request was rejected.
   *
   * @return If the client has permission to execute the method and
   * the request is valid, true will be returned. If not, false will
   * be returned and reason will be set to the reason for
   * disallowing access.
   *
   */
  virtual bool accessAllowed(const ClientInfo client,
                             const ClientRequest request,
                             string &reason,
                             FaultCode &reasonCode) const;
};
```

Note that the request handled by MethodChecker handler objects is checking to see whether a given client can access the given method hosted on the XML-RPC server. The request handling method in the MethodChecker class is `accessAllowed()`.

The implementation of the MethodChecker class is as follows:

```
// *****************************************************
MethodChecker::~MethodChecker() {
  if (nextHandler != NULL) {
    delete nextHandler;
  }
  nextHandler = NULL;
}

// *****************************************************
MethodChecker::MethodChecker(TrustLevel newLevel,
                             MethodChecker *newNextHandler) {
  level = newLevel;
  nextHandler = newNextHandler;
}

// *****************************************************
bool MethodChecker::addMethod(MethodDefinition method) {

  // Is the method already handled by this method checker?
  if (methods.find(method.getName()) != methods.end()) {

    // It is a duplicate. Do not add it.
    return false;
  }

  // The method is not a duplicate. Track it.
  methods[method.getName()] = method;

  return true;
}

// *****************************************************
bool MethodChecker::accessAllowed(const ClientInfo client,
                                  const ClientRequest request,
                                  string &reason,
                                  FaultCode &reasonCode) const {

  // Is the desired method tracked by this method checker?
  string calledMethodName = request.getAsCalledSig().getName();
  if (methods.find(calledMethodName) == methods.end()) {

    // Pass the request on to the next handler in the chain.
    return nextHandler->accessAllowed(client, request, reason, reasonCode);
  }

  // If we get here the method is tracked by this method checker.

  // Does the client have sufficient permissions to execute the
  // method?
  if (!(checkCredentials(client.getTrustLevel)) {
    // No, it does not. Reject the request.
    reason = string("") +
      "The client trust level (" +
      toString<TrustLevel>(client.getTrustLevel()) + ") is not " +
      "sufficient to execute method " + calledMethodName + " (" +
      toString<TrustLevel>(level) + ").";
    reasonCode = UNAUTHORIZED_METHOD;
    return false;
  }

  // If we get here the client can execute the method.

  // Did the client call the method with valid arguments?
  MethodDefinition formalMethodDef = methods.find(calledMethodName)->second;
  if (request.getAsCalledSig() != formalMethodDef) {

    // No, it does not. Reject the request.
    reason = string("") +
      "The as-called method signature is " +
      toString<MethodDefinition>(request.getAsCalledSig()) + ". The " +
      "required signature is " +
      toString<MethodDefinition>(formalMethodDef) + ".";
    reasonCode = INVALID_METHOD_ARGUMENTS;
    return false;
  }

  // If we get here the method exists, the client has sufficient
  // permission to execute the method, and the method was called
  // correctly. The server can execute the method.
  return true;
}
```

The general MethodChecker class is used for handlers that handle methods that are actually hosted on the XML-RPC server. The handler for methods that are not hosted on the XML-RPC server are handled by instantiations of the UnknownMethodChecker class:

```
/**
 * The UnknownMethodHandler class is the default handler class to
 * handle checking methods that are not tracked by any other method
 * checker. It will appear at the end of the secure chain of
 * responsibility and handle any methods that have been passed down
 * by all the previous checkers. It simply rejects all requests as
 * requests to unknown methods.
 */
class UnknownMethodChecker : public MethodChecker {

public:

  ~UnknownMethodChecker() {
    // There is no next checker here, so no need to free anything.
  };
  UnknownMethodChecker(const TrustLevel newLevel,
                       MethodChecker *newNextHandler);

  bool accessAllowed(const ClientInfo client,
                     const ClientRequest request,
                     string &reason,
                     FaultCode &reasonCode) const;
};
```

The implementation of the UnknownMethodChecker class is as follows:

```
// *******************************************************
// This checker never uses the trust level field or the next checker
// field, so set them to unusable values.
UnknownMethodChecker::UnknownMethodChecker(const TrustLevel newLevel,
                                           MethodChecker *newNextHandler)
  : MethodChecker(BOGUS,NULL) {}

// *******************************************************
bool UnknownMethodChecker::accessAllowed(const ClientInfo client,
                                         const ClientRequest request,
                                         string &reason,
                                         FaultCode &reasonCode) const {

  // If this checker gets a request, it means the desired method is
  // unknown. Reject the method.
  string calledMethodName = request.getAsCalledSig().getName();
  reason = string("") + "The method '" + calledMethodName +
    "' is not known by the server.";
  reasonCode = UNKNOWN_METHOD;
  return false;
}
```

In the Secure XML-RPC server implementation clients are classified according to the following levels of trust:

- **Banned** – The client is not allowed to even connect to the XML-RPC server.
- **None** – The client is not trusted, but still allowed to connect to the server. The client may access a limited set of methods hosted on the server.
- **Little –** The client is minimally trusted.
- **Somewhat** – The client is trusted to some degree, but not completely trusted.
- **Complete** – The client is completely trusted and can access all methods hosted on the server.

Instances of the MethodChecker class will be used to check client requests for methods hosted on the XML-RPC server that require minimum client trust levels of **Complete**, **Somewhat**, **Little**, and **None**. Banned clients are not allowed to connect to the Secure XML-RPC server, so a method checker is not required for the **Banned** trust level. Methods that are not known by the Secure XML-RPC server are handled by an instance of the UnknownMethodChecker class.

The secure chain of responsibility for checking client method requests may be set up as follows:

```
// The default checker handles all methods that are not known by
// the other checkers (it rejects them all).
MethodChecker *defaultChecker =
  new UnknownMethodChecker(BOGUS, NULL);

// Set up the checkers for known methods.
methodCheckers[NONE] =
  new MethodChecker(NONE, defaultChecker);
methodCheckers[LITTLE] =
  new MethodChecker(LITTLE, methodCheckers[NONE]);
methodCheckers[SOMEWHAT] =
  new MethodChecker(SOMEWHAT, methodCheckers[LITTLE]);
methodCheckers[COMPLETE] =
  new MethodChecker(COMPLETE, methodCheckers[SOMEWHAT]);

// We initially start checking methods at the COMPLETE trust level.
firstMethodChecker = methodCheckers[COMPLETE];
```

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.