6. Set the concrete secure factory defined in step 5 as the default factory provided by the abstract factory defined in step 3.

If the application being written does not need to support easily changing the secure factory being used, it is possible to implement the Secure Factory secure design pattern using the non-abstract Factory pattern. This may be done by skipping step three (creation of the AbstractSecureFactory class) and then making use of the single concrete secure factory (defined in step 5) in the application.

3.1.8 Sample Code

Sample code using the Secure Factory secure design pattern to select a Builder object [Gamma 1995] given security information is provided in the Secure Builder Factory section (Section 3.3).

Sample code using the Secure Factory secure design pattern to select a Strategy object [Gamma 1995] given security information is provided in the Secure Strategy Factory section (Section 3.2).

3.1.9 Known Uses

Secure XML-RPC Server Library

3.2 Secure Strategy Factory

3.2.1 Intent

The intent of the Secure Strategy Factory pattern is to provide an easy to use and modify method for selecting the appropriate strategy object (an object implementing the Strategy pattern) for performing a task based on the security credentials of a user or environment. This secure design pattern is an extension of the Secure Factory secure design pattern (Section 3.1) and makes use of the existing Strategy pattern [Gamma 1995].

In brief, the Secure Strategy Factory pattern operates as follows:

1. A caller asks an implementation of the Secure Strategy Factory pattern for the appropriate strategy to perform a general system function given a specific set of security credentials.
2. The Secure Strategy Factory pattern implementation uses the given security credentials to select and return the appropriate object implementing the Strategy pattern that will correctly perform the desired general system function.

3.2.2 Motivation (Forces)

Various general functions performed by a secure system may behave differently based on the security credentials of a user of the system or the particular environment in which the system is operating. For example, a system may generate user error messages containing a varying amount of information based on the trust level of the user, with untrusted users getting error messages containing less information than trusted users. Rather than spreading the logic needed to select the appropriate security specific behavior for a general system function throughout the system, the Secure Strategy Factory pattern concentrates the logic needed to choose the correct specific behavior for a general system function in a single centralized location. The use of this pattern will result in an implementation that is easier to modify, test, and verify than a system where the securi-

ty-credential based specific behavior for a general system function is spread throughout the tem's code base.

As this pattern is an extension of the Abstract Factory pattern [Gamma 1995], it is relatively easy to generate custom variants of a system that provide different security-credential based specific behavior for general system functions.

3.2.3 Applicability

The Secure Strategy Factory pattern is applicable if

- The system has varying security-credential-based specific behavior for a general system function.
- The various versions of the general system function can be implemented by classes using the Strategy pattern. From [Gamma 1995], the Strategy pattern is applicable if
  - Many related classes differ only in their behavior. Strategies provide a way to configure a class with one of many different behaviors.
  - You need different variants of an algorithm.
  - An algorithm uses data that clients shouldn't know about.
  - A class defines many behaviors, and these appear as multiple conditional statements in its operations. Instead of many conditionals, move related conditional branches into their own Strategy class.
- The selection of the appropriate specific behavior for a general system function (that is, the selection of the appropriate Strategy object) can be performed given only a set of security credentials. In other words, the security credentials contain all of the information needed to select the appropriate strategy to perform the general system function.

3.2.4 Structure

Figure 8 shows the structure Secure Strategy Factory pattern.

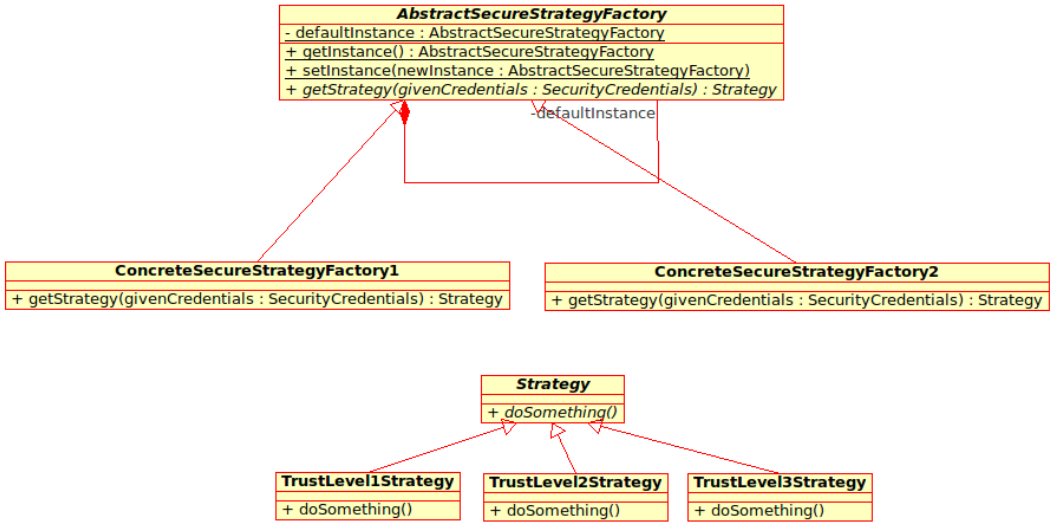

Figure 8: Secure Strategy Factory Pattern Structure

Note that as with the more general Secure Factory secure design pattern, it is possible to implement the Secure Strategy Factory secure design pattern using the non-abstract Factory pattern. See the Secure Factory secure design pattern for additional discussion of the use of the non-abstract Factory pattern.

3.2.5 Participants

- **Client** – The client tracks the security credentials of a user and/or the environment in which the system is operating. Given the security credentials of interest, the client uses the `getInstance()` method of the AbstractSecureStrategyFactory to get a concrete instance of the secure strategy factory, and then calls the `getStrategy()` method of the concrete factory to get the appropriate strategy given the current security credentials.
- **SecurityCredentials** – The SecurityCredentials class provides a representation of the security credentials of a user and/or operating environment.
- **AbstractSecureStrategyFactory** – The AbstractSecureStrategyFactory class serves several purposes.
  - It provides a concrete instance of a secure strategy factory via the `getInstance()` method of the factory.
  - It allows the system to set the actual concrete secure strategy at runtime via the `setInstance()` method. This makes it relatively easy to change the strategy selection methodology by specifying a different concrete secure strategy at runtime.
  - It defines the abstract `getStrategy()` method that must be implemented by all concrete implementations of AbstractSecureStrategyFactory.
- **ConcreteSecureStrategyFactoryN** – Different strategy selection methodologies are implemented in various concrete implementations of AbstractSecureStrategyFactory. Each concrete secure strategy factory provides an implementation of `getStrategy()`, which is responsible for selecting an appropriate strategy for the given security credentials.
- **Strategy** – The abstract Strategy class defines a method representing a general system function. All concrete implementations of the abstract Strategy class must provide an implementation of this method.
- **TrustLevelNStrategy** – One concrete implementation of the abstract Strategy class must be provided for each different security-credential based specific behavior for a general system function. Viewing the security credentials as a method for defining several levels of trust, one concrete TrustLevelNStrategy will be implemented for each level of trust.

The Secure Strategy Factory secure design pattern shares many common participants with the Secure Factory secure design pattern (Section 3.1). The primary difference in the participants between the two patterns is the type of object returned by the concrete implementations of the factory interface. The `getStrategy()` method of the ConcreteSecureStrategyFactoryN classes returns an instance of a class implementing the Strategy pattern while the `getObject()` method of the ConcreteSecureFactoryN classes returns an object of an unspecified type.

3.2.6 Consequences

- The logic of using security credentials to select the appropriate specific behavior for a general system function is hidden from the portions of the system that make use of the general system function.
- As with the Secure Factory secure design pattern, the black box nature of the Secure Strategy Factory secure design pattern makes it easy to change the security credential dependent behavior of the system. Changes to the strategy selection logic or the provided strategies themselves will require little or no changes to the code making use of the Secure Strategy Factory pattern implementation.

3.2.7 Implementation

The general process of implementing the Secure Strategy Factory 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.
2. Use the Strategy pattern to define the interface for classes implementing the general system function.
3. Write concrete implementations of the interface defined in step two. One concrete implementation will be written for each security specific version of the general system function.
4. Use the basic Abstract Factory pattern as described in the Structure section to implement the AbstractSecureStrategyFactory.
5. Identify the information needed to determine the trust level of a user or environment. This information will be used to define the SecurityCredentials class or data structure.
6. Implement a concrete secure strategy factory that selects the appropriate strategy defined in step three given a security credential defined in step four.
7. Set the concrete secure strategy factory defined in step five as the default factory provided by the abstract factory defined in step three.

3.2.8 Sample Code

The sample code provided in this section is C++ code showing how to implement the Secure Strategy Factory 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 the content of XML-RPC fault messages is dependent on the trust level of an XML-RPC client. Untrusted clients are given XML-RPC fault messages that contain less information than trusted clients.

In this example the general system functionality to be implemented in trust-level specific strategies is the generation of XML-RPC fault messages. Using the Strategy pattern, the abstract class defining the interface to XML-RPC fault message generators is as follows:

```
//! Fault codes for the various types of XML-RPC failures.
enum FaultCode {
  //! A general problem on the XML-RPC server occurred.
  SERVER_ERROR = 8,
  //! The client request exceeded the maximum request length.
  REQUEST_TOO_LONG = 1,
  //! A general problem with the client request occurred.
  CLIENT_ERROR = 0,
```

```
  //! The client has issued too many bad requests and is now banned.
 TOO_MANY_FAULTY_REQUESTS = 2,
  //! The XML in the client request was invalid.
  MALFORMED_CLIENT_REQUEST = 3,
  //! The client requested a method not known by the server.
  UNKNOWN_METHOD = 4,
  //! The client requested a method they do not have permission to execute.
  UNAUTHORIZED_METHOD = 5,
  //! The client called a method with incorrect arguments.
  INVALID_METHOD_ARGUMENTS = 6,
  //! The executed method generated invalid XML as a response.
  MALFORMED_RESPONSE = 7,
  //! The # of fault codes. <b>Update this if fault codes are added.</b>
  NUM_FAULT_CODES = 9
};

/**
 * A MessageGenerator object generates the appropriate XML for an
 * XML-RPC failure message for a given failure. This interface must
 * be implemented by concrete classes that implement generating
 * failure messages with varying amounts of information. It is
 * expected that there will be one concrete message generator class
 * for each trust level.
 */
class MessageGenerator {

protected:

  /**
   * A utility method for generating a string containing the XML for
   * a fault message. Note that this method simply creates an
   * XML-RPC fault message containing the given information, it does
   * not filter or block the information based on the trust level of
   * a client.
   */
  static string fillOutMessage(FaultCode code, string moreInfo);

  /**
   * Given an XML-RPC fault code, get a text string describing the
   * fault.
   *
   * @param code The XML-RPC fault code (defined in the enum in
   * MessageGenerator,hpp).
   *
   * @return A text string describing the fault.
   */
  static string getErrorString(FaultCode code);

public:

  MessageGenerator() {};

  /**
   * Generate the XML for an XML-RPC fault message, as a string. The
   * text in the fault message will be selected based on the given
   * fault code and the additional fault information string.
   *
   * @param code The fault code of the XML-RPC failure. These are
   * defined in the enum in MessageGenerator.hpp.
   *
   * @param moreInfo A string containing additional information
   * about the XML-RPC failure.
   *
```

```
   * @return A string containing the XML of the appropriate fault
   * message, given the clients trust level.
   */
  virtual string genFaultMessage(FaultCode code, string moreInfo="") = 0;
};
```

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 XML-RPC 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 XML-RPC server.

The Secure XML-RPC server determines the amount of information to provide in XML-RPC fault messages sent to the client based on the trust level of the client. The amount of information sent, broken up by client trust level, is as follows:

- **Complete** – All available information regarding the failed XML-RPC request is sent in the fault message to the client.
- **Somewhat**, **Little** – Only general information regarding the failed XML-RPC request is sent to the client. Specific information about the failed request is not sent to the client.
- **None** – The only information included in the XML-RPC fault message is whether the failed request is due to a problem with the server or a problem with the client's request. No other information is included.

Banned clients are not allowed to connect to the Secure XML-RPC server at all. No XML-RPC messages are sent to banned clients.

The XML-RPC fault message generation strategy for completely trusted clients includes all available fault information. The implementation of the abstract XML-RPC fault message generation strategy for clients with complete trust is as follows:

```
string CompleteTrustMessageGenerator::genFaultMessage(FaultCode code,
                                                       string moreInfo) {
  // To completely trusted clients we will always return as much
  // information as possible regarding faults.
  return fillOutMessage(code, getErrorString(code) + " " + moreInfo);
}
```

The XML-RPC fault message generation strategy for somewhat or little trusted clients only includes general fault information. The implementation of the abstract XML-RPC fault message generation strategy for clients with somewhat or little trust is as follows:

```
  string SomewhatTrustMessageGenerator::genFaultMessage(FaultCode code,
                                                        string moreInfo) {
    // To somewhatly trusted clients we will return the basic error
    // information, but leave out the additional error information.
    return fillOutMessage(code, getErrorString(code));
  }
```

The XML-RPC fault message generation strategy for untrusted clients only includes information on whether the failure occurred due to a problem on the server or due to a problem with the client's request. The implementation of the abstract XML-RPC fault message generation strategy for untrusted clients is as follows:

```
string NoneTrustMessageGenerator::genFaultMessage(FaultCode code,
                                                  string moreInfo) {
  // For clients that are not trusted, we will just tell them if the
  // error was a client or server error. They will get no more
  // information.
  FaultCode newCode = SERVER_ERROR;
  switch (code) {
  case SERVER_ERROR:
  case MALFORMED_RESPONSE: {
    newCode = SERVER_ERROR;
    break;
  }
  case CLIENT_ERROR:
  case REQUEST_TOO_LONG:
  case TOO_MANY_FAULTY_REQUESTS:
  case MALFORMED_CLIENT_REQUEST:
  case UNKNOWN_METHOD:
  case UNAUTHORIZED_METHOD:
  case INVALID_METHOD_ARGUMENTS: {
    newCode = CLIENT_ERROR;
    break;
  }
  }
  return fillOutMessage(code, getErrorString(newCode));
}
```

The logic for selecting the appropriate XML-RPC fault message generation strategy depends on being able to determine the trust level of a client. The trust level information (that is, the security credentials) of a client are tracked using the ClientInfo class in the Secure XML-RPC server:

```
/**
 * A ClientInfo object stores information about a single XML-RPC
 * client.
 *
 * The information tracked includes:
 * - The IP address from which the client connects.
 * - The trust level associated with the clients IP address.
 * - The number of faulty requests from the client.
 * - The maximum allowed request size for the client.
 */
class ClientInfo {

private:

  //! The IP address from which the client connected.
  __be32 ipAddr;

  //! The trust level of the client.
  TrustLevel trustLevel;

  //! The number of faulty requests made by this client;
  unsigned int numFaultyRequests;

public:

  //! The # of faulty requests before a client is banned.
```

```
  static unsigned int maxBadRequests;

  //! The trust level to assign to banned clients.
  static TrustLevel bannedTrustLevel;

  /**
   * Create a new client information object for the given IP address
   * with the given trust level.
   *
   * @param newIpAddr The IP address of the client.
   *
   * @param trust The trust level of the client.
   *
   * The trust level enumerated type is defined in Server.cpp.
   */
  ClientInfo(__be32 newIpAddr, TrustLevel trust);

  /**
   * Copy constructor.
   *
   * @param info The client information object to copy.
   */
  ClientInfo(const ClientInfo &info);

  /**
   * Overwrite all of the fields of the ClientInfo object with zeros
   * prior to destroying the object.
   */
  ~ClientInfo();

  /**
   * Get the # of faulty requests submitted by the client.
   *
   * @return The # of faulty requests submitted by the client.
   */
  unsigned int getNumFaultyRequests() const;

  /**
   * Increment the # of faulty request submissions for the
   * client. The faulty request count will be incremented by 1.
   */
  void trackFaultyRequest();

  /**
   * Get the trust level of the client.
   *
   * @return The trust level of the client.
   */
  TrustLevel getTrustLevel() const;

  /**
   * Get the IP address of the client.
   *
   * @return The IP address of the client.
   */
  __be32 getIpAddr() const;
};
```

The Abstract Factory pattern is used to define the interface for concrete factories that contain the logic for selecting the appropriate XML-RPC fault message generation strategy based on given client information. The interface for the abstract XML-RPC fault message generation strategy factory is as follows:

```
/**
 * Given the trust level of a client, the MessageFactory selects the
 * appropriate fault message generation strategy object and returns
 * it.
 */
class MessageFactory {

private:

  //! The current default concrete message generator factory.
  static MessageFactory *instance;

public:

  virtual ~MessageFactory() {};

  /**
   * Based on the given client trust level, return the appropriate
   * MessageGenerator object for generating failure messages for the
   * client.
   *
   * @param trust The trust level of the client.
   *
   * @return The appropriate generator for generating failure
   * messages for the client.
   */
  virtual MessageGenerator *getMessageGenerator(TrustLevel trust) = 0;

  /**
   * Get the current default concrete message generator factory.
   *
   * @return The current default concrete message generator factory.
   */
  static MessageFactory *getInstance();

  /**
   * Set the current default concrete message generator factory. Use
   * this to use a message generator factory other than the default
   * factory (TrustBasedMessageFactory).
   *
   * @param newFactory The new factory instance to use.
   */
  static void setInstance(MessageFactory *newFactory);
};
```

The user of the abstract XML-RPC fault message generation strategy factory gets the currently used concrete implementation of the factory via the static `getInstance()` method of the abstract factory class. In the Secure XML-RPC server the default concrete implementation of the abstract factory class is a XML-RPC fault message generation strategy factory that uses the client trust level to determine the correct strategy for generating the fault messages. The definition of the trust-based XML-RPC fault message generation strategy factory is as follows:

```
/**
 * The trust based message generator factory will return a different message
 * generator object based on the trust level of the client for which XML-RPC
 * fault messages are to be generated.
 */
class TrustBasedMessageFactory : public MessageFactory {

private:
```

```
  //! The message generator objects to use, indexed by trust level.
  MessageGenerator *generators[HIGHEST_TRUST_LEVEL+1];

public:
  //! Make a new trusted based message generator factory.
  TrustBasedMessageFactory() throw (XmlRpcException);
  //! Clean up the trusted based message generator factory.
  ~TrustBasedMessageFactory();
  MessageGenerator *getMessageGenerator(TrustLevel trust);
};
```

An XML-RPC fault message for a client whose information is stored in the variable `currClient` of type `ClientInfo` is then generated as follows:

```
// First, get the current concrete message generation strategy factory.
MessageFactory *messageGenFactory = MessageFactory::getInstance();

// Then get the appropriate strategy for generating a message given
// the current clients trust level.
MessageGenerator *currMsgGen =
  messageGenFactory->getMessageGenerator(currClient.getTrustLevel());

// Generate the XML-RPC fault message with the correct amount of
// information.
// This example fault message is sent when a client request is too long.
string errMsg = currMsgGen->genFaultMessage(REQUEST_TOO_LONG,
                                string("The maximum request length is ") +
                                toString<int>(maxRequestSize));
```

3.2.9 Known Uses

Secure XML-RPC Server Library

3.3 Secure Builder Factory

3.3.1 Intent

The intent of the Secure Builder Factory secure design pattern is to separate the security dependent rules involved in creating a complex object from the basic steps involved in actually creating the object. A *complex object* is generally defined as a library object that is made up from many interrelated elements or digital objects [Arms 2000]. In the current context, a complex object is an object that makes use of several simpler objects.

In brief, the Secure Builder Factory pattern operates as follows:

1. A caller asks an implementation of the Secure Builder Factory pattern for the appropriate builder to build a complex object given a specific set of security credentials.
2. The Secure Builder Factory pattern implementation uses the given security credentials to select and return the appropriate object implementing the Builder pattern [Gamma 1995] that will correctly build a complex object given the security rules identified by the given security credentials.

3.3.2 Motivation (Forces)

A secure application may make use of complex objects whose allowable contents are dictated by the level of trust the application has in a user or operating environment. The content of the complex objects may be controlled in several ways: