```
  //! 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:

- The complex object itself may be given information about the user/environment trust level, which would be used by the object to control its content. This creates a tight coupling between the security-based content creation logic and the complex object implementation
- The secure application creating the complex object could use the information about the user/environment trust level to directly control the setting of the complex object contents. This creates a tight coupling between the security-based content creation logic and the secure application implementation
- The logic needed to set the complex object contents based on the user/environment trust level could be decoupled from the complex object and the application code creating the complex object by the use of the Secure Builder Factory secure design pattern. This creates a loose coupling between the security-based content creation logic and the complex object implementation and a loose coupling between the security-based content creation logic and the secure application implementation.

The loose coupling between the security-based content creation logic and the complex object implementation and a loose coupling between the security-based content creation logic and the secure application implementation makes it easier to verify, test, and modify the security-based complex object content creation logic.

As a simple example of the use of the Secure Builder Factory secure design pattern, consider an example application that makes use of some form of a persistent data management system (PDMS). The allowable queries the application may make to the PDMS are dictated by the trust level of the current user of the system. The application could be implemented by defining a class that is capable of representing all possible PDMS queries, a PDMS query builder class for each trust level, and a builder factory responsible for selecting the correct PDMS query builder object for a given set of security credentials. The application would then use the query builder factory to get the correct PDMS query builder and use the builder to build a query. The selected query builder would only be capable of building queries appropriate for the trust level of the current user.

3.3.3 Applicability

The Secure Builder Factory pattern is applicable if

- The system may make use of an object implementing the Builder pattern to create complex objects. From [Gamma 1995], the Builder pattern is applicable if
  - The algorithm for creating a complex object should be independent of the parts that make up the object and how they are assembled.
  - The construction process must allow for different representations for the object that is constructed.
- The behavior of the Builder is dependent on the security credentials of a user/operating environment. In other words, different variants of a complex object will be constructed given different user/operating environment security credentials.
- The construction of the appropriate complex object can be performed given only a set of security credentials. In other words, the security credentials contain all of the information needed to construct the correct complex object.

3.3.4 Structure

Figure 9 shows the structure of the Secure Builder Factory pattern.

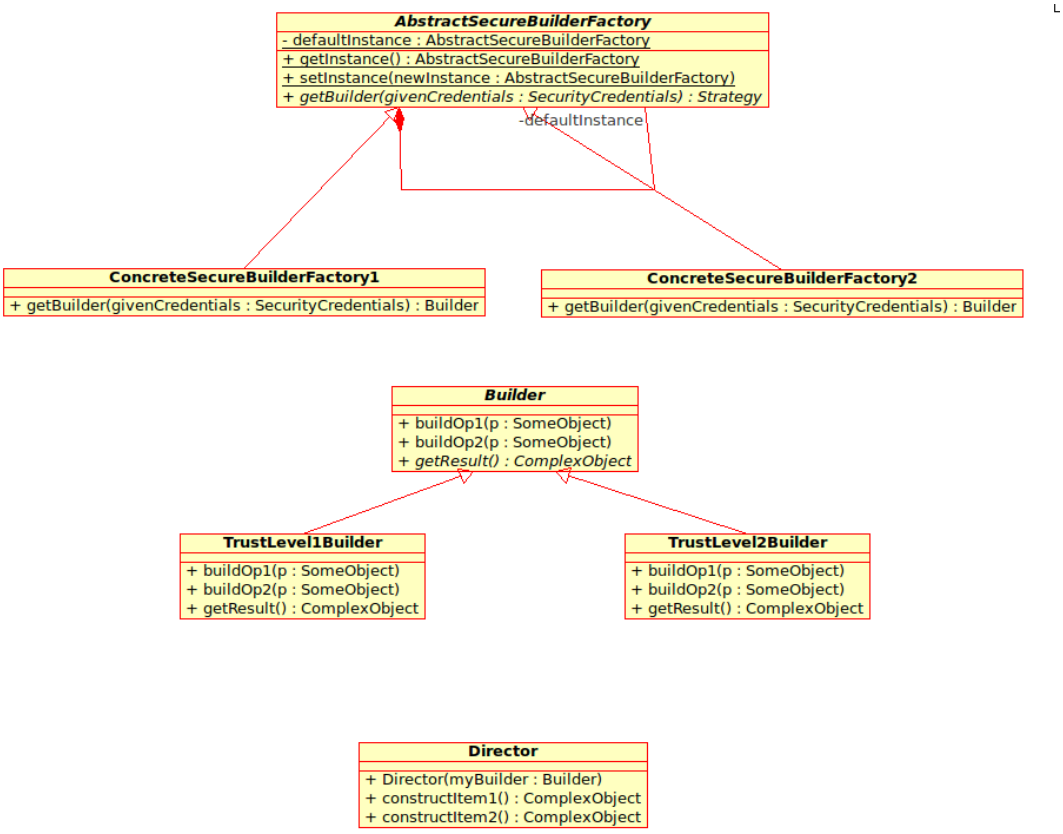

Figure 9: Secure Builder Factory Pattern Structure

Note that as with the more general Secure Factory secure design pattern, it is possible to implement the Secure Builder 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.

3.3.5 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 AbstractBuilderFactory to get a concrete instance of the secure builder factory, and then calls the `getBuilder()` method of the concrete factory to get the appropriate builder given the current security credentials.
- **Director** – This is the Director element of the Builder pattern [Gamma 1995]. The director builds specific complex objects given a builder instance. The client instantiates a Director with a builder chosen by the secure builder factory and then calls methods of the Director to create specific complex objects.
- **SecurityCredentials** – The SecurityCredentials class provides a representation of the security credentials of a user and/or operating environment.

- **AbstractSecureBuilderFactory** – The AbstractSecureBuilderFactory class serves several purposes:
  - It provides a concrete instance of a secure builder factory via the `getInstance()` method of the factory.
  - It allows the system to set the actual concrete secure builder factory at runtime via the `setInstance()` method. This makes it relatively easy to change the builder selection methodology by specifying a different concrete secure builder factory at runtime.
  - It defines the abstract `getBuilder()` method that must be implemented by all concrete implementations of AbstractSecureBuilderFactory.
- **ConcreteSecureBuilderFactoryN –** Different builder selection methodologies are implemented in various concrete implementations of AbstractSecureBuilderFactory. Each concrete secure builder factory provides an implementation of `getBuilder()`, which is responsible for selecting an appropriate builder for the given security credentials.
- **Builder –** The abstract Builder class defines the basic builder methods applicable for all builders of the complex object.
- **TrustLevelNBuilder –** One concrete implementation of the abstract Builder class must be provided for each different security-credential based set of complex object building behaviors. Viewing the security credentials as a method for defining several levels of trust, one concrete TrustLevelNBuilder class will be implemented for each level of trust.

3.3.6 Consequences

- The security-credential dependent selection of the appropriate complex object builder is hidden from the portions of the system that make use of the builder. An implementation of the Secure Builder Factory pattern operates as a black box supplying the appropriate builder to the caller. This in turn hides the security dependent complex object building behavior from the caller.
- The black box nature of the Secure Builder Factory secure design pattern makes it easy to change the security credential dependent behavior of the system. Changes to the builder selection logic or the provided builders themselves will require little or no changes to the code making use of the Secure Builder Factory implementation.

3.3.7 Implementation

The general process of implementing the Secure Builder Factory pattern is as follows:

1. Identify a complex object whose construction depends on the level of trust associated with a user or operating environment. Define the general builder interface using the Builder pattern for building complex objects of this type.
2. Implement the concrete builder classes that implement the various trust level specific construction rules for the complex object.
3. Use the basic Abstract Factory pattern as described in the Structure section to implement the AbstractSecureBuilderFactory.
4. 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.

5. Implement a concrete secure builder factory that selects the appropriate builder defined in step 2 given security credentials defined in step 4.
6. Set the concrete secure builder factory defined in step 5 as the default factory provided by the abstract factory defined in step 3.
7. Implement a Director to build specific types of complex objects using a given builder.

3.3.8 Sample Code

The sample code provided in this section is C++ code showing how to implement the Secure Builder Factory secure design pattern. The sample code in this section expands on the example provided in the Motivation section. The code implements the creation of appropriate queries to an underlying persistent data management system (PDMS) based on the level of trust assigned to a user. This example assumes that the underlying PDMS does not support a sufficient level of access control to support the security requirements of the application.

In more detail, the data stored in the PDMS for the example application is employee and customer data. For each employee or customer the following information is stored:

- Social Security Number
- Address
- First Name
- Last Name
- Date of Birth

The application supports the following PDMS actions:

- Viewing of a record.
- Modification of a record.

The application limits the PDMS actions allowed to be performed by a user based on the user's trust level. The trust levels supported by the application are

- **Banned** – The user is not allowed to access the PDMS.
- **Little –** The user is minimally trusted. They only have limited access to the PDMS. A user with little trust may only view the names and date of births of employees in the PDMS. They may not change the contents of the PDMS.
- **Complete** – The user is completely trusted. They have full access to the PDMS. The user may view and modify the full contents of the PDMS.

The abstract query builder class for building query objects is defined as follows:

```
class QueryBuilder {

public:

  virtual void addField(const string &f) throw(string) {}
  virtual void addConstraint(const string &f) throw(string) {}
  virtual void setSourceEmployee() throw(string) {}
  virtual void setSourceCustomer() throw(string) {}
  virtual void setActionView() throw(string) {}
  virtual void setActionModify() throw(string) {}
```

```
  virtual Query getQuery() throw(string) = 0;
};
```

A concrete query builder class will be implemented for each user trust level requiring different query building behavior. The concrete query builder class for completely trusted users is as follows:

```
class CompleteTrustQueryBuilder : public QueryBuilder {

private:

  //! Track the action to be performed.
  string action;

  //! Track the data source being used.
  string source;

  //! Track a list of fields the query uses.
  list<string> fields;

  //! Track the constraints on the query.
  list<string> constraints;

public:

  CompleteTrustQueryBuilder() {
    source = "";
    action = "";
  }

  void addField(const string &f) throw(string) {

    // A completely trusted user can access all fields.
    if ((f == "SSN") ||
        (f == "ADDRESS") ||
        (f == "FIRST_NAME") ||
        (f == "LAST_NAME") ||
        (f == "BIRTH_DATE")) {
      fields.append(f);
    }

    // The field provided is not known.
    else {
      throw "Field " + f + " is unknown.";
    }
  }

  void addConstraint(const string &f) throw(string) {
    constraints.append(f);
  }

  void setSourceEmployee() throw(string) {
    // A completely trusted user can access all data sources.
    source = "employee";
  }

  void setSourceCustomer() throw(string) {
    // A completely trusted user can access all data sources.
    source = "customer";
  }

  void setActionView() throw(string) {
```

```
    // A completely trusted user perform all actions.
    action = "view";
  }

  void setActionModify() throw(string) {
    // A completely trusted user perform all actions.
    action = "modify";
  }

  Query getQuery() throw(string) throw(string) {

    // The action, data source, and at least one field must be set to
    // create the query.
    if ((action == "") || (source == "") || (fields.size() == 0)) {
      throw "Cannot create query. Information is missing.";
    }

    // Create and return the query.
    // .
    // .
    // .
  }
};
```

The query builder for users with little trust is as follows:

```
class LittleTrustQueryBuilder : public QueryBuilder {

private:

  //! Track the action to be performed.
  string action;

  //! Track the data source being used.
  string source;

  //! Track a list of fields the query uses.
  list<string> fields;

  //! Track the constraints on the query.
  list<string> constraints;

public:

  CompleteTrustQueryBuilder() {
    source = "";
    action = "";
  }

  void addField(const string &f) throw(string) {

    // A little trusted user can only access the names and DOBs in the
    // PDMS.
    if ((f == "FIRST_NAME") ||
        (f == "LAST_NAME") ||
        (f == "BIRTH_DATE")) {
      fields.append(f);
    }

    // The user cannot access the given field.
    else if ((f == "SSN") ||
             (f == "ADDRESS")) {
      throw "The user may not access the " + f + " field.";
```

```
    }

    // The field provided is not known.
    else {
      throw "Field " + f + " is unknown.";
    }
  }

  void addConstraint(const string &f) throw(string) {
    constraints.append(f);
  }

  void setSourceEmployee() throw(string) {
    // A little trusted user can access the employee data source.
    source = "employee";
  }

  void setSourceCustomer() throw(string) {
    // A little trusted user cannot access the customer data source.
    throw "The user cannot access the customer data source."
  }

  void setActionView() throw(string) {
    // A little trusted user can view records.
    action = "view";
  }

  void setActionModify() throw(string) {
    // A little trusted user cannot modify records.
    throw "The user cannot modify records.";
  }

  Query getQuery() throw(string) {

    // The action, data source, and at least one field must be set to
    // create the query.
    if ((action == "") || (source == "") || (fields.size() == 0)) {
      throw "Cannot create query. Information is missing.";
    }

    // Create and return the query.
    // .
    // .
    // .
  }
};
```

The query builder for banned users is as follows:

```
class BannedTrustQueryBuilder : public QueryBuilder {

public:

  Query getQuery() throw(string) {

    // A banned user cannot access the PDMS.
    throw "A banned user may not access the PDMS.";
  }
};
```

The logic for selecting the appropriate query builder depends on being able to determine the trust level of a user. The trust level information (that is, the security credentials) of a user will be

tracked using the ClientInfo class. The ClientInfo class will not be explicitly implemented in this example.

The Abstract Factory pattern is used to define the interface for concrete factories that contain the logic for selecting the appropriate query builder based on given user information. The interface for the abstract query builder factory is as follows:

```
/**
 * Given the trust level of a user, the QueryBuilderFactory selects the
 * appropriate query builder object and returns it.
 */
class QueryBuilderFactory {

private:

  //! The current default concrete query builder factory.
  static QueryBuilderFactory *instance;

public:

  virtual ~QueryBuilderFactory() {};

  /**
   * Based on the given user trust level, return the appropriate
   * QueryBuilder object for building queries.
   *
   * @param trust The trust level of the user.
   *
   * @return The appropriate builder for building queries.
   */
  virtual QueryBuilder *getBuilder(TrustLevel trust) = 0;

  /**
   * Get the current default concrete query builder factory.
   *
   * @return The current default query builder factory.
   */
  static QueryBuilderFactory *getInstance();

  /**
   * Set the current default concrete query builder factory. Use
   * this to use a query builder factory other than the default
   * factory (TrustBasedQueryBuilderFactory).
   *
   * @param newFactory The new factory instance to use.
   */
  static void setInstance(QueryBuilderFactory *newFactory);
};
```

The user of the abstract query builder factory gets the currently used concrete implementation of the factory via the static `getInstance()` method of the abstract factory class. In the example code the default concrete implementation of the abstract factory class is a query builder factory that uses the user trust level to determine the correct query builder. The definition of the trust-based query builder factory is as follows:

```
/**
 * The trust based query builder factory will return a different builder
 * object based on the trust level of the user.
 */
class TrustBasedQueryBuilderFactory : public QueryBuilderFactory {
```

```
public:
  QueryBuilder *getBuilder(TrustLevel trust) {
    if (trust.level == COMPLETE) {
      return new CompleteTrustQueryBuilder();
    }
    else if (trust.level == LITTLE) {
      return new LittleTrustQueryBuilder();
    }
    else {
      return new BannedTrustQueryBuilder();
    }
  }
};
```

A Director class actually uses a given builder to build various specific types of queries. Note that the Director implementation does not directly base its behavior on the security credentials of the user. The security-credential-based behavior is handled transparently in the client by the selection of the proper builder by the Secure Builder Factory object. The Director uses the given builder without needing to know how the builder was selected.

```
class QueryDirector {

private:

  QueryBuilder *builder;

public:

  QueryDirector(QueryBuilder *newBuilder) {builder = newBuilder;}

  // Query building methods.

  string makeQuery1() throw(string) {

    // Build the query.
    builder->setSourceCustomer();
    builder->setActionModify();
    builder->addField("SSN");
    string query = builder->getQuery();

    // Set some constraints on the query.
    // .
    // .
    // .

    return query;
  }

  // Additional query building methods...
};
```

Using this implementation, a query may be created as follows:

```
  // .
  // .
  // .
  ClientInfo currClient = getClientInfo();
  // .
  // .
  // .
```

```
  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: