3 The Design-Level Patterns

3.1 Secure Factory

3.1.1 Intent

The intent of the Secure Factory secure design pattern is to separate the security dependent logic involved in creating or selecting an object from the basic functionality of the created or selected object.

In brief, the Secure Factory secure design pattern operates as follows:

1. A caller asks an implementation of the Secure Factory pattern for the appropriate object given a specific set of security credentials.
2. The Secure Factory pattern implementation uses the given security credentials to select and return the appropriate object.

The Secure Factory secure design pattern presented here is a security specific extension of the Abstract Factory pattern [Gamma 1995]. Note that it is also possible to implement the Secure Factory secure design pattern using the non-abstract Factory pattern. Although the Abstract Factory pattern is more complex than the non-abstract Factory pattern, the Abstract Factory pattern is presented here as the basis for the Secure Factory secure design pattern due to the Abstract Factory pattern's support for easily and transparently changing the underlying concrete factory implementation.

Specializations of the Secure Factory secure design pattern are the Secure Strategy Factory (Section 3.2) and Secure Builder Factory (Section 3.3) secure design patterns.

3.1.2 Motivation (Forces)

A secure application may make use of an object whose behavior is dependent of the level of trust the application places in a user or operating environment. The logic defining the trust level dependent behavior of the object may be implemented in several ways:

- The object itself may be given information about the user/environment trust level, which would be used by the object to control its behavior. This creates a tight coupling between the security-based behavior of the object and otherwise general object functionality.
- The secure application creating the object could use the information about the user/environment trust level to directly choose between different objects implementing different security-based behavior. This creates a tight coupling between the security-based object selection logic and the more general secure application functionality.
- The logic needed to choose the correct object based on the user/environment trust level could be decoupled from the object and the application code creating the object by the use of the Secure Factory secure design pattern. This creates a loose coupling between the security-based object selection logic and the object implementation and a loose coupling between the security-based object selection logic and the secure application implementation.

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

3.1.3 Applicability

The Secure Factory secure design pattern is applicable if

- The system constructs different versions of an object based on the security credentials of a user/operating environment.
- The available security credentials contain all of the information needed to select and construct the correct object. No other security-related information is needed.

3.1.4 Structure

Figure 7 shows the structure of the Secure Factory secure design pattern.

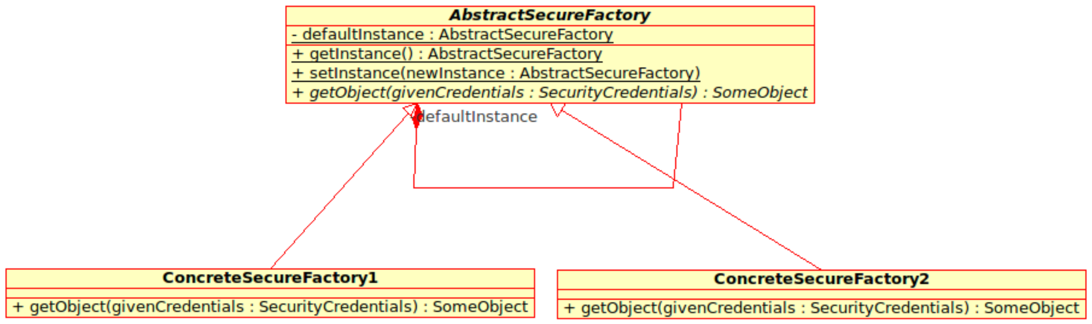
AbstractSecureFactory
- defaultInstance : AbstractSecureFactory
+ getInstance() : AbstractSecureFactory
+ setInstance(newInstance : AbstractSecureFactory)
+ getObject(givenCredentials : SecurityCredentials) : SomeObject
defaultInstance
ConcreteSecureFactory1
+ getObject(givenCredentials : SecurityCredentials) : SomeObject
ConcreteSecureFactory2
+ getObject(givenCredentials : SecurityCredentials) : SomeObject

Figure 7: Secure Factory Pattern Structure

3.1.5 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 AbstractSecureFactory class to get a concrete instance of the secure factory, and then calls the `getObject()` method of the concrete factory to get the appropriate object given the current security credentials.
- SecurityCredentials – The SecurityCredentials class provides a representation of the security credentials of a user and/or operating environment.
- AbstractSecureFactory – The AbstractSecureFactory class serves several purposes:
  - It provides a concrete instance of a secure factory via the `getInstance()` method of the factory.
  - It allows the system to set the default concrete secure factory at runtime via the `setInstance()` method. This makes it relatively easy to change the object selection methodology by specifying a different concrete secure factory at runtime.
  - It defines the abstract `getObject()` method that must be implemented by all concrete implementations of AbstractSecureFactory. The `getObject()` method is called by the client to get the appropriate object given some security credentials.

- ConcreteSecureFactoryN – Different object selection methodologies are implemented in various concrete implementations of AbstractSecureFactory. Each concrete secure factory provides an implementation of the `getObject()` method.
- SomeObject – The abstract SomeObject class defines the basic interface implemented by the objects returned by the secure factory.
- ConcreteObjectN – A concrete implementation of SomeObject will be created for each set of object behaviors dictated by the current security information. For example, if an application classifies users as having complete, little, or no trust, three concrete implementations of SomeObject will be created, one for each trust level. A concrete implementation of SomeObject will only contain functionality appropriate for the concrete implementation's corresponding trust level.

3.1.6 Consequences

- The security-credential dependent selection of the appropriate object is hidden from the portions of the system that make use of the selected object. The Secure Factory operates as a black box supplying the appropriate object to the caller. This hides the security dependent object selection logic from the caller.
- The objects created by the Secure Factory only need to implement functionality appropriate to their corresponding trust level. Functionality that is not appropriate for the object's corresponding trust level will not be implemented in the object.
- The objects created by the Secure Factory do not need to check to see if an action implemented in the object is allowed given information about the current user or operating environment. Those checks have already been performed by the Secure Factory that created the object.
- The black box nature of the Secure Factory secure design pattern makes it easy to change the security credential dependent behavior of the system. Changes to the object selection logic or the provided objects themselves will require little or no changes to the code making use of the Secure Factory.

3.1.7 Implementation

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

1. Identify an object whose construction or choice depends on the level of trust associated with a user or operating environment. Define an abstract class or interface describing the general functionality supported by the object.
2. Implement the concrete classes that implement the trust level specific behavior of the object. One concrete implementation will be created for each identified trust level.
3. Use the basic Abstract Factory pattern as described in the Structure section to define the AbstractSecureFactory class.
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 factory that selects the appropriate object defined in step two given security credentials defined in step 4.

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-