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.

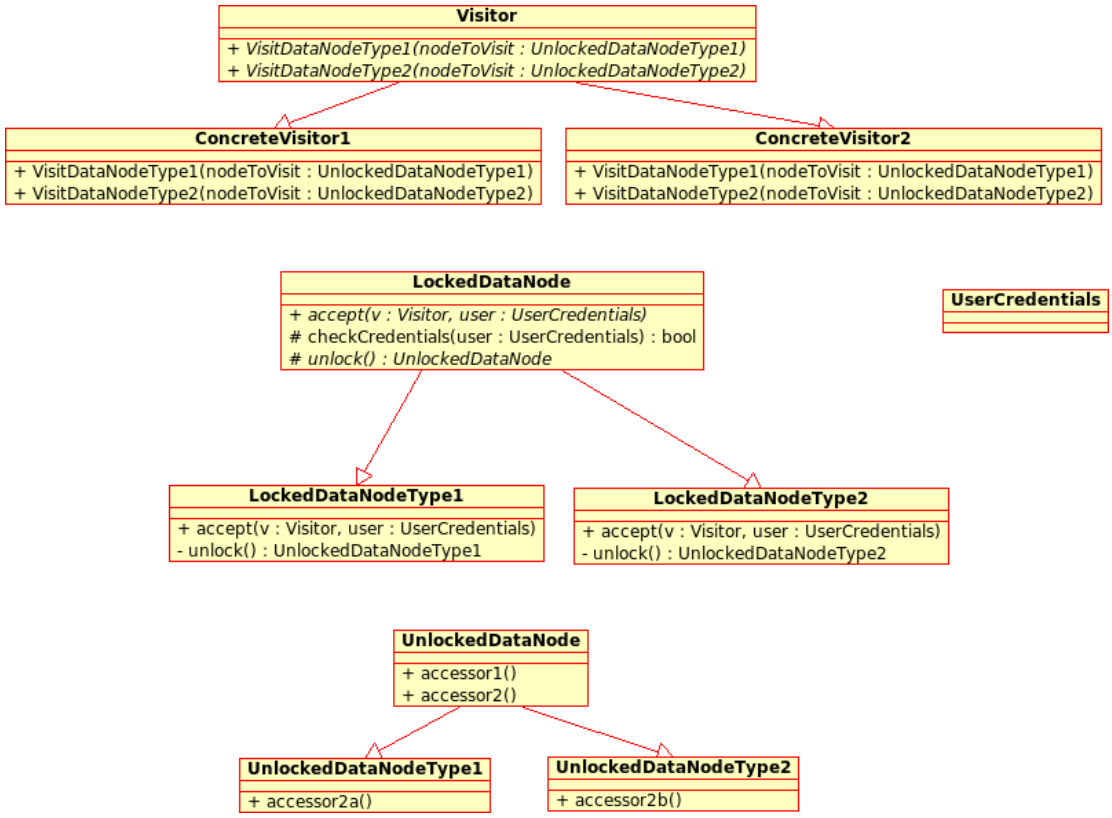

Figure 14: Secure Visitor Pattern Structure

3.6.5 Participants

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

- **Visitor** (HierarchicalDataVisitor). The Visitor participant in the secure visitor pattern is almost exactly the same as the Visitor participant in the standard Visitor pattern. The primary difference in the patterns is that the various visit methods take unlocked node objects in the Secure Visitor pattern, whereas the visit methods in the standard Visitor pattern simply take a node object (the standard Visitor pattern has no concept of locked and unlocked data nodes).
- **ConcreteVisitor**. As with the standard Visitor pattern, the ConcreteVisitor classes implement the operations defined in the abstract Visitor class.
- **LockedDataNode** (LockedDataNode). The LockedDataNode class defines an `accept()` operation that accepts a visitor. In addition, the LockedDataNode class also defines an operation for checking a user's credentials and for unlocking the current locked node. Note that a locked node presents no public operations for viewing the data in the node or changing the data in the node. All access to the node must be directed through the node's `accept()` operation. The `accept()` operation will check the user's credentials. If the credentials are valid for the user to view the data in the current node, the node will unlock itself using the `unlock()` operation and pass the unlocked version of itself to the visit method of the visitor.

- **LockedDataNodeTypeN** (LockedNodeType1). The LockedDataNodeTypeN classes implement the operations defined in the abstract LockedDataNode class. This includes the `unlock()` operation to unlock the various locked node objects and return the unlocked versions of the nodes.
- **UnlockedDataNode** (UnlockedDataNode). This class represents the unlocked version of a locked data node. The unlocked version of a node has some important characteristics:
  - It has no access to the parent(s) or children of its corresponding locked node. It only contains the data specific to the node itself, that is, the data that the user has been granted permission to see.
  - It has no `accept()` operation. The traversal of a hierarchical data structure with a secure visitor is done on the locked nodes, not the unlocked nodes.
- **UnlockedDataNodeTypeN** (UnlockedNodeType1). The concrete implementations of UnlockedDataNode implement the operations defined in the abstract class.
- **UserCredentials**. The UserCredentials represent the current user of the system and/or the permissions assigned to the current user. The Secure Visitor pattern does not place many restrictions on the specific implementation of the user credentials. The only requirement is that it is possible for a node to use the credentials to control access to the node's data.

3.6.6 Consequences

In addition to the set of consequences associated with the standard Visitor pattern, the Secure Visitor pattern has these additional consequences:

- *It clearly separates security mechanisms from user-level functionality.* The use of this pattern requires that the nodes in the data hierarchy, not the visitors themselves, implement security. This makes it easy to
  - test and verify the security aspects 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 implemented in the nodes in the data hierarchy and not in the various visitors of the data hierarchy, the security mechanism can be changed without requiring any modifications to the visitors.
- *It prevents programmatic access to the user-level functionality that avoids security.* Because the only way to access a locked node in the data hierarchy is via the `accept()` method of the Visitor pattern and the only class allowed to create an unlocked version of a node is its corresponding locked node, it is difficult or impossible to programmatically access the data in a node without supplying valid credentials for the node.

3.6.7 Implementation

In addition to the implementation considerations associated with the standard Visitor pattern, the Secure Visitor pattern has the following implementation consideration:

How is the data in a locked node protected? The goal of the Secure Visitor design pattern is to make it difficult to read the data in a locked node without supplying the appropriate credentials for the node. While the pattern itself makes it difficult to programmatically read a locked node data without the appropriate credentials, it still may be possible to read the raw bytes making up the locked node and thereby gain access to the data in the locked node. This implies that the data in the locked node must actually be "locked" in some manner. Data can be locked in a locked node using encryption or off-line storage.

- **Encryption**. The data in a locked node can be encrypted and only decrypted as part of the process of making an unlocked version of the node after accepting the credentials of a visitor.
- **Off-line storage**. The actual data in a locked node can be stored in some sort of an external, protected data management system like a database. The actual node data would only be loaded from the external source after accepting the credentials of a visitor.

3.6.8 Sample Code

The collaboration diagram in Figure 15 represents the basic behavior of the example code presented in this section:

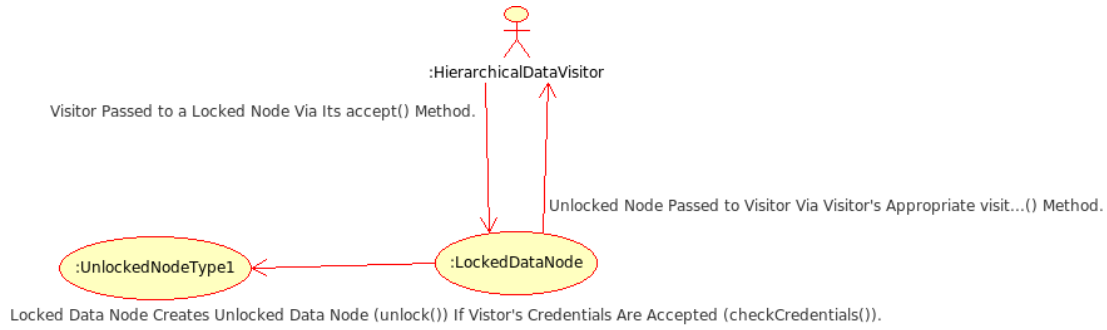

Figure 15: Secure Visitor Example Code Collaboration Diagram

This class represents the credentials of the user associated with a visitor; that is, the visitor is visiting the data in response to some action performed by the user represented by the given credentials. This class will not be sketched out in this example.

```
class UserCredentials;
```

This is a forward declaration for an unlocked version of the locked node. This will be defined later in the example.

```
class UnlockedDataNode;
```

This is a forward declaration for the visitor of the locked nodes in the hierarchical data. This will be defined later in the example.

```
class HierarchicalDataVisitor;
```

This defines the general interface for a locked node in the Secure Visitor pattern. It looks just like the interface in the standard Visitor pattern.

```
class LockedDataNode {
```

```
public:
  virtual void accept(HierarchicalDataVisitor& visitor,
                      UserCredentials user);

private:

  // Each type of node will have some way of checking the visitor's
  // credentials to see if the user has permission to access the
  // node.
  virtual bool checkCredentials(UserCredentials user);
};
```

The visitor interface in the Secure Visitor looks very much like the visitor interface in the standard Visitor pattern. The only difference is that the various `visit…()` methods accept the unlocked version of a node, not the locked version.

```
class HierarchicalDataVisitor {

public:

  virtual ~HierarchicalDataVisitor()
  virtual void visitNodeType1(UnlockedNodeType1 *node);

protected:

  HierarchicalDataVisitor();
};
```

Each concrete node in the data hierarchy has both a locked and unlocked version. Only a locked node will be able to create an unlocked node.

```
class LockedNodeType1 : LockedDataNode {

public:

  // The only way to access the data in the data hierarchy in the
  // Secure Visitor pattern is via the accept() method that accepts a
  // node visitor and the current user's credentials.
  void accept(HierarchicalDataVisitor& visitor,
              UserCredentials user);

private:

  // If the locked node accepts the visitor's credentials, it will
  // create an unlocked version of itself to pass to the visitor for
  // processing. Only a LockedDataNode can create an
  // UnlockedDataNode.
  UnlockedNodeType1 unlock();

  // Each type of node will have some way of checking the visitor's
  // credentials to see if the user has permission to access the
  // node.
  bool checkCredentials(UserCredentials user);
```

```
  // Track the children of the node somehow...
  // ...
};
```

The accept method for a locked node in the data hierarchy checks the user's credentials and unlocks the node and passes it on to the visitor if the credentials are valid for the node.

```
void LockedNodeType1::accept(HierarchicalDataVisitor& visitor,
                             UserCredentials user) {

  // Are the credentials valid for this node?
  if (checkCredentials(user)) {

    // The user has access to this node. Unlock the node and pass it
    // on to the visitor.
    visitor.visitNodeType1(unlock());
  }

  // Visit the children of the node...
  // ...
}
```

Note that the constructor for an unlocked node is private and that the corresponding locked node class is its friend. This means that an unlocked node can be created only by a locked node.

```
class UnlockedNodeType1 {

public:

  ...Data access methods, etc. ...

private:
  UnlockedNodeType1();
  friend class LockedNodeType1;
}
```