36

STRATEGY

This pattern was previously described in GoF95.

DESCRIPTION

The Strategy pattern is useful when there is a set of related algorithms and a client object needs to be able to dynamically pick and choose an algorithm from this set that suits its current need.

The Strategy pattern suggests keeping the implementation of each of the algorithms in a separate class. Each such algorithm encapsulated in a separate class is referred to as a *strategy*. An object that uses a `Strategy` object is often referred to as a *context object*.

With different `Strategy` objects in place, changing the behavior of a `Context` object is simply a matter of changing its `Strategy` object to the one that implements the required algorithm.

To enable a `Context` object to access different `Strategy` objects in a seamless manner, all `Strategy` objects must be designed to offer the same interface. In the Java programming language, this can be accomplished by designing each `Strategy` object either as an implementer of a common interface or as a subclass of a common abstract class that declares the required common interface.

Once the group of related algorithms is encapsulated in a set of `Strategy` classes in a class hierarchy, a client can choose from among these algorithms by selecting and instantiating an appropriate `Strategy` class. To alter the behavior of the context, a client object needs to configure the context with the selected strategy instance. This type of arrangement completely separates the implementation of an algorithm from the context that uses it. As a result, when an existing algorithm implementation is changed or a new algorithm is added to the group, both the context and the client object (that uses the context) remain unaffected.

STRATEGIES VERSUS OTHER ALTERNATIVES

Implementing different algorithms in the form of a method using conditional statements violates the basic object-oriented, open-closed principle. Designing each algorithm as a different class is a more elegant approach than designing all

different algorithms as part of a method in the form of a conditional statement. Because each algorithm is contained in a separate class, it becomes simpler and easier to add, change or remove an algorithm.

Another approach would be to subclass the context itself and implement different algorithms in different subclasses of the context. This type of design binds the behavior to a context subclass and the behavior executed by a context subclass becomes static. With this design, to change the behavior of the context, a client object needs to create an instance of a different subclass of the context and replace the current `Context` object with it.

Having different algorithms encapsulated in different `Strategy` classes decouples the context behavior from the `Context` object itself. With different `Strategy` objects available, a client object can use the same `Context` object and change its behavior by configuring it with different `Strategy` objects. This is a more flexible approach than subclassing.

Also, sometimes subclassing can lead to a bloated class hierarchy. We have seen an example of this during the discussion of the Decorator pattern. Designing algorithms as different `Strategy` classes keeps the class growth linear.

STRATEGY VERSUS STATE

From the discussion above, the Strategy pattern looks very similar to the State pattern discussed earlier. One of the differences between the two patterns is that the Strategy pattern deals with a set of related algorithms, which are more similar in what they do as opposed to different state-specific behavior encapsulated in different `State` objects in the State pattern.

Table 36.1 provides a detailed list of similarities and differences between the State and the Strategy patterns.

EXAMPLE

During the discussion of the Decorator pattern we designed a decorator class `EncryptLogger` that encrypts an incoming message before sending it to the `FileLogger` instance it contains for logging. For encrypting the message text, the `EncryptLogger` calls its `encrypt(String)` method. The encryption algorithm implemented inside the `encrypt(String)` method is very simple in that the characters of the message text are all shifted to the right by one position.

In general, there are many different ways of encrypting a message text using different encryption algorithms. Let us consider four different encryption algorithms including the simple encryption used by the `EncryptLogger` in the existing design.

`SimpleEncryption`

When this encryption is applied, characters in the plain text message are shifted to the right or left by one position.

Table 36.1 State versus Strategy

State Pattern	*Strategy Pattern*
Different types of possible behavior of an object are implemented in the form of a group of separate objects (`State` objects).	Similar to the State pattern, specific behaviors are modeled in the form of separate classes (`Strategy` objects).
The behavior contained in each `State` object is specific to a given state of the associated object.	The behavior contained in each `Strategy` object is a different algorithm (from a set of related algorithms) to provide a given functionality.
An object that uses a `State` object to change its behavior is referred to as a `Context` object. A `Context` object needs to change its current `State` object to change its behavior.	An object that uses a `Strategy` object to alter its behavior is referred to as a `Context` object. Similar to the State pattern, for a `Context` object to behave differently, it needs to be configured with a different `Strategy` object.
Often, when an instance of the context is first created, it is associated with one of the default `State` objects.	Similarly, a context is associated with a default `Strategy` object that implements the default algorithm.
A given `State` object itself can put the context into a new state. This makes a new `State` object as the current `State` object of the context, changing the behavior of the `Context` object.	A client application using the context needs to explicitly assign a strategy to the context. A `Strategy` object cannot cause the context to be configured with a different `Strategy` object.
The choice of a `State` object is dependent on the state of the `Context` object.	The choice of a `Strategy` object is based on the application need. Not on the state of the `Context` object.
A given `Context` object undergoes state changes. The order of transition among states is well defined. These are the characteristics of an application where the State pattern could be applied. **Example:** A bank account behaves differently depending on the state it is in when a transaction to withdraw money is attempted. When the minimum balance is maintained — no transaction fee is charged. When the minimum balance is not maintained — transaction fee is charged. When the account is overdrawn — the transaction is not allowed.	A given `Context` object does not undergo state changes. **Example:** An application that needs to encrypt and save the input data to a file. Different encryption algorithms can be used to encrypt the data. These algorithms can be designed as `Strategy` objects. The client application can choose a strategy that implements the required algorithm.

Example:

Plain text:This is a message
Cipher text:eThis is a messag

CaesarCypher

In its simplest form, the Caesar cipher is a rotation-substitution cipher where characters are shifted to the right by one position. It involves replacing the letter A with B, B with C, and so on, up to Z, which is replaced by A. This is called the rotate-1 Caesar cipher because it involves rotating the alphabet in the plain text by one position.

Example:

Plain text:This is a message
Cipher text:Uijt jt b nfttbhf

Similarly, a rotate-2 Caesar cipher replaces letter A with C, B with D, … Z with B.

Julius Caesar is known to have used this simple rotate-*n* replacement cipher and hence the name *Caesar cipher*.

SubstitutionCypher

This encryption algorithm uses a letter substitution table to replace different letters in the plain text with corresponding entries from the substitution table.

Table 36.2 shows an example letter substitution table.

To encrypt a given plain text, look up letters from the plain text in the top row of the letter substitution table and replace it with the corresponding letter from the bottom row in the same column.

Example:

Plain text: This is a Message
Cipher text: mWNR NR T DXRRTnX

Table 36.2 Sample Letter Substitution Table

A	T	i	B	h	s	a	e	m	X	Y	M	P	C	g	F	Q	w	r	t
s	m	N	o	W	R	T	X	Y	A	B	D	F	I	n	d	i	a	U	S

CodeBookCypher

This algorithm involves replacing words from the plain text with corresponding word entries from a code-book table.

Table 36.3 shows an example code-book table.

Table 36.3 Sample Code-Book Table

This	Design
Is	Patterns
Book	CD
A	Are
Sun	Hello
True	Really
Moon	Country
Statement	Useful
Discovery	Old
Channel	Vaccum

To encrypt a given plain text message, look up every word from the plain text message in the first column of the code-book table and replace it with the corresponding word from the second column of the same row.

Example:
Plain text:This Is A True Statement
Cipher text:Design Patterns Are Really Useful

Let us suppose that clients of the `EncryptLogger` would like to be able to dynamically select and use any of the aforementioned encryption algorithms.

This requirement can be designed in different ways, including:

- Implementing all algorithms inside the existing `encrypt(String)` method of the `EncryptLogger` class using conditional statements
- Applying inheritance, with each subclass of the `EncryptLogger` implementing a specific encryption algorithm

Though these options look straightforward, as discussed earlier under the "Strategies versus Other Alternatives" section, applying the Strategy pattern results in a more elegant and efficient design.

Applying the Strategy pattern, each of the encryption algorithms can be encapsulated in a separate (*strategy*) class (Listing 36.1 through Listing 36.4). Table 36.4 shows the list of these strategy classes and the algorithms they implement.

Let us define a common interface to be implemented by each of the strategy classes, in the form of a Java interface `EncryptionStrategy`, as follows:

```
public interface EncryptionStrategy {
  public String encrypt(String inputData);
}
```

Figure 36.1 shows the resulting class hierarchy.

Listing 36.1 `SimpleEncryption` Class

```
public class SimpleEncryption implements EncryptionStrategy {
  public String encrypt(String inputData) {
    inputData = inputData.substring(inputData.length() - 1) +
                inputData.substring(0, inputData.length() - 1);
    return inputData;
  }
}
```

Listing 36.2 `CaesarCypher` Class

```
public class CaesarCypher implements EncryptionStrategy {
  public String encrypt(String inputData) {
    char[] chars = inputData.toCharArray();
    for (int i = 0; i < chars.length; i++) {
      char c = chars[i];
      if (c == 'z') {
        c = 'a';
      }
      if ((c >= 'a') && (c < 'z')) {
        ++c;
      }
      chars[i] = c;
    }
    return new String(new String(chars));
  }
}
```

Each of the strategy classes listed in Table 36.4 provides the implementation of the algorithm it represents as part of the `encrypt` method declared by the `EncryptionStrategy` interface.

Because all of the strategy classes listed in Table 36.4 share the same interface, a client object that is designed to use an object of the `EncryptionStrategy` type will be able to access the encryption services offered by different strategy objects in a seamless manner.

Listing 36.3 `SubstitutionCypher` Class

```
public class SubstitutionCypher implements EncryptionStrategy {
  char[] source = {'a','b','c','d','e','f','g','h','i','j','k',
                   'l','m','n','o','p','q','r','s','t','u','v',
                   'w','x','y','z'};
  char[] dest = {'m','n','o','p','q','r','a','b','c','d','e',
                 'f','g','h','i','j','k','l','y','z','s','t',
                 'u','v','w', 'x'};
  public String encrypt(String inputData) {
    char[] chars = inputData.toCharArray();
    for (int i = 0; i < chars.length; i++) {
      char c = chars[i];
      for (int j = 0; j < source.length; j++) {
        if (source[j] == chars[i]) {
          c = dest[j];
        }
      }
      chars[i] = c;
    }
    return new String(chars);
  }
}
```

With each encryption algorithm encapsulated in a separate strategy class, the `EncryptLogger` is no longer required to contain any implementation to encrypt an input message. The idea is that the `EncryptLogger` can make use of the services of any of the strategy objects as required to encrypt a message. To facilitate the usage of different strategy objects by the `EncryptLogger` in a seamless manner, the `EncryptLogger` needs to be redesigned (Figure 36.2 and Listing 36.5) so that:

- It contains an object reference variable `currEncryptionStrategy` of the `EncryptionStrategy` type. This variable is used to hold its current encryption strategy.
- It configures itself with the default encryption `Strategy` object when it is first created.
- It offers a method `setEncryptionStrategy` to enable a client object to configure it with a different `Strategy` object.
- As part of the `encrypt` method implementation, it accesses the encryption services offered by the `EncryptionStrategy` object that it is configured

Listing 36.4 `CodeBookCypher` Class

```
public class CodeBookCypher implements EncryptionStrategy {
  HashMap codeContents = new HashMap();
  private void populateCodeEntries() {
    codeContents.put("This”,"Design");
    codeContents.put("is”,"Patterns");
    codeContents.put("a”,"are");
    codeContents.put("true”,"really");
    codeContents.put("statement”,"useful");
    //……
    //……..
  }
  public String encrypt(String inputData) {
    populateCodeEntries();
    String outStr = "";
    StringTokenizer st = new StringTokenizer(inputData);
    while (st.hasMoreTokens()) {
      outStr = outStr + " " +
               codeContents.get(st.nextToken());
    }
    return new String(outStr);
  }
}
```

Table 36.4 Different Encryption Strategies

Strategy	*Encryption Algorithm*
`CaesarCypher`	Caesar
`CodeBookCypher`	Code-Book
`SimpleEncryption`	Basic
`SubstitutionCypher`	Substitution

with. In other words, the implementation of its `encrypt` method transforms to a simple method call to the `encrypt` method of its current encryption `Strategy` object stored in the `currEncryptionStrategy` instance variable.

Figure 36.3 shows the overall class association.

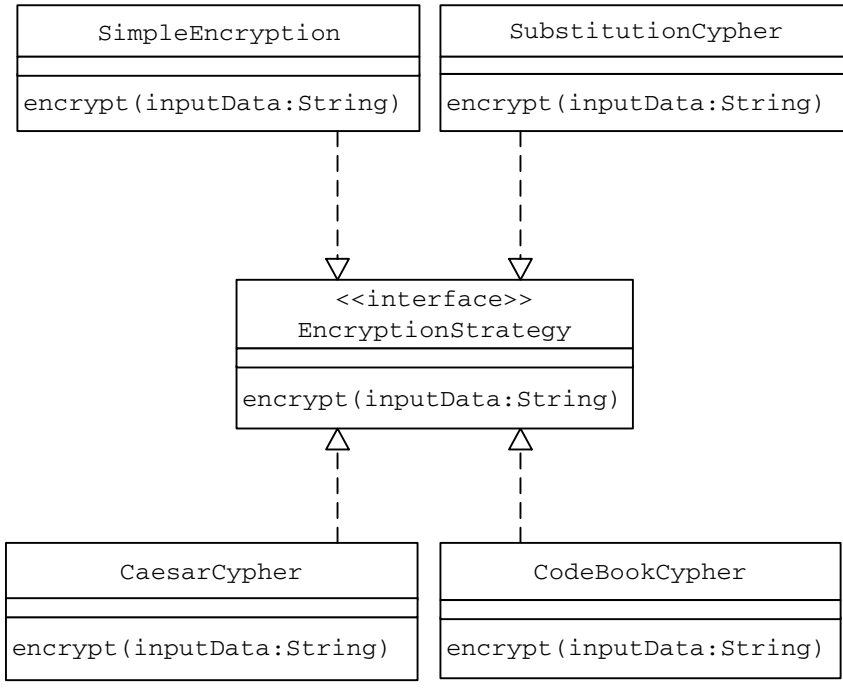

Figure 36.1 `EncryptionStrategy` Class Hierarchy

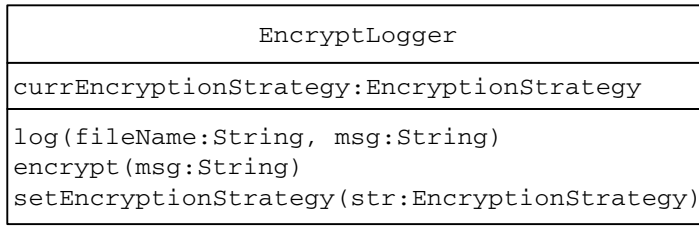

Figure 36.2 `EncryptLogger`

Note: The `EncryptLogger` contains an object reference of `FileLogger` type. This relationship is not included in the Figure 36.3 class diagram as it is not part of the pattern implementation.

The `EncryptLogger` uses different `Strategy` objects and hence acts as the context. A client object such as the `LoggerClient` that wants to log an encrypted message needs to create an instance of the `EncryptLogger` and invoke its `log` method. When the `EncryptLogger` is first instantiated, its current encryption strategy is set to `SimpleEncryption` inside its constructor. The `EncryptLogger` uses this strategy until the client explicitly changes the strategy to be used.

The client can create a different `Strategy` object and set it to be used as the current strategy by passing it to the `EncryptLogger` as part of the

Listing 36.5 `EncryptLogger` Class: Revised

```
public class EncryptLogger {
  private EncryptionStrategy currEncryptionStrategy;
  private FileLogger logger;
  public EncryptLogger(FileLogger inp_logger) {
    logger = inp_logger;
    //set the default encryption strategy
    setEncryptionStrategy(new SimpleEncryption());
  }
  public void log(String fileName, String msg) {
    /*Added functionality*/
    msg = encrypt(msg);
    /*
      Now forward the encrypted text to the FileLogger
      for storage
    */
    logger.log(fileName, msg);
  }
  public String encrypt(String msg) {
    /*
      Apply encryption using the current encryption strategy
    */
    return currEncryptionStrategy.encrypt(msg);
  }
  public void setEncryptionStrategy(
    EncryptionStrategy strategy) {
    currEncryptionStrategy = strategy;
  }
}
```

`setEncryptionStrategy` method call. The `EncryptLogger` uses this new strategy until again changed by the client.

```
class LoggerClient {
  public static void main(String[] args) {
    FileLogger logger = new FileLogger();
    EncryptLogger eLogger = new EncryptLogger(logger);
    eLogger.log("log1.txt",
                "this message is to be encrypted & logged");
    EncryptionStrategy strategy = new SubstitutionCypher();
```

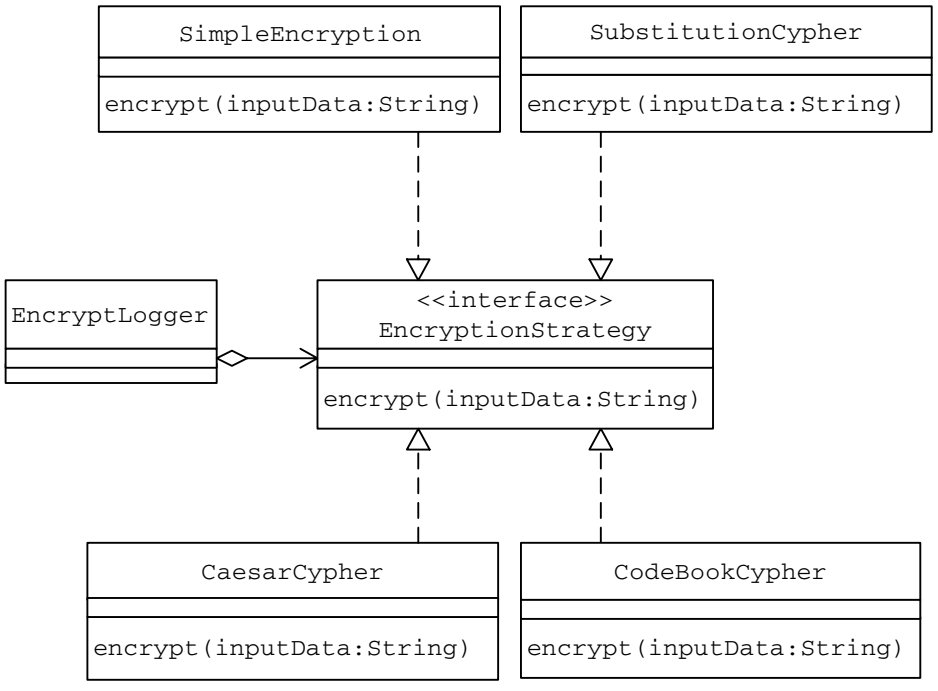

Figure 36.3 Class Association

```
        eLogger.setEncryptionStrategy(strategy);
        eLogger.log("log2.txt",
                    "this message is to be encrypted & logged");
        strategy = new CodeBookCypher();
        eLogger.setEncryptionStrategy(strategy);
        eLogger.log("log3.txt","This is a true statement");
      }
    }//End of class
```

In the new design, the `EncryptLogger` (the context) is not affected when changes such as adding, changing or removing an algorithm are made. In addition, making such changes will be simpler as each algorithm is contained in a separate class.

The sequence diagram in Figure 36.4 depicts the message flow when a client uses the `CodeBookCypher` to encrypt a message.

PRACTICE QUESTIONS

1. Identify how the Strategy pattern is used when you build an applet setting its layout manager.
2. Design and implement an application to search for an item from a list of items. The application should decide the search algorithm to be used and

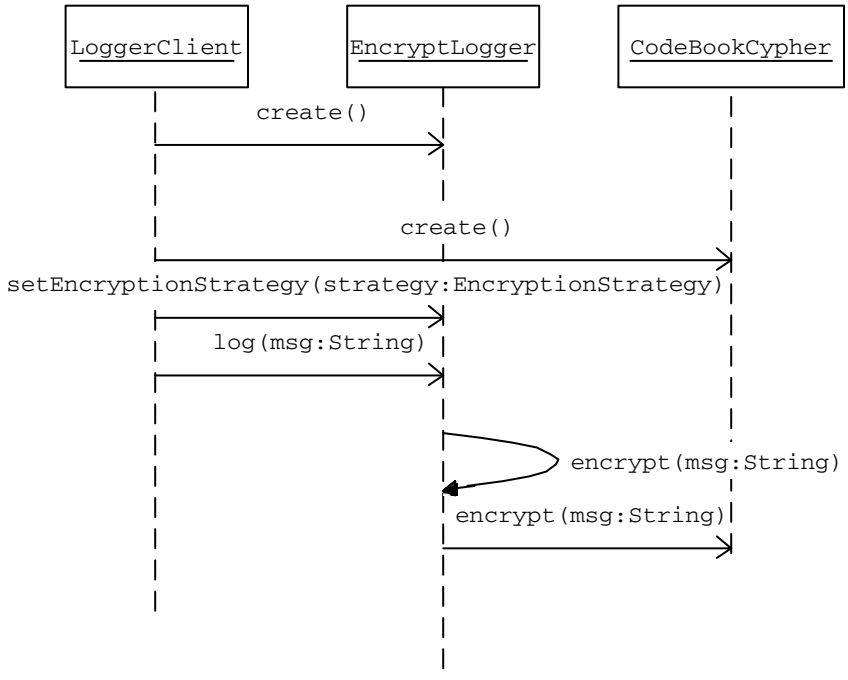

Figure 36.4 Client Object Using `CodeBookCypher` for Encryption

configure a search manager object (context) with this algorithm. For example, if the list is already sorted, the application should use the binary search algorithm as opposed to the linear search algorithm. Implement each algorithm as a different `Strategy` class.

3. The tax calculation varies from state to state in the United States. Design an application using the Strategy pattern to calculate taxes for different states in the United States.
4. Design an application that calculates simple and the compound interest. Identify the advantages and disadvantages of using the Strategy pattern in this case, compared to other alternatives.