The Mutator Pattern

Mirko Raner*
Parasoft Corporation
mirko@parasoft.com

July 31, 2006

Abstract

The Mutator pattern is a simple behavioral pattern that applies a series of successive modifications to a mutable object. The Mutator pattern is similar to the Iterator pattern but has significant advantages in situations where a small modification to an existing object is more efficient than creating a new object from scratch.

Introduction

The discovery of the Mutator pattern began with a bug that was particularly hard to track down. The method in which the bug occurred was supposed to execute a sequence of unit tests that were all generated from a common test case template and subsequently collected in a repository. The sequence of test cases was made available by means of an Iterator pattern [GHJV95]. However, instead of executing a number of different variations of the common template, the code apparently executed always the very last variation in each iteration (example code written in Java 5; see [AGH05]):

```
import java.util.*;
public class TestCaseExecutor
{
    public void executeTestCases(Iterator<ConcreteTestCase> testCases,
    Comparator<ConcreteTestCase> executionOrder)
    {
        List<ConcreteTestCase> testList
        testList = new ArrayList<ConcreteTestCase>();
        while (testCases.hasNext())
        {
            testList.add(testCases.next());
        }
        Collections.sort(testList, executionOrder);
        Iterator<ConcreteTestCase> execution = testList.iterator();
        while (execution.hasNext())
        {
            execution.next().run();
        }
    }
}
```

After a long search, the cause of the problem was found: the client code assumed that the iterator that was passed as an argument would return a sequence of distinct objects, but that was not the case:

```
import java.util.*;
class ConcreteTestCaseIterator implements Iterator<ConcreteTestCase>
{
    private TestCaseTemplate template;
    private TestCaseParameters[] parameters;
    private ConcreteTestCase testCase;
    private int index;

    ConcreteTestCaseIterator(TestCaseTemplate template,
    TestCaseParameters[] parameters)
    {
        this.template = template;
        this.parameters = parameters;
        testCase = new ConcreteTestCase(template, parameters[0]);
    }

    public boolean hasNext()
    {
        return index < parameters.length;
    }

    public ConcreteTestCase next()
    {
        if (!hasNext())
        {
            throw new NoSuchElementException();
        }
        if (index > 0)
        {
            testCase.setParameters(parameters[index]);
        }
        index++;
        return testCase;
    }

    public void remove()
    {
        throw new UnsupportedOperationException();
    }
}
```

As can be seen from the source code, the concrete test cases are not actually pre-existing objects. The code reuses a single `ConcreteTestCase` object for all iterations. A new `ConcreteTestCase` is instantiated only once. Subsequent iterations only modify the object's parameterization and then return the same object again.

Though `ConcreteTestCaseIterator` does implement the `Iterator` interface, it violates the expected semantics of the underlying Iterator pattern in several aspects. As an iterator is supposed to provide "a way to access the elements of an aggregate object sequentially" [GHJV95] client code will commonly assume that each iteration will return a distinct object (unless the original aggregate object indeed contained multiple references

to the same object). In the example, sorting the incoming test cases in a specific order does not have any effect because the `Collections.sort` method is applied to multiple references to the same object (rather than different objects). Also, the modification of the iterated objects is not part of the responsibilities of an iterator. Most client code will not be prepared to deal with iterators that modify the iterated objects (or any other objects).

However, a closer examination of this bug scenario showed that some valuable lessons could be learned. The reason for choosing such an unusual implementation for the iterator was that, in this particular case, creating a new `ConcreteTestCase` object was a much more expensive operation than changing the parameters of an existing object. Object creation is usually an expensive operation in an object-oriented system. Especially the creation of complex object graphs can be very inefficient. However, there are many situations where a series of objects is processed in a strictly sequential manner and where each object in the series is very similar to its predecessor. In those cases, it may be possible to use a single object that mutates its state so that it effectively at some point assumes the state of each object in the series. If the differences between successive objects are relatively small it is more efficient to create only a single object and then apply a succession of modifications to that object. The code shown above actually ran faster and used less memory than the straightforward implementation approach, which would have created all the objects in advance, stored them in a collection data structure and then used a standard iterator.

Though the optimization was an abuse of the Iterator pattern (for the reasons stated above), the optimized reuse of the same object was a valuable and useful new pattern by itself. Instead of returning a sequence of pre-created objects this pattern applies a series of modifications – or mutations – to a single object. This new pattern is therefore subsequently called the Mutator pattern.

In its simplest form, a mutator requires only two methods: a method `hasMoreMutations` that determines whether the mutator can apply additional mutations to a given object, and a method `applyNextMutation` that modifies the object so that its new state reflects the next logical mutation in the sequence. A mutator is very similar to an iterator. The main difference is that it does not return an object from an existing collection but modifies an object that was passed as a parameter.

The following sections contain a detailed description of the Mutator pattern, loosely based on the format introduced by the Gang of Four [GHJV95].

1 Intent

The intent of the Mutator pattern is to apply a series of successive modifications to a mutable object, specifically as an alternative to successively creating new object instances.

2 Problem

Certain algorithms can produce a large number of objects, which are then passed to some sort of client component that processes the incoming objects in a sequential manner. The algorithm that originally produces the objects often stores the newly created objects in a collection and then uses an iterator or a similar pattern to pass the objects to the clients. When the generated objects have large object graphs it can be very inefficient to generate new objects from scratch. Creating the objects from scratch also does not take advantage of possible similarities between successive objects in the sequence. Objects that are only processed once and then discarded also impose a heavy strain on the system memory and the garbage collector (if present).

3 Solution

Instead of maintaining a separate object for each iteration a single object is reused and successively modified. Thus, the overhead of object creation occurs only once, regardless of the number of elements. If the differences between two successive objects are sufficiently small and can be applied efficiently then the overall sequence of objects can be traversed much more quickly. Also, memory space for only a single object is required, and there is no need for repeated garbage collection or deallocation of already processed objects.

4 Applicability

The Mutator pattern is applicable when:

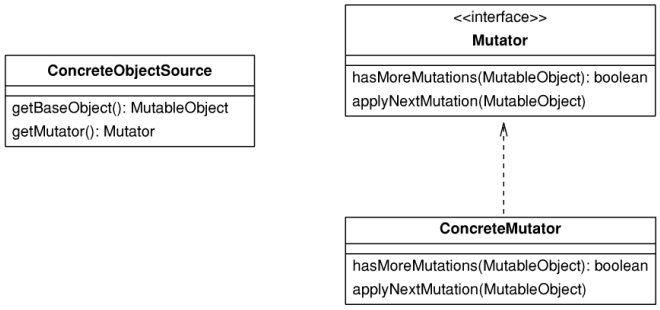
- an algorithm operates on a sequence of complex objects whose individual creation is rather expensive
- the objects are created on-the-fly, they do not exist yet
- the objects in the sequence are all relatively similar to each other
- the objects in the sequence are mutable, and applying minor modifications to an object is a relatively inexpensive operation
- the client algorithm processes the objects in a strictly sequential manner, i.e., after an object was processed by the algorithm that object is no longer needed; also, the algorithm must never require references to two or more objects from the series at the same time
- the object modifications may have different implementations but should still be accessible under a common interface

Potential scenarios where a Mutator pattern makes sense can be quite hard to identify. Some sophisticated profiling tools can pinpoint code that creates large numbers of new objects. If it also becomes apparent that all these objects are very similar then this may point to a possible candidate for the Mutator pattern. As a starting point, it can also be helpful to examine existing uses of the Iterator pattern and the aggregate

objects over which they iterate. If an aggregate object is only accessed via iterators after it was originally created and populated it may be beneficial to replace the aggregate object and its iterator by a mutator. To determine whether such a replacement is feasible, the client code has to be examined with respect to all the above listed criteria.

5 Structure

The Mutator design pattern has the following structure:



The client code that uses the mutator is not shown in the diagram. The generic Mutator interface has only two methods:

- `hasMoreMutations(MutableObject)`
  Determines whether a given object can be mutated by the mutator in its current state. If the mutator is capable of applying further mutations this method will return `true`; if the mutator has reached its end this method will return `false`
- `applyNextMutation(MutableObject)`
  Performs the given object's transition into its next mutation state. This method must be called only if the preceding invocation of `hasMoreMutations` returned `true`. After `hasMoreMutations` has already returned `false` this method will always fail, for example, by throwing an exception

6 Participants

In a typical Mutator pattern, the following participants are collaborating:

- **Mutator**
  defines the general interface for the mutator; this interface can also be a parameterized interface that can be bound to different types of mutable objects
- **ConcreteMutator**
  a concrete implementation of the interface; this class implements a specific series of mutations for a specific type of objects

- **MutableObject**
  provides an interface or parameterized type for mutable objects; as a fallback, a general type like, for example, `java.lang.Object` can be used
- **ConcreteObjectSource**
  the class that provides a base object and a mutator for that object (base object and mutator could also be provided in a different manner or from different sources, though)
- Client Code
  the code that processes the individual mutations of the object; this code is typically very similar to the client code for using an iterator

7 Collaborations

A concrete mutator implementation directly modifies the mutable object that was passed to it. The mutator may also carry additional internal information that determines the next mutation and keeps track of the sequence of mutations undergone so far.

8 Consequences

The use of the Mutator pattern has a number of beneficiary consequences:

- it saves time by eliminating the repetitive creation of objects
- it saves memory by using only a single mutable object

However, there are also some drawbacks to the Mutator pattern:

- it requires mutable objects, which can be more problematic to handle than immutable ones
- it has a sizeable list of prerequisites that limits its applicability
- it may require extensive restructuring of the code if one of its prerequisites suddenly no longer holds

The use of mutable objects often entails a number of problems. For example, mutable objects are unsafe as keys into hashed data structures and prone to issues of concurrent modification. References to mutable objects that participate in a Mutator pattern should be kept as local as possible.

The Mutator is applicable only in those situations that fulfill all of its prerequisites (see section 4). If one of the prerequisites can no longer be maintained a fairly large restructuring of the code may be necessary. In some cases such a restructuring may effectively cancel out the benefits of the Mutator pattern. The example of the `ConcreteTestCaseIterator` demonstrates such a problem: the `TestCaseExecutor` needs to sort the incoming test cases, which requires references to two test case objects at the same time for the purpose of comparison. This violates one of the Mutator's prerequisites, which in turn caused the described bug. Turning the code into a proper application of the Mutator pattern could prove very difficult here.

9 Implementation

Mutators can be implemented in a stateless or stateful fashion.

A stateless mutator carries no state information in addition to the mutable object that is passed. Stateless mutators either derive their termination condition solely from the mutated object or may produce mutations ad infinitum, in which case it is up to the client code how many mutations are requested.

Stateful mutators carry additional information, for example the number of mutations that was already applied. They can also store a reference to the mutated object. This allows for creating mutators that are specifically designed for a particular mutable object and may only be used on that particular object. For example, a stateful mutator could compare the passed mutated object with the internally stored reference and throw an exception if they do not match. If the mutated object is already passed to the mutator's constructor (and the mutator is only supposed to work on that particular object) the methods `hasMoreMutations` and `applyNextMutation` do not need a parameter that specifies the mutated object.

The basic mutator interface may also be extended to include methods for undoing the previous mutation or "rewinding" the mutated object to its original state.

10 Sample Code

In Java 5 [AGH05], a generic interface for the main participant of the Mutator pattern can be defined as follows:

```
public interface Mutator<MutableObject>
{
    boolean hasMoreMutations(MutableObject object);

    void applyNextMutation(MutableObject object);
}
```

A sample implementation that mutates a `StringBuffer` could look like this:

```
public class StringBufferMutator implements Mutator<StringBuffer>
{
    private int position;
    private int mutation;

    public boolean hasMoreMutations(StringBuffer buffer)
    {
        return (position < buffer.length()-1)
            || (position < buffer.length() && mutation < 2);
    }

    public void applyNextMutation(StringBuffer buffer)
    {
        switch (mutation)
        {
            case 1:
```

```
                buffer.setCharAt(position,
                    (char)(buffer.charAt(position)-2));
                mutation = 2;
                break;
            case 2:
                buffer.setCharAt(position,
                    (char)(buffer.charAt(position)+1));
                position++;
                /* fallthru */
            case 0:
                buffer.setCharAt(position,
                    (char)(buffer.charAt(position)+1));
                mutation = 1;
                break;
            default:
                throw new RuntimeException();
        }
    }
}
```

For each character in the `StringBuffer`, the mutator will first increase the character's value by 1 and then decrease it by 2 in the next mutation (effectively decreasing the original value by 1). In a practical application, this could be used for testing how a certain method reacts to slight variations of the original input.

The client code that would mutate the string "MUTATOR" would look like this:

```
public class Client
{
    public static void main(String[] arg)
    {
        StringBuffer buffer = new StringBuffer("MUTATOR");
        StringBufferMutator mutator = new StringBufferMutator();
        while (mutator.hasMoreMutations(buffer))
        {
            mutator.applyNextMutation(buffer);
            System.err.println(buffer);
        }
    }
}
```

For the example string "MUTATOR", the above code will produce these mutations: "NUTATOR", "LUTATOR", "MVTATOR", "MTTATOR", "MUUATOR", "MUSATOR", "MUTBTOR", "MUT@TOR", "MUTAUOR", "MUTASOR", "MUTATPR", "MUTATNR", "MUTATOS", and "MUTATOQ".

The same effect can, of course, also be achieved with an iterator:

```
import java.util.*;
public class StringBufferIterator implements Iterator<StringBuffer>
{
    private List<StringBuffer> sequence;
    private int index = 0;

    public StringBufferIterator(String original)
    {
        sequence = new ArrayList<StringBuffer>();
```

```
        sequence.add(new StringBuffer(original));
        for (int position = 0; position < original.length();
                                                        position++)
        {
            for (int mutation = -1; mutation <= 1; mutation += 2)
            {
               StringBuffer buffer = new StringBuffer(original);
               buffer.setCharAt(position,
                           (char)(buffer.charAt(position)-mutation));
               sequence.add(buffer);
            }
        }
    }

    public boolean hasNext()
    {
        return index < sequence.size();
    }

    public StringBuffer next()
    {
        return sequence.get(index++);
    }

    public void remove()
    {
        throw new UnsupportedOperationException();
    }
}

public class IteratorClient
{
    public static void main(String[] arg)
    {
        StringBufferIterator iterator;
        iterator = new StringBufferIterator("MUTATOR");
        while (iterator.hasNext())
        {
            System.err.println(iterator.next());
        }
    }
}
```

In contrast to the mutator, the iterator first creates all the different `StringBuffer` objects and then returns them in a straightforward fashion. Constructing the iterator may take a long time if many `StringBuffer`s have to be created. Also, all objects must be stored in memory at the same time.

The `StringBufferMutator` just provides a simple illustrative example; in practice, there would probably be little difference in efficiency if new `String` or `StringBuffer` objects were created from scratch. In typical real-world applications of the Mutator pattern, the creation of new objects is usually by orders of magnitude more expensive than the modification of an existing object.

11 Known Uses

Possible applications of the Mutator pattern include genetic algorithms, processing of large trees or graphs, as well as execution of parameterized unit tests [TS05] or unit test generation by means of permutation or perturbation (for an explanation of perturbation testing, see [OX04]).

For example, a genetic algorithm might have to examine a large number of tree structures to determine which one has the highest value according to a certain metric. The tree structures are generated according to a fixed set of rules, and whereas no two trees are exactly identical, the variations between two trees are typically very minor. In such a scenario, a mutator is likely to have time and memory advantages over an iterator.

12 Related Patterns

The Mutator pattern is closely related to the Iterator pattern, and the use of mutators is very similar to the use of iterators. These are the main differences between mutators and iterators:

	Mutator	Iterator
Number of pre-existing objects	one	one for each iteration
Source of object(s)	supplied by client code	supplied by aggregate that is being iterated
Method of iteration	implicit; by successive modification	explicit; as defined by aggregate
Concurrency safety	only if separate mutable objects are used	typically always
Applicable objects	only Value Objects	Value Objects and Reference Objects[1]

13 Conclusion

The Mutator pattern provides a good alternative to iterators and similar patterns in scenarios where, by using the Iterator pattern, a large number of similar objects are created from scratch and processed in a sequential fashion. By mutating a single object through a predefined series of states the Mutator pattern requires only a single object instance and replaces expensive object creation with less expensive object modification.

Before choosing the Mutator pattern to solve a particular problem, developers should make sure that all of the pattern's prerequisites are met and are not likely to be broken by future development of the code. Typical applications of the Mutator pattern include genetic algorithms and unit testing by means of permutation or perturbation.

[1] see [Fow03], pp. 73f. for an explanation of Value Objects versus Reference Objects.

Acknowledgements

The Mutator pattern was originally inspired by my professional work as a member of Parasoft's Jtest development team. The discovery of this pattern could not have happened without my daily interaction with fellow team members in San Diego (USA), Krakow (Poland), and Novosibirsk (Russia). Special thanks go to Philipp Bachmann of the Institute for Medical Informatics and Biostatistics in Basel (Switzerland) for shepherding my submission to PLoP 2006. Philipp's insightful comments and detailed suggestions greatly improved the quality and clarity of the description of this new pattern.

References

[AGH05] Ken Arnold, James Gosling, and David Holmes. *The Java Programming Language*. Addison-Wesley, 4th edition, 2005.

[Fow03] Martin Fowler. *UML Distilled – A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley, 3rd edition, 2003.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[OX04] Jeff Offutt and Wuzhi Xu. Generating test cases for web services using data perturbation. *ACM SIGSOFT Software Engineering Notes*, 29(5):1–10, 2004.

[TS05] Nikolai Tillmann and Wolfram Schulte. Parameterized unit tests. In *Proceedings of the 10th European Software Engineering Conference held jointly with the 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC-FSE'05)*, pages 253 – 262, Lisbon, Portugal, 2005.