# The Mutator Pattern

Mirko Raner
Parasoft Corporation
mirko@parasoft.com

## 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 operates only on a single object as opposed to a collection of objects. Mutators have a significant performance benefit 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 (also known as a "parameterized test case" [7]) and subsequently collected in a repository. The sequence of test cases was passed to the execution method by means of an Iterator pattern [4]. The method did execute the right number of test cases, but instead of using the different variations of the test case only the last variation in the sequence was executed over and over again.

After a long search, the cause of the problem was found: the execution code assumed that the supplied iterator would return a sequence of distinct objects, but that was not the case. In fact, the iterator returned the same test case object for all iterations. A new test case object was instantiated only once. Subsequent iterations only modified the original test case's parameter settings and then returned the same object again. Though this "iterator" did implement Java's `Iterator` interface, it violated 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" [4] 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). Also, the modification of the iterated objects is not part of the usual responsibilities of an iterator. Most client code will not be prepared to deal with "iterators" that modify the iterated objects (or any other objects, for that matter).

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 test case object was a much more expensive operation than changing the parameters of an existing test case object. Object creation is usually an expensive operation in an object-oriented system. Especially the creation of complex object graphs can be very time-consuming. 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.

Although the described optimization was an abuse of the Iterator pattern (for the reasons outlined above), the optimized reuse of the same object was a valuable and useful new pattern by itself. As demonstrated by the bug scenario, this pattern is, however, not an Iterator and must not be confused with one. Instead of returning a sequence of precreated objects the pattern applies a series of modifications – or mutations – to a single object. This new pattern is therefore subsequently called the Mutator pattern.

In its basic 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.

The following sections contain a detailed description of the Mutator pattern, loosely based on the format introduced by the Gang of Four [4] with additional elements from the format used by Alur et al. in "Core J2EE Design Patterns" [1] (Problem, Solution, and Forces sections).

A preliminary version of this paper was workshopped at Pattern Languages of Programming (PLoP) '06 October 21-23, 2006, Portland, OR, USA.
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission. Copyright is held by the author. ISBN: 978-1-50448-372-3

*Revision* : 1.7

## 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 produce a large number of objects, which are then passed to some sort of client component that processes these 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 garbage collector (if present).

Figure 1 shows a Java 5 [2] class that instantiates and executes a number of concrete test cases based on a test case template. It uses an `Iterator` to iterate over the list of test case parameter sets. The source shows a simplified example scenario, it is not an excerpt from actual production code:

```
import java.util.*;
public class IterativeTestCaseExecutor extends BaseTestCaseExecutor
{
    public void executeTestCases(TestCaseTemplate template,
    List<TestCaseParameters> parameterSets)
    {
        Iterator<TestCaseParameters> parameterSetIterator;
        parameterSetIterator = parameterSets.iterator();
        while (parameterSetIterator.hasNext())
        {
            TestCaseParameters parameterSet;
            parameterSet = parameterSetIterator.next();
            ParameterizedTestCase testCase;
            testCase = template.instantiate(parameterSet);
            testCase.runParameterized();
        }
    }
}
```

**Figure 1: Code example for test case execution using an iterator**

This example code works fine but has a serious performance problem. To understand this issue better, it is important to know some more details about the test case generation process. The test case template consists of a possibly very large tree structure that represents the test case's execution sequence (similar to an Abstract Syntax Tree). A small number of tree nodes are "parameterized", that is, their values can be varied to create different test cases from the common template. The `instantiate(TestCaseParameters)` method creates a new tree in which the parameterized nodes are replaced with concrete values from the `TestCaseParameters`. This operation is slow and creates a new concrete tree for each parameter set. Also, it is important to realize that two object trees that were created in direct succession will usually only differ in very few nodes (those nodes that contained different parameters).

## 3. FORCES

The Mutator pattern typically appears in the context of algorithms that sequentially process a number of objects. The pattern resolves the following forces:

- You want to reduce the number of objects that are being created in the context of a sequential processing loop
- You want to take advantage of similarities between two successive objects in the processing sequence
- You want to encapsulate an algorithm for successive object modifications into a separate object
- You want to provide a uniform interface for various different algorithms that perform successive object modifications

```
import java.util.*;
public class MutativeTestCaseExecutor extends BaseTestCaseExecutor
{
    public void executeTestCases(TestCaseTemplate template,
    List<TestCaseParameters> parameterSets)
    {
        Iterator<TestCaseParameters> parameterSetIterator;
        parameterSetIterator = parameterSets.iterator();

        // Create a "blank" instantiation of the template:
        //
        ParameterizedTestCase testCase;
        testCase = template.instantiate(null);

        // Create the mutator and apply the mutations:
        //
        TestCaseMutator mutator;
        mutator = new TestCaseMutator(parameterSetIterator);
        while (mutator.hasMoreMutations(testCase))
        {
            mutator.applyNextMutation(testCase);
            testCase.runParameterized();
        }
    }
}

class TestCaseMutator implements Mutator<ParameterizedTestCase>
{
    private Iterator<TestCaseParameters> parameterSets;

    public TestCaseMutator(Iterator<TestCaseParameters> parameters)
    {
        this.parameterSets = parameters;
    }

    public boolean hasMoreMutations(ParameterizedTestCase testCase)
    {
        return parameterSets.hasNext();
    }

    public void applyNextMutation(ParameterizedTestCase testCase)
    {
        testCase.applyParameters(parameterSets.next());
    }
}
```

**Figure 2: Code example for test case execution using a mutator**

## 4. SOLUTION

Instead of creating and 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 changes can be applied efficiently then the overall sequence of objects can be traversed much more quickly. Also, there is no need for repeated garbage collection or deallocation of already processed objects.

Using the Mutator pattern, the example from section 2 can be resolved as shown in figure 2. The original algorithm is replaced by a slightly different algorithm that only uses a single test case object which is brought into the right state and then executed. This new scenario is also further illustrated in figure 4.

## 5. 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 objects allow reuse (for example, executable objects must allow to be executed more than once)
- 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 or referenced, i.e., the algorithm must never require references to two or more objects from the series at the same time

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 once via an iterator (and not otherwise used) after it was originally created and populated then 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.

## 6. STRUCTURE

The Mutator design pattern has the structure shown in figure 3 (rendered in UML 2.0 [3]).

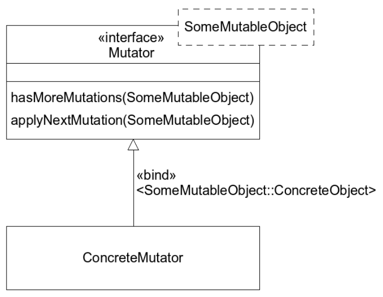
SomeMutableObject
«interface»
Mutator
hasMoreMutations(SomeMutableObject)
applyNextMutation(SomeMutableObject)
«bind»
<SomeMutableObject::ConcreteObject>
ConcreteMutator

**Figure 3: Mutator class diagram (UML 2.0)**

The client code that uses the mutator is not shown in the diagram. Concrete implementations of the `Mutator` interface will modify a specific class of objects. `SomeMutableObject` is just a type parameter which can be bound to any class or interface whose objects are mutable. In programming languages that do not support parameterized types (for example, older versions of Java) the `Mutator` interface can be defined using a general root class (for example, `java.lang.Object` in Java) as parameter to its methods. The generalization of the interface as shown in figure 3 is analogous to "pattern interfaces" like `java.util.Iterator` in Java. However, it is not necessary to have an abstract interface for the Mutator pattern at all: the performance benefit will still be there (but not the benefit of the common interface for different types of object modifications).

The generic Mutator interface has only two methods:

- `hasMoreMutations(SomeMutableObject)`
  Determines whether a given object can be mutated by the mutator depending on the mutator's and the object's current state. If the mutator is capable of applying further mutations this method will return `true`, otherwise it will return `false`
- `applyNextMutation(SomeMutableObject)`
  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`, otherwise it will fail, for example, by throwing an exception

## 7. 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

- **SomeMutableObject**
  defines the objects on which the mutator operates; as a fallback, a general type like, for example, `java.lang.Object` can be used

- Client Code *(not shown in figure 3)*
  the code that defines the object to be mutated and performs the further processing or transmission of the various object states; this code is typically very similar to the client code for using an iterator and may also involve a separate "processor" component

## 8. COLLABORATIONS

A concrete mutator implementation directly modifies the mutable object that was passed to it (usually by invoking one of the object's methods). The mutator may also carry additional internal information that determines the next mutation and keeps track of the sequence of mutations undergone so far. Figure 4 shows a typical scenario where a client uses a mutator to process a sequence of objects (which is, in fact, one and the same object traversing different states):

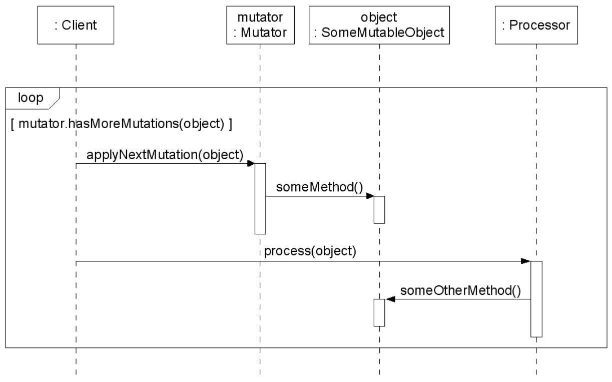

**Figure 4: Mutator sequence diagram (UML 2.0)**

It is important to note that the mutable object is provided by the client code and not created by the mutator. The client code can either create the mutable object and its mutator or can obtain those objects from somewhere else. The Processor component shown in the diagram is technically also part of the client code and could also be shown as part of the Client component in the diagram. Similar to an iterator, the pattern repeats a loop until the mutator can no longer apply any further mutations.

Another interesting aspect of the Mutator pattern is that during a mutation the mutated object's "ownership" is temporarily transferred from the client code to the mutator. The client code hands the object to the mutator and lets the mutator do whatever it deems necessary to that object. When control is transferred back from the mutator to the client code the object's state will be different.

## 9. CONSEQUENCES

The use of the Mutator pattern has the main beneficiary consequence that it saves time by eliminating the repetitive creation of similar objects. 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 limit its applicability (see section 5)

- 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 inherently 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 5). 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.

## 10. 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 the creation of mutators that are specifically designed for a particular mutable object instance 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 actually need the parameter that specifies the mutated object.

The basic mutator interface may also be extended to include methods for undoing the previous mutation, "rewinding" the mutated object to its original state, or determining the number of mutations that are left and the number of mutations that were already executed.

## 11. SAMPLE CODE

In Java 5 [2], a generic interface for the main participant of the Mutator pattern can be defined as follows:

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

    void applyNextMutation(SomeMutableObject 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 `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.

## 12. KNOWN USES

Currently, the only "known" and verifiable use of the Mutator pattern is in the execution system of Parasoft's Jtest [5], a commercial testing solution for Java.[1]

Possible applications of the Mutator pattern, however, include genetic algorithms, processing of large trees or graphs, as well as general execution of parameterized unit tests [7] or unit test generation by means of permutation or perturbation (for an explanation of perturbation testing, see [6]). 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 an advantage over an iterator.

## 13. 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[2] |

The Mutator pattern is also similar to the "Generator" pattern. The Generator pattern is not recorded in any of the major pattern catalogues, but the term is occasionally used to refer to a pattern that creates objects on-the-fly rather than returning pre-existing objects from a collection (which, in turn, makes the Generator pattern similar to the Factory pattern). The classic abstraction known as input stream or

[1]also, during the workshop session several participants mentioned that they had encountered the described pattern before

[2]see [3], pp. 73f. for an explanation of Value Objects versus Reference Objects.

a random number generator are instances of the Generator pattern. Again, the important difference between Mutator and Generator is that a mutator does not generate or produce any objects but instead modifies an existing object.

The Strategy pattern described in [4] also shares a number of common traits with the Mutator pattern. A mutator can be seen as encapsulating a "mutation strategy" that is applied in an iterative fashion. The original Strategy pattern, however, lacks this iterative aspect (though a strategy can obviously also be executed more than once). By combining the Strategy pattern with an Iterator pattern benefits similar to those of the Mutator can be achieved.

## 14. 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.

## 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). In particular, I wish to thank the participants of the PLoP 2006 workshop session who reviewed and discussed the original paper about the Mutator pattern (listed in no particular order): Kanwardeep Ahluwalia of Wipro Technologies (India); Sachin Bammi of Schlumberger Limited (USA); Maurice Rabb, Danny Dig, and Brian Foote of the University of Illinois at Urbana-Champaign (USA); Andrew Black of Portland State University (USA); Daniel Vainsencher of the Technion (Israel); Ademar Aguiar of the University of Porto (Portugal), and Ricardo Lopez of Qualcomm, Inc. (USA). 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. Last but not least, I'd like to thank Ward Cunningham for leading the workshop session about the Mutator pattern. Ward made sure that the workshop always stayed on track and steered the discussion into the right direction.

## 15. REFERENCES

[1] D. Alur, J. Crupi, and D. Malks. *Core J2EE Design Patterns – Best Practices and Design Strategies.* Prentice Hall, 2nd edition, 2003.

[2] K. Arnold, J. Gosling, and D. Holmes. *The Java Programming Language.* Addison-Wesley, 4th edition, 2005.

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

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[5] Jtest. http://parasoft.com/jsp/products/home.jsp?product=Jtest.

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

[7] N. Tillmann and W. 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.