The Memento Design Pattern

Daniel Geier

Abstract

Managing state restoration is a commonly occurring task in software development. Memento is a design pattern used for saving and restoring the (partial) state of objects. After examining reasons for using Memento, a comprehensive overview covering the static and dynamic structure is given. Examples from a C++ source are presented. Benefits, drawbacks and implementation details in C++ and `Java` are discussed.

1 Introduction

The average project size in software development is constantly growing and so is code complexity. On the technical side, design patterns, made popular by Gamma *et al.* [1], are an attempt to tame this complexity with a structured object-oriented approach. The impact of design patterns were critically examined in a number of empirical investigations (e.g., [2], [3], [4], [5], [6], [7]), which led to mixed conclusions on the benefits patterns may bring. Nonetheless patterns, when carefully used, can bring key advantages to the design and implementation process.

The analysis in this paper concentrates on the Memento design pattern as it is presented in [8].

2 Motivation

Why invent a pattern to manage state changes in the first place? Why not use a simpler approach?

When preserving the state of an object, the first technique that comes to mind is copying the variables that represent the state and copy them back to the object at a later time. Though seductive in its simplicity, this technique breaks encapsulation. Another possible solution, letting the stateful object manage its previous states by itself, violates separation of concern.

Memento works by splitting the burden of managing state restoration between multiple classes, thus preserving good development practices.

3 Structure

The structure of a pattern can broken down into a *static structure* and a *dynamic structure*. The *static structure* is composed of the object-oriented classes, their member variables, and their functions while the interactions between these static elements constitute the *dynamic structure*.

3.1 Static Structure

Three classes make up Memento: `Originator`, `Caretaker` and `Memento`.

`Originator` is the class whose state will be encapsulated by our memento.

`Memento` encapsulates this state. It provides set and get mechanisms for this state which are accessible only to the originator. This is achieved by defining a *narrow* interface for the caretaker and a *wide* one for the originator.

`Caretaker` is the user of the originator. It is responsible for storing the mementos returned by the originator and returning them back to it. The caretaker never accesses the memento directly.

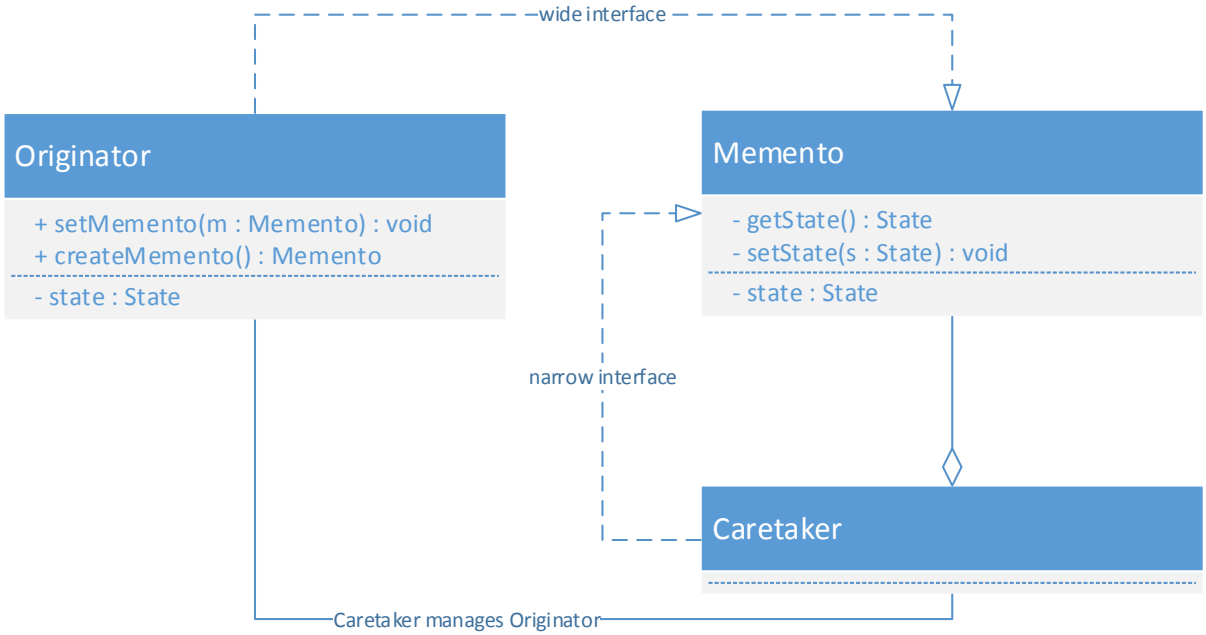

Figure 1: Class diagram showing the static structure with classes, their methods and class variables used

As one can see in figure 1, `Originator` implements the method `createMemento` for creating a memento, while `setMemento` is used to load a previous state. `Memento` implements accessors for its state, which should only be accessible to the `Originator`. `Caretaker` doesn't implement any methods specific to the pattern.

3.1.1 A *Wide* and a *Narrow* Interface

The key to Memento lies in implementing two different interfaces for `Originator` and `Caretaker`. The details of `Memento` should be opaque to all but `Originator` which accesses `Memento` through a *wide* interface. This way, `Originator` can access the state of `Memento` while all other participants interact with `Memento` through an *narrow* interface, which leaves it opaque to them, i.e.,

they cannot access `Memento`'s state or call its methods.

This poses difficulties in programming languages who lack mechanism to define such constructs. Section 4.3 discusses mechanisms in C++ and `Java` for defining those interfaces.

3.2 Dynamic Structure

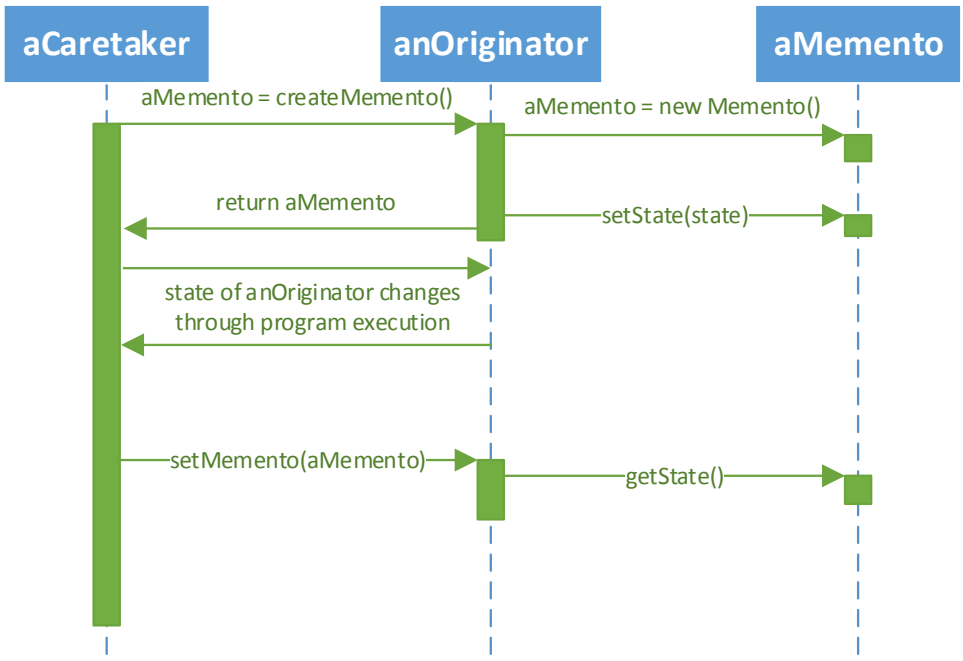

Figure 2: Sequence diagram showing the dynamic structure through method calls being made

Figure 2 shows how the pattern is used in practice. `aCaretaker` receives `aMemento` from `anOriginator` and saves it through the subsequent program execution. As soon as `aCaretaker` wants to restore `anOriginator` to a previous state, it returns `aMemento` back to `anOriginator` through the `setMemento` method.

4 Sample Code

The sample application is a Mandelbrot fractal explorer written in C++. We use a c-value of 1 to obtain the most famous of the Mandelbrot fractals. The fractal drawing and rendering is managed in the `Mandelbrot` class. It is responsible for calculating the values of the Mandelbrot set and saving them in an array of pixels.

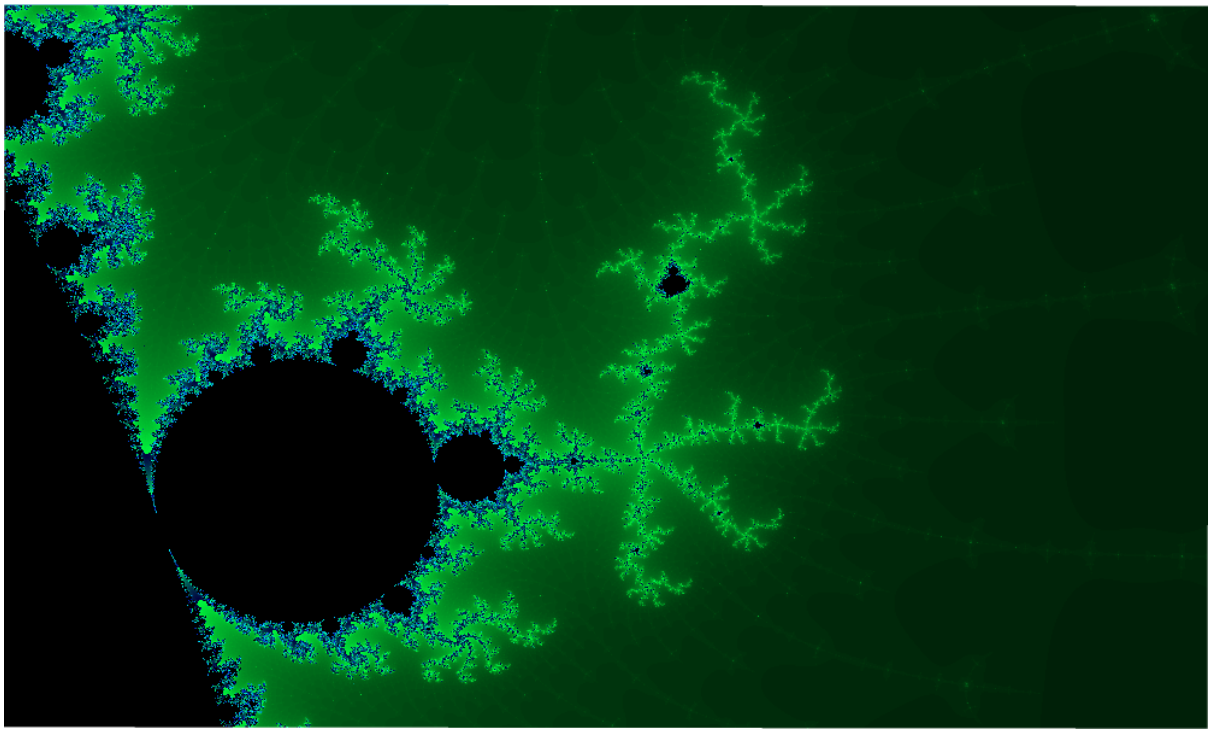

Figure 3: Part of the Mandelbrot figure as rendered by our program.

4.1 About Fractals

A fractal is a phenomenon that displays repeating patterns at every level of scale. The Mandelbrot fractal in specific is the set of complex numbers c for which the sequence $c_{n+1} = c_n + c$ does not converge to infinity for $n \to \infty$. Observations about the intrinsics involved in the calculation of the Mandelbrot and related fractals are made by Mandelbrot himself [9].

4.2 Code

The Memento pattern is used for saving the state of the calculation and the calculated image in Listings 1 and 2.

Listing 1: Mandelbrot.h

```
1 class Mandelbrot
2 {
3 public:
4         Memento* createMemento();
5         void setMemento(Memento* memento);
6
7         /* ... */
```

```
8
9  private:
10         // Wide interface for the memento
11         friend class Memento;
12
13         // Variables saved within the memento
14         Dimensions bounds;
15         uint32_t* pixels;
16
17         // Independent state variables
18         int iterations;
19         int width, height;
20         bool doCalc;
21
22         /* ... */
23         };
```

Listing 2: Mandelbrot.cpp

```
1  Memento* Mandelbrot::createMemento()
2  {
3          Memento* m = new Memento();
4          m->setPixels(pixels, width * height);
5          m->setBounds(bounds);
6          return m;
7  }
8
9  void Mandelbrot::setMemento(Memento* memento)
10 {
11         // Copy pixels from memento
12         memcpy(pixels,memento->getPixels(),
13                 width * height * sizeof(uint32_t));
14         bounds = memento->getBounds();
15         doCalc = false;
16 }
```

In Listings 3 and 4, the `Memento` class saves the state which consists of the image data, `pixels`, and the logical position of the Mandelbrot section, `bounds`. A wide interface to `Mandelbrot` is achieved by declaring it as a `friend`.

Listing 3: Memento.h

```
1  class Memento
2  {
3  public:
```

```
4           ~Memento();
5
6   private:
7           friend class Mandelbrot;
8
9           uint32_t* pixels;
10          Dimensions bounds;
11
12          Memento();
13          void setPixels(uint32_t* oldPixels, int size);
14          void setBounds(Dimensions bounds);
15
16          uint32_t* getPixels();
17          Dimensions getBounds();
18  };
```

Listing 4: Memento.cpp

```
1   void Memento::setPixels(uint32_t* oldPixels, int size)
2   {
3           pixels = new uint32_t[size];
4           // Copy pixels from mandelbrot
5           memcpy(pixels, oldPixels,
6                   size * sizeof(uint32_t));
7   }
8
9   void Memento::setBounds(Dimensions oldBounds)
10  {
11          bounds = oldBounds;
12  }
13
14  Memento::~Memento()
15  {
16          delete[] pixels;
17  }
18
19  uint32_t* Memento::getPixels()
20  {
21          return pixels;
22  }
23
24  Dimensions Memento::getBounds()
25  {
```

```
26          return bounds;
27  }
```

We want to allow the user to zoom into the fractal. For this, she has to select a portion of the image by drawing a rectangle. Subsequently `zoomIn()` is called.

Listing 5 show how a Memento is created and put aside in dynamic storage before zooming in. The memento will restored at a later point to zoom out of the image.

Listing 5: Main.cpp

```
1  void zoomIn() {
2          // mementos is a list of previous mementos
3          mementos.push_back(mandelbrot.createMemento());
4          mandelbrot.zoomIn(zoomRect);
5          mandelbrot.render(renderer, screenTexture);
6  }
```

The user may want to zoom out of the image to explore another part of the fractal. She can do so by clicking a button whereupon `zoomOut()` is called. In Listing 6 a previously stored memento is retrieved from storage. The state of the `Mandelbrot` object is restored.

Listing 6: Main.cpp

```
1  void zoomOut() {
2          if (mementos.size() > 0) {
3                  // Retrieve last memento
4                  Memento* m = mementos.back();
5                  mementos.pop_back();
6
7                  mandelbrot.setMemento(m);
8                  mandelbrot.render(renderer,
9                          screenTexture);
10
11                 delete m;
12         }
13 }
```

4.3 Implementation Details

Not every programming language has features to facilitate the creation of wide and narrow interfaces. We examine two languages, Java and C++, that provide such features and show how they are used.

In Java, *static nested classes* can access private members of its surrounding class and vice versa. In such a way the memento can be implemented as a static nested class of the originator class. Listing 7 shows this concept in practice.

Listing 7: Memento.java

```
class Originator {

        private String data;

        public Memento getMemento() {
                return new Memento(data);
        }

        public void setMemento(Memento m) {
                this.data = m.data;
        }

        // This is a static nested class -  its private members
        // can only be accessed by its outer class.
        // In this context, 'static' means that a Memento object
        // can be instantiated independently from an Originator
        // object.
        static class Memento {
                private String data;

                private Memento(String data) {
                        this.data = data;
                }
        }
}
```

In C++ a class that is marked with the `friend` keyword may access private members of the class. Hence the memento has to declare the originator as a `friend`, as does the originator with the memento. How to do so in practice is shown in Section 4.2.

5 Discussion

Now that we have examined Memento in theory and practice we can discuss when its suitable to use, when we can fit it to a problem and when something else can be used to a greater effect.

5.1 Benefits

Encapsulation. Encapsulation has been shown to provide key benefits in the understandability and changeability of a computer program [10]. As already explored in Section 2, simply copying member variables is unsuitable for maintaining good development practices; exposing state variables involves confiding implementation-specific details of the class. The memento object encapsulates the state of the originator. Inaccessible for all but the originator, the state is safe

from unsound modifications from outside objects.

Separation of concerns. Before using Memento, the originator had to manage all state-restoring functionality by itself to maintain proper encapsulation. With this functionality moved into Memento, the originator is simplified. An in-depth treatment on separation of concerns is [11].

5.2 Drawbacks

Memory overhead. Depending on how easily the state can be refactored out of the originator, there may be a substantial memory overhead in creating a memento.

Defining different interfaces. Some programming languages may lack facilities for declaring both a narrow and wide interface.

Hidden costs. As the implementation of the memento is hidden from the caretaker, it doesn't know how much state the memento is managing. It will therefore be hard to assess the memory consumption of a possibly otherwise lightweight caretaker.

5.3 Comparison with Other Patterns

A similar pattern, is *Memoization*. *Memoization* is most often used for program-speed optimization. It is the simple storing of function values to avoid repetitively calculating the same values over and over again. This is feasible if a time-consuming function has a manageable number of possible inputs and outputs; The most common outputs can be saved in an appropriate data structure like a hash table.

6 Conclusions

We have seen that Memento can be useful for managing state restoration.

Though a number of drawbacks limit the usefulness of Memento, wisely used it can be an asset for suitable applications. Just as much as any other design strategy, the use of a pattern has to be weighed carefully against the added complexity it adds to the project. Only when a right fit of pattern to problem is determined, the pattern unfolds its potential.

References

[1] Erich Gamma, Richard Helm, Ralph E Johnson, and John M Vlissides. Design patterns: Abstraction and reuse of object-oriented design. In *Proceedings of the 7th European Conference on Object-Oriented Programming*, pages 406–431. Springer-Verlag, 1993.

[2] Sebastien Jeanmart, Yann-Gael Gueheneuc, Houari Sahraoui, and Naji Habra. Impact of the visitor pattern on program comprehension and maintenance. In *Proceedings of the 2009*

3rd International Symposium on Empirical Software Engineering and Measurement, pages 69–78. IEEE Computer Society, 2009.

[3] Foutse Khomh and Y-G Guéhéneuc. Do design patterns impact software quality positively? In *12th European Conference on Software Maintenance and Reengineering, 2008. CSMR 2008.*, pages 274–278. IEEE, 2008.

[4] Massimiliano Di Penta, Luigi Cerulo, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. An empirical study of the relationships between design pattern roles and class change proneness. In *IEEE International Conference on Software Maintenance, 2008. ICSM 2008.*, pages 217–226. IEEE, 2008.

[5] Lutz Prechelt, Barbara Unger, Walter F. Tichy, Peter Brossler, and Lawrence G. Votta. A controlled experiment in maintenance: comparing design patterns to simpler solutions. *IEEE Transactions on Software Engineering*, 27(12):1134–1144, 2001.

[6] Marek Vokac. Defect frequency and design patterns: An empirical study of industrial code. *IEEE Transactions on Software Engineering*, 30(12):904–917, 2004.

[7] Marek Vokáč, Walter Tichy, Dag IK Sjøberg, Erik Arisholm, and Magne Aldrin. A controlled experiment comparing the maintainability of programs designed with and without design patterns—a replication in a real programming environment. *Empirical Software Engineering*, 9(3):149–195, 2004.

[8] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Pearson Education, 1994.

[9] Benoit B Mandelbrot. Fractal aspects of the iteration of $z \to \lambda z(1-z)$ for complex λ and z. *Annals of the New York Academy of Sciences*, 357(1):249–259, 1980.

[10] Alan Snyder. Encapsulation and inheritance in object-oriented programming languages. *SIGPLAN Not.*, 21(11):38–45, June 1986.

[11] Walter L. Hürsch and Cristina Videira Lopes. Separation of concerns. Technical report, College of Computer Science, Northeastern University, February 1995.