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.* [4], 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., [1], [9], [6], [7], [2], [10], [12], [13]), 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 [3].

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.[1]

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

[1] Section 5.1 has more details on the benefits of encapsulation and the separation of concerns.

3 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.

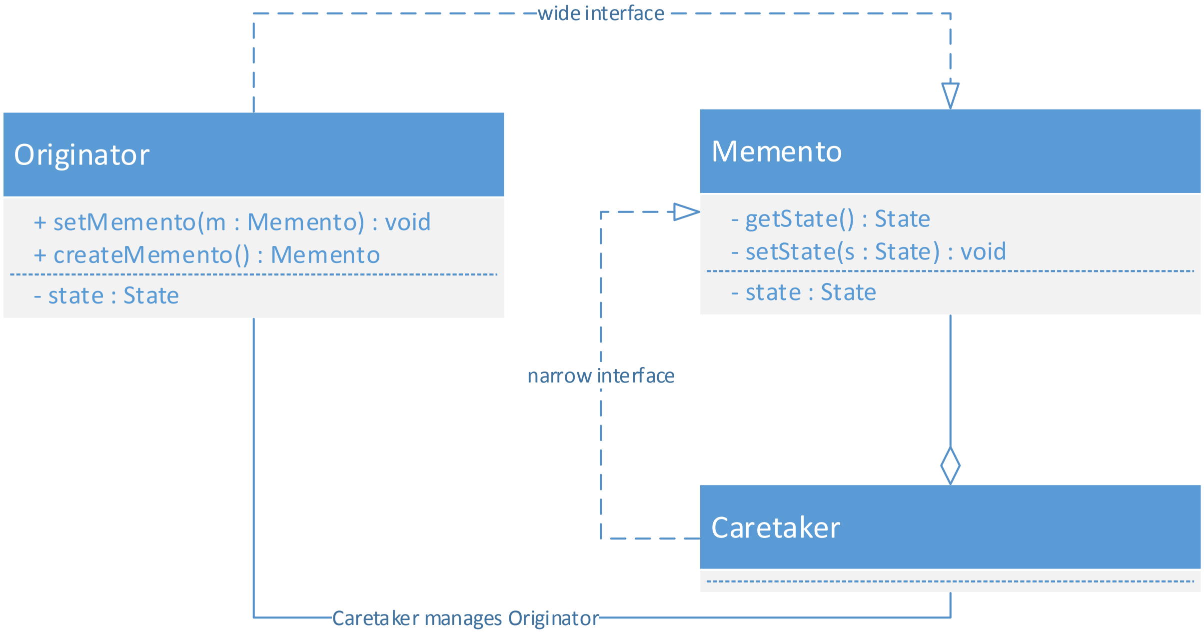

Figure 1: Class diagram showing the static structure with classes, their methods and class variables used

As one can see in figure 1, `Originator` implements the method `createMemento` for creating a memento, while `setMemento` is used to load the 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.

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. All other participants interact with `Memento` through an *narrow* interface, which leaves `Memento` opaque to them. This poses difficulties in programming languages who lack mechanism to define such constructs. Section 4.1 discusses mechanisms in C++ and `Java` for defining those interfaces.

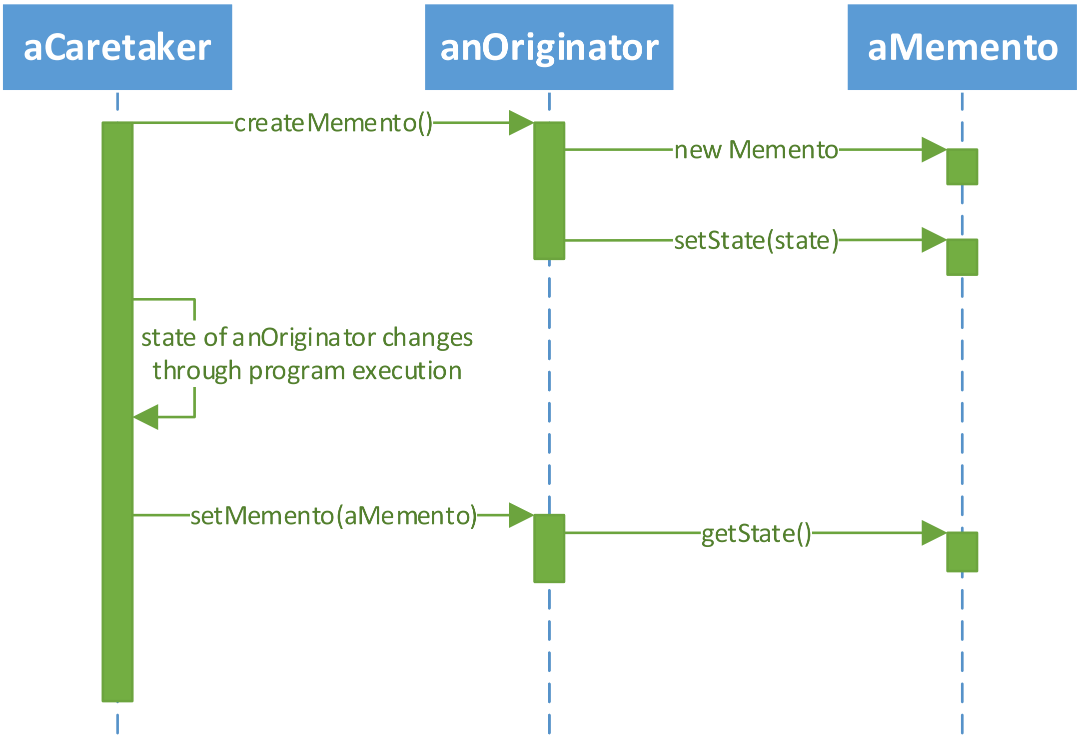

Figure 2: Sequence diagram showing the dynamic structure through method calls being made

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[2] fractal explorer written in C++. 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 a array of pixels.

Listing 1: Mandelbrot.h

```
        class Mandelbrot
        {
        public:
                Memento* createMemento();
```

[2]Observations about the intrinsics involved in the calculation of the Mandelbrot and related fractals are made in [8].

```
		void setMemento(Memento* memento);

		/* ... */

	private:
		// Wide interface for the memento
		friend class Memento;

		// Variables saved within the memento
		Dimensions bounds;
		uint32_t* pixels;

		// Independent state variables
		int iterations;
		int width, height;
		bool doCalc;

		/* ... */
	};
```

The Memento pattern is used for saving the state of the calculation and the calculated image.

Listing 2: Mandelbrot.cpp

```
	Memento* Mandelbrot::createMemento()
	{
		Memento* m = new Memento();
		m->setPixels(pixels, width * height);
		m->setBounds(bounds);
		return m;
	}

	void Mandelbrot::setMemento(Memento* memento)
	{
		// Copy pixels from memento
		memcpy(pixels,memento->getPixels(),
			width * height * sizeof(uint32_t));
		bounds = memento->getBounds();
		doCalc = false;
	}
```

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 a `friend` (see Section 4.1).

Listing 3: Memento.h

```
	class Memento
	{
	public:
		~Memento();

	private:
		friend class Mandelbrot;

		uint32_t* pixels;
		Dimensions bounds;

		Memento();
		void setPixels(uint32_t* oldPixels, int size);
		void setBounds(Dimensions bounds);

		uint32_t* getPixels();
		Dimensions getBounds();
	};
```

Listing 4: Memento.cpp

```
	void Memento::setPixels(uint32_t* oldPixels, int size)
	{
		pixels = new uint32_t[size];
		// Copy pixels from mandelbrot
		memcpy(pixels, oldPixels,
			size * sizeof(uint32_t));
	}

	void Memento::setBounds(Dimensions oldBounds)
	{
		bounds = oldBounds;
	}

	Memento::~Memento()
	{
		delete[] pixels;
	}

	uint32_t* Memento::getPixels()
	{
		return pixels;
```

```
        }

        Dimensions Memento::getBounds()
        {
                return bounds;
        }
```

In the main event loop, `zoomIn` is called, when the eponymous action is to be performed. A memento is created and put aside in dynamic storage.

Listing 5: Main.cpp

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

`zoomOut` is called, when the user wants to return to the previous section of the fractal. A memento is retrieved from storage and the state of the `Mandelbrot` object restored.

Listing 6: Main.cpp

```
        void zoomOut() {
                if (mementos.size() > 0) {
                        // Retrieve last memento
                        Memento* m = mementos.back();
                        mementos.pop_back();

                        mandelbrot.setMemento(m);
                        mandelbrot.render(renderer,
                                screenTexture);

                        delete m;
                }
        }
```

4.1 Implementation Details

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.

In C++ the `friend` keyword signals that the stated class 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.

5 Discussion

5.1 Benefits

Encapsulation. Encapsulation has been shown to provide key benefits in the understandability and changeability of a computer program [11]. 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[3]. 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.

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.

References

[1] James M Bieman, Greg Straw, Huxia Wang, P Willard Munger, and Roger T Alexander. Design patterns and change proneness: An examination of five evolving systems. In *Software metrics symposium, 2003. Proceedings. Ninth international*, pages 40–49. IEEE, 2003.

[2] 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 *Software Maintenance, 2008. ICSM 2008. IEEE International Conference on*, pages 217–226. IEEE, 2008.

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

[3] An in-depth treatment on separation of concerns is [5].

[4] 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.

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

[6] 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.

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

[8] 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.

[9] Gerardo Cepeda Porras and Yann-Gaël Guéhéneuc. An empirical study on the efficiency of different design pattern representations in uml class diagrams. *Empirical Software Engineering*, 15(5):493–522, 2010.

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

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

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

[13] 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.