Memento Pattern

Anupam Srivastava
Object Oriented Analysis and Design

April 28, 2019

Contents

1 Introduction

Memento pattern is used to save and restore the state of an object. Memento pattern falls under behavioral pattern category. It naturally lends itself to implementing undo/redo functionality.

2 Definition

Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.[1]

3 Description

The memento pattern is used under the following constraints:

- State of an object must be stored externally
- Object's encapsulation must not be violated

This is solved by making the object itself responsible for consolidating the state information (Originator), making it encapsulated (Memento) and then saving it externally (Caretaker).

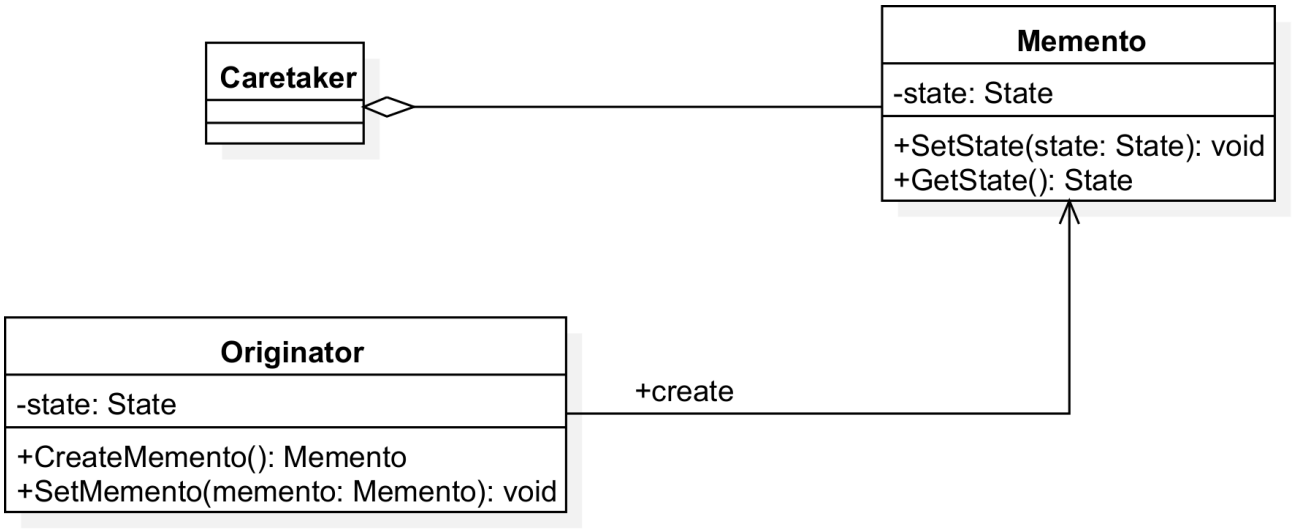

Figure 1: Class diagram

From figure 1 on page 2, we can see that **Caretaker** has the business logic to know when to save state of the **Originator**, and accordingly, it asks for a **Memento** from the **Originator**. **Originator** creates a **Memento** by storing its state information in it then passes this **Memento** to **Caretaker** who stores it for later use. At the time of restoration (such as undo or redo), **Caretaker** provides this **Memento** object to **Originator** so that it can restore itself to stored state.[2]

4 Example

4.1 Real life scenario from *Aristocrat Technologies, Inc.*

4.1.1 Introduction

Aristocrat Technologies is a gaming company that primarily sells slot machines. In slot games, a typical game of round starts with the user pressing the spin button followed by slots stopping in sequence and at the end providing a *win* if winning symbols have landed.

Aristocrat Technologies uses C++ for developing such games. For the mobile platform, C++ code is converted to JavaScript using `asmjs` library.

Scenario A user might already see a winning symbol in 1st slot and expect a win even before rest of the slots have landed. If the game is interrupted due to external or internal factors (player accidentally closes the game or due to a crash etc.), the player does not expect to lose the win that already looked imminent. Also, at no point, any external entity should be able to read or tamper with the state of the game.

Expectation As soon as the state of a round is known, the game should be able to save it. Whenever a game is launched, it should restore itself to the previous state. To maintain the fairness of the gamble, the game state should remain private.

Solution Memento pattern is used to save and restore the game state. The game state is stored on disk the moment it becomes final. No complex data type is stored, but all stored data is serialized to ease and optimize the retrieval from disk.

4.1.2 Components

1. **GameClient** - *Caretaker*
2. **GameEngine** - *Originator*
3. **ValueDictionary** - *Memento*

4.1.3 Description

GameClient requests GameEngine to give a memento of type ValueDictionary, which is similar to a hash map, by calling SaveState() function at appropriate times, such as when the Stopping symbols of each slot have been determined, or when an amount is credited or debited from a user's account. GameEngine then stores all such information in a ValueDictionary by either directly storing it as String or by asking complex data type objects (such as Scene objects) to return a ValueDictionary of their own state. All the types are eventually converted to a serialized string and stored in the hash map. The final hash map object is the memento.[1]

GameClient stores this memento on a persistent data storage such as a disk or a remote server. When the game is restored, such as when the machine is rebooted or when the player

[1]While memento's API should not be public, encapsulation is maintained by the *key* of the hash map which is only known to Originator, i.e., GameEngine.

switches back to the game on hardware where it was not possible to save game in-memory (such as a mobile device with low RAM), GameClient reads the data from persistent data-storage and passes it to GameEngine. GameEngine unwinds the hash map to get all the relevant data and restores its state accordingly.

4.1.4 UML Diagrams

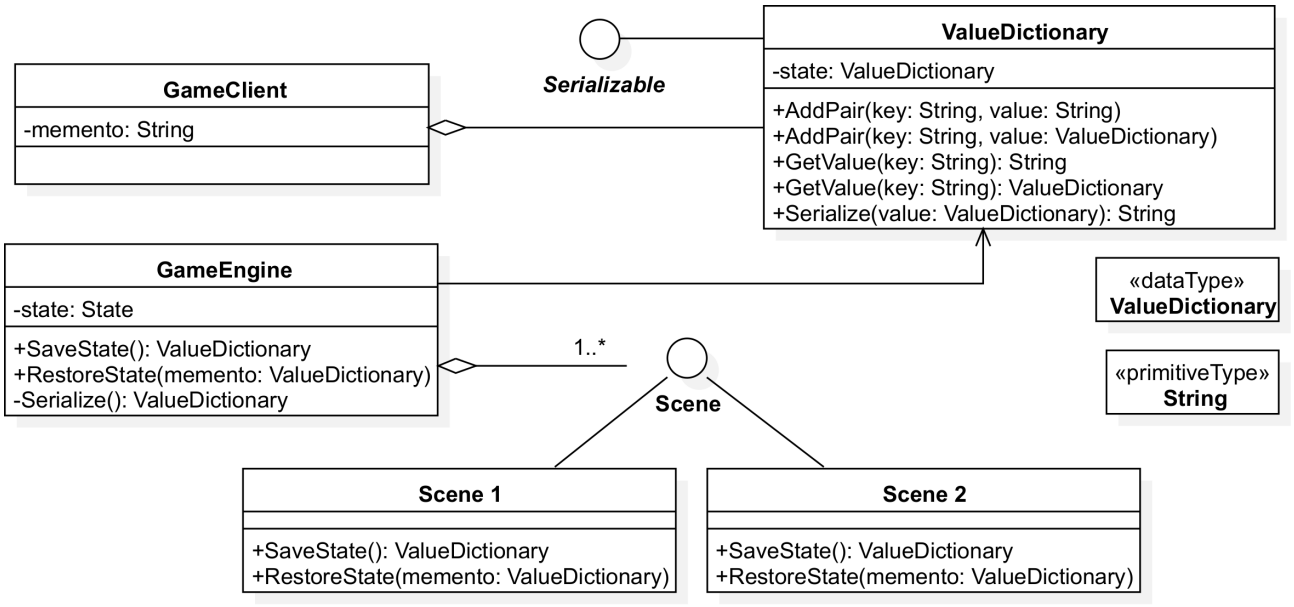

Figure 2: Class diagram for saving game state

Figure 2 shows the class diagram. Figure 3 shows the sequence diagram of storing a memento. Figure 4 shows the sequence diagram for restoring from a memento. Figure 5 and Figure 6 show the activity diagram for storing and restoring of states.

4.2 Agreement with GRASP guidelines

GRASP stands for *G*eneral *R*esponsibility *A*ssignment *S*oftware *P*atterns. It is a set of guidelines for achieving good object-oriented design. The different patterns used in GRASP are:

4.2.1 Creator

The responsibility of creating an object remains with the class most closely related to it. The game state is saved by GameEngine by creating a ValueDictionary. Conversely, while restoring the game state it is GameClient that reads the persistent data and creates ValueDictionary.

4.2.2 Information Expert

The information of state is contained and remains with the GameEngine. Neither GameClient or ValueDictionary can deduce the state of a game by themselves.

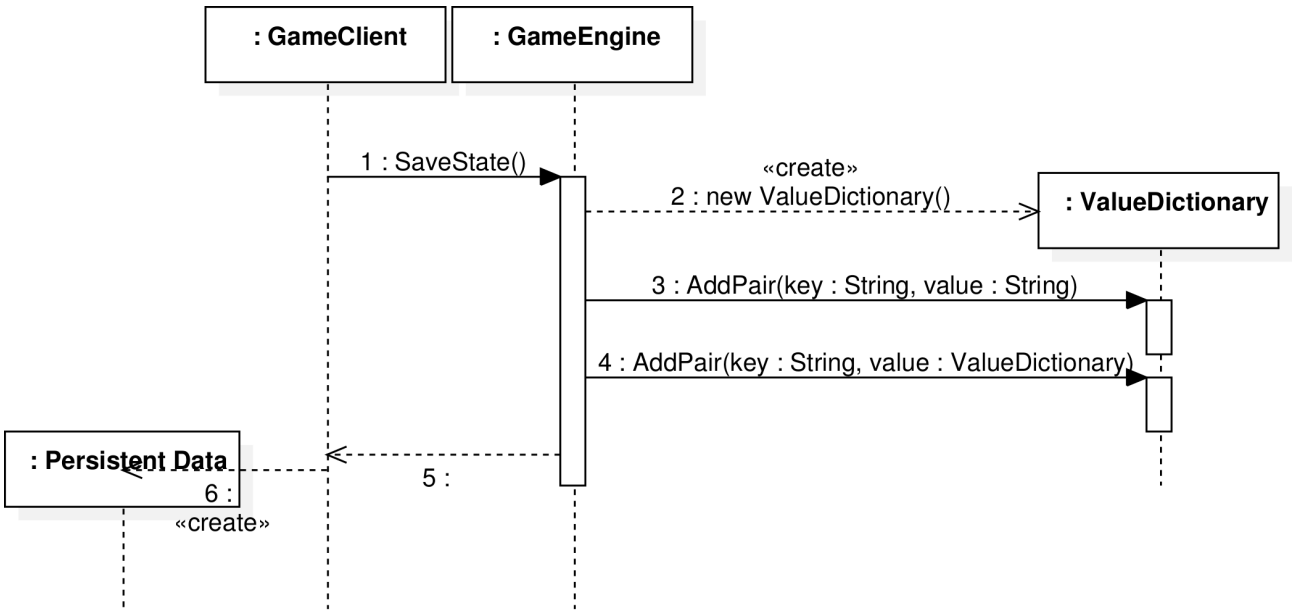

Figure 3: Sequence diagram for saving game state

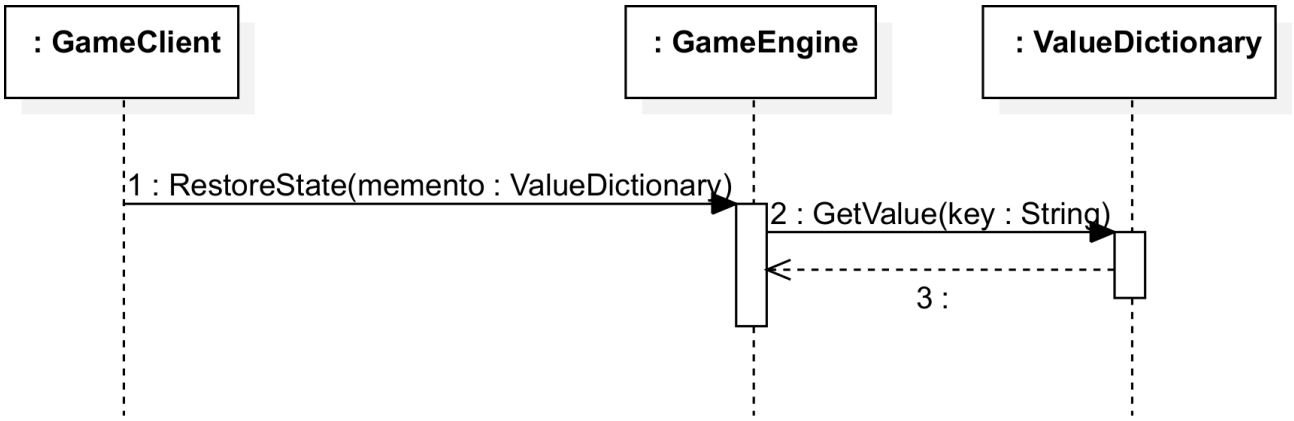

Figure 4: Sequence diagram for restoring the game state

4.2.3 Low Coupling

All components work independently from each other.

a) GameClient is independent of the type of Memento if it can call Serialize() on it. It only needs a stream to write to persistent data.

b) ValueDictionary implements the *Serializable* interface so that it can convert the data in the hash-map into a string(-stream). It remains independent of the data stored in the hash-map.

c) GameEngine saves and restores from ValueDictionary but is not dependent on when and where it needs to do it.

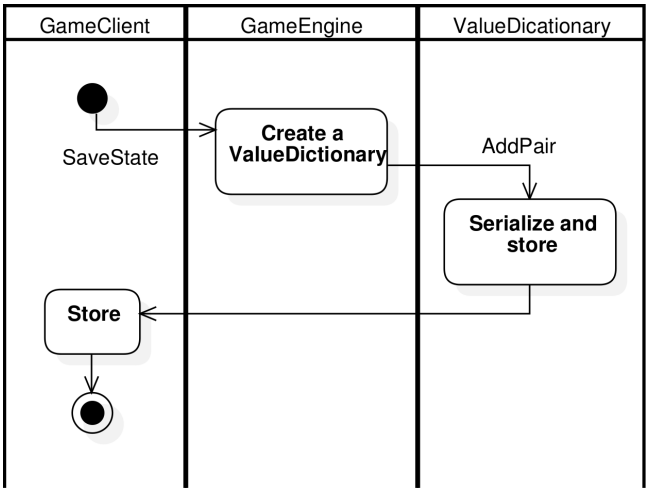

Figure 5: Activity diagram for storing the game state

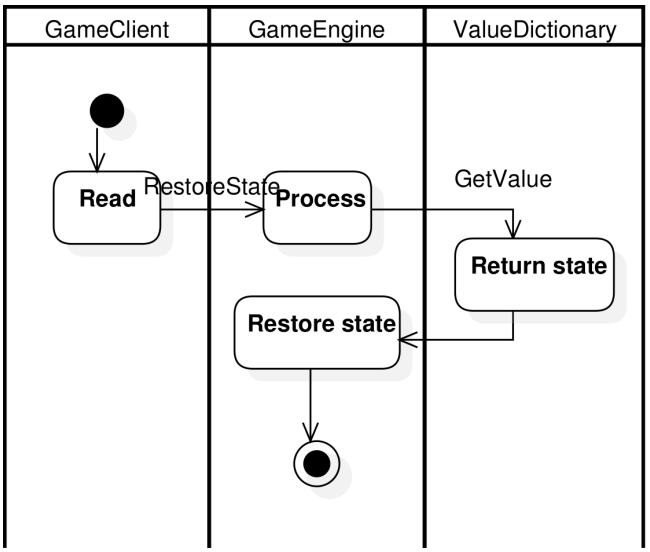

Figure 6: Activity diagram for restoring game state

4.2.4 High Cohesion

Each class has strongly related and focused responsibilities. GameClient is focused on input/output from persistent data. GameEngine is focused on maintaining the game state consistency. ValueDictionary is focused on storing and retrieving data in an optimized and serialized form.

4.2.5 Controller

While in the use case discussed above there is no role of UI, looking at a broader picture the GameClient acts like a controller. GameClient handles gets all the events generated from the UI and propagates it to relevant parts of GameEngine. Since our use case only deals with restoring from a saved state for extreme cases of non-user generated interruption, the memento design pattern doesn't need a controller.

4.2.6 Polymorphism

ValueDictionary class implements Serializable interface. GameClient only needs an object of the class that implements the Serializable interface. This allows us to further extend memento object by supporting serialization of other more complex data types beside string. Similarly, different Scenes maintain their own states and GameClient only talks to them via a common interface.

4.2.7 Pure Fabrication

Pure fabrication is not directly involved in memento design pattern, although it is used in *Aristocrat Technologies* by having GameClient interact with a domain agnostic fabricated class to handle I/O with persistent data.

4.2.8 Protected Variations

There can be multiple types of games with multiple gaming techniques requiring different types of objects to maintain the state. GameClient remains achieves independence of having ValueDictionary implement Serializable interface. If the memento implements this interface, GameClient remains protected of any variation in the type of memento or the data stored in it.

4.2.9 Indirection

A game involves multiple parts on the screen that work independently. For example, one part may be animating the slots while another slot may be showing the player's current credit amount. These are referred to as **Scenes** and each of these Scenes create their own mementos. GameClient only talks to GameEngine instead of requesting each Scene for a memento. GameEngine consolidates all the different mementos and returns that to GameEngine. This de-couples GameClient from the Scenes which hold the game state.

4.3 Agreement with SOLID principles

SOLID is short for the following 5 principles:

a) *S*ingle Responsibility Principle

b) *O*pen Closed Principle

c) *L*iskov Substitution Principle

d) *I*nterface Segregation Principle

e) *D*ependency Inversion Principle

4.3.1 Single Responsibility Principle

Each class maintains a single responsibility.

- GameClient is responsible for storing and retrieving the memento at the appropriate time.

- GameEngine is responsible for mediating between different scenes.
- ValueDictionary is responsible for containing any data that can be serialized.
- Scene is responsible for updating and retrieving data from ValueDictionary.

4.3.2 Open Closed Principle

In our case, ValueDictionary can be easily extended by adding more serializable objects without modifying the original class.

4.3.3 Liskov Substitution Principle

Liskov Substitution states that all derived classes must be substitutable with the base class. In our case, we have two derived classes, ValueDictionary and Scene. Both are derived from a generic enough interface so that all mementos and Scenes have a common interface.

4.3.4 Interface Segregation Principle

Only the memento class (ValueDictionary) implements the Serializable interface.

4.3.5 Dependency Inversion Principle

GameClient is a higher-level class that aggregates ValueDictionary, but it does not depend on ValueDictionary class. Instead, it stores objects of abstract type Serializable which ValueDictionary implements.

5 Pitfalls

Memento pattern suffers majorly from the following three factors that affect its performance:

a) *Creating memento* — During a normal save/restore cycle a memento is created twice. Normally, creating an object is the slowest part of execution as memory is allocated on the heap. Hence it is very easy to see a steep decline in performance when the creation of an object is request multiple times.

b) *Size of memento* — As the state information of a program grows, the size of memento increases too. This has an adverse effect on memory consumption.

c) *No object* — A memento does not store an object itself but just its state. Hence it may not be enough to redo an *action* or re-run a *procedure*.

One can mitigate the 1st issue by reducing the number of times Caretaker makes to Originator. This can be done by having a single memento object that gets updated by the Originator.

Size of a memento object may be decreased by pruning derivative states from it and having Originator store only most relevant state information. Having a single memento object also helps reduce memory consumption.

If the ability of undoing/redoing is not a critical part of the program, the caretaker may keep the memento object in-memory.

To store an object, Command pattern may be used along with the memento pattern.

6 Conclusion

Memento pattern is a basic design pattern that is used to store and restore an object's state. It is commonly used in any program that has undo/redo functionality. To further generalize the state information, a memento object may use serialization. In contrast to Command Pattern, Memento doesn't store the object itself.

References

[1] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1 edition, 1994.

[2] Grady Booch, Robert A. Maksimchuk, Michael W. Engle, Bobbi J. Young, and Jim Conallen. *Object-oriented analysis and design with applications, Third Edition.* Addison Wesley object technology series. Addison-Wesley, 2007.