Memento Pattern

1st Herrmann, Axel
Knowledge Foundation @ Reutlingen University
Stuttgart, Germany
axel.herrmann@weiterbildung-reutlingen-university.de

I. Einleitung

Entwurfsmuster bieten bewährte Lösungen für häufige Designprobleme in der Softwareentwicklung, fördern Wiederverwendbarkeit und Flexibilität des Codes und erleichtern die Kommunikation im Team durch gemeinsame Begriffe. Sie helfen Entwicklern, komplexe Probleme effizienter zu lösen und robuste, wartbare Software zu erstellen. Sehr bekannte Entwurfsmuster sind die von Gamma et al. [1] vorgestellten Patterns, welche in Erzeugungsmuster, Strukturmuster und Verhaltensmuster unterteilt werden. Das Memento Pattern ist eines dieser Entwurfsmuster, welches als Verhaltensmuster eingeordnet wird.

II. Kontext

In vielen Anwendungen gehen Nutzer heutzutage davon aus, dass vorherige Zustände wiederhergestellt werden können. Beispiele finden sich in Texteditoren mit der „Undo"-Funktion, in Computerspielen mit speicherbaren Spielständen oder in der Datenbankentwicklung, wo Transaktionen zurückgerollt werden können. Solche Anforderungen können durch die Speicherung von Zustandsinformationen eines Objekts umgesetzt werden. Um den Zustand eines Objekts zu speichern und diesen zu einem späteren Zeitpunkt potentiell wiederherstellen zu können, muss der zustandsrelevante Teil des Objekts kopiert und abgespeichert werden. Dieser Prozess dessen ist non-trivial, da eine einfache Kopie das Objekts gegen das Prinzip der Kapselung verstößt. Kapselung ist wichtig, um die Integrität des Zustands zu schützen [2]. Bei Zugriff anderer Objekte auf die Zustandsinformationen können unerwünschte Nebeneffekte auftreten. Um also gleichzeitig von der Qualitätserhöhung von Software durch eingehaltene Kapselung und umgesetzten Rollback-Möglichkeiten zu profitieren, sollte eine klare Methodik verwendet werden, die diese beiden, auf den ersten Blick widersprüchlich wirkenden Forderungen, vereint.

III. Ansatz

Dieses Problem versucht das Memento Pattern, vorgestellt von Gamma et al. [1], und auch *Snapshot* genannt [3], zu lösen. Die Idee dabei ist, dass das zu speichernde Objekt, der *Originator*, eine Momentaufnahme seines Zustands in einem separaten Objekt, welches *Memento* genannt wird, speichert. Ausgelöst wird dieses Verhalten durch ein weiteres Objekt, das *Caretaker* genannt wird. Dieses verwaltet eine Historie von Momentaufnahmen und kann sie später verwenden, um den Zustand des Originators wiederherzustellen. Der Fokus des Entwurfsmusters liegt unter anderem darauf, die Kapselung des Originator-Objekts zu bewahren, indem der Caretaker keinen direkten Zugriff auf die internen Details erhält [4]. Dadurch wird eine klare Trennung zwischen den Rollen des Originators und des Caretakers gewährleistet. Für das Erreichen dieses Zielzustands gibt es nicht nur einen offensichtlichen besten Lösungsweg, sondern mehrere Implementierungsansätze, abhängig von den Anforderungen der jeweiligen Anwendung.

IV. Implementierung

Je nach Programmiersprache und gewünschter Sicherheit der Einhaltung der Kapselung gibt es verschiedene konkrete Umsetzungsmöglichkeiten des Patterns. In den folgenden Abschnitten werden drei gängige Ansätze nach [3] vorgestellt.

A. Verschachtelte Klasse

Eine Möglichkeit der Implementierung liegt in der Nutzung des Prinzips der verschachtelten Klassen, welches in einigen Programmiersprachen wie Java, C++ und C# möglich ist. Bei diesem

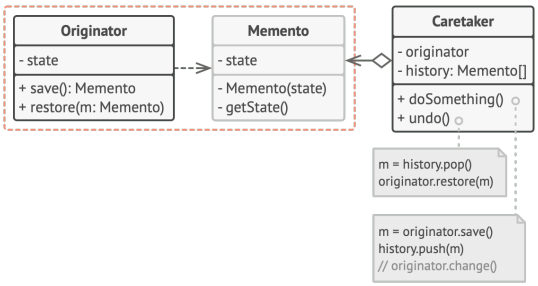

Abbildung 1. Implementierung des Memento Patterns nach [3] mit Nutzung einer verschachtelten Klasse.

Ansatz wird die Memento-Klasse als innere Klasse innerhalb des Originators definiert. Der Originator besitzt dadurch direkten Zugriff auf die privaten Felder und Methoden des Mementos, was die Implementierung vereinfacht. Die Kapselung wird aufrechterhalten, da die Memento-Klasse für den Caretaker und andere externe Objekte vollständig verborgen bleibt. Von außen sieht die interne Struktur des Mementos wie eine Blackbox aus und lediglich bewusst zugängig gemachte Methoden können verwendet werden. Typischerweise können Caretaker so beispielsweise die Zeitpunkte oder Namen der Mementos einsehen.

Ein praktisches Beispiel ist die Speicherung von Spielständen in einem Videospiel. Der Spielstand (Memento) wird in der Engine (Originator) als innere Klasse definiert. Dadurch kann die Engine den Zustand direkt speichern und wiederherstellen, während der Caretaker, z. B. eine Benutzeroberfläche, lediglich den Spielstand verwaltet, ohne Details darüber zu kennen. Dort kann dem Nutzer dann der Name des Spielstands oder der Zeitpunkt der letzten Speicherung angezeigt werden.

Dieser Ansatz eignet sich besonders dann, wenn die Sprache eine einfache Unterstützung für innere Klassen bietet und die Anforderungen an die Kapselung hoch sind. Ein Nachteil kann jedoch die enge Kopplung zwischen Originator und Memento sein, da die beiden Klassen in diesem Modell untrennbar verbunden sind.

B. Zwischengeschaltetes Interface

Eine alternative Implementierung nutzt ein Interface, das die Interaktion zwischen dem Caretaker und dem Memento regelt.

Dabei regelt das Prinzip der Polymorphie den Zugriff auf den

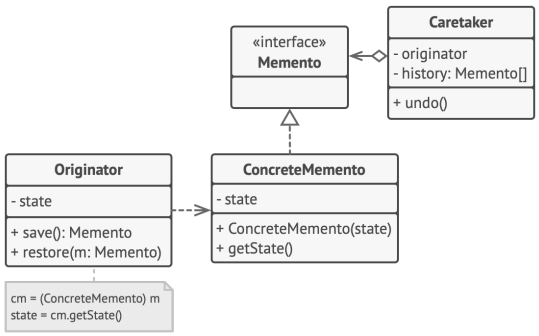

Abbildung 2. Implementierung des Memento Patterns nach [3] mit Nutzung eines zwischengeschalteten Interfaces.

internen Zustand des Mementos. Das Memento ist nämlich in ein Interface, welches lediglich die öffentlich zugänglichen Methoden definiert und eine *ConcreteMemento*-Klasse, welche dieses Interface implementiert, aufgeteilt. Die Idee ist, dass externe Objekte wie Caretaker das Memento lediglich als Interface kennen und somit nur die öffentlich einsehbaren Attribute und Methoden verwenden können, wohingegen der Originator die Mementos als ConcreteMemento referenziert.

Dieser Ansatz bietet mehrere Vorteile. Zum einen wird die Interaktion zwischen den Klassen klar definiert und getrennt, was die Wartbarkeit des Codes verbessert. Zum anderen können mehrere unterschiedliche Memento-Typen verwendet werden, solange sie das gleiche Interface implementieren. Das ist nützlich, wenn ein System mehrere verschiedene Zustände oder Datenarten speichern und wiederherstellen muss.

Ein Beispiel ist ein Texteditor, der unterschiedliche Memento-Typen für Textformatierungen, Cursorpositionen oder geöffnete Tabs verwendet. Der Caretaker muss in diesem Fall nur wissen, wie man das Interface verwendet, während der Originator die genaue Funktionsweise kontrolliert. Ein Nachteil dieser Methode ist die erhöhte Komplexität, da zusätzliche Klassen und Schnittstellen definiert werden müssen. Für kleine Projekte kann das die Übersichtlichkeit verringern und zu unnötigem Overhead führen.

C. Strikte Kapselung

Ein Ansatz zur maximalen Wahrung der Kapselung ist die vollständige Trennung von Originator und Caretaker, sodass der Caretaker ausschließlich mit abstrahierten oder anonymisierten Mementos arbeitet. Diese Art der Implementierung ermöglicht

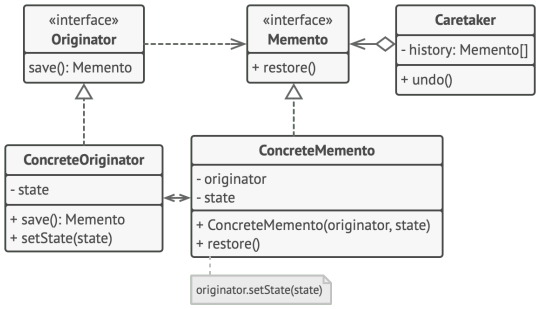

Abbildung 3. Implementierung des Memento Patterns nach [3] mit starker Absicherung der Kapselung.

die Wiederverwendung eines Originator- und Memento-Interfaces für mehrerer Typen der beiden. Dabei würde jede konkrete Implementierung eines Originators mit einer Memento-Implementierung verknüpft sein. So sind externe Objekte wie Caretaker explizit daran gehindert, den Zustand der Mementos zu betrachten oder verändern. Außerdem ist die Caretaker-Klasse in dieser Version unabhängig vom Originator, da das Memento-Interface die restore-Funktion anbietet. Originator und Memento sind in dieser Implementierung explizit auf Objektebene gekoppelt, indem bei der Erstellung von Mementos das Originator Objekt mit übergeben wird. Dadurch kann das Memento den Zustand des Originators zurücksetzen, wenn dieser passende Zugriffsmethoden definiert hat.

Ein potenzieller Nachteil dieser Methode ist der zusätzliche Implementierungsaufwand. Schon an der Größe der Übersicht ist zu sehen, dass diese Implementierung konzeptionell im kompliziertesten ist. In vielen Fällen wird vermutlich eine der vorherigen beiden Methoden ebenfalls ausreichende Sicherheit der Kapselung bei weniger Aufwand bieten.

V. Diskussion

Das beschriebene Vorgehen nach Memento Pattern kann wirksam genutzt werden, um den früheren Zustand einer Applikation durch das Erstellen eines Snapshots wiederherzustellen. Besonders im Fokus des Patterns liegt dabei das Aufrechterhalten der Kapselung des Originator Objekts, was zugleich der große Vorteil bei Verwendung des Entwurfsmusters ist. Außerdem unterstützt das Pattern das Prinzip der *Separation of Concerns*, indem es die Zustandsverwaltung vom Originator entkoppelt [4].

Zugleich bringt das Muster allerdings auch Nachteile mit sich. Mit der Speicherung mehrerer Zustände geht natürlich, abhängig von der Größe der Zustände und auch der Anzahl der gespeicherten Zustände, mitunter eine nicht unerhebliche Hauptspeicher-Last einher. Außerdem ist eine große Motivation für das Memento Pattern der Wunsch nach Kapselung, welcher jedoch vor allem in dynamischen Programmiersprachen nicht vollkommen garantiert werden kann. Als Beispiel dafür kann Typescript genannt werden, wobei zwar zur Compile-Zeit ein Zugriff auf private Attribute verhindert werden kann, jedoch durch die Kompilierung zu Javascript zur Laufzeit keine Versicherungen gemacht werden können. Dagegen könnte allerdings argumentiert werden, dass dieses Problem bei der Wahl einer solchen Programmiersprache im Allgemeinen bekannt ist und akzeptiert werden muss. Abgesehen davon wäre bei Lösung des Memento Patterns über Polymorphie auch in statisch typisierten Programmiersprachen eine Konvertierung möglich, durch die der Kapselungsmechanismus umgangen werden könnte.

VI. Fazit

Abschließend lässt sich sagen, dass das Memento Pattern eine gute Möglichkeit bietet, auf eine praktische Art und Weise „Undo“-Funktionalität zu einem Programm hinzuzufügen und zugleich eine hohe Code-Qualität zu wahren, indem Kapselung trotzdem nicht verletzt wird. Die Schwierigkeit in der Anwendung sollte allerdings nicht unterschätzt werden, da es nicht nur eine konkrete Implementierungsmöglichkeit gibt und allgemein die Code-Komplexität erhöht wird. Zuletzt sollte nicht vergessen werden, dass die genannten Implementierungsmethoden die Kapselung nicht mit voller Sicherheit gewährleisten können.

Literatur

[1] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, “Design patterns: Abstraction and reuse of object-oriented design,” in

ECOOP'93—Object-Oriented Programming: 7th European Conference Kaiserslautern, Germany, July 26–30, 1993 Proceedings 7. Springer, 1993, pp. 406–431.

[2] S. Zweben, S. Edwards, B. Weide, and J. Hollingsworth, "The effects of layering and encapsulation on software development cost and quality," *IEEE Transactions on Software Engineering*, vol. 21, no. 3, pp. 200–208, 1995.

[3] [Online]. Available: https://refactoring.guru/design-patterns

[4] [Online]. Available: https://www.geeksforgeeks.org/memento-design-pattern/