Design Patterns im Projekteinsatz: Erfahrungen aus der Praxis

Problembereiche, Fallstricke, Tipps zur richtigen Anwendung

Karl Nieratschker, SKT Nieratschker

Als Lösungsbausteine für häufig wiederkehrende Aufgabenstellungen gehören die Entwurfsmuster der „Gang of Four“ (GoF-Design Patterns) heute zu den wichtigen Grundelementen eines guten objektorientierten Softwareentwurfs. Aufgrund der enormen Leistungssteigerung der Hardware gilt dies immer häufiger auch für Software in Embedded Systemen. Unterstützt durch umfangreiche Fachliteratur und unzählige Beispiele im Internet sollte die Anwendung der Muster eigentlich kein Problem sein. In der Praxis stellt sich allerdings oft heraus, dass die Implementierung eines Patterns schwieriger ist, als erwartet.

In der objektorientierten Softwareentwicklung sind Entwurfsmuster insbesondere durch das Buch „Design Patterns. Elements of Reusable Object-Oriented Software” bekannt geworden. Bereits bei der Einarbeitung in die dort beschriebenen 23 Entwurfsmuster können die ersten Probleme auftreten, denn die Patterns werden bewusst in einer stark abstrahierten Form präsentiert. Zwar gibt es zu jedem Muster auch ein Beispiel, aber diese Beispiele sind für Embedded Systeme eher untypisch. Zudem sind nicht alle Beispiele vollständig ausprogrammiert oder setzen mitunter Kenntnisse der Programmiersprache Smalltalk voraus. Außerdem gibt es zu jedem Beispiel auch eine Art Klassendiagramm, das allerdings in der Regel nicht ganz UML-konform ist, da das Buch vor der Einführung von UML veröffentlicht wurde (**Bild 1**).

Wo kann ein Pattern eingesetzt werden?

Hat man die Patterns einmal verinnerlicht, kann man damit beginnen, nach Stellen im Softwareentwurf zu suchen, wo Patterns sinnvoll eingesetzt werden können. Die Schwierigkeit dabei liegt erfahrungsgemäß oft darin, dass man zwar weiß, wie die Patterns funktionieren, nicht aber, für welche Anwendungssituationen sie gedacht sind. Ein Dekorierer lässt sich z.B. - unabhängig davon, wie er genau funktioniert - typischerweise in Situationen einsetzen, in denen man in einfacheren Fällen mit Ableitung arbeiten würde. Somit ist „Vererbung“ eine Art „Auslöser“ für eine mögliche Anwendung des Dekorierer-Musters. Mithilfe solcher „Trigger“ lassen sich sehr viel leichter Stellen finden, an denen Patterns sinnvoll eingesetzt werden können.

Welches Pattern ist das richtige?

Mitunter fällt es auch schwer zu entscheiden, *welches* Pattern eingesetzt werden soll, sobald die richtige Stelle gefunden ist. Langjährige Programmierpraxis führt oft dazu, dass man sich leichter Implementierungen als Abstraktionen merkt. Da es aber deutlich weniger Implementierungsmöglichkeiten als Entwurfsmuster gibt, müssen sich zwangsläufig mehrere Patterns Implementierungen teilen, obwohl sie semantisch völlig unterschiedlich sind. Ein Beispiel dafür ist das Zustands- und Strategiemuster. Insbesondere hat dies auch Auswirkungen auf die grafischen Darstellungen der Patterns, die sich von ihrer Struktur her zwar ähnlich sehen, aber verschiedene Bedeutungen haben (**Bild 1** und **Bild 2**). Da sich Bilder in der Regel besser einprägen als Textinformationen, sind es mitunter gerade die grafischen Darstellungen die zur Verwirrung beitragen. Um nicht in diese Falle zu tappen, ist es wichtig, sich bewusst zu machen, dass die Absicht eines Musters zunächst nichts mit der Implementierung zu tun hat, und deshalb als eigenständige Qualität betrachtet werden muss.

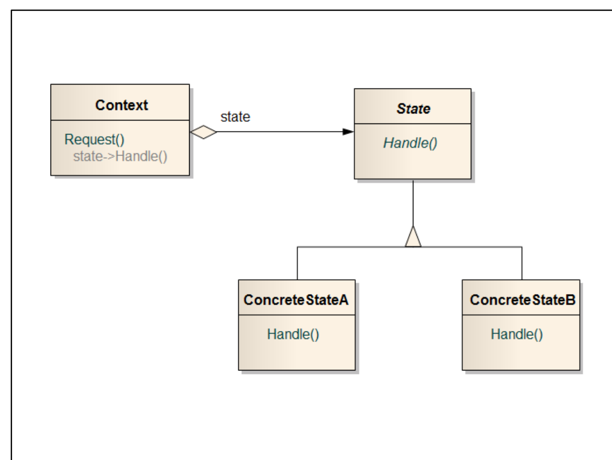

Bild 1: Zustandsmuster (State Pattern)

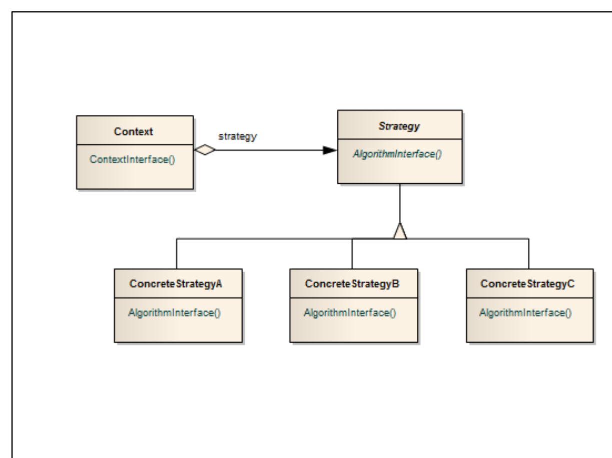

Bild 2: Strategiemuster (Strategy Pattern)

Realisierung des Patterns

Sobald der Einsatzort und die Art des Patterns festgelegt sind, kann man mit der Abbildung des abstrahierten Patterns auf reale Klassen und Relationen beginnen. Die Schwierigkeit dieses Vorgangs wird in der Praxis oft unterschätzt. Aufgrund des hohen Abstraktionsgrades der Patterns sind die dort verwendeten Bezeichnungen z.T. äußerst allgemein gehalten (z.B. „Subject“, „Component“, „Handle“). Häufig werden diese Namen bei der Anwendung des Patterns direkt übernommen und nicht für die Anwendung konkretisiert. Dies kann zwar, wie z.B. beim Command-Pattern, mitunter sinnvoll sein, i.A. sollte dies aber vermieden werden. Ein weiteres Problem ist, dass die grafische Darstellung nicht alle Aspekte eines Patterns darstellen kann. Z.B. geht aus den Diagrammen i.A. nicht hervor, ob ein „Kästchen“ eine Klasse oder ein Interface darstellt, oder ob vielleicht sogar beides möglich ist (z.B. Zustandsmuster). Generell wird immer nur *eine* Implementierung gezeigt, auch wenn mehrere Varianten möglich sind (z.B. Beobachtermuster), und es wird i.d.R. die minimale Implementierung gezeigt, was manchmal sogar irreführend sein kann (z.B. Zustandsmuster). Schließlich muss beim Einsatz von Patterns in Embedded Systemen noch geprüft werden, ob die Speicherplatz- und Laufzeitkosten des Patterns akzeptabel sind.

Patterneinsatz im Projektumfeld

Für den erfolgreichen Einsatz von Entwurfsmustern im Projektumfeld ist es wichtig, dass *alle* Entwickler im Team mit Design Patterns vertraut sind. Ist dies nicht der Fall, dann müssen Pattern-Lösungen beim Softwareentwurf im Team erst aufwändig erklärt werden. Häufig werden Entwurfsmuster dabei als unnötig kompliziert empfunden und nur deshalb abgelehnt, weil zu wenig Zeit bleibt, ihre Nützlichkeit deutlich zu machen. Mitunter wird auch auf den Einsatz von Entwurfsmustern nur deshalb verzichtet, weil Kollegen mit geringen oder fehlenden Design-Pattern-Kenntnissen beim Debuggen oder bei der Wartung sonst Schwierigkeiten haben könnten. Aber auch das Gegenteil kann passieren: übertriebener Einsatz von Design Patterns kann zu einem „Over-Engineering“ führen und die Applikation tatsächlich unnötig komplex machen.

Generell sollte der Einsatz von Patterns immer wieder trainiert werden. Leider sind die Möglichkeiten dafür im Projekt eher selten gegeben. In dieser Situation kann es hilfreich sein, Patternlösungen in nicht selbst geschriebener Software (z.B. Open Source Software) zu suchen und zu analysieren, wie sie implementiert sind. Man kann auch in der eigenen Software nach Stellen suchen, wo Patterns eingesetzt werden könnten, ohne dies aber sofort umzusetzen. Ergibt sich zu einem späteren Zeitpunkt die Notwendigkeit, den Code - z.B. zur Behebung eines Fehlers - ändern zu müssen, dann kann dieser Wissensvorsprung sofort genutzt werden.

Tipps für die Praktische Anwendung

Eines der bekanntesten Entwurfsmuster ist das **Singleton-Pattern** (**Bild 3**). Es stellt sicher, dass es von einer Klasse, die nach diesem Konzept entworfen wurde, nur eine einzige Instanz gibt, auf die von jeder beliebigen Stelle im Programm aus zugegriffen werden kann. Obwohl es sich hier um ein leicht zu verstehendes und sehr nützliches Pattern handelt, wird es leider oft falsch angewendet. Aufgrund ihrer universellen Ansprechbarkeit werden Singletons nämlich gern als „legitimer“ Ersatz für globale Objekte verwendet, die es aus objektorientierter Sicht nicht geben sollte. Aus diesem Grund wird das Singleton-Muster heute oft schlechtgeredet, manche halten es gar für ein „Antipattern“ und weigern sich deshalb, es anzuwenden. Dabei hat dieses Muster definitiv seine Berechtigung, nämlich dann, wenn wirklich sichergestellt werden muss, dass es nur eine einzige Instanz gibt. Ein charakteristischer Anwendungsfall hierfür wäre z.B. ein ID-Generator. Deshalb sollte man es sich gut überlegen, ob man auf ein grundsätzlich sinnvolles Pattern generell verzichtet, nur weil es in der Praxis oft falsch anwendet wird.

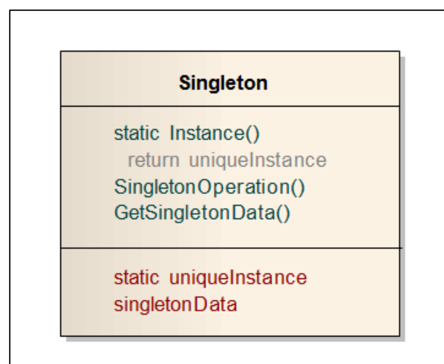

Bild 3: Singleton Pattern

Ein anderes, sehr häufig verwendetes Muster ist das **Beobachtermuster** oder **Observer-Pattern** (**Bild 4**). Der Name dieses Musters drückt aus, dass Objekte ein bestimmtes Zielobjekt „beobachten“, um Zustandsänderungen des Zielobjektes erkennen zu können. Dadurch wird suggeriert, dass ein Beobachter aktiv ein Objekt beobachtet, während das beobachtete Objekt zwar seinen Zustand ändert, sich aber in Bezug auf den Beobachter passiv verhält. Die Realisierung sieht allerdings völlig anders aus: die Beobachter werden beim beobachteten Objekt registriert und verhalten sich sonst passiv, während das beobachtete Objekt bei einer Zustandsänderung aktiv wird und jeden registrierten Beobachter durch den Aufruf seiner Update-Methode informiert. So betrachtet ist der Name des Patterns sogar eher irreführend. Wie die Praxis zeigt, kann dies sogar dazu führen, dass die Rollen

der Klassen falsch zugeordnet werden. Darüber hinaus macht diese Diskrepanz auch deutlich, warum Absicht und Implementierung eines Patterns immer getrennt betrachtet werden sollten.

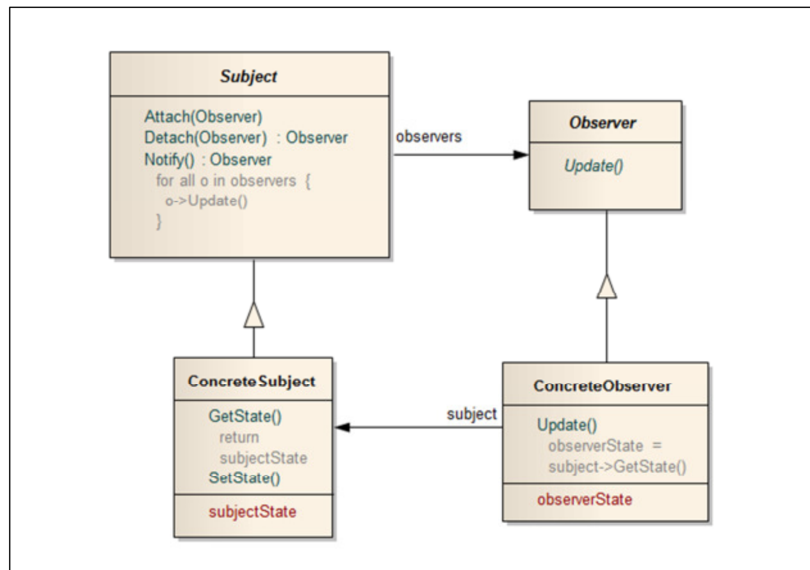

Bild 4: Beobachtermuster (Observer Pattern)

Die Verbindung zwischen Beobachter und beobachtetem Objekt wird mithilfe eines Interfaces realisiert, das in der Patternbeschreibung den Namen „Observer" trägt und eine parameterlose Methode mit dem Namen Update() enthält. Eine typische Designfalle besteht darin, dieses Interface der Einfachheit halber 1:1 in die Applikation zu übernehmen. In trivialen Fällen ist dies auch ausreichend. Stellt sich im Lauf der Zeit aber heraus, dass Observer mehrere Objekte oder unterschiedliche Arten von Zustandsänderungen beobachten müssen, dann hat dieses Modell aufwändigen Observer-Code zur Folge oder es müssen nachträglich teure Interfaceänderungen vorgenommen werden. Deswegen sollte unbedingt von Anfang an auf die richtige Schnittstellenauslegung geachtet werden.

Kaum ein Embedded System kommt in der Praxis ohne einen Zustandsautomaten aus. Das **Zustandsmuster** oder **State-Pattern** (**Bild 5**) bildet jeden Zustand auf eine jeweils eigene, von einer gemeinsamen Basisklasse abgeleitete Zustandsklasse ab. Ereignisse werden als Methoden (Eventhandler) in den Zustandsklassen realisiert. Ein typischer Fehler, der hier mitunter gemacht wird, ist es, zustandsübergreifende Daten in der gemeinsamen Basisklasse der Zustandsklassen unterzubringen. Dies funktioniert nicht, weil diese Daten sich in die jeweiligen Zustandsklassen vererben und somit zu zustandsspezifischen Variablen werden. Für diesen Zweck sieht das Muster eine eigene Kontextklasse vor, die die zustandsübergreifenden Daten und

einen Pointer auf das momentan aktive Zustandsobjekt enthält. Typischerweise wird die Adresse des Kontextes bei jedem Aufruf eines Ereignishandlers als Parameter übergegeben. Damit der Eventhandler auf die Daten des Kontextes zugreifen kann, werden in der Kontextklasse üblicherweise Zugriffsmethoden implementiert. Dies kann in der Praxis allerdings sehr aufwändig werden und zu unerwünschten Zugriffsmöglichkeiten führen. Im Übrigen muss unbedingt darauf geachtet werden, dass ein Ereignishandler, der einen Zustandswechsel initiiert, unmittelbar danach zurückkehrt. In jedem anderen Fall wird nämlich Code im falschen Kontext ausgeführt, was zu schwer auffindbaren Fehlern führen kann.

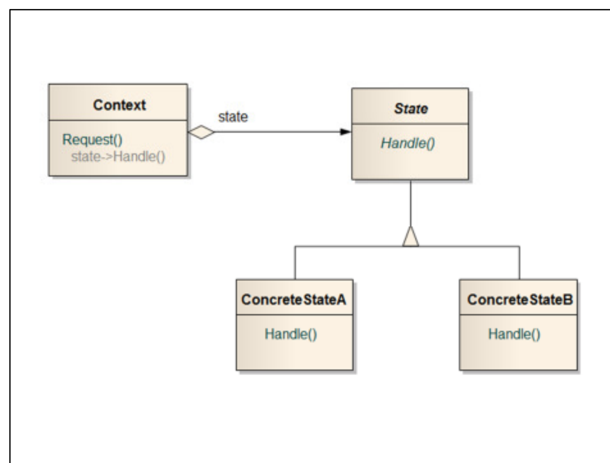

Bild 5: Zustandsmuster (State Pattern)

Quellenverzeichnis/Literaturverzeichnis:

„Design Patterns. Elements of Reusable Object-Oriented Software", Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides

Autor

Basierend auf einem Studium der technischen Informatik verfügt Karl Nieratschker über eine mehr als 30-jährige Erfahrung im Bereich der Embedded-Programmierung und betriebssystemnahen Softwareentwicklung. Als selbständiger Trainer, Softwareberater und Coach unterstützt er seit 1999 Kunden bei der Einführung neuer Softwaretechnologien. Besondere Schwerpunkte dabei sind „Objektorientierte Programmierung in Embedded Systemen", sowie „Multithreading-„ und „Multicore-Programmierung".

Internet: www.skt-nieratschker.de

Email: office@skt-nieratschker.de