Hausarbeit
Software Design Patterns

Template Method, State, Strategy

René Speck

13. Oktober 2009

Betreuer: Frank Schumacher
Axel Ngonga
Martin Gebauer

Universität Leipzig
Fakultät für Mathematik und Informatik
Institut für Informatik
Abt. Betriebliche Informationssysteme

Inhaltsverzeichnis

Glossar

Framework	Eine Rahmenvorgabe für die Struktur einer Software.
Library	ist eine Sammlung von Klassen und Routinen zur Unterschtüzung der Softwareentwicklung.
konkrete Klasse	ist eine Unterklasse.
MVC	Model-View-Controller ist ein Architekturmuster zur Trennung von Daten, Präsentation und Logik.
OC-Principle	Das Open-Closed-Principle ist ein objektorientiertes Prinzip beim Entwurf von Software.
P2P	Peer to Peer ist ein Architekturmuster für die dezentrale Vernetzung von Rechnersystemen.
SDP	Software Design Patterns sind Entwurfsmuster.
Unterklasse	ist in der objektorientierten Programmierung eine Klasse die von einer anderen Klasse erbt.

1 Einleitung

1.1 Muster

Am Anfang einer Realisierung steht für gewöhnlich ein Problem, zu welchem ein Lösungsentwurf benötigt wird. Um möglichst schnell eine flexible und wiederverwendbare Lösung für dieses Problem zu finden, greifen Entwickler daher stets auf die schon vorhandenen Lösungen und auf die Erfahrungen von Experten zurück. Muster stellen dieses Expertenwissen bereit und beschreiben somit Problemlösungen für erprobte Probleme, die sich im Laufe der Zeit bewährt haben und weiterentwickelt wurden. Bereits in den siebziger Jahren verwendete Christopher Alexander - der „Architekturtheoretiker des 20. Jahrhunderts“[1] - Diese, bei der Stadt- und Raumplanung.

> *Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.* [AIS77]

Muster bilden darüber hinaus auch eine Kommunikationsgrundlage[2] für Entwickler, da sie einheitliche Bezeichnungen für Problemlösungen und deren Elemente definieren.

Die folgenden Musterarten lassen sich im Bereich der Softwareentwicklung unterscheiden:

- Architekturmuster - Lösungen für die gesamte Architektur einer Software.
- Entwurfsmuster - Lösungen für konkrete Probleme beim Softwaredesign.
- Idiome - Lösungen für Implementierungsprobleme in einer bestimmten Programmiersprache.

Architekturmuster beschreiben den Gesamtaufbau eines Systems mit einem hohen Abstraktionsgrad. Bekannte Beispiele dafür sind *MVC* und *P2P*. Entwurfsmuster beschreiben einzelne Module und wie sie miteinander in Verbindung treten. Im folgenden werden drei sehr grundlegende Entwurfsmuster vorgestellt, die von der *Gang of Four* [3] 1995 veröffentlich wurden. Idiome beschreiben spezifische programmiersprachenabhängige Lösungen mit einem sehr geringen Abstraktionsgrad. Bekannte Beispiele für C++ sind *Counted Body* und *Virtual Constructor*.

> *The pattern is, in short, at the same time a thing which happens in the world, and the rule which tells us how to create that thing, and when we must create it. It is both a process and a thing; both a description of a thing which is alive, and a description of the process which will generate that thing.* [ALE79]

[1] de.wikipedia.org/wiki/Christopher_Alexander
[2] de.wikipedia.org/wiki/Mustersprache
[3] [GOF94]

1.2 Entwurfsmuster

Entwurfsmuster dienen der Konstruktion und der Dokumentation von Software und der Kommunikation zwischen Softwareentwicklern. Sie benennen, abstrahieren und identifizieren die relevanten Merkmalen eines Entwurfs und beschreiben ein ganz bestimmtes objektorientiertes Entwurfsproblem. Sie bestehen aus den folgenden vier Elementen. [4]

- Mustername - Ein oder zwei Wörter, die das Muster sinnvoll beschreiben.
- Problem - Wann ist das Muster anzuwenden und welchen Kontext hat es.
- Lösung - Beschreibt die Elemente aus denen das Muster besteht.
- Konsequenz - Vor- und Nachteile des Musters und spezielle Implementierungsaspekte.

Entwurfsmuster lassen sich anhand der zwei Kriterien *Aufgabenbereich* und *Gültigkeitsbereich* klassifizieren. [5]

Aufgabenbereich

Wenn ein Entwurfsmuster den Prozess der Objekterzeugung zur Aufgabe hat, dann sprechen wir von einem *Erzeugungsmuster*[6]. Ein *Strukturmuster*[6] hingegen hat die Aufgabe, den Aufbau von Klassen und Objekten zu regeln. Wenn die Aufgabe des Musters darin besteht, die Interaktionen zwischen Objekten und komplexen Kontrollflüssen zu beschreiben und die Art und Weise, in der Klassen und Objekte zusammenarbeiten und Zuständigkeiten aufteilen, dann sprechen wir von einem *Verhaltensmuster*[6].

Gültigkeitsbereich

Der Gültigkeitsbereich teilt die Entwurfsmuster in *klassenbasierte* und *objektbasierte* Muster auf. Wobei klassenbasierte Muster Vererbung verwenden und sich auf die Klassen und deren Unterklassen konzentrieren. Somit sind sie statisch und liegen zur Übersetzungszeit fest vor. Objektbasierte Muster beschreiben Objektbeziehungen und wie diese zur Laufzeit geändert werden können. Sie verwenden anstelle von Vererbung vorrangig Assoziation oder Aggregation und sind dynamisch.

[4] [GOF01] Seite 3 ff.
[5] [GOF01] Seite 14 ff. Entwurfsmuster können aber auch nach anderen Kriterien organisiert werden.
[6] engl.: Creational Pattern, Structural Pattern, Behavioral Pattern

2 Verhaltensmuster

2.1 Template Method

Das *Template Method* [1] Entwurfsmuster ist ein sehr einfaches und grundlegendes klassenbasiertes Verhaltensmuster. Es definiert in einer Methode das Grundgerüst und die feste Struktur eines abstrakten Algorithmus. Unterklassen können einzelne Schritte des Algorithmus neu definieren.

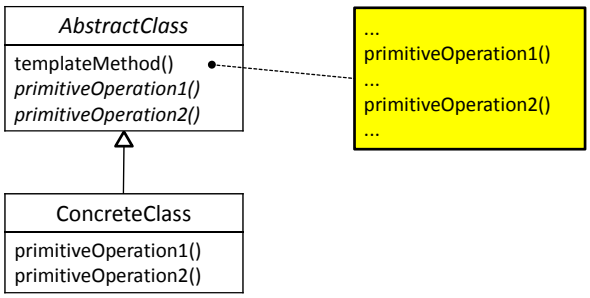

Abb. 2.1: Klassendiagramm nach [GOF94]

Mit diesem Muster erstellen wir also in einer Methode ein Vorlage, die - unter Verwendung von primitiven Operationen [2] - unseren Algorithmus als eine Folge von abzuarbeitenden Schritten festlegt. Unterklassen erweitern den Algorithmus, indem sie die primitiven Operationen implementieren. Somit wird durch die gleichnamige Template-Methode, die Struktur fest vorgegeben, während die Unterklassen das spezifische Verhalten bestimmen.

2.1.1 Problem

Wir verwenden mehrere verhaltensähnliche Algorithmen mit invarianten Teilen, die somit viel redundante Quelltextzeilen erfordern. Oder wir kennen die varianten Teile der verschiedenen verhaltensähnlichen Algorithmen noch nicht und möchten die invarianten Teile bereits implementieren, dabei aber die Kontrolle über den Algorithmus behalten, d.h. der Algorithmus soll in seinem Verhalten flexibel bleiben aber einer festen und bekannten Struktur folgen.

Im Kontext steht ein Framework mit dem wir verschiedene typenähnliche Algorithmen implementieren können.

[1] Ist auch als *Schablonenmethode* bekannt.
[2] Sind abstrakte Methoden, die von konkreten Klassen implementert werden.

2.1.2 Lösung

Wir erstellen eine abstrakte Klasse mit primitiven Operationen und einer Template-Methode. Die primitiven Operationen werden von den erbenden konkreten Klassen implementiert und definieren somit die speziellen Schritte des Algorithmus bzw. die invarianten Teile des Algorithmus. Die Template-Methode schützt den Algorithmus und gibt dessen Grundgerüst vor, d.h. die Methode nutzt die primitiven Operationen ohne vorher deren konkrete Implementierung zu kennen.

Dieses objektorientierte Prinzip wird auch als *invertierter Kontrollfluss* [3] oder als *Hollywood-Prinzip* bezeichnet und kommt in vielen Entwurfsmustern zum Einsatz (Strategy, Factory Method, Observer,...). Es ermöglicht Objekten sich in die Operationen eines übergeordneten Objektes bzw. eines Moduls einzubringen ohne dabei bestimmen zu können, inwiefern diese Operationen genutzt werden. Das übergeordnete Modul definiert also, wie und zu welchem Zeitpunkt die Operationen aufgerufen werden, die von den darunter liegenden Modulen erweitert wurden - „Don‘t call us, we‘ll call you“[4].

Darüber hinaus können weitere Operationen von der Template-Methode verwendet werden, die nicht von erbenden Klassen erweitert wurden. Zum Beispiel *Hook*-Methoden, die in der abstrakten Klasse als nicht abstrakt definiert sind und folglich eine Implementierung besitzen, die per se auch aus einem leeren Rumpf bestehen kann. Erbende Klassen können - aber sie müssen nicht - diese *Hook*-Methoden überschreiben. Die Aufgabe der abstrakten Klasse ist es somit, die invarianten Schritte und die invarianten Implementierungen des Algorithmus vorzugeben.

2.1.3 Konsequenz

Vor- und Nachteile

Dieses Entwurfsmuster stellt ein grundlegendes Mittel zur Wiederverwendung von Code dar und löst das Problem von redundantem Code in verhaltensähnlichen Algorithmen durch herausfaktorisieren - „Refaktorisierung zur Verallgemeinerung“[5] - von gemeinsamen Verhalten. Somit werden Quelltextzeilen wiederverwendet, was dem Entwickler eine erneute Implementierung erspart. Das hat zur Folge, dass der Entwurf schneller fertig gestellt werden kann und die Implementation kleiner wird. Aber auch die Sicht auf die einzelnen Schritte des Algorithmus wird verbessert und ermöglicht eine schnellere und einfachere Modifikation. Das Design kann jedoch unnötig komplizierter werden, wenn Unterklassen sehr viele Methoden implementieren müssen, um den Algorithmus zu konkretisieren. Unterklassen sollten relativ einfach den abstrakten Algorithmus individualisieren. Es kann zum Nachteil werden, dass der Algorithmus einer fixen Struktur folgt, die wir nicht mehr ändern können.

Implementierungsaspekte

In C++ können wir den Zugriff auf die primitiven Operationen schützen und somit festlegen, dass nur die Template-Methode Zugriff auf sie hat. Deklarieren wir unsere primitiven Operationen als rein virtuelle Methoden, so müssen die Unterklassen die primitiven Operationen

[3] engl.: Inversion of Control oder Call-Back
[4] The Mesa Programming Environment - Richard E. Sweet
[5] Creating abstract superclasses by refactoring - William F. Opdyke, Ralph E. Johnson

überschreiben. Die Template Mehtode selbst sollte nicht überschrieben werden und kann als nicht virtuell definiert werden. In Java bietet sich für die Deklaration der Template-Methode eine finale Methode an.
Die Anzahl der zu überschreibenden Methoden sollte dabei so gering wie möglich gehalten werden, damit das Implementieren eines konkreten Algorithmus nicht unnötig erschwert wird. Als Namenskonvension für alle zu überschreibenden Methoden ist es vernünftig einen Präfix wie zum Beispiel `Do` zu verwenden, somit sehen andere Entwicker auf den ersten Blick, welche Methoden sie für ihre Erweiterungen zur Verfügung haben.

2.1.4 Beispiel

Die in diesem Beispiel illustrierten Implementationen stellen keine originalen Implementationen eines Compilers dar, sie beschreiben vielmehr den Ablauf der in diesem Beispiel verwendeten Methoden und wurden auf das Wesentliche gekürzt.

In vielen Programmiersprachen wird für die Ausgabe von Informationen an ein Ausgabegerät ein Ausgabestrom genutzt, der die Information nicht direkt an das entsprechenden Ausgabegerät weiter leitet, sondern einen Puffer zum Zwischenspeichern verwendet. Ist dieser Puffer gefüllt, so werden die enthaltenen Informationen an das Ausgabegerät weitergegeben und der Puffer wird geleert. Dieses Vorgehen erhöht die Effizienz der Ausgabe erheblich.

Auch in der Programmiersprache C++ werden solche Ausgabeströme (`ostream`) und zugehörige Puffer (`streambuf`) verwendet. Unterklassen von `ostream` sind u.a. `ofstream` und `ostringstream`, die sich am verwendeten Puffer unterscheiden (`filebuf` bzw. `stringbuf`). Stream-Klassen setzen ihr eigenes Streambuffer-Objekt. Die Streambuffer-Klassen `filebuf` und `stringbuf` definieren somit spezielle Puffer, da sie Teile der vererbten `streambuf`-Klasse neu implementieren.
Geben wir mit einem `ostream` Objekt (`cout`) eine Ausgabe mit dem «-Umleitungsoperator auf der Konsole aus, so könnte der zugehörige Aufruf wie folgt aussehen:

```
cout << "Hallo␣Welt\n";
```

Eine mögliche Implementierung des öffentlichen «-Umleitungsoperators bzw. der verwendeten Schreibmethode [6] könnte folgende Gestalt haben:

```
ostream& ostream::operator<<(const char s[])
{
    int i = 0;
    while (s[i] != '\0')
    {
        rdbuf()->sputc(s[i]);
        i++;
    }
    return *this;
}
```

Die `ostream`-Klasse erbt von der `ios`-Klasse unter anderem die `rdbuf`-Methode, die einen Zeiger auf ein `streambuf`-Objekt zurück gibt. Unterklassen von `ostream` (wie zum Beispiel die `ostringstream`-Klasse) implementieren diese Methode neu und geben einen entsprechenden Puffer zurück. In diesen Puffer schreiben wir mit der öffentlichen Member-Methode `sputc` [7]

[6] Eine mögliche Implementierung der Schreibmethode des Operators ist in der GNU C++ Library zu finden.
[7] Eine mögliche Implementierung dieser Methode ist zum Beispiel in der GNU C++ Library zu finden.

alle Zeichen die im übergebenen Parameter an unseren Operator enthalten waren.

```
int streambuf::sputc(char c)
{
  return (pptr >= epptr) ? overflow(c) : *pptr++ = c;
}
```

Die `sputc`-Methode ist in der `streambuf`-Klasse nicht als *virtual* deklariert und wird von Unterklassen wie der `stringbuf`-Klasse nicht überschrieben. Wenn beim schreiben des Puffers der `pptr`-Zeiger [8] die gleiche Stelle erreicht hat, wie der `epptr`-Zeiger [9] , dann ist der Puffer gefüllt und die Member-Methode `overflow` wird aufgerufen. Sie ist als *protected virtual* deklariert und kann somit von Unterklassen erweitert werden. Die `stringbuf`-Klasse enthält eine eigene Implementation der `overflow`-Methode und erbt von der `streambuf`-Klasse die `sputc`-Methode, die eine Template-Methode darstellt, da sie die `overflow`-Methode verwendet, ohne vorher die konkrete Implementierung in der `stringbuf`-Klasse zu kennen.

2.2 State

Das *State* [10] Entwurfsmuster ist ein einfaches objektbasiertes Verhaltensmuster. Es teilt das zustandsabhängige Verhalten von einem Algorithmus in Unterklassen auf und kapselt es vom Kontext ab. Somit kann sich - verborgen vor dem Klienten - der innere Zustand eines Objekts und damit auch sein Verhalten ändert. Es hat nach außen den Anschein, als hätte das Objekt seine Klasse gewechselt.

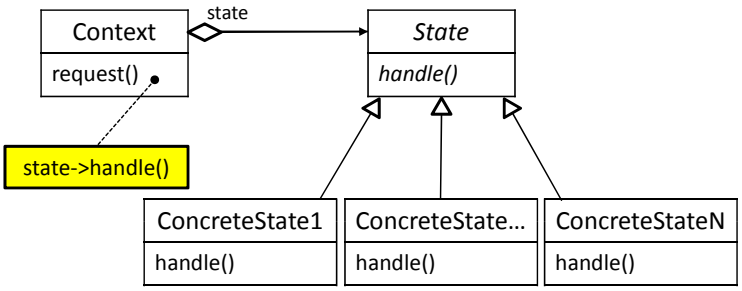

Abb. 2.2: Klassendiagramm nach [GOF94]

Mit diesem Muster erstellen wir also eine Klassenhierarchie, die zustandsspezifisches Verhalten in Zustandsunterklassen verlagert und mit der wir die entsprechenden zustandsabhängigen Objekte ohne Hilfe des Klienten verwalten und zur Laufzeit variieren können.

2.2.1 Problem

Unsere Klasse enthält schlecht erweiterbare und unübersichtliche zustandsverändernde Logik, die dahin tendiert, sich über die gesamte Klasse zu verbreiten und sehr komplex zu werden. Je nach Zustand wird nur ein geringer Teil der Klasse verwendet, die übrige - von anderen

[8] put pointer - aktuelle Position der Ausgabesequenz
[9] pointer to end - Endposition der Ausgabesequenz
[10] Ist auch als *objects for states* oder *Zustand* bekannt.

Zuständen abhängige - Logik bleibt ungenutzt. Dabei wird das zustandsabhängige Verhalten in strukturähnlichen Bedingungsanweisungen repräsentiert.

In diesem Kontext kann die verwendete Klasse als Zustandsautomat ausgedrückt werden, dabei müssen die einzelnen Zustände und die Übergangsbedingungen dem Klienten nicht bekannt sein und sollten relativ leicht austauschbar und erweiterbar bleiben.

2.2.2 Lösung

Wir verwenden eine abstrakte Zustandsklasse, die eine Schnittstelle für das zustandsabhängige Verhalten einer Kontextklasse definiert. Jede Klasse die diese Schnittstelle implementiert, stellt eine konkrete Zustandsklasse dar, in der ein bestimmtes zustandsabhängiges Verhalten der Kontextklasse gekapselt ist. Somit lagern wir jeden Zweig der Bedingungsanweisungen in eine eigene konkrete Zustandsklasse aus. Eine Kontextklasse verwendet ein Objekt einer konkreten Zustandsklasse, die den aktuellen Zustand repräsentiert. Das heißt, die Kontextklasse delegiert alle zustandsabhängigen Anfragen über die angebotene Schnittstelle an das aktuelle Zustandsobjekt. Das zu einem bestimmten Zeipunkt zu nutzende konkrete Zustandsobjekt kann vom Kontextobjekt bestimmt werden oder die konkreten Zustandsobjekte rufen selbst Zustandsübergänge hervor. Die Kontextklasse bietet außerdem dem Klienten eine Schnittstelle an.

2.2.3 Konsequenz

Vor- und Nachteile

Das Kapseln von zustandsverändernder bedingter Logik in eine Familie von Zuständen, kann ein einfacheres Design hervorbringen, dass die Sicht auf die Übergänge zwischen den Zuständen verbessert. Weil durch das Kapseln auch Teile der bedingten Logik aus dem Quelltext entfernt werden können, wird er leichter zu lesen und zu modifizieren sein. Wenn die Zustandsübergänge einer Klasse jedoch als einfach aufzufassen und keine Zustandserweiterungen geplant sind, kann sich - durch die Verwendung des *State* Entwurfsmuster - die Sicht auf die Logik verschlechtern. Dieses Muster bietet also eine bessere Möglichkeit zum strukturieren von großen bedingten Anweisungsblöcken und macht Zustandsübergänge explizit.

Implementierungsaspekte

Wie bereits erwähnt, können Zustandsübergänge vom Kontextobjekt oder von den konkreten Zustandsobjekten selbst hervorgerufen werden. Wenn die Kriterien, welche die Zustandswechsel definieren, flexibel bleiben sollen, ist es sinnvoll die Übergänge in der Kontextklasse zu implementieren und modifizierbar zu gestalten. Wenn hingegen Zustandsübergänge von den konkreten Zustandsklassen definiert werden, dann können sich die Kriterien nur schwer ändern lassen und die Kontextklasse muss des Weiteren eine Schnittstelle anbieten, um es den Zustandsobjekten zu ermöglichen, einen neuen Zustand explizit setzen zu können. Die konkreten Zustandsklassen werden häufig auch als *Singleton* implementiert. Dies ermöglicht eine leichtere Handhabung bei Erweiterungen des Entwurfs und gewährleistet, dass nur ein Zustandsobjekt von einem Zustand existieren kann.

2.3 Strategy

Das *Strategy* [11] Entwurfsmuster ist ein einfaches objektbasiertes Verhaltensmuster. Es definiert eine Familie von Algorithmen und kapselt sie. Die einzelnen verwandten Algorithmen können leicht ausgetauscht und zur Laufzeit variiert werden.

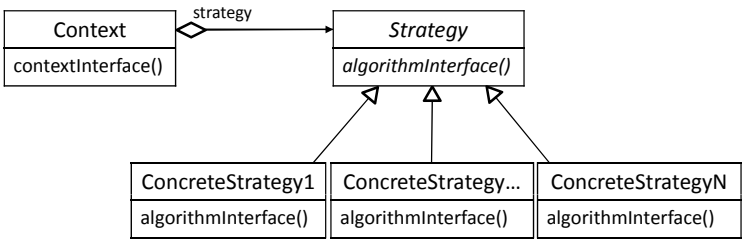

Abb. 2.3: Klassendiagramm nach [GOF94]

Mit diesem Muster erstellen wir also eine Klassenhierarchie von verhaltensähnlichen Algorithmen und kapseln jeden in einer Unterklasse vom Kontext ab.

2.3.1 Problem

Wir haben verschiedene Typen eines Algorithmus oder verschiedene Implementierungen des selben Typs eines Algorithmus, die womöglich in unübersichtlichen und schlecht erweiterbaren Bedingungsanweisungen definiert sind. Die Variierung der verschiedenen Algorithmen zur Laufzeit kann vom Klienten abhängig sein, dabei greifen die einzelnen Algorithmen auf gemeinsame Daten und eigene, typenspezifische Daten zu. Die Datenstruktur von unserem Algorithmus liegt außerdem unnötig offen und erschwert die Sicht auf die übrige Logik.

In diesem Kontext können unsere verschiedenen Algorithmen als *Library* aufgefasst werden.

2.3.2 Lösung

Wir erstellen eine abstrakte Strategieklasse, die eine Schnittstelle für alle zu unterstützenden Algorithmen deklariert. Eine konkrete Strategieklasse definiert einen konkreten Algorithmus unter Verwendung dieser Schnittstelle. Desweiteren erstellen wir eine Kontextklasse, die wir mit einer konkreten Strategieklasse konfigurieren wollen, d.h. das Kontextobjekt nutzt die Schnittstelle der konkreten Strategieklasse, um auf den konkreten Algorithmus zuzugreifen. Die Kontextklasse kann eine Schnittstelle für die konkreten Strategieobjekte bieten, damit sie auf die notwendigen kontextabhängigen Daten Zugriff haben.

2.3.3 Konsequenz

Vor- und Nachteile

Ein offensichtlich resultierender Vorteil des *Strategy* Musters ist die Wiederverwendbarkeit der einzelnen Algorithmen aus der Klassenhierarchie.

[11] Ist auch als *policy* oder *Strategie* bekannt.

Die Kapselung der Verhaltensweisen in einzelne Strategieklassen ermöglicht eine leichte Auswechlsung und Erweiterbarkeit der speziellen Algorithmen. Wie auch das *State* Muster, bietet das *Strategy* Muster eine Alternative zu bedingten Anweisungen und kann das Lesen und Modifizieren des Quelltextes erleichtern. Besteht für den Klienten eine Möglichkeit der Auswahl einer konkreten Strategie, so muss er diese und evtl. auch deren Vor- und Nachteile im Vorfeld kennen. Ein weiterer Nachteil kann entstehen, wenn die Algorithmen sehr unterschiedlich in ihren Implementierungen sind und nicht alle Operationen der angebotenen Schnittstelle nutzen. So kann es möglich sein, dass der Kontext zu stark gekapselt ist und Parameter erzeugt, die vom aktuellen Algorithmus nicht benötigt werden. Durch die Verwendung von gekapselten Algorithmen in Unterklassen entstehen bei der Anwendung eine höhere Anzahl an Objekten die verwaltet werden müssen. Ähnlich wie beim *State* Muster können die konkreten Algorithmen auch als *Flyweight* implementiert werden und somit von Kontextobjekten gemeinsam genutzt werden, was die Anzahl der konkreten Strategieobjekte verringert.

Implementierungsaspekte

Dieses Entwurfsmuster verfolgt - wie viele andere Muster auch - das *OC-Principle* und schütz somit vor Modifikationen und unterstützt Modulerweiterungen.

> *[...]when we inherit a module we keep the flexibility of changing what is not adapted any more to our new context - a flexibility without which we would lose one of the main attractions of the object-oriented method.* [MEY88]

Dieses Prinzip sollte beim Entwurf beachtet werden.

Die benötigten Daten, die von einer konkreten Strategie verwendet werden, können vom Kontext als Parameter übergeben werden. Ein anderer Ansatz ist, das Kontextobjekt selbst als Parameter mit zu übergeben, welches eine Schnittstelle für den Zugriff auf die nötigen Daten anbietet. Die Kontextklasse kann auch eine default Strategie anbieten, das bedeutet, wenn der Klient keine konkrete Strategie ausgewählt hat, wird eine vordefinierte zur Bearbeitung herangezogen.

2.3.4 Beispiel

Wie wir bereits aus dem *Template Method* Beispiel 2.1.4 wissen, stehen uns in C++ eine Reihe von unterschiedlichen Pufferarten zur Verfügung, die unter anderem die `overflow`-Methode neu definieren können. Die `streambuf`-Klasse „bietet virtuelle Funktionen für die Operationen, bei denen sich die Pufferstrategien unterscheiden (z.B. Funktionen, die Überläufe und Unterläufe behandeln“[12]. Jeder dieser Puffer ist von der `streambuf`-Klasse abgeleitet und stellt einen eigenen Datentyp dar.

Das folgende kleine Programm zeigt, wie wir ein `ostream`-Objekt - welches unserem Kontext entspricht - mit einer anderen Pufferstrategie konfigurieren können. Die verwendete Strategie ist hier ein `filebuf`-Objekt, dass die konkrete Strategie repräsentiert und die Strategieschnittstelle der `streambuf`-Klasse anbietet.

[12] [STR00] Seite 692 ff.

```
#include <iostream>
#include <fstream>

int main(){
  //speichern des cout Puffers
  std::streambuf *backup = std::cout.rdbuf();
  //ein Dateistrom-Objekt mit der Zieldatei 'foo.bar' anlegen
  std::ofstream ofs("foo.bar");
  //cout-Objekt mit dem Dateipuffer von unserem Dateistrom-Objekt konfigurieren
  std::cout.rdbuf(ofs.rdbuf());
  //Ausgabe in die zugehoerige Datei des Dateipuffers
  std::cout << "Das␣steht␣in␣der␣Datei␣'foo.bar'";
  //Ausgangszustand wiederherstellen
  std::cout.rdbuf(backup);
  //Dateistrom schliessen
  ofs.close();
  return 0;
}
```

2.4 Template Method vs. Strategy

Das *Strategy* ist dem *Template Method* Entwurfsmuster sehr ähnlich, da sie beide die Möglichkeit bieten, einen Algorithmus flexibler zu gestalten. Sie unterscheiden sich aber im Anwendungsbereich und in der Methodik. Das *Strategy* Muster wird verwendet, um verschiedene Algorithmen für eine Aufgabe variieren zu lassen, das *Template Method* Muster hingegen, erlaubt eine Variierung der Schritte in einem Algorithmus. Wir können also sagen, dass das *Strategy* Muster eine 'gröbere' Vorgabe, als das *Template Method* Muster ist, da dieses wiederum eine 'feiner' Kontrolle über die Operationen eines Algorithmus erlaubt. Ein weiterer Unterschied ist, dass das *Strategy* Muster Aggregation verwendet und somit Anfragen an einen bestimmten referenzierten Algorithmus delegiert. Das *Template Method* Muster verwendet Vererbung und bietet Unterklassen die Möglichkeit, Methoden zu überschreiben.

2.5 State vs. Strategy

Das *State* und das *Strategy* Muster haben strukturgleiche Klassendiagramme, aber verfolgen unterschiedliche Ziele. Das *Strategy* Entwurfsmuster dient dazu, dass ein Objekt die Ausführung eines Algorithmus an ein konkretes Strategieobjekt delegiert, was den auszuführenden Algorithmus bestimmt. Das *State* Entwurfsmuster hingegen, ermöglicht es einem Objekt auf einfache Art und Weise zwischen konkreten Zustandsobjekten zu variieren.

2.6 State/Strategy vs. Flyweight

Wenn die konkreten Zustands- bzw. Strategieobjekte nur durch ihren Typ definiert sind und somit keinen intrinsischen Zustand besitzen, dann sprechen wir von konkreten *Flyweight*-Objekten, die gemeinsam genutzt werden können.

Index

Literaturverzeichnis

[AIS77] Christopher Alexander, Sara Ishikawa, Murray Silverstein, Max Jacobson, Ingrid Fiksdahl-King, Shlomo Angel *A Pattern Language* Oxford University Press, New York, 1977

[ALE79] Christopher Alexander *The Timeless Way Of Building* Oxford University Press, New York, 1979

[FSB05] Eric Freeman, Elisabeth Freeman, Kathy Sierra, Bert Bates *Entwurfsmuster von Kopf bis Fuß* O'Reilly, 2005, ISBN: 3-89721-421-0

[GOF94] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides *Design Patterns - Elements of Reusable Object-Oriented Software* Addison-Wesley, 1995 (1st edition), ISBN: 0201633612

[GOF01] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides *Entwurfsmuster - Elemente wiederverwendbarer objekt-orientierter Software* Addison-Wesley, 2001 (korrigierter Nachdruck), ISBN: 3-8273-1862-9

[GOL02] Brandon Goldfedder *Entwurfsmuster einsetzen* Addison-Wesley, 2002, ISBN: 3-8273-1983-8

[MEY88] Bertrand Meyer *Object-Oriented Software Construction* Prentice Hall, 1988

[STR00] Bjarne Stroustrup *Die C++-Programmiersprache* Addison-Wesley, 2000, ISBN: 978-3-8273-1660-8