Software Design Patterns:
Paket „Kommando“

Gliederung:

1. **Das Pattern „Command“**
    1. **Zweck, Synonyme**
    2. **Kontext, Problem, Lösung**
    3. **Beispiel**
    4. **Vorteile**
    5. **Nachteile**
    6. **Varianten und Verwandschaften**
2. **Das Pattern „Command Processor“**
    1. **Zweck, Synonyme**
    2. **Kontext, Problem, Lösung**
    3. **Beispiel**
    4. **Varianten und Verwandschaften**
    5. **Vorteile**
    6. **Nachteile**
3. **Das Pattern „Visitor“**
    1. **Zweck, Synonyme**
    2. **Kontext, Problem, Lösung**
    3. **Beispiel**
    4. **Vorteile**
    5. **Nachteile**
    6. **Varianten und Verwandschaften**
4. **Quellenangaben**

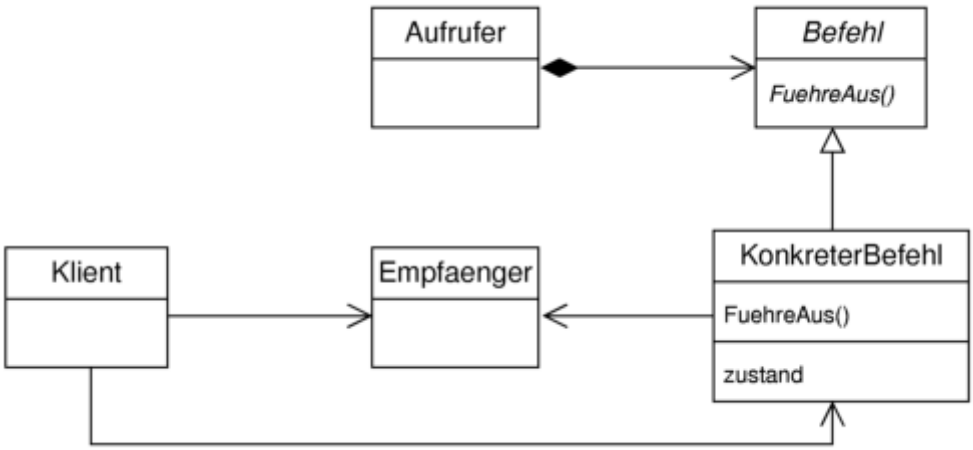
1. Das Pattern „Command“

1. Zweck, Synonyme:

Das Pattern „Command“ [kə’mænd] ist auch bekannt unter den Namen „Action“ [’ækʃən], „Transaction“ [træn’zækʃən], sowie deren deutschen Übersetzungen „Kommando“, „Befehl“, „Aktion“ und „Transaktion“. Es soll dazu dienen, Anfragen bzw. Methodenaufrufe als eigenständige Objekte zu kapseln. Es gehört zur Gruppe der Verhaltensmuster (behavioral pattern).

2. Kontext, Problem, Lösung:

Manchmal ist es notwendig, Anfragen an Objekte zu schicken, ohne irgendwelche Details über die auszuführende Aktion, das empfangende Objekt oder das die Aktion ausführende Objekt zu kennen. Dieses Problem tritt z.B. recht deutlich bei der Entwicklung von GUI-Toolkits auf. Dort wären es die einzelnen Elemente der GUI (Buttons, Menüeinträge, etc.), die Anfragen abschicken möchten, während sie gleichzeitig nicht wissen sollen, wer die Anfrage annimmt, wer sie ausführt oder was die Bearbeitung der Anfrage im Detail zur Folge hat.
In diesem Fall kann man sich des Command-Patterns bedienen, das die entsprechenden Methodenaufrufe kapselt und so den Aufruf einer Methode von deren Ausführung entkoppelt. Folgendes UML-Diagramm zeigt die am Pattern beteiligten Klassen:



Die beteiligten Klassen sind:

- Das Interface „Befehl“ mit der Methode „FuehreAus“.
- „KonkreterBefehl“, die das Interface „Befehl“ implementiert. Von dieser Klasse kann es beliebig viele geben (KonkreterBefehl1, KonkreterBefehl2, etc.). Sie speichert einen Verweis auf die Klasse „Empfaenger“ und der Rumpf ihrer Implementierung von „FuehreAus“ enthält typischerweise nur einen Aufruf einer Methode dieses Empfängers. „FuehreAus“ wäre also in diesem Fall nur eine Weiterleitung.
- „Klient“ erzeugt die einzelnen konkreten Befehle, weist ihnen Empfänger zu und weist den verschiedenen Aufrufern die konkreten Befehle zu.
- „Empfaenger“ führt dann tatsächlich die angeforderte Aktion aus und weiß als einzige Klasse, wie diese Ausführung im Detail aussehen soll. Empfänger kann

prinzipiell jede Klasse sein.
- „Aufrufer“ ist die Klasse, die die Ausführung eines Befehls auslöst, indem sie seine „FuehreAus“-Methode aufruft. Aufrufer hat eine Referenz auf „Befehl“ gespeichert und kennt tatsächlich nur das Interface „Befehl“, nicht die konkreten Implementierungen.

3. Beispiel:

Wie schon erwähnt, kann man das Pattern sehr gut in GUI-Anwendungen einsetzen. Es könnte z.B. die Anwendung (die hier den „Klient“ repräsentiert und die GUI-Funktionalität vielleicht nur als externes Modul einbindet) einen Button und einen Menüeintrag erzeugen (die hier die „Aufrufer“ repräsentieren), deren „OnClick“-Methoden jeweils nur die „FuehreAus“-Methode eines gespeicherten „Befehl“-Objekts aufrufen. Dann erzeugt die Anwendung zwei konkrete Befehle, ein mal den Befehl „ErzeugeNeueDatei“ und ein mal den Befehl „SpeichereDatei“. Beide Befehle implementieren das Interface „Befehl“, haben also die Methode „FuehreAus“. Die „FuehreAus“-Methode von „ErzeugeNeueDatei“ enthält nur den Methodenaufruf eines bestimmten Empfängerobjekts, das die tatsächliche Erzeugung einer neuen Datei ausführen kann. Entsprechend für „SpeichereDatei“.
Die Anwendung stellt die beiden Empfängerobjekte, die für die Ausführung verantwortlich sind, zur Verfügung und speichert jeweils einen Verweis auf den passenden Empfänger in „ErzeugeNeueDatei“ und „SpeichereDatei“. Schließlich speichert es einen Verweis auf das Objekt „SpeichereDatei“ im Button-Objekt und einen Verweis auf „ErzeugeNeueDatei“ im Menüeintrag.
Wenn nun der Anwender auf den Button klickt, wird dessen „OnClick“-Methode aufgerufen, welche selbst „FuehreAus“ beim gespeicherten Befehlsobjekt „SpeichereDatei“ ausführt. Diese führt schließlich die eigentlich zuständige Methode beim gespeicherten Empfängerobjekt aus.
Weitere Beispiele sind unter anderem:
- elektronische Finanztransaktionen (die Finanztransaktion wird z.B. an einem öffentlichen Terminal in Auftrag gegeben, aber erst von den zentralen Servern der Bankgesellschaft ausgeführt)
- Anfragen an Webserver
- Anfragen an Datenbanken
- aber auch: remote shells (rsh, ssh, telnet, ftp) und ähnliches wie z.B. Aktionen eines Spielers in einem Computerspiel, die zwar vom Client erzeugt, aber erst vom Server ausgeführt werden
- allgemein Benutzer-Interfaces (nicht nur grafische), bei denen Tasten bzw. Tastenkombinationen bestimmte Aktionen zugewiesen bekommen (siehe auch hier wieder Computerspiele, bei denen diese Zuweisung oft frei konfigurierbar ist)

4. Vorteile:

- Der Aufrufer muss die Implementierungsdetails der konkreten Befehle nicht kennen. Dadurch sind erstens konkrete Implementierungen austauschbar und zweitens kann man die Aufrufer-Klassen unabhängig vom Rest implementieren (z.B. von einem anderen Entwicklerteam).
- Man kann Befehle dynamisch, d.h. während der Laufzeit austauschen.
- Man kann in der „FuehreAus“-Methode eines konkreten Befehls auch mehr als

eine Methode aufrufen und auf diese Weise Makro-Befehle erstellen.
- Man kann Code und damit Platz sparen, indem man nur eine Reihe sehr grundlegender Befehle implementiert und für alle komplexeren Funktionen Makro-Befehle verwendet.

5. Nachteile:

- Man hat zusätzlichen Overhead beim Aufruf, da man ja eine Methode mehr aufrufen muss als in der Variante ohne das Befehlsobjekt.
- Die Klassenstruktur wird aufwändiger, da ja für jeden einzelnen konkreten Befehl eine eigene Klasse implementiert werden muss.
- Man hat Schwierigkeiten, an zusätzliche Parameter heranzukommen, da ja das Interface „Befehl“ fest ist und somit „FuehreAus“ immer die gleichen Parameter verlangt, egal zu welchem konkreten Befehl die Methode gehört. Auch die konkreten Befehle selbst kennen jeweils nur ihren Empfänger. Beim konkreten Befehl „SpeichereDatei“ möchte man z.B. wissen, welche Datei gemeint ist, da ja durchaus mehrere geöffnet sein können.

6. Varianten und Verwandschaften:

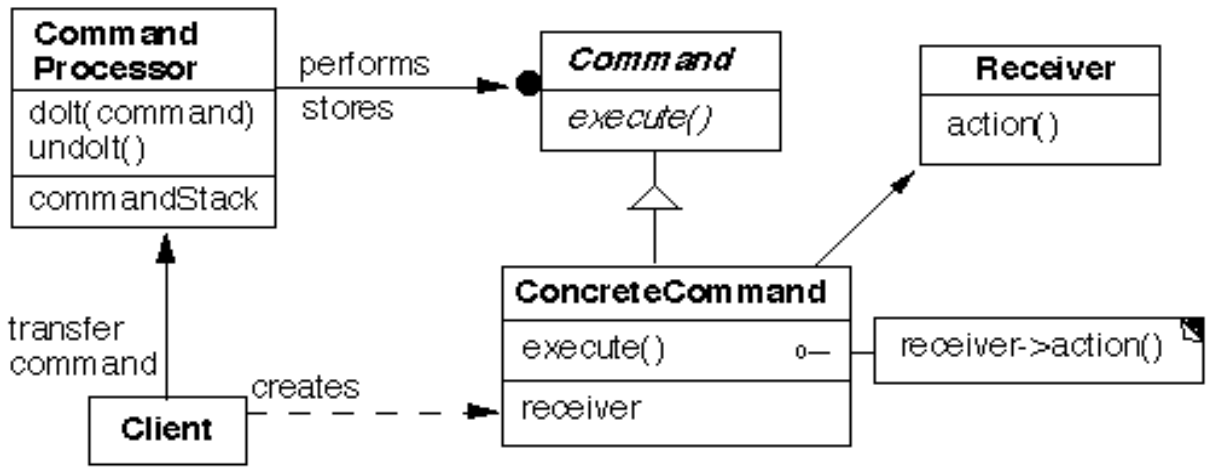
Eine Variante des Patterns könnte vor dem Aufruf von „FuehreAus“ in einer der Methoden von „Aufrufer“ testen, ob die Referenz auf das gespeicherte Befehlsobjekt eine NULL-Referenz ist, um in diesem Fall einen Absturz zu verhindern und stattdessen eine sinnvolle Fehlermeldung wie z.B. „Funktion noch nicht implementiert“ zu liefern. Dasselbe kann man auch in den Rümpfen der „FuehreAus“-Methoden der konkreten Befehle tun, um zu überprüfen, ob die Referenz auf den gespeicherten Empfänger gültig ist.

Das Pattern „Command“ ist verwandt mit einer Reihe von anderen Patterns und Konzepten:

- Composite: wenn man Makrobefehle verwendet, ist „Befehl“ eine „Component“, ein einfacher (kein Makro) konkreter Befehl ist ein „Leaf“, und Makrobefehle sind „Composites“
- Null Object: die andere Lösung, um „noch nicht implementiert“ umzusetzen: statt Tests auf Null-Pointer übergibt man einfach einen konkreten Befehl, dessen „FuehreAus“-Methode leer ist, und der folglich gar nichts tut
- Interpreter: die zu interpretierenden Ausdrücke dieses Patterns kann man als „Befehl“ betrachten
- Callbackfunktionen: das Pattern „Befehl“ ist die objektorientierte Variante dieses Prinzips, bei dem man einer Funktion einen Pointer auf eine andere Funktion übergibt, damit diese dann zum geeigneten Zeitpunkt „zurückgerufen“ werden kann. Auch hier kennt die aufrufende Funktion die Implementierung der aufgerufenen Funktion nicht und kann einen Pointer auf jede beliebige Funktion verwenden, solange diese die richtige Signatur (=das richtige Interface) hat.
- C++ function objects: eine Art Mittelweg zwischen Callbackfunktion und dem Pattern „Command“, bei dem der Funktionsaufrufoperator überladen wird.
- Closures: ein wichtiges Konzept in funktionalen Programmiersprachen, in denen Funktionen „first class values“ sind und damit als Parameter übergeben werden können wie alle anderen Variablen auch. Eine Funktion bekommt zu diesem Zweck die Information über alle ihre freien Variablen „mitgeliefert“.

2. Das Pattern „Command Processor“

1. Zweck, Synonyme:
Das Pattern „Command Processor“ [kə'mænd 'prəʊsesə] ist ebenfalls ein Verhaltensmuster und soll das Muster „Command“ um zusätzliche Funktionen erweitern sowie das Management der Befehle von der Ausführung trennen.
2. Kontext, Problem, Lösung:
Ein Programm, das eine Undo-Funktion mit beliebig vielen Undo-Levels implementieren möchte (und idealerweise schon das Pattern „Command“ verwendet), ist der übliche Kontext für das Pattern „Command Processor“. Es baut dabei auf das Pattern „Command“ auf und löst das Problem durch Erweiterung des „Command“-Interface um die neue Methode „Undo“ sowie durch Einführung einer neuen Klasse „Command Processor“, die sich um die Verwaltung aller auszuführenden und bereits ausgeführten Befehle kümmert und Namensgeber des Musters ist.
Folgendes UML-Diagramm erläutert die Klassenstruktur des Musters:



Beteiligt sind:
- Das Interface „Befehl“ (hier „Command“ genannt) mit den Methoden „FuehreAus“ und „Undo“
- verschiedene Klassen „KonkreterBefehl“, die genauso wie im Muster „Command“ funktionieren, jetzt aber zusätzlich noch „Undo“ implementieren müssen und folglich dafür sorgen müssen, dass sie den Zustand vor ihrer Ausführung wiederherstellen können (z.B. indem sie eine komplette Kopie des vorherigen Zustands erzeugen und im Speicher halten). Das Erzeugen der Wiederherstellungsinformationen erfolgt in der Methode „FuehreAus“, bevor die eigentliche Aktion bei der gespeicherten „Empfaenger“-Klasse ausgelöst wird
- die Klasse „Command Processor“, die jetzt zentraler Anlaufpunkt für das Ausführen von Befehlen ist. Ein Befehl wird jetzt nicht mehr ausgeführt, indem man seine „FuehreAus“-Methode direkt aufruft, sondern indem man die „FuehreAus“-Methode (hier „doIt“ genannt) des „Command Processor“ mit dem auszuführenden Befehl als Parameter aufruft. Diese Methode ruft dann ihrerseits

die „FuehreAus“-Methode des übergebenen Befehls auf und legt danach diesen Befehl auf einen Stack, den Undo-Stack. Die zweite Methode dieser Klasse, „Undo“ (hier „undoIt“ genannt), nimmt den obersten Befehl vom Undo-Stack und ruft dessen „Undo“-Methode auf

- verschiedene Klassen „Empfaenger“(hier:„Receiver“),die wie zuvor funktionieren
- die Klasse „Klient“ (hier: „Client“), die ebenfalls wie zuvor funktioniert
- verschiedene Klassen „Aufrufer“ (die in diesem Diagramm mit „Client“ zusammenfallen), die wie zuvor funktionieren, jetzt aber „FuehreAus“ beim „Command Processor“ aufrufen, anstatt beim gespeicherten Befehl selbst

3. Beispiel:

Praktisch alle Beispiele für das Pattern „Command“ sind auch Beispiele für „Command Processor“. Um beim oben angeführten GUI-Beispiel zu bleiben: In der OnClick-Methode des Buttons wird jetzt nicht mehr die „FuehreAus“-Methode des „SpeichereDatei“-Objekts aufgerufen, sondern es wird die „FuehreAus“-Methode des „Command Processor“-Objekts mit dem Objekt „SpeichereDatei“ als Parameter aufgerufen. Entsprechend für den Menüeintrag mit seinem Befehl „ErzeugeNeueDatei“. Man würde jetzt zur Nutzung der Undo-Funktionalität einen weiteren Button (oder irgendein vergleichbares GUI-Element) einbauen, dessen „OnClick“-Methode nur die „Undo“-Methode des Command Processor aufruft. Der Command Processor mit seinem Undo-Stack und die konkreten Befehle mit ihren „FuehreAus“- und „Undo“-Methoden kümmern sich darum, dass das wie erwartet funktioniert.

4. Varianten und Verwandschaften:

Zusätzlich zur Undo-Funktion ist es auch nicht weiter schwer, eine Redo-Funktion zu implementieren. Dazu muss der Command Processor nur einen weiteren Stack, den Redo-Stack, einführen. Immer wenn die „Undo“-Methode des Command Processor aufgerufen wird, wird dann nicht einfach der oberste Befehl vom Undo-Stack entfernt, sondern auf den Redo-Stack gelegt. Wenn die „Redo“-Methode des Command Processor aufgerufen wird, wird die „FuehreAus“-Methode des obersten Befehls des Redo-Stack ausgeführt und dieser Befehl wird wieder vom Redo-Stack auf den Undo-Stack gelegt.

Der Command Processor kann sich außerdem darum kümmern, dass beim Ausführen der Befehle jeweils ein Eintrag in einem Log angelegt wird.

Und er kann sich darum kümmern, dass Befehle auf mehrere Ausführungseinheiten (Prozessoren, Prozessorkerne, Prozessorkomponenten, Computer) aufgeteilt werden, falls mehrere zur Verfügung stehen. Das heißt, der Command Processor kümmert sich um Scheduling.

Falls die verarbeiteten Befehle Finanztransaktionen sind, kann sich der Command Processor auch um die Buchhaltung (Accounting) kümmern.

Das Pattern „Command Processor“ ist außerdem verwandt mit:

- dem Pattern „Memento“, das man verwenden kann, um die Zustandsinformation für die „Undo“-Methode der Befehle zu verwalten
- Singleton, da man im Normalfall immer nur genau ein „Command Processor“-Objekt benötigt
- „Strategy“, um verschiedene, austauschbare Logging-Strategien verwenden zu

können (vollständiges Logging, Logging nur für die wichtigsten Informationen, gar kein Logging, etc.)
- noch ein mal „Strategy“, um verschiedene Scheduling-Strategien verwenden zu können
- „Null Object“ in Zusammenarbeit mit „Strategy“ für eine Logging-Strategie, die gar nichts logt

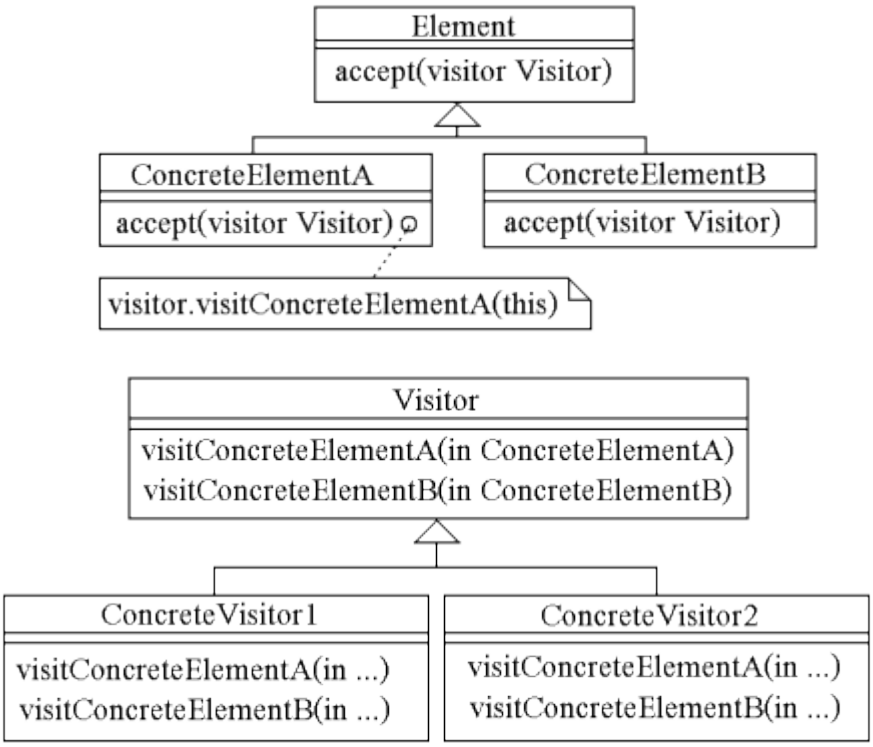
5. Vorteile:
- Man kann durch das Scheduling die Rechenarbeit auf mehr Prozessoren aufteilen, hat so mehr Rechenleistung zur Verfügung und kann die Rechenleistung gezielt an den Bedarf anpassen
- Logging & Accounting
- Autorisierung (der Command Processor führt Befehle nur aus, wenn der Aufrufer dazu berechtigt ist)
- Undo/Redo

6. Nachteile:
- Noch mehr zusätzlicher Overhead als schon beim Pattern „Command“, denn jetzt sind zwei zusätzliche Aufrufe notwendig („FuehreAus“ von „Command Processor“ und „FuehreAus“ des konkreten Befehls)
- Scheduling bringt bei out-of-order-execution neue Probleme mit sich
- Die Undo-Funktion kann eventuell viel Speicher kosten, falls wirklich für jeden ausgeführten Befehl ein komplettes Abbild des alten Zustands zurückbehalten werden muss
- Der Aufrufer weiß nicht, ob ein Befehl tatsächlich ausgeführt wurde oder nicht, da ja Aufruf von Ausführung getrennt wurde und insbesondere durch Autorisierung und Scheduling die erfolgreiche Beendigung eines Befehls einige Zeit nach dem Aufruf von „FuehreAus“ eintreten kann
- zusätzlicher Overhead durch Logging
- „Aufrufer“ muss jetzt nicht nur das Interface „Command“ kennen, sondern auch die Klasse „Command Processor“

3. Das Pattern „Visitor“

1. Zweck, Synonyme:
Das Pattern „Visitor“ ['vɪsɪtə], auch bekannt unter dem deutschen Namen „Besucher“, ist ebenfalls ein Verhaltensmuster und soll dazu dienen, Struktur und Funktion (oder auch: Daten und Operationen) zu trennen. Es hilft dadurch, das open/closed-Prinzip (offen für Erweiterungen, geschlossen für Änderungen) zu realisieren.

2. Kontext, Problem, Lösung:
Man hat eine Anwendung mit vielen unterschiedlichen Klassen zur Repräsentation der Daten, die alle eine Reihe von Funktionen mit identischer Signatur und ähnlicher Funktionsweise haben. Die Anzahl und Implementierung dieser Klassen ändert sich im Laufe der Zeit kaum oder gar nicht. Dafür möchte man aber regelmäßig neue Funktionen zu allen diesen Klassen hinzufügen (zu allen gleichzeitig) oder bestehende ändern, ohne dabei einen unüberschaubaren Aufwand zu haben.
Dieses Problem wird durch das „Visitor“-Pattern gelöst, indem es die für alle Klassen gleichen Funktionen selbst in eigene Klassen verschiebt, so dass es dann zusätzlich zu den vorhandenen (Daten-)Klassen noch eine weitere Klasse für jede Funktion gibt. Auf diese Weise simuliert das „Visitor“-Pattern „double dispatch“ in Sprachen, die eigentlich nur „single dispatch“ unterstützen.
„double dispatch“ meint dabei, dass für eine mehrfach überladene Funktion zur Laufzeit eine konkrete Funktion ausgewählt wird, abhängig von den Typen von zwei Argumenten der Funktion. Bei „single dispatch“ kann dagegen der Typ von nur einem Argument der Funktion zur Auswahl der konkreten Funktion herangezogen werden.
Die folgenden beiden Diagramme zeigen die Klassenstruktur des Musters:



Die beteiligten Klassen sind:

- Das Interface „Element“ mit der Methode „Akzeptiere“. Dieses Interfaces wird von allen bisherigen (Daten-)Klassen implementiert und die Methode „Akzeptiere“ ersetzt alle Funktionen dieser Klassen, die bisher für alle (Daten-) Klassen implementiert waren.
- Verschiedene Klassen „KonkretesElement“, die die bisherigen (Daten-)Klassen ersetzen und die alle das Interface „Element“ implementieren. Für „Akzeptiere“ wird „double dispatch“ erreicht (dispatch basierend auf der konkreten Unterklasse von „Element“ und der konkreten Unterklasse des übergebenen „Visitor“-Objekts) , indem alle Implementierungen von „Akzeptiere“ den Methodenaufruf nur weiterleiten zu einer konkreten Methode des übergebenen „Visitor“-Objekts. So besteht der Rumpf von „KonkretesElementA.Akzeptiere“ nur aus dem Aufruf der Methode „Besucher.BesucheKonkretesElementA“ mit dem Objekt selbst (also „this“) als Argument
- Das Interface „Besucher“, das für jeden existierenden Element-Typ (also für jede Unterklasse von „Element“) „KonkretesElementX“ eine Methode „BesucheKonkretesElementX“ beinhaltet
- Verschiedene Klassen „KonkreterBesucher“, die das Interface „Besucher“ implementieren und eine Objekt-Abstraktion der bisher in allen Element-Klassen vorhandenen Funktionen darstellen.

3. Beispiel:

Das Paradebeispiel ist ein Compiler, dessen primäre Datenstruktur ein abstrakter Syntaxbaum ist. Dieser Baum besteht aus vielen, heterogenen Knoten, die jeweils noch einen oder mehrere Unterknoten haben können. Das „heterogen“ bezieht sich dabei auf die Typen der Knoten. Mögliche Knotentypen sind z.B. „Integer-Variable“, „Zuweisungs-Anweisung“, „For-Schleife“, etc. . Üblicherweise existiert wenigstens für jedes syntaktische Konstrukt der Quellsprache des Compilers ein Knotentyp. Die unterschiedlichen Knotentypen gehen also in die Dutzende, verändern sich dafür aber über Jahre hinweg gar nicht und werden auch nicht mehr oder weniger, da ja die Quellsprache sich nur langsam weiterentwickelt oder sogar standardisiert ist.
Auf jedem der Knotentypen sind nun eine Reihe von Funktionen wie „TypÜberprüfung“, „ErzeugeCode“, „PrettyPrint“, „ListeBezeichnerAuf“, „Optimiere“, etc. definiert.
Bei Verwendung des „Visitor“-Pattern würde jeder Knotentyp Unterklasse von „Besucher“ und die einzige auf allen Knotentypen definierte Methode wäre „Akzeptiere“. Jede der Funktionen „TypÜberprüfung“, etc. würde zu einer eigenen Unterklasse von „Besucher“ und würde für jeden Knotentyp eine eigene Methode implementieren. Auf diese Weise ist es nun möglich, neue Funktionen (z.B. eine weitere „ErzeugeCode“-Funktion, die anderen, eventuell besseren Code erzeugt) zu implementieren, ohne irgendetwas an der vorhandenen Klassenstruktur ändern zu müssen. Alle existierenden Unterklassen von „Element“ und alle existierenden Unterklassen von „Besucher“ würden unverändert bestehen bleiben und müssten (falls sie in eigenen Modulen stehen) nicht ein mal neu kompiliert werden.
Zum Aufruf von „ErzeugeCode“ auf der Wurzel des Baums würde man jetzt nicht mehr die „ErzeugeCode“-Methode des Wurzelknotens direkt aufrufen, sondern man

würde die „Akzeptiere“-Methode des Wurzelknotens aufrufen und ihr ein Objekt vom Typ „ErzeugeCodeBesucher“ übergeben.
Ein anderes Beispiel sind 3D-Anwendungen, deren Daten ebenfalls oft aus Bäumen mit heterogenen Knotentypen bestehen. Die einzelnen Knotentypen sind z.B. „NURBS-Objekt“, „Mesh-Objekt“, „TransformationsObjekt“, etc. . Auf diesen Knotentypen sind nun verschiedene Funktionen wie „RendereAufBildschirm“, „RendereInDatei“, „SpeichereInDatei“, „RendereAufBildschirm2“ (anderer Algorithmus), etc. definiert.

4. Vorteile:

- Das Hinzufügen neuer Funktionen ist einfacher
- Der komplette Code eines Algorithmus ist gebündelt in einer Klasse und nicht verstreut in dutzenden Klassen
- „Besucher“-Klassen können auch einen Zustand haben, der von einem Aufruf bis zum nächsten bestehen bleibt

5. Nachteile:

- Alle „KonkreterBesucher“-Klassen implementieren das gleiche Interface, obwohl unterschiedliche Algorithmen eventuell unterschiedliche Parameter benötigen und unterschiedliche Ergebnisse zurück liefern. Die Übergabe und Zurückgabe von Parametern wird dadurch erschwert und der Zustand der „Besucher“-Klassen wird zwingend notwendig
- Man kann das Pattern nur schwer nachträglich implementieren, da das ja ein refactoring des gesamten bisherigen Codes erfordert, eventuell sogar mehr als nur ein refactoring. Das bedeutet im Prinzip, dass man sich schon vor der Implementierung für oder gegen das Pattern entscheiden muss
- Das Hinzufügen neuer Knotentypen (also Unterklassen von „Element“) ist nun schwierig, da man dafür ja „Besucher“ und alle seine Unterklassen anpassen muss
- Zusätzlicher Overhead wegen des simulierten „double dispatch“ in „Akzeptiere“
- Die einzelnen konkreten Besucher benötigen üblicherweise eine ganze Reihe von Attributen der besuchten Elemente/Knoten. Da sie nun aber keine Methoden dieser Elemente mehr sind, benötigen sie dafür viele „Get“- und vielleicht sogar „Set“-Methoden, was im Widerspruch zum Kapselungsprinzip der objektorientierten Programmierung steht

6. Varianten und Verwandschaften:

Eine Frage bei der Implementierung des Pattern dreht sich darum, wer sich um die Traversierungsreihenfolge des Baums kümmern soll. Man kann die Verantwortung dafür entweder an die Unterklassen von „Element“ übergeben, so dass die „Akzeptiere“-Methode jeder „KonkretesElementX“-Klasse selbst die „Akzeptiere“-Methoden der Unterelemente/Unterknoten aufruft. In dem Fall müssen alle Besucher die gleiche Reihenfolge verwenden. Man kann die Verantwortung auch an die „Besucher“-Unterklassen übergeben. In dem Fall müsste man den Traversierungscode in jeder dieser Klassen neu implementieren. Oder man kann die Verantwortung an eine zusätzliche „Iterator“-Klasse übergeben.
Verwandt ist „Besucher“ hauptsächlich mit „Composite“ (dank des Interface „Element“ und seiner Unterklassen) und mit „Interpreter“ (die „Besucher“-Unterklassen führen die Interpretation der Knoten aus).

4. Quellenangaben

Meine Quellen sind:

- Buschmann, Henney, Schmidt: Pattern-oriented Software Architecture
- Eilebrecht, Starke: Patterns kompakt
- Gamma, Helm, Johnson, Vlissides: Design Patterns
- www.wikipedia.org
- Professor Roger Whitney von der San Diego State University (www.eli.sdsu.edu)

Die Illustrationsquellen sind:

- Abbildung Seite 2: http://upload.wikimedia.org/wikipedia/de/9/93/KommandoMuster_Klassen.png , von Wikipedia-Benutzer Thomasv selbst gezeichnet
- Abbildung Seite 5: http://www.eli.sdsu.edu/courses/spring98/cs635/notes/command/cmdProcessorStructure.gif (© 1998, All Rights Reserved, SDSU & Roger Whitney)
- Abbildung 1 & 2 auf Seite 8: selbst gezeichnet mittels Microsoft Paint und IrfanView