Paket Kommando

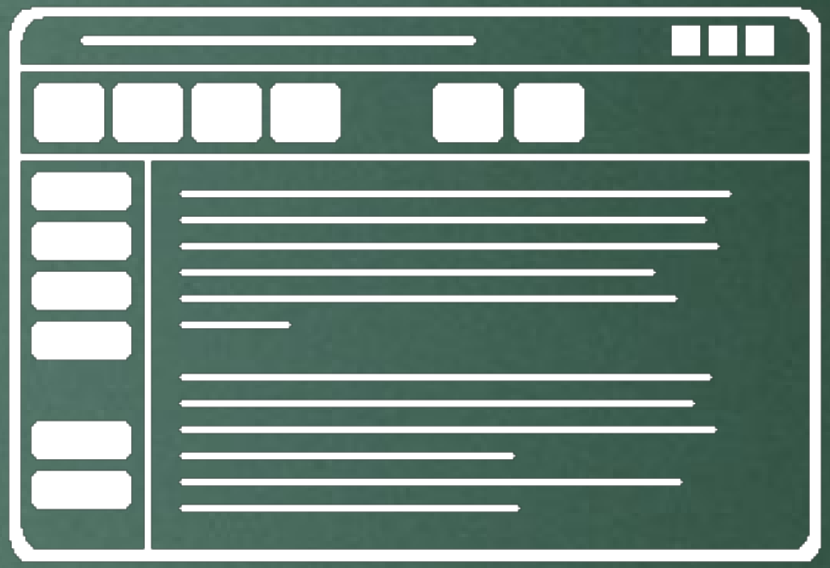
- Command
- Command Processor
- Visitor

Command

- Command [kə'mænd]
  - aka: Action ['ækʃən]
  - aka: Transaction [træn'zækʃən]
- Verhaltensmuster
- Zweck: Anfragen/Methoden-aufrufe als Objekte kapseln

Command: Motivation

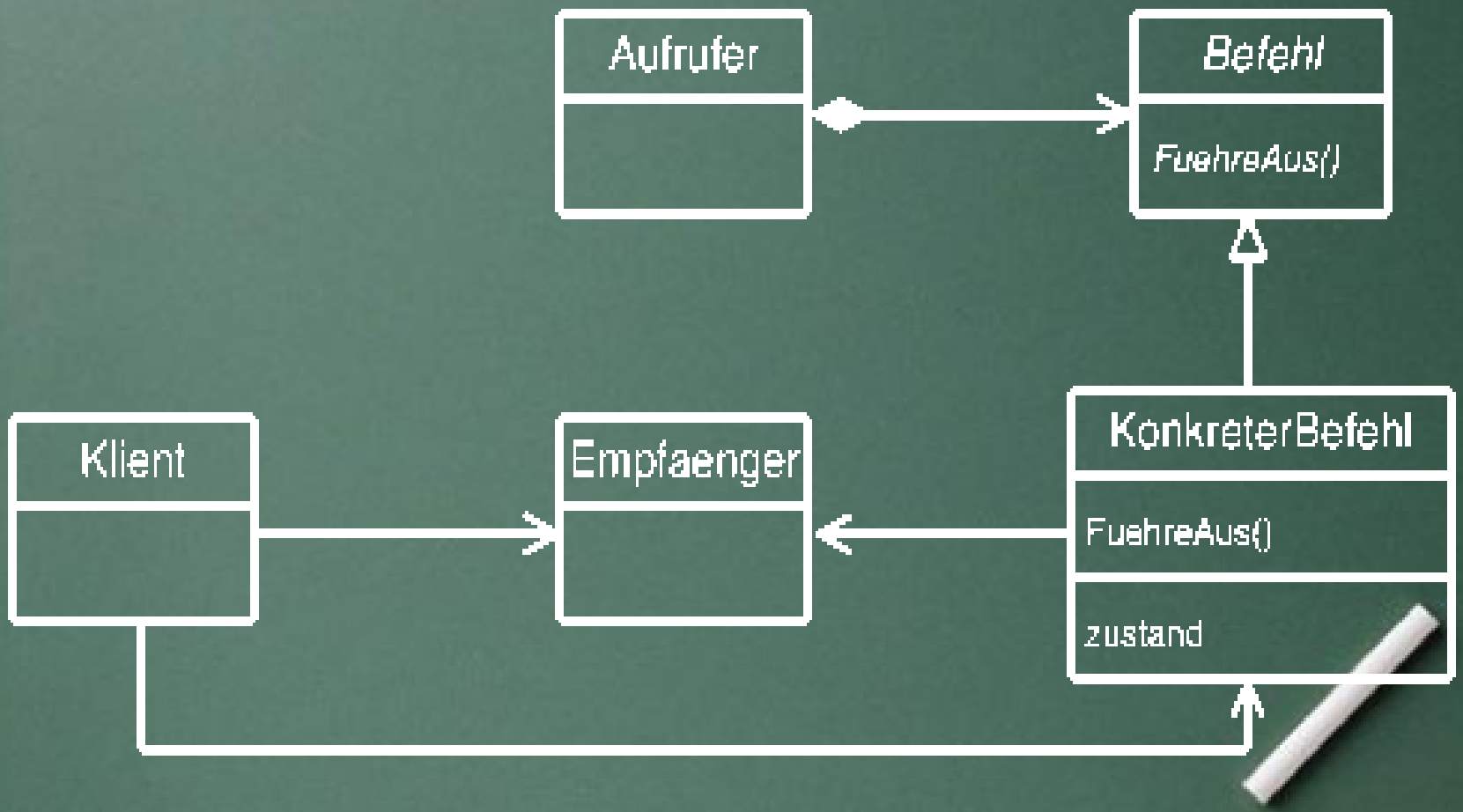
- In GUI lösen Buttons und Menüeinträge Aktionen aus, dürfen aber die Implementierungsdetails nicht kennen.



Command: Lösung

- Entkopple Erstellung und Ausführung der Befehle durch Kapselung in Objekte
- Übergib an jeden Button und jede Schaltfläche der GUI geeignetes Befehlsobjekt (z.B. "Datei Speichern")
- Erstellungs- und Ausführungszeitpunkt nicht mehr identisch

Command: Lösung

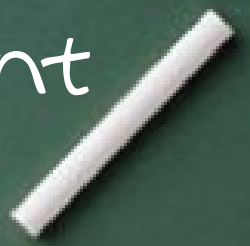


Command: Mitspieler

- Befehl: Interface mit Methode FuehreAus
- KonkreterBefehl:
  - Definiert Bindung zwischen Empfaenger & Aktion
  - Implementiert FuehreAus(), indem entsprechende Operation bei Empfaenger aufgerufen wird
- Klient: erzeugt KonkreterBefehl, setzt Empfaenger

Command: Mitspieler

- Empfaenger: weiß, wie Operation auszuführen ist, die mit einem bestimmten Befehl verbunden ist. Kann prinzipiell jede Klasse sein.
- Aufrufer: Fordert einen Befehl auf, seine Aktion auszuführen. Kennt nur das Interface Befehl.



Command: weitere Beispiele

- elektronische Finanztransaktionen
- Anfragen an Webserver
- Anfragen an Datenbanken
- Im weitesten Sinne: remote shells und ähnliches (rsh,ssh,telnet,ftp)

Command: Pro's

- Aufrufer des Commands kennt Implementierungsdetails nicht:
  - Implementierung austauschbar
  - Aufrufer sieht Command als Black Box
- Commands dynamisch austauschbar
- Makro-Commands möglich
- Platz sparen (siehe Ende)

Command: Con's

- Zusätzlicher Overhead beim Aufruf
- Aufwändigere Klassenstruktur:
  - Für jedes konkrete Command muss eigene Klasse implementiert werden
- Schwierigkeit, an zusätzliche Parameter heranzukommen (Command kennt nur Receiver)

Command: Verwandschaft

- Command ist verwandt mit:
  - Composite
  - Null Object
  - Interpreter
  - CallBackfunktionen
  - Closures
  - C++ Function Objects

Command Processor

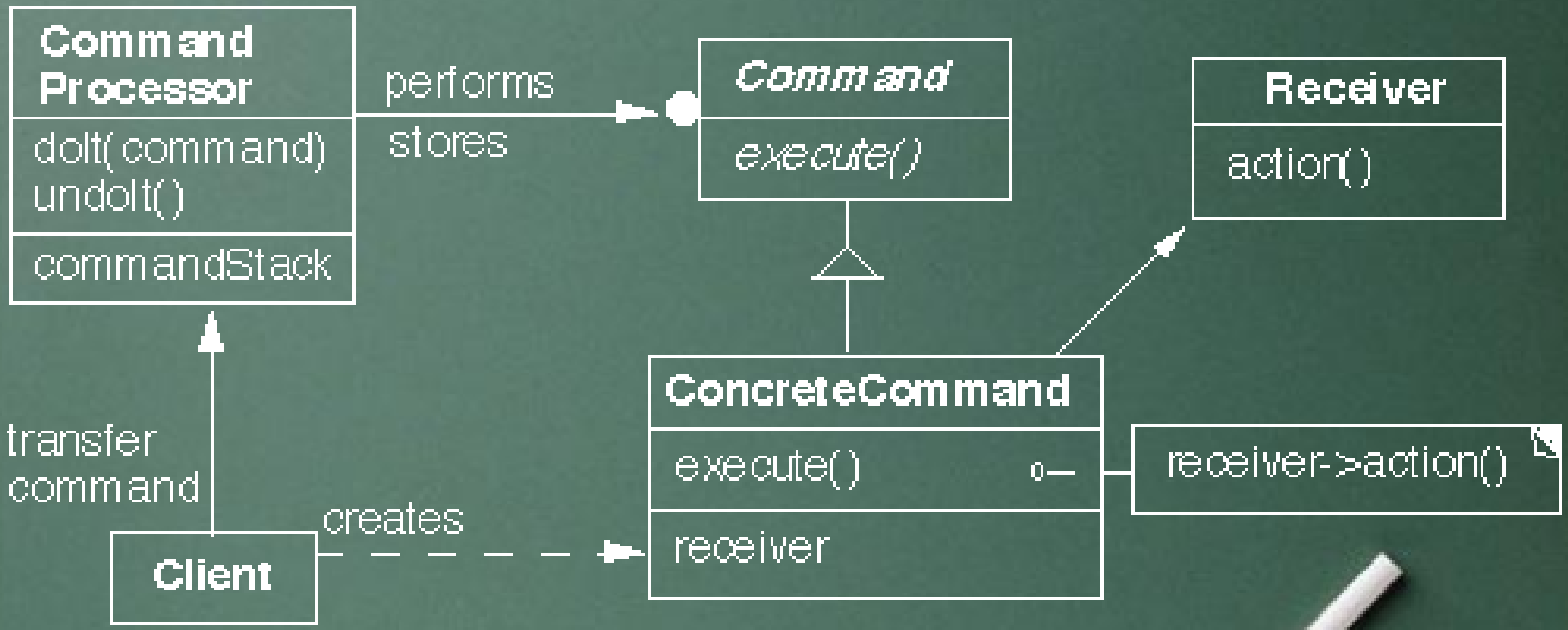
Command Processor

- Command Processor [kə'mænd 'prəʊsesə]
- Verhaltensmuster
- Zweck: Designmuster Command um zusätzliche Funktionen erweitern, Management der Commands von Ausführung trennen

Cmd. Processor: Motivation

- Ein Programm (verwendet idealerweise schon das Command Pattern) möchte Undo-Funktion mit beliebig vielen Levels implementieren
- Naive Lösung beschränkt auf ein Undo-Level oder führt zu Problemen

Command Processor: Lösung



Cmd. Processor: Beteiligte

- Command Processor: Speichert und verwaltet alle Commands, kümmert sich eventuell um Scheduling
- Command: wie im Pattern davor, aber jetzt mit Undo-Funktion
- ConcreteCommand: wie zuvor + Undo

Cmd. Processor: Beteiligte

- Receiver, Client: wie zuvor
- Aufrufer: in dieser Graphik identisch mit Client
- Manchmal zusätzlich "Controller": übernimmt Rolle von Client und Aufrufer, erzeugt Commands aus Anfragen

Command Processor: Pro's

- Ausnutzung mehrerer Prozessoren durch Scheduling
- Logging & Accounting/Buchhaltung
- Autorisierung
- Undo
- Redo (zweiter Stack)

Command Processor: Con's

- Zusätzlicher Overhead gegenüber Command: doppelt indirekter Aufruf
- Scheduling: out-of-order-execution
- Undo kostet eventuell viel Speicher
- Aufrufer weiß nicht, ob Command tatsächlich ausgeführt
- Overhead durch Logging

Command Processor: Die Sippe

- Memento (Undo)
- Strategy (fürs Logging)
- Strategy (fürs Scheduling)
- Null Object (Logging)
- Singleton

Visitor

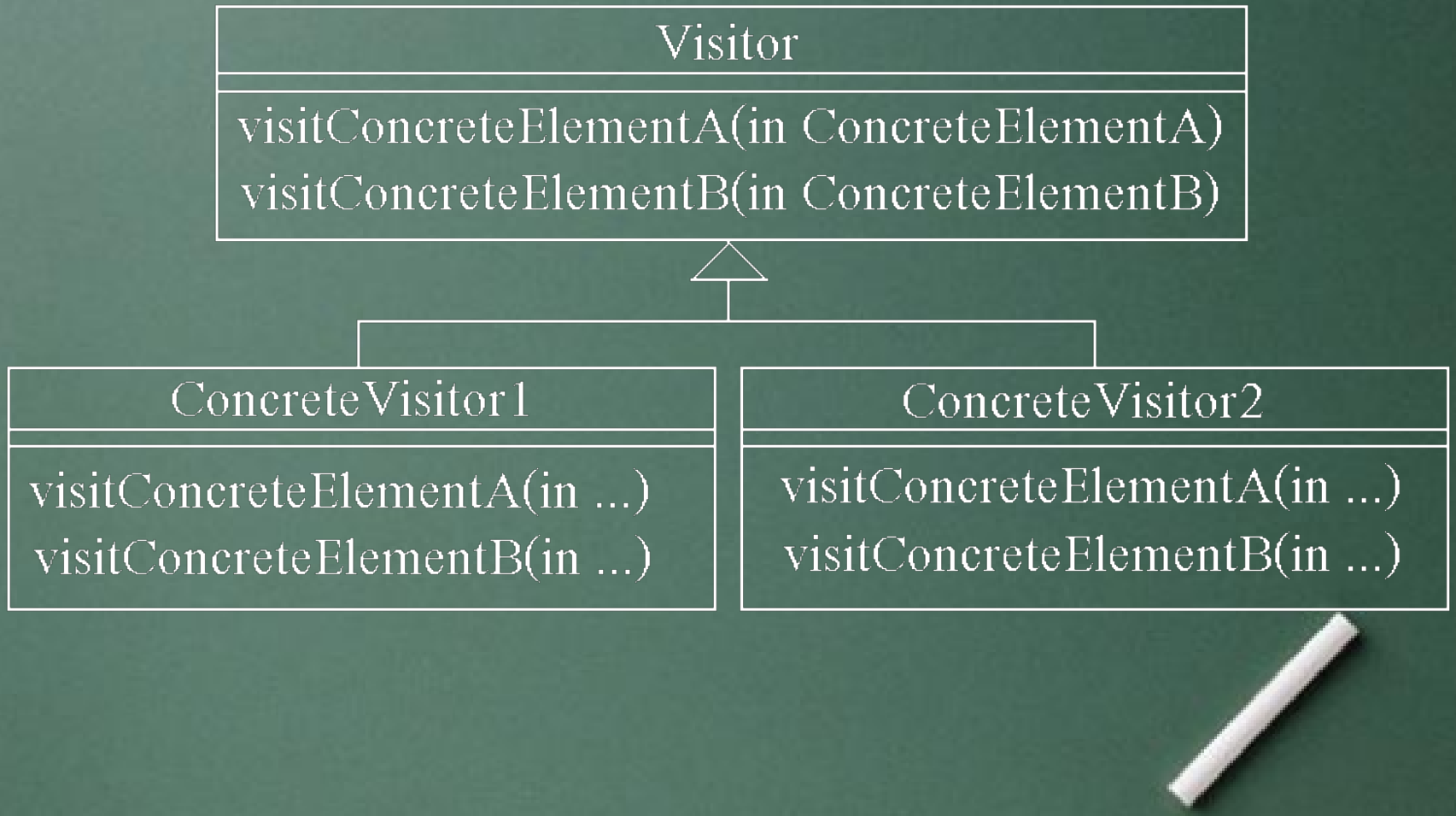
Visitor

- Visitor ['vɪsɪtə]
- Verhaltensmuster
- Zweck: Trennung von Struktur und Funktion (Daten und Operationen). Realisiert open/closed-Prinzip.

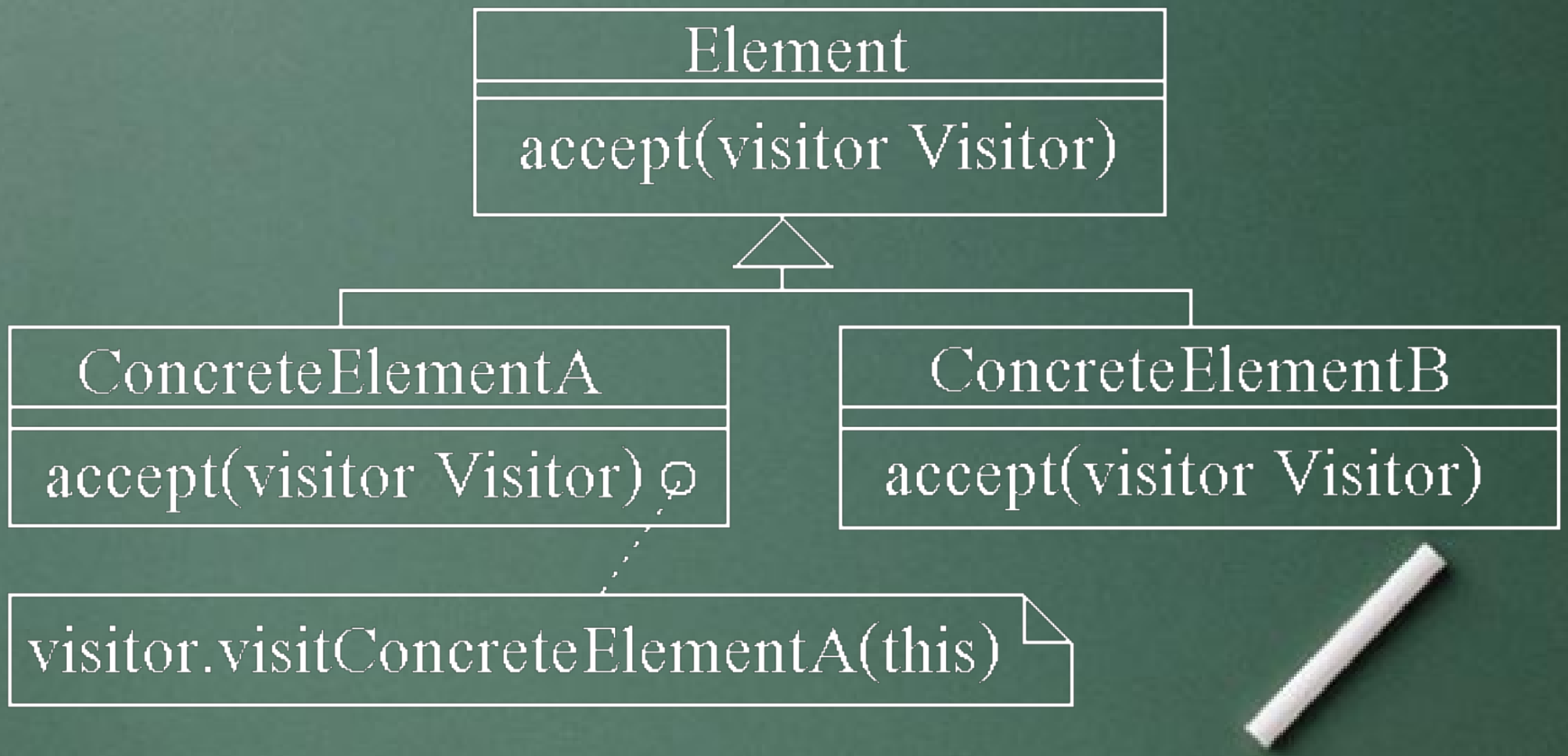
Visitor: Motivation

- Ein Compiler implementiert auf einer Menge von Knotentypen Funktionen wie TypeCheck, GenerateCode, PrettyPrint
- Hinzufügen neuer Funktionalität (wie z.B. GenerateOptimizedCode) betrifft alle Knotentypen -> großer Änderungsaufwand, alles neu kompilieren

Visitor: Lösung



Visitor: Lösung



Visitor: Wer macht mit

- Visitor: Interface. Enthält eine visitElementXY-Methode für jeden Element-Typ XY
- ConcreteVisitor: implementiert Visitor. Jede der implementierten Methoden ist ein Teil des jeweiligen Algorithmus

Visitor: Wer macht mit

- Element: Interface. Enthält nur die Methode accept(Visitor)
- ConcreteElement: implementiert Element. Simuliert double dispatch in der Implementierung von accept durch Aufruf der jeweils für sich selbst passenden Methode von Visitor

Visitor: weitere Beispiele

- 3D-Objekte: Bestehen aus vielen Elementtypen (z.B. NURBS, triangle mesh, transformation, etc.), die jeweils wieder Kinder haben können. Auf diese werden Funktionen wie renderViaRasterizing, renderViaRaytracing, etc. angewandt

Visitor: Variante

- Falls alle konkreten Visitors die gleiche Traversierungsreihenfolge verwenden, kann man Traversierung auch in der accept-Methode belassen:
  - accept leitet Visitor selbständig an accept der Kindelemente weiter & ruft davor/danach/dazwischen die visit-Funktion des Visitors auf

Visitor: Pro's

- Visitor kann auch Zustand (State) haben, der von einem Aufruf zum nächsten Bestehen bleibt
- Hinzufügen neuer Algorithmen leicht
- Der komplette Code eines Algorithmus ist in einer Klasse konzentriert, nicht verstreut

Visitor: Con's

- Alle konkreten Visitors implementieren gleiches Interface -> Parameterübergabe & Funktionswertrückgabe erschwert -> State zwingend notwendig
- Visitor Pattern ist nur schwer nachträglich zu implementieren
- Hinzufügen neuer Elementtypen schwer

Visitor: Con's

- Zusätzlicher Overhead beim Aufruf wegen double dispatch
- Viele get-Methoden contra Kapselung

Visitor: Verwandschaft

- Iterator: Im Gegensatz zu Iterator kann Visitor Elemente verschiedener Typen Besuchen
- Interpreter: Visitor führt die Interpretation aus
- Composite

But now for something completely different

Antipatterns

- Design by committee
- Escalation of commitment (siehe auch Sunk Cost Fallacy)
- Vendor lock-in
- Groupthink
- Gold plating

- Circular dependency
- God object

Danke, dass Sie nicht eingeschlafen sind!

Quellen

- Buschmann, Henney, Schmidt: Pattern-oriented Software Architecture
- Eilebrecht, Starke: Patterns kompakt
- Gamma, Helm, Johnson, Vlissides: Design Patterns
- www.wikipedia.org

Quellen

- Prof. Roger Whitney, San Diego State University (www.eli.sdsu.edu)
- www.thelinuxBox.org/?p=24