Entwurfsmuster für Notes-Entwickler

Session 1: Entwicklung

Thomas Bahn
assono GmbH
www.assono.de
tbahn@assono.de
+49/4307/900-401

Worum geht es?

- Motivation
  - Was sind Entwurfsmuster (Design Patterns)?
  - Wir können Entwurfsmuster **dir** helfen?

- Grundlagen
  - Objektorientierte Programmierung (OOP)
  - OO-Entwurfsprinzipien
  - Entwurfsmuster – eine Definition
  - Beschreibungen von Entwurfsmustern
  - Klassifizierung

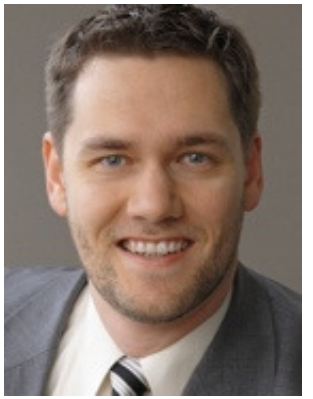
Worum geht es? (forts.)

- Einige wichtige Entwurfsmuster im Detail
  (in der Reihenfolge, wie ich sie für das Beispiel brauche)
  - Kompositum – Composite
  - Strategie – Strategy
  - Iterator
  - Dekorierer – Decorator
  - Beobachter – Observer
  - Proxy
  - Befehl – Command
  - Befehlsprozessor – Command Processor
  - Singleton

Worum geht es? (forts.)

- Weitere Entwurfsmuster in aller Kürze
  - Fassade – Façade
  - Adapter
  - Schablonenmethode – Template Method
  - Zuständigkeitskette – Chain of Responsibility
  - Fabrik, Fabrikmethode & Abstrakte Fabrik – Simple Factory, Factory Method & Abstract Factory
  - Model View Controller (MVC)
- OOP in LotusScript – Einschränkungen und (einige) Lösungen
- Quellen

Organisatorisches



- Wer bin ich?
  - Thomas Bahn, IT-Berater, Dipl.-Math., 37 Jahre
  - Mitgründer und -inhaber der assono GmbH
  - seit 1997 entwickle ich mit Java und RDBMS, z. B. Oracle (Oracle Certified Professional)
  - seit 1999 mit IBM Lotus Notes/Domino (IBM Certified Advanced Application Developer – Lotus Notes and Domino R4 – 7, IBM Certified Advanced System Administrator – Lotus Notes and Domino 6 - 7)
  - Mein Schwerpunkt liegt auf Anwendungen mit Schnittstellen zu anderen Systemen, z. B. RDBMS, SAP R/3, und interaktiven Web-Anwendungen

Organisatorisches (forts.)

- Zwischenfragen erwünscht!
- Handys bitte aus- oder stumm schalten
- Bitte am Ende die Bewertungsbögen ausfüllen
  - sie sind für mich **sehr wichtig**!

Wo sind wir jetzt?

Motivation

Grundlagen

Einige wichtige Entwurfsmuster im Detail

Weitere Entwurfsmuster in aller Kürze

OOP in LotusScript – Einschränkungen und Lösungen

Quellen

Was sind Entwurfsmuster (Design Patterns)?

- Objektorientierte Programmierung (OOP) **kann** …
  - die Wiederverwendbarkeit von Code steigern
  - seine Verständlichkeit und Wartbarkeit verbessern
  - Flexibilität und Erweiterbarkeit erhöhen
  - die Entwicklung beschleunigen und
  - die Lücke zwischen Modellierung (z. B. UML) und Entwicklung schließen
- aber auch bei OOP geht das nicht von ganz alleine.
  Du musst dich darum kümmern!

Was sind Entwurfsmuster? (forts.)

- Probleme wiederholen sich!
  - Wie oft hast du eine Schleife über alle Dokumente einer Ansicht programmiert?
- Du bist **nicht** der einzige Entwickler in der Welt
- Meistens hatte andere vor dir das gleiche Problem und habe eine gute Lösung dafür gefunden.
- Mit der Zeit hat sich diese Lösung zu einer „Best Practice“ weiterentwickelt und wurde katalogisiert
- Solche Lösungen heißen dann

  Entwurfsmuster

  (englisch: **Design Pattern**)

Was sind Entwurfsmuster? (forts.)

- Was sind Entwurfsmuster **nicht**:
  - Script-Bibliotheken
  - Sammlung von Klassen
  - „fertiger“ Code

Wir können Entwurfsmuster **dir** helfen?

- Die genaue Kenntnis und die geeignete Verwendung von Entwurfsmustern an den richtigen Stelen **wird** …
  - die Wiederverwendbarkeit von Code steigern
  - seine Verständlichkeit und Wartbarkeit verbessern
  - Flexibilität und Erweiterbarkeit erhöhen
  - die Entwicklung beschleunigen und

- Aber wie immer gilt:
  Man sollte dieses „Werkzeug“ nicht überbeanspruchen!

Wir können Entwurfsmuster **dir** helfen? (forts.)

- Wenn das ganze Team Entwurfsmuster kenn, kann das die Kommunikation vereinfachen: sie wird knapper und prägnanter
- Zum Beispiel:

  „Ich habe die Klasse als Singleton realisiert."

  gegenüber

  „Ich habe sichergestellt, dass es von dieser Klasse maximal eine Instanz gleichzeitig geben kann, und dass man auf dieses eine Objekt von über all aus leicht zugreifen kann. Wenn es bis dahin nicht existiert, wird es notfalls automatisch erzeugt."

- Und die Beschreibung des Singleton-Entwurfsmusters ist noch relativ kurz…

Wo sind wir jetzt?

Motivation

Grundlagen

Einige wichtige Entwurfsmuster im Detail

Weitere Entwurfsmuster in aller Kürze

OOP in LotusScript – Einschränkungen und Lösungen

Quellen

Objektorientierte Programmierung (OOP)

- Abstraktion
  - Abstraktion ist die Vereinfachung der komplexen Realität durch Modellbildung. In OOP bedeutet das: Modellierung durch für ein spezifisches Problem geeignete Klassen und passende Vererbungshierarchien.
- Klassen
  - Eine Klasse definiert die abstrakten Charakteristiken eines Objekts der realen Welt. Dazu gehören die Eigenschaften des Objekts (OOP: Attribute) und sein Verhalten (OOP: Methoden).
- Objekte
  - Ein Objekt einer Klasse (OOP: Objekt oder Instanz) repräsentiert dann ein reales Objekt mit seinen konkreten Eigenschaften.

Objektorientierte Programmierung (forts.)

- Kapselung
  - Eine Klasse verbirgt die Details ihrer internen Vorgänge.
  - Andere Klassen kennen nur öffentliche Attribute und Methoden.
  - Private Attribute und Methoden können sich problemlos ändern, solange sich die öffentliche Schnittstelle nicht ändert.

Objektorientierte Programmierung (forts.)

- Vererbung
  - Unterklassen erben Eigenschaften und Verhalten ihrer Oberklassen.
  - Oberklassen sind allgemeiner, Unterklassen spezifischer.
  - Beispiel: ein Lastwagen IST-EIN Kraftfahrzeug
- Komposition und Delegation
  - Eine Klasse benutzt eine andere Klasse, um bestimmte Funktionen bereitzustellen.
  - Sie hat dazu ein (meist privates) Attribut für ein Objekt der genutzten Klasse.
  - Beispiel: ein Kraftfahrzeug HAT-EIN Motor

Objektorientierte Programmierung (forts.)

- Polymorphismus
  - Es kann mehrere gleichnamige Methoden mit unterschiedlichen Parametersignaturen geben (nicht in LotusScript)
  - Methoden einer Unterklasse können gleichnamige Pendants der Oberklasse überschreiben. Nur sie können die überschriebene Methode der Oberklasse aufrufen.
  - Methoden einer Oberklasse können „abstrakt“ sein. Sie haben dann keinen Methodenrumpf, sondern müssen in Unterklassen implementiert werden (nicht in LotusScript). Eine Klasse mit mindestens einer abstrakten Methode ist selbst auch „abstrakt“.

OO-Entwurfsprinzipien

- KISS – Keep It Simple, Stupid
  - Vermeide unnötige Komplexität, bevorzuge einfache Lösungen
- Prinzip der minimalen Verwunderung (Principle of Least Astonishment)
  - Erstaunliche Lösungen sind häufig schwerer zu verstehen.
- Vermeide Wiederholungen (DRY – Don't Repeat Yourself)
  - Vermeide Code-Duplikation.
- Prinzip der einzelnen Verantwortlichkeit (Separation of Concerns)
  - Jede Klasse sollte nur genau eine Verantwortlichkeit haben.
- Offen-Geschlossen-Prinzip (Open-Closed Principle)
  - Klassen sollte offen sein für Erweiterungen, aber geschlossen gegenüber Veränderungen.

OO-Entwurfsprinzipien (forts.)

- Kapsele, was sich ändert
  - So beeinflussen Änderungen nicht den stabilen Teil deines Codes.
- Bevorzuge Komposition gegenüber Vererbung
  - So kann sich das Verhalten zur Laufzeit ändern, nicht nur beim Kompilieren
- Strebe nach lose-gekoppelten Entwürfen
  - Veränderungen wirken dann nur lokal.
- Prinzip der Umkehrung von Abhängigkeiten (Dependency Inversion Principle)
  - Klassen sollten nur von Abstraktionen abhängen, nicht von konkreten Implementierungen.

Entwurfsmuster – eine Definition

- [HFDP]: “A Design Pattern is a solution to a problem in a context.”
- Ein Entwurfsmuster ist eine Lösung zu einem Problem in einem Kontext.
  - Kontext – die Situation, in der das Muster angewandt werden kann
  - Problem – das zu erreichende Ziel und die einzuhaltenden Bedingungen; Ziel und Bedingungen werden zusammen manchmal Kräfte (forces) genannt
  - Lösung – ein allgemeines Design, das angewandt werden kann, um das Problem zu lösen

Entwurfsmuster – noch eine Definition

- „Jedes Muster beschreibt ein in unserer Umwelt beständig wiederkehrendes Problem und erläutert den Kern der Lösung für dieses Problem, so daß Sie diese Lösung beliebig oft anwenden können, ohne sie jemals ein zweites Mal gleich auszuführen“

  Christopher Alexander in
  A Pattern Language, Oxford University Press, New York, 1977

- Oder ganz kurz:
  Entwurfsmuster sind Best-Practice-Lösungen von häufig auftretenden Problemen (im Bereich Software-Entwurf)

Beschreibungen von Entwurfsmustern

- Übliche Teile der Beschreibung eines Entwurfsmusters sind
  - Name
  - Klassifizierung
  - Ziel oder Absicht (Problem)
  - Motivation/Szenario (Kontext)
  - Anwendbarkeit
  - Lösung
    - Struktur (meist in Form von UML-Klassendiagrammen)
    - Teilnehmer und ihr Zusammenwirken
  - Einfache Implementierung
  - Konsequenzen, Vor- und Nachteile
  - Bekannte Verwendungen, Varianten und verwandte Muster

Klassifizierung von Entwurfsmustern

- Entwurfsmuster werden häufig folgenden Klassen zugeordnet:
  - Erzeugungsmuster
    - Erzeugungsmuster beschäftigen sich mit der Erzeugung von Objekten und entkoppeln den Benutzer eines Objekts von dessen Erzeugung.
  - Verhaltensmuster
    - Verhaltensmuster beschäftigen sich mit der Interaktion zwischen Klassen bzw. Objekten
  - Strukturierungsmuster
    - Bei Strukturierungsmustern geht um Strukturen von mehreren Klassen bzw. Objekten

Wo sind wir jetzt?

Motivation

Grundlagen

Einige wichtige Entwurfsmuster im Detail

Weitere Entwurfsmuster in aller Kürze

OOP in LotusScript – Einschränkungen und Lösungen

Quellen

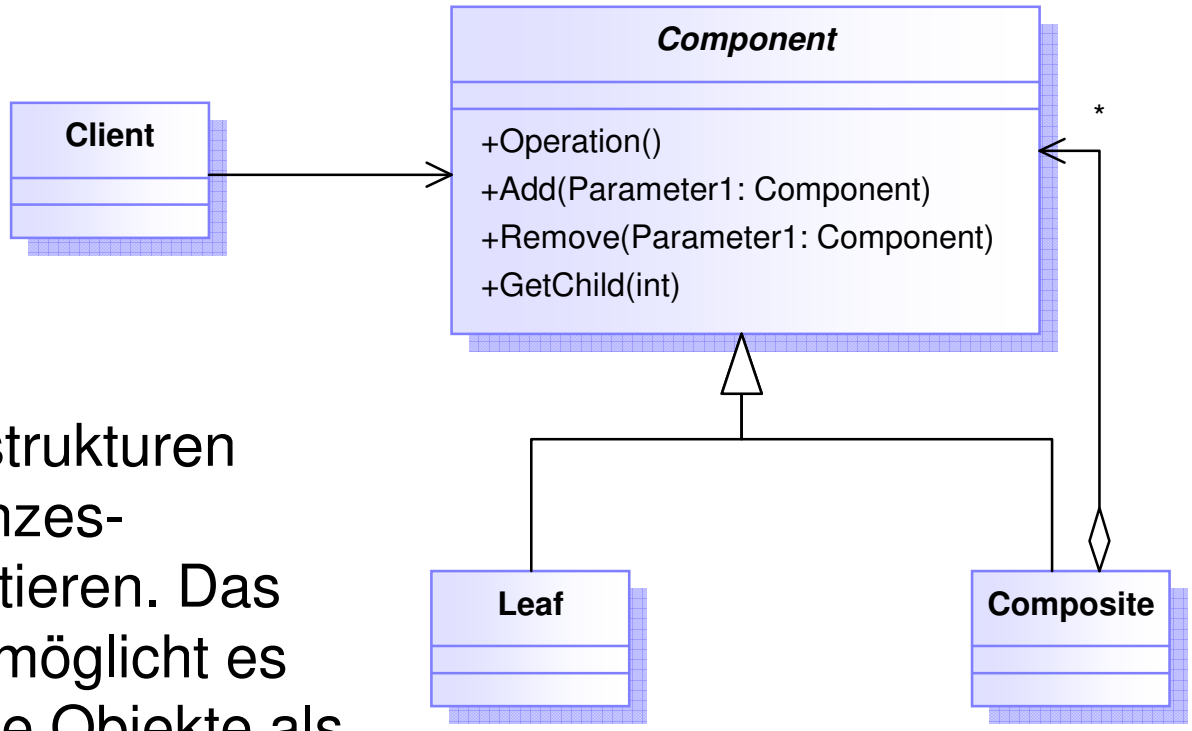
Kompositum – Composite

- Ich will Verzeichnisse (directories) mit Ordnern (folders) und Dateien (files) modellieren
- Es sollen sowohl lokale Dateisysteme, als auch entfernte Verzeichnisse, z. B. FTP-Verzeichnisse, modelliert werden.
- gemeinsame Attribute und Methoden von Ordnern und Dateien:
  - Beide haben einen Namen, ein Zeitpunkt der letzten Änderung und können versteckt (hidden) sein. Beide sollen eine Methode haben, um ihren aktuellen Zustand zurückzugeben (ToString)
  - Dateien haben eine Größe (in Bytes), Ordner nicht
  - Ordner repräsentieren Unterverzeichnisse mit ggf. weiteren Ordnern und Dateien als Einträgen
- Verzeichnisse haben einen „Ort“ (location) – im lokalen Dateisystem = Pfad

Kompositum – Composite (forts.)

- Also möchte ich eine Oberklasse für Ordner und Dateien erstellen: DirectoryEntry (Verzeichniseintrag)
- Verzeichnisse sind baumartige Strukturen.
- Sehen wir uns dazu mal das Kompositum (Composite) näher an:

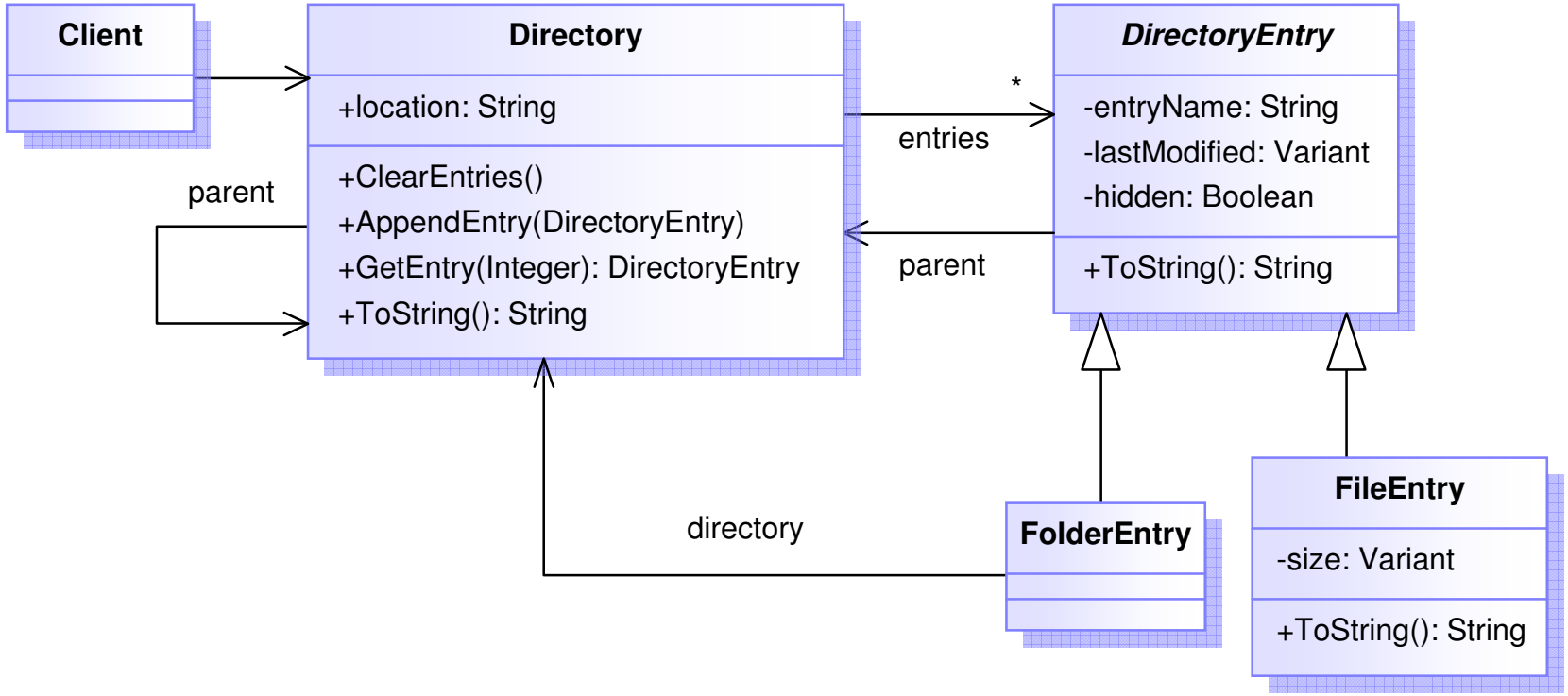
Kompositum – Composite (forts.)



- Definition [GoF]:

  „Füge Objekte zu Baumstrukturen zusammen, um Teil-Ganzes-Hierarchien zu repräsentieren. Das Kompositions-muster ermöglicht es Klienten, sowohl einzelne Objekte als auch Kompositionen von Objekten einheitlich zu behandeln.“

Kompositum – Composite (forts.)



- Achtung: Das ist kein „richtiges“ Kompositum!
  - Siehst du, warum nicht?

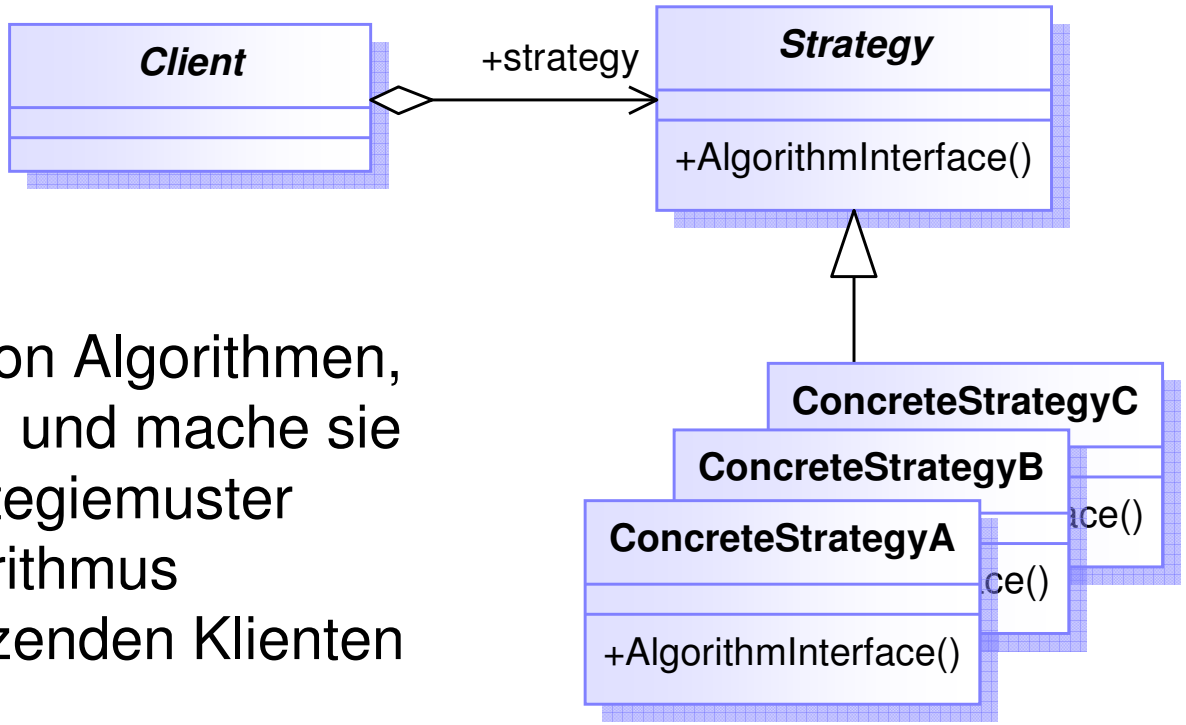
Kompositum – Composite (forts.)

- Beim Kompositum haben auch die Blatt-Unterklassen (Leaf) die Methoden zum Setzen und Entfernen von Kind-Objekten.
- Ich mag das nicht, also habe ich die Component-Klasse zweigeteilt: Directory kümmert sich um die Einträge, DirectoryEntry um die gemeinsamen Attribute und Methoden.
- In einem echten Kompositum können die Client-Klassen die Composite- und Leaf-Objekte absolut gleich behandeln, hier nicht.
- Aber da es für jeden FolderEntry ein Directory gibt, reicht diese Lösung für meine Zwecke.
- [HFDP]: “Patterns are tools not rules – they need to be tweaked and adapted to your problem.”, Erich Gamma

Strategie – Strategy

- Unabhängig davon, um welche Art von Verzeichnis es sich handelt (lokal oder entfernt), soll es eine Methode zum Auslesen geben.
- Es gibt noch keine Entscheidung, welche Arten von entfernten Verzeichnissen (FTP, Web Services usw.) unterstützt werden sollen. Es ist nur klar, dass später wahrscheinlich weitere dazu kommen.
- Machen wir aus der entsprechenden Methode ReadDirectory() eine abstrakte Methode, die dann von Unterklassen erst implementiert wird.
- Zusätzlich lagern wir die Verantwortlichkeit zum Auslesen von Verzeichnissen in eine weitere Klasse DirectoryReader aus.

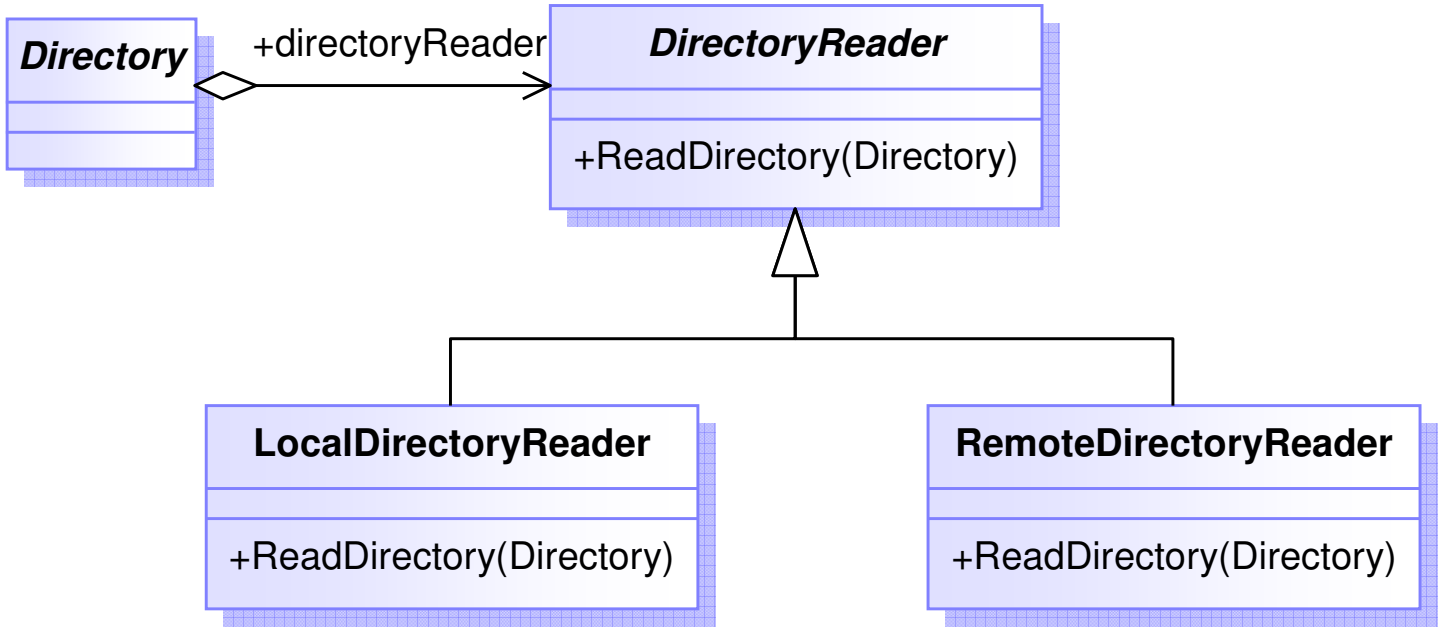
Strategie – Strategy (forts.)



- Definition [GoF]:

  „Definiere eine Familie von Algorithmen, kapsele jeden einzelnen und mache sie austauschbar. Das Strategiemuster ermöglicht es, den Algorithmus unabhängig von ihn nutzenden Klienten zu variieren.“

Strategie – Strategy (forts.)



Strategie – Strategy (forts.)

- Der Client, in diesem Fall also Directory, hängt nur von der abstrakten Oberklasse DirectoryReader ab.
- Aber er muss auch die (mindestens eine) konkrete Unterklasse kennen, damit er von ihr Objekte erstellen kann.
- Weitere typische Anwendungen des Strategie-Musters:
  - Sortierung von Mengen von Objekten mit unterschiedlichen Algorithmen
  - Internationalisierung, z. B. unterschiedliche Kalender

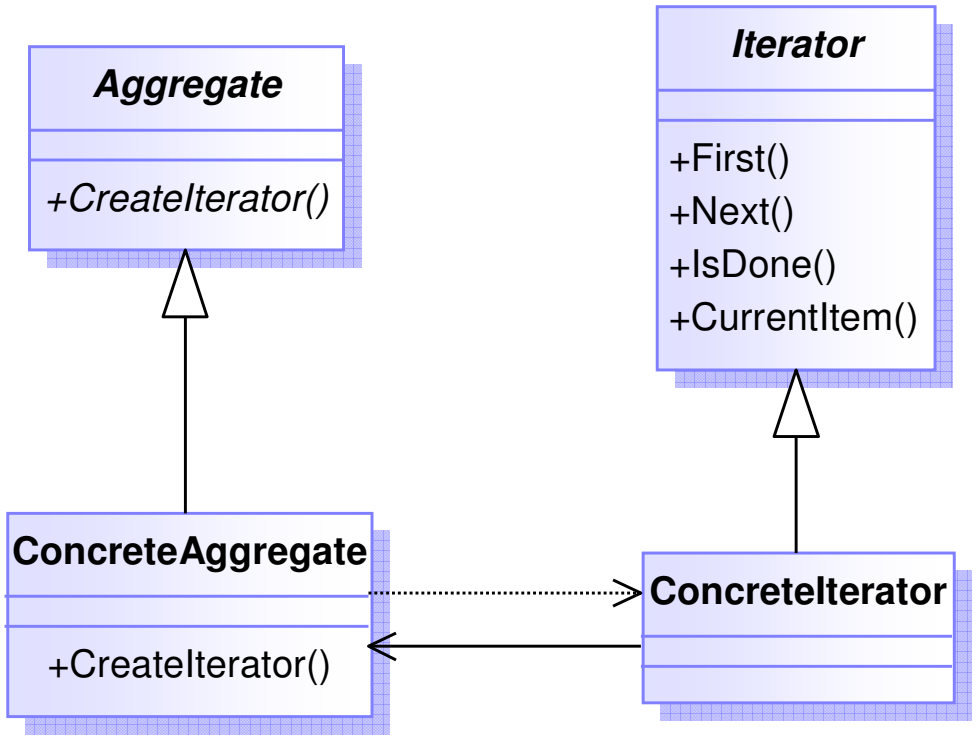
Iterator

- Ich möchte alle Einträge eines Verzeichnisses nacheinander durchgehen (und etwas damit tun).
- Das könnte zum Beispiel dazu genutzt werden, um ein Verzeichnis auszugeben oder eine Dateioperation auf allen seinen Einträgen auszuführen.
- Der Client (z. B. ein DirectoryPrinter) sollte die interne Struktur von Directory möglichst nicht kennen (Entkoppelung), so dass diese geändert werden kann.
- Alle Clients sollten unabhängig voneinander (und gleichzeitig) ein Verzeichnis durchlaufen können.

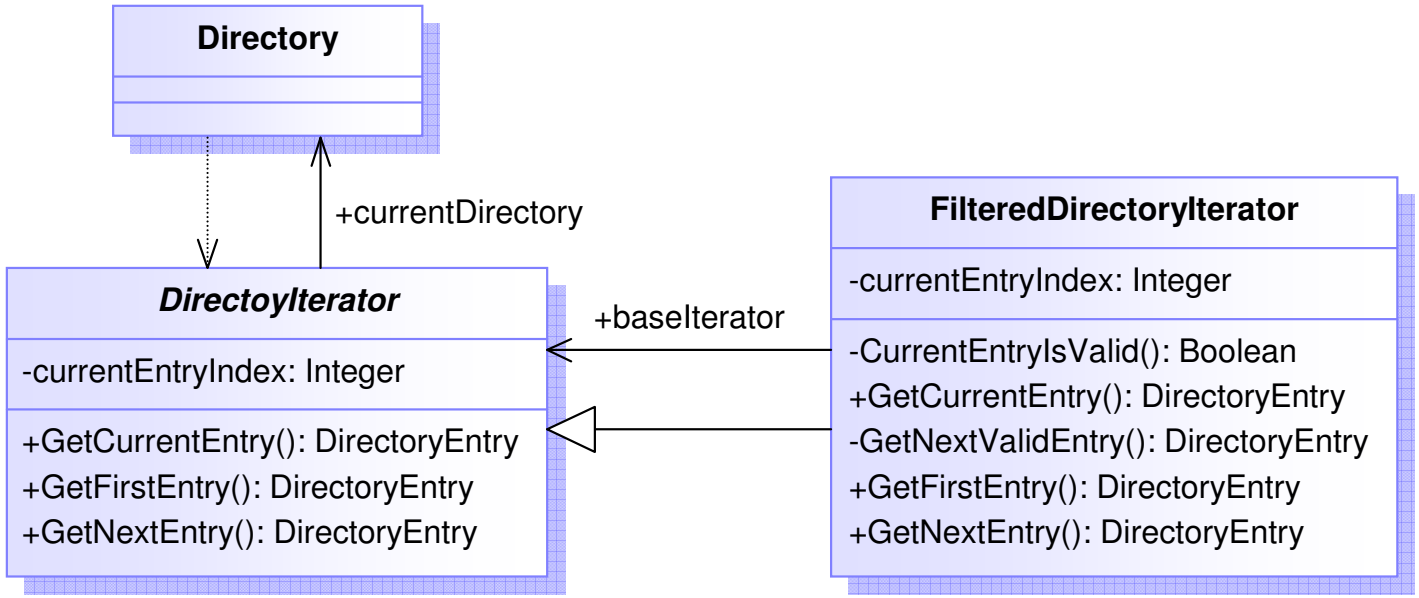
Iterator (forts.)

- Definition [GoF]:

  „Ermögliche den sequentiellen Zugriff auf die Elemente eines zusammengesetzten Objekts, ohne seine zugrundeliegende Repräsentation offenzulegen.“



Iterator (forts.)



Iterator (forts.)

- Wieder habe ich das Muster für meine Zwecke angepasst:
  - Ich habe die abstrakten Klassen Aggregate und Iterator entfernt.
  - DirectoryIterator ist eine konkrete Oberklasse, kann also auch selbst schon verwendet werden.
- Der Client von DirectoryIterator kennt nicht die Interna von Directory.
- Da jede DirectoryIterator-Instanz einen eigenen „Cursor“ hat, sind die Objekte voneinander unabhängig.
- Wenn man Iterator benutzt, muss man sich darum kümmern, was passiert, wenn sich die unterliegende Struktur, also das Aggregate, sich verändert, also z. B. DirectoryEntry-Objekte hinzugefügt oder entfernt werden.

Iterator (forts.)

- Warum soll man überhaupt die Aufgabe „Durchlaufen der enthaltenen Objekte“ vom Aggregate trennen?
- Zum Beispiel gibt es in NotesView und NotesCollection gleich selbst die entsprechenden Methoden:
  - GetFirstDocument, GetNextDocument…
  - GetNthDocument
- Es kommt auf die persönlichen Vorlieben und die Flexibilität der Lösung an…

Iterator (forts.)

- Vorteile der eingebauten „Durchlauf“-Funktionalität
  - weniger Klassen
  - keine Notwendigkeit die Änderungen am Aggregate an den Iterator zu melden, und es ist einfacher, auf solche Änderungen zu reagieren (mehr „internes“ Know-how)
- Vorteile externer Iterator-Klassen
  - Erinnere dich an die Entwurfsprinzipien: Trenne Verantwortlichkeiten
  - mehr Flexibilität: Der Iterator kann durch Unterklassen leichter erweitert werden; getrennte Vererbungshierarchien für Aggregate und Iterator sind möglich.
  - ein Iterator kann ggf. für verschiedene Aggregates genutzt werden

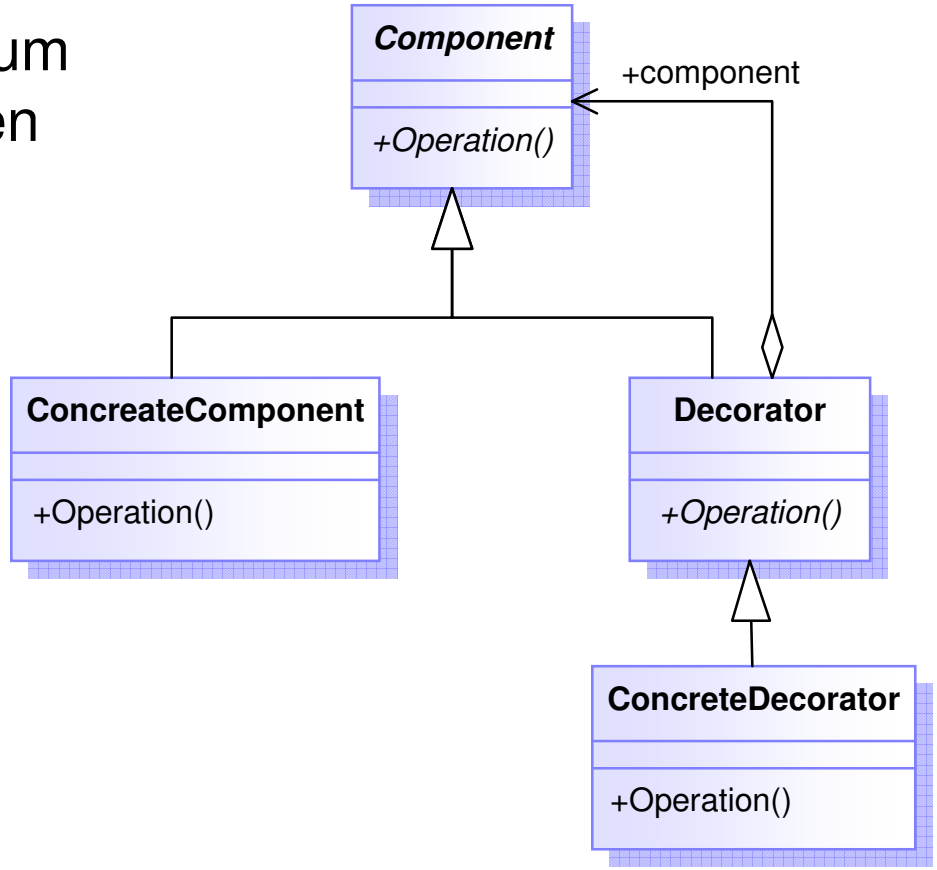
Dekorierer – Decorator

- Ich möchte nun ein Verzeichnis durchlaufen – aber nur einige bestimmte Einträge verarbeiten, z. B.
  - nur sichtbare Einträge
  - nur Datei-Einträge
  - nur Ordner-Einträge
- Später sollen noch andere Erweiterungen möglich sein, z. B. Sortierungen nach unterschiedlichen Kriterien
- Wie kann man den Iterator einfach und flexibel erweitern? Vererbung!
- Wie kann man es machen, dass man **zur Laufzeit** erst entscheidet, welche Einträge in welcher Reihenfolge durchlaufen werden sollen? Delegation!
- Beides zusammen ergibt den Dekorierer:

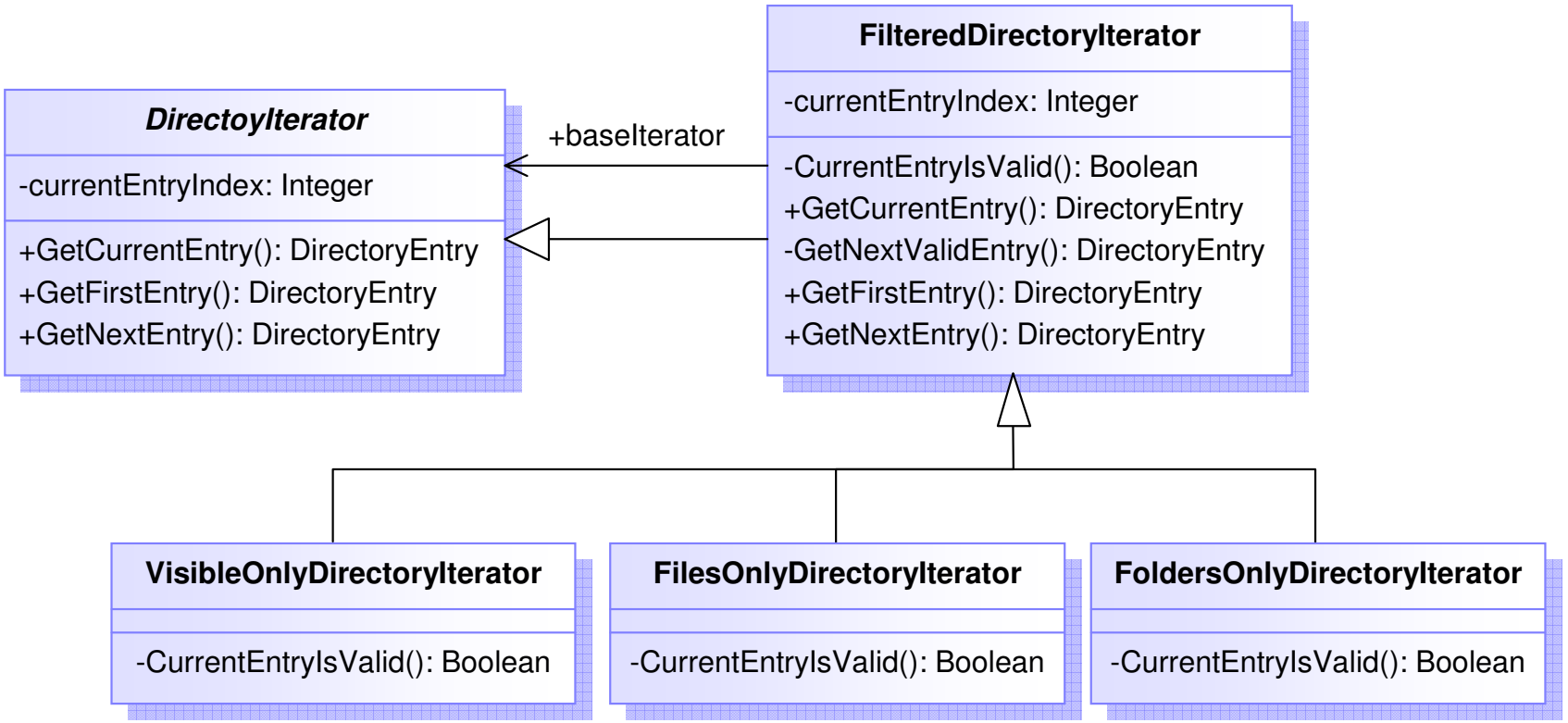
Dekorierer – Decorator (forts.)

- Definition [GoF]:

  „Erweitere ein Objekt dynamisch um Zuständigkeiten. Dekorierer bieten ein flexible Alternative zur Unterklassenbildung, um die Funktionalität einer Klasse zu erweitern.“



Dekorierer – Decorator (forts.)



Dekorierer – Decorator (forts.)

- Durch Vererbung kann man Klassen erweitern – Änderungen sind aber nur beim Entwickeln möglich (compile-time)
- Mit Delegation kann man das Verhalten zur Laufzeit ändern
- Ein Dekorierer „umhüllt“ die Component und kann neues Verhalten vor oder nach dem Aufruf der Operation-Methode hinzufügen oder es sogar komplett ersetzen.
- Dekorierer-Instanzen können überall an Stelle von Component-Objekten verwendet werden.
- Man kann die Dekorierer sogar schachteln:
  - new FilesOnlyDirectoryIterator(
    new VisibleOnlyDirectoryIterator(
    new DirectoryIterator(someDirectory)))
- Siehst du die Verwendung von Strategy beim Dekorierer?

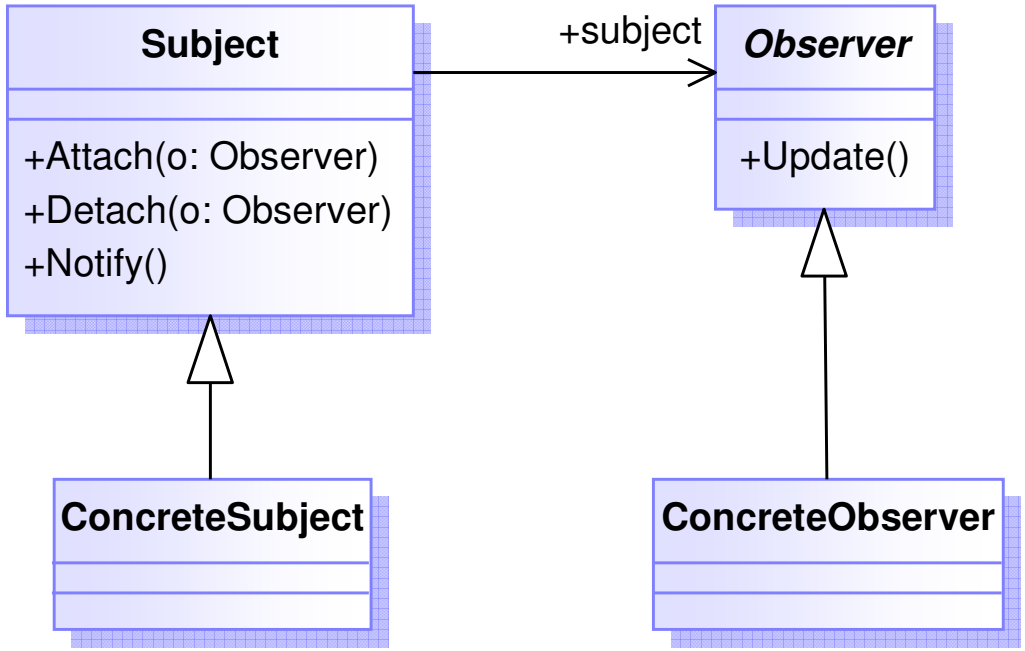
Beobachter – Observer

- Erinnerst du dich an das Problem beim Iterator:
  - Wie soll man auf Änderungen des Aggregate reagieren?
  - Und wie erfährt der Iterator überhaupt davon?
- Eine einfache Lösung für die erste Frage:
  - Markiere den Iterator als ungültig (invalid).
  - Bei verschachtelten Iteratoren muss diese Markierung weitergereicht werden.
- Lösung:
  - Jeder DirectoryIterator soll sich bei seiner Datenquelle registrieren.
  - Die Quelle sendet bei Veränderungen ein „Signal“.
  - Der DirectoryIterator markiert sich als ungültig und sendet nun seinerseits das Signal an die bei ihm registrierten Objekte.

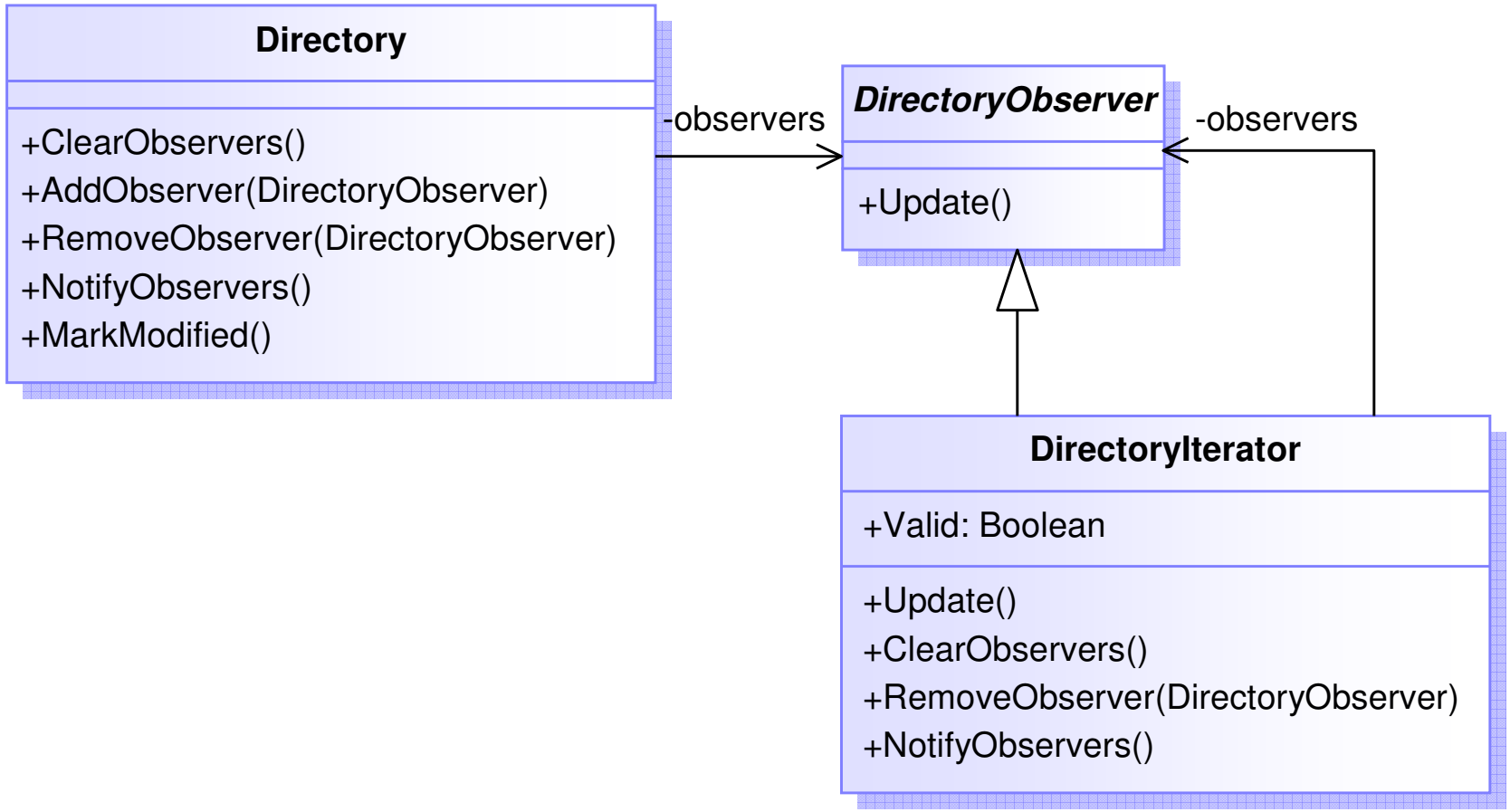
Beobachter – Observer (forts.)

- Definition [GoF]:

  „Definiere eine 1-zu-n-Abhängigkeit zwischen Objekten, so daß die Änderung des Zustands eines Objekts dazu führt, daß alle abhängigen Objekte benachrichtigt und automatisch aktualisiert werden.“



Beobachter – Observer (forts.)



Beobachter – Observer (forts.)

- Wenn ein DirectoryIterator erstellt wird, dann wird er automatisch bei seiner Datenquelle als Observer registriert.
- Wenn das Directory geändert wird (im Beispiel: MakeNotification() wird aufgerufen), ruft es NotifyObservers() auf. Diese Methode ruft wiederum die Update()-Methode aller Observer in der Liste auf.
- DirectoryIterator ist auch ein Subject!
- In seiner Update()-Methode markiert es sich selbst als ungültig und ruft dann NotifyObservers() auf.
- Subject weiß nichts über die Observer, außer dass sie Unterklassen von DirectoryObserver sind: Die Klassen sind nur lose gekuppelt!
- Das Beobachter-Muster ist eines der am häufigsten genutzten Entwurfsmuster.

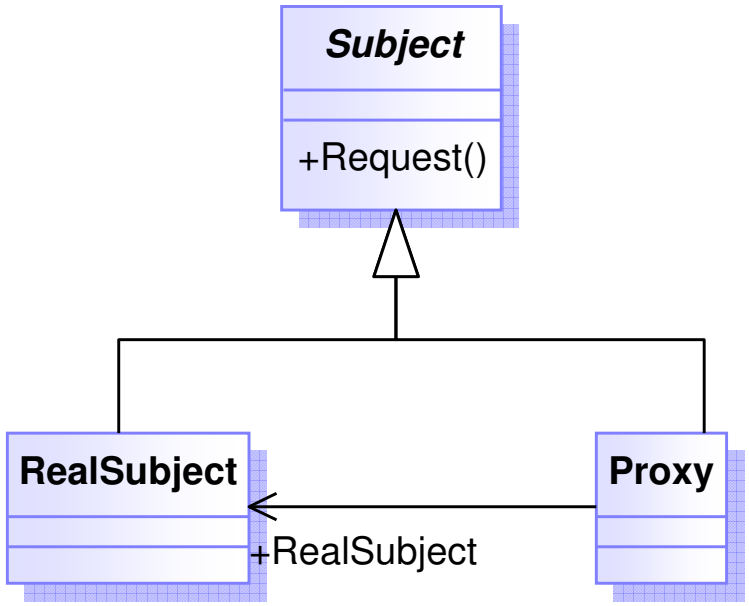
Proxy

- FolderEntry hat eine Referenz auf sein Directory.
- Wann soll dieses eingelesen und damit das Objekt initialisiert werden?
- Das Einlesen könnte „teuer“ sein (im Allgemeinen bei entfernten Verzeichnissen, z. B. per Web Service).
- Und das Unterverzeichnis könnte weitere Unterverzeichnisse haben.
- Wann sollen diese eingelesen werden?
- Lösung:
  - Directories sollen spätest möglich gelesen werden: Lazy loading
  - Idealerweise sollte sich FolderEntry gar nicht darum kümmern müssen, wann sein Directory gelesen wird. Es muss nur sichergestellt sein, dass es „da ist“, wenn es gebraucht wird.

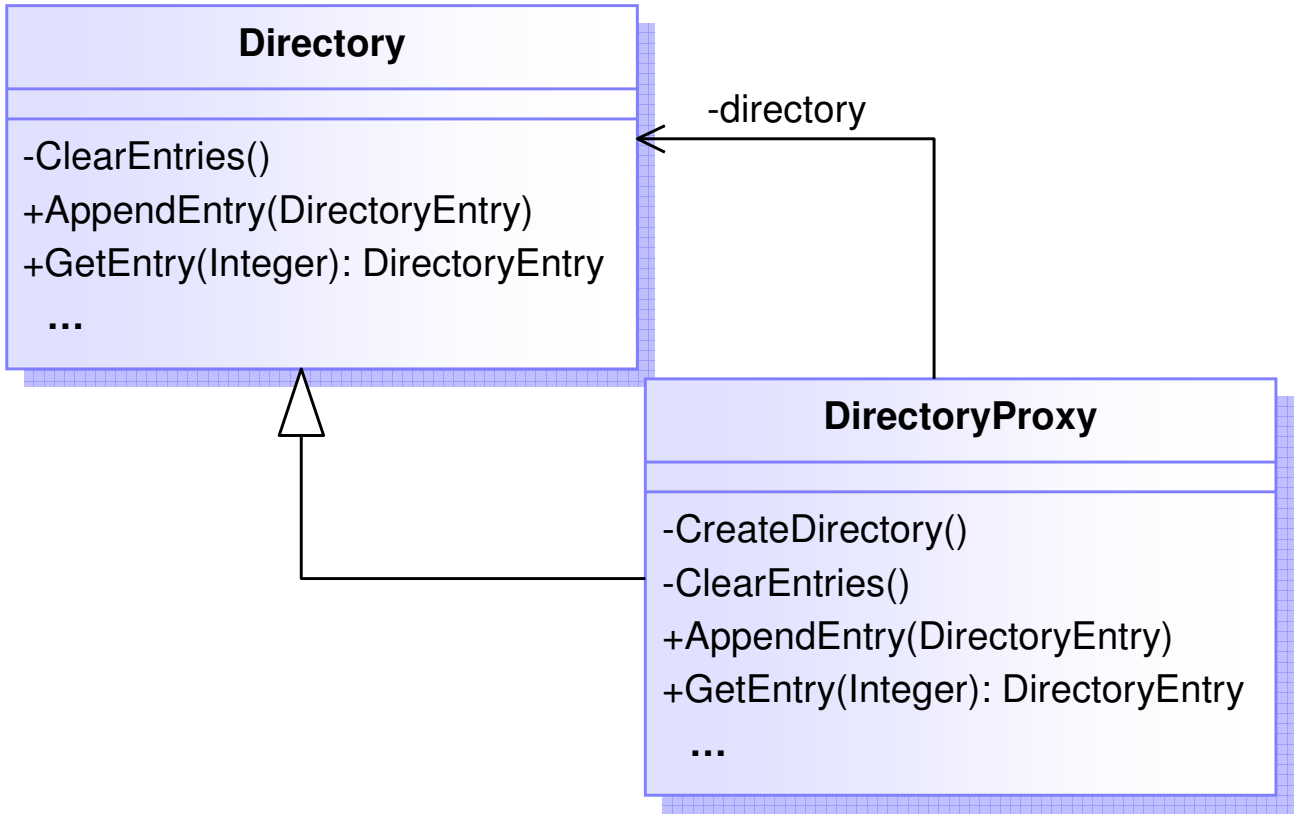
Proxy (forts.)

- Definition [GoF]:

  „Kontrolliere den Zugriff auf ein Objekt mit Hilfe eines vorgelagerten Stellvertreterobjekts.“



Proxy (forts.)


Directory
-ClearEntries()
+AppendEntry(DirectoryEntry)
+GetEntry(Integer): DirectoryEntry
...
-directory
DirectoryProxy
-CreateDirectory()
-ClearEntries()
+AppendEntry(DirectoryEntry)
+GetEntry(Integer): DirectoryEntry
...

Proxy (forts.)

- Im Konstruktor von DirectoryProxy, wird das Attribut directory auf Nothing gesetzt.
- Eine typische öffentliche Methode des Proxy kann dann so aussehen:

```
Public Function GetEntry(index As Integer) As DirectoryEntry
        Call CreateDirectory
        Set GetEntry = directory.GetEntry(index)
End Function

Private Sub CreateDirectory()
        If Me.directory Is Nothing Then
                Set directory = New Directory(Me.location, True)
                Call directory.SetParent(parent)
        End If
End Sub
```

Proxy (forts.)

- Jeder FolderEntry hat ein DirectoryProxy im directory-Attribut.
- Aber der FolderEntry „weiß“ nichts über den Proxy, sondern „glaubt“, es handle sich um ein echtes Directory-Objekt.
- Das erste Mal, wenn irgendeine (öffentliche) Methode des Proxies aufgerufen wird, die der Proxy nicht selbst beanworten kann, lädt der Proxy im Hintergrund das Verzeichnis und benutzt es als Delegate!
- Ab jetzt delegiert es alle Aufrufe an sein Directory-Objekt.

Proxy (forts.)

- Proxies werden genutzt, wenn die Generierung der realen Objekte „teuer“ ist (über das Netzwerk, ressourcen-intensiv), oder um den Zugriff zu regeln oder zu protokollieren.
- Web Services sind ein gutes Beispiel für das Proxy-Muster.
- Sowohl Web Service-Anbieter, als auch –Konsumenten werden häufig mit entsprechenden Werkzeugen aus einer WSDL-Datei generiert, z. B.
  - Apache Axis(2) WSDL2Java,
  - Stubby (OpenNTF.org),
  - Lotus Designer 7 (nur Anbieter) / 8 (Anbieter & Konsumenten)
- Diese Werkzeuge generieren Rumpf-Klassen. Die Konsumenten-Rümpfe sind Proxies, die vom lokalen Programm aufgerufen werden und ihrerseits den entfernten Web Services nutzen. Dieser Vorgang ist aber für das lokale Programm unsichtbar.

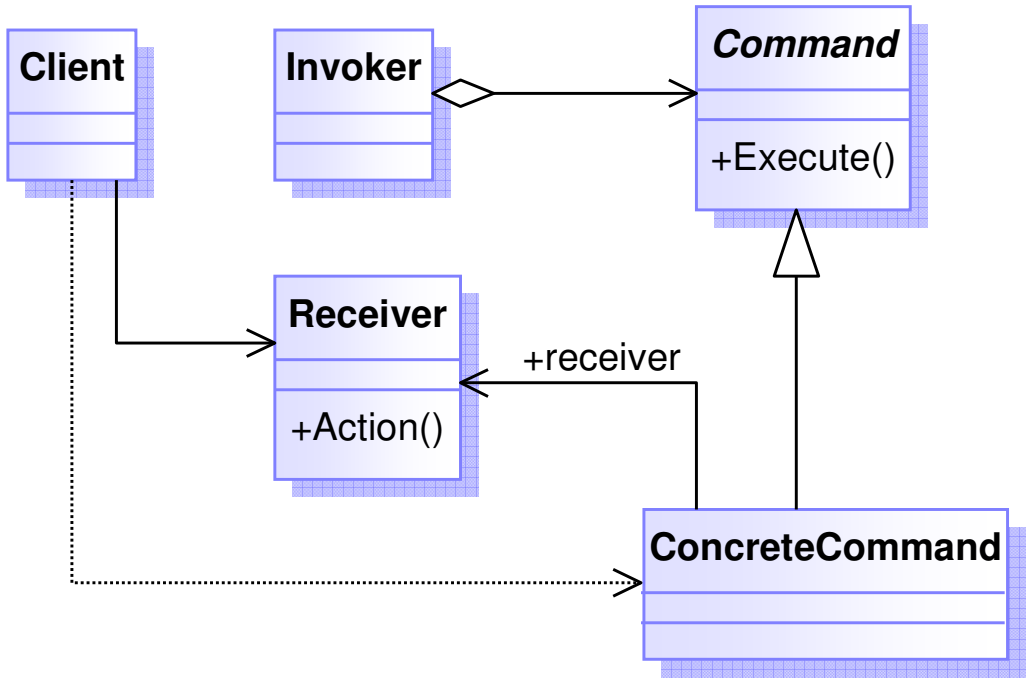
Befehl – Command

- Jetzt will ich einige Operationen auf DirectoryEntries ausführen, z. B.:
  - Eintrag löschen,
  - Eintrag umbenennen oder
  - Eintrag in anderen Ordner verschieben.
- Soweit möglich sollen die Operationen durch Aufruf einer Undo()-Methode rückgängig gemacht werden können.
- Die Operationen sollen zusammengefasst in Batches laufen können.
- Lösung:
  - Wir brauchen eine generische Klasse für diese Operationen und Unterklassen für die einzelnen Operationen.
  - Für reversible Operationen soll es eine Klasse mit einer abstrakten Undo()-Methode geben. Die konkreten Unterklassen müssen alle Informationen speichern, die sie für das Undo() brauchen.

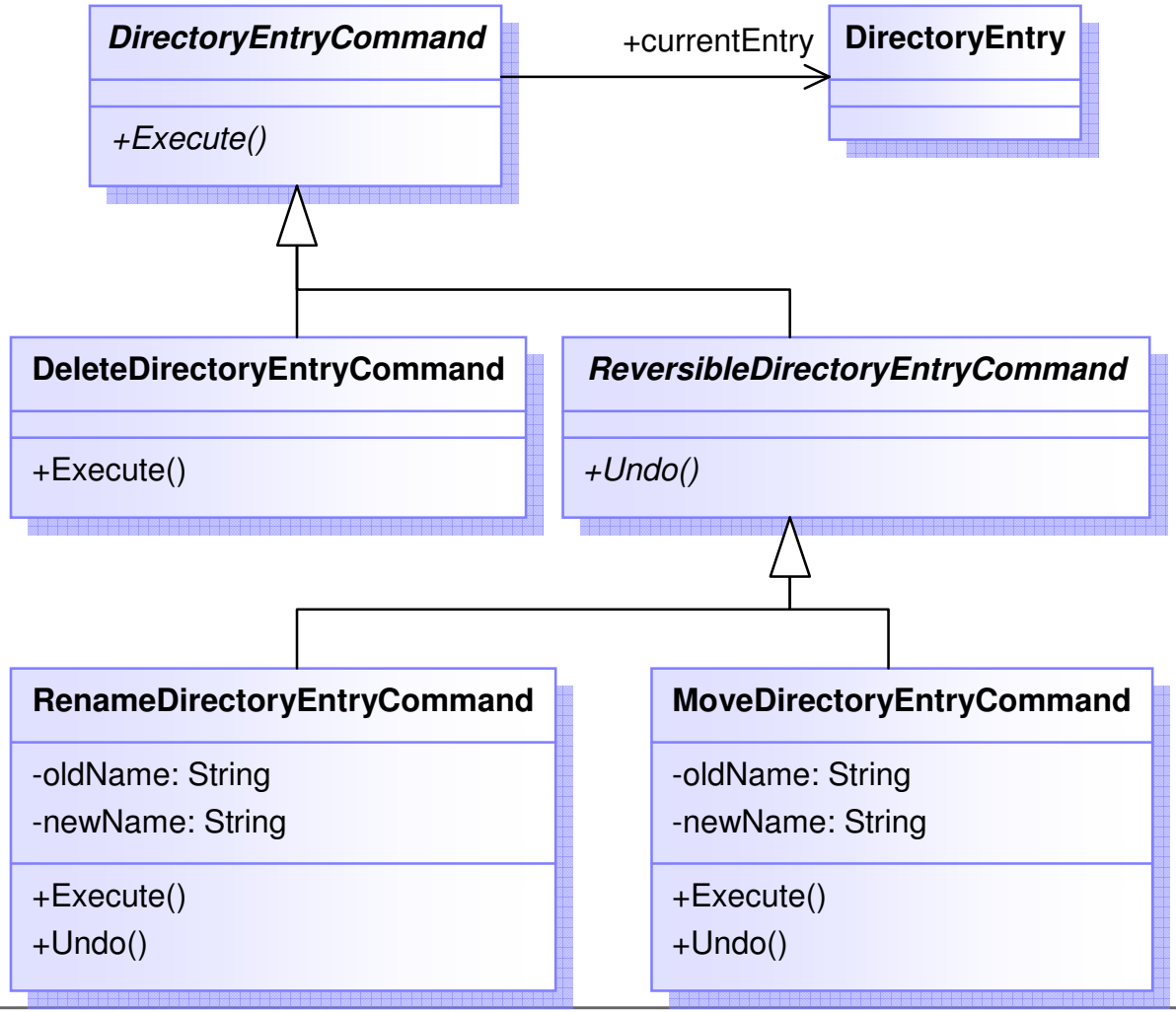
Befehl – Command (forts.)

- Definition [GoF]:

  „Kapsle einen Befehl als ein Objekt. Dies ermöglicht es, Klienten mit verschiedenen Anfragen zu parametrisieren, Operationen in eine Schlange zu stellen, ein Logbuch zu führen und Operationen rückgängig zu machen.“



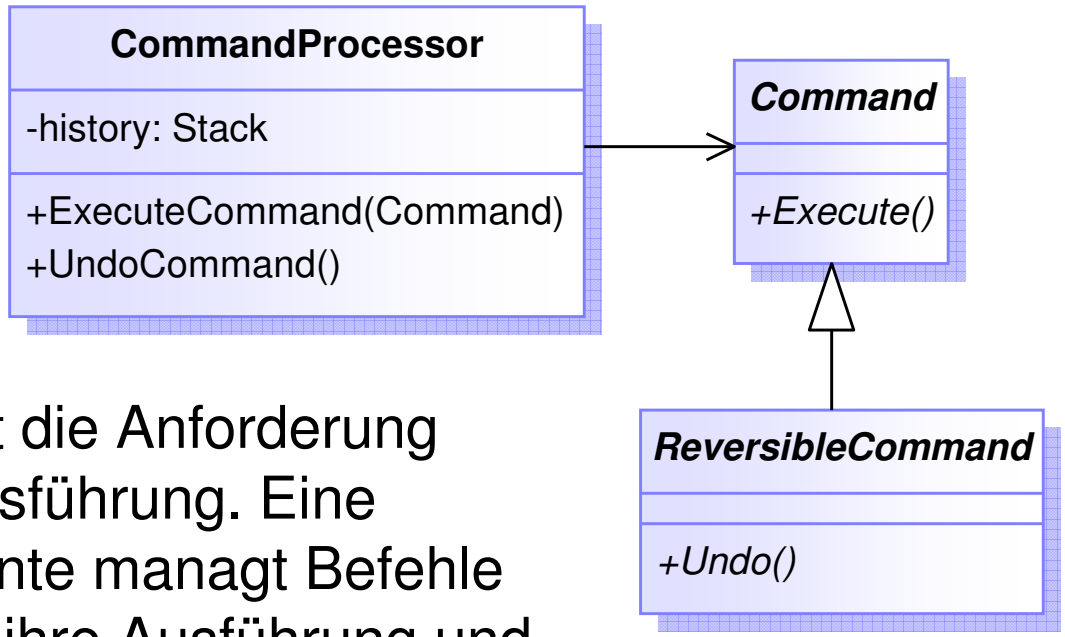
Befehl – Command (forts.)



Befehl – Command (forts.)

- Durch die Kapselung der Operation in Objekte können wir
  - den Aufruf (Erstellung des Befehlsobjekts) von seiner Ausführung trennen und so eine verzögerte und Batch-Ausführung ermöglichen
  - diese Objekte speichern, z. B. in Stacks oder Queues
  - die gespeicherten Objekte für das Rückgängigmachen nutzen
  - die Ausführung der Befehle protokollieren
- Das führt uns direkt zum nächsten Entwurfsmuster, das uns das Management von Befehlen ermöglicht:

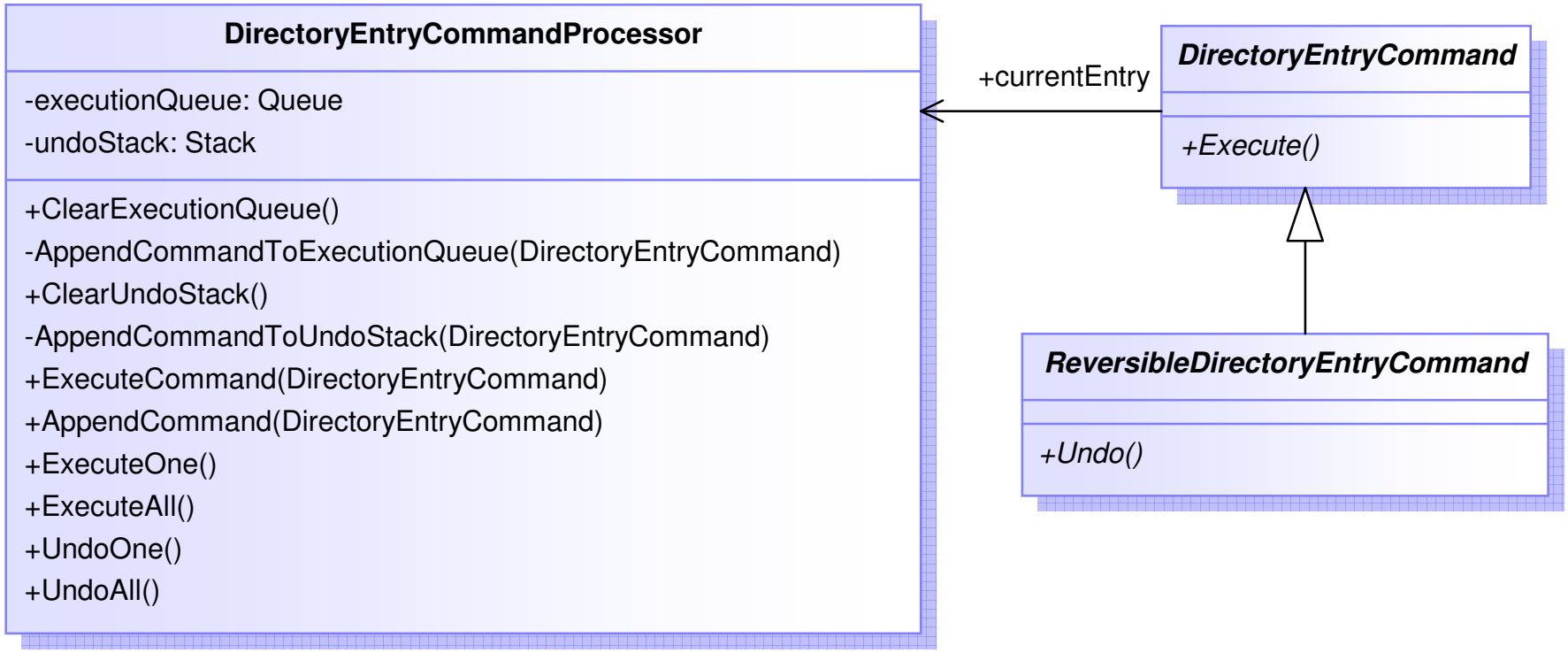
Befehlsprozessor – Command Processor



- Definition [POSA]:

  „Der Befehlsprozessor trennt die Anforderung eines Befehls von seiner Ausführung. Eine Befehlsprozessor-Komponente managt Befehle als getrennte Objekte, plant ihre Ausführung und bietet zusätzliche Dienste wie das Speichern von Befehlsobjekten für späteres Rückgängig-machen.“

Befehlsprozessor – Command Processor (forts.)



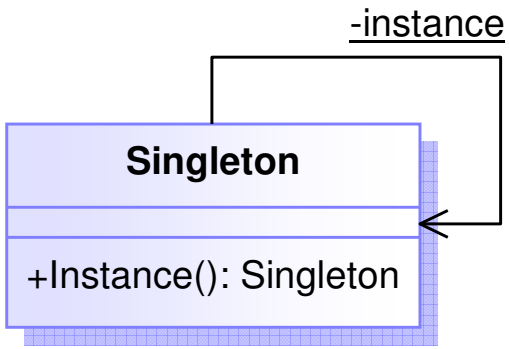
Befehlsprozessor – Command Processor (forts.)

- Der DirectoryEntryCommandProcessor nutzt eine Queue für die verzögerte Ausführung von DirectoryEntryCommands und einen Stack für ReversibleDirectoryEntryCommands.
- Der undoStack wird geleert, sobald ein nicht-reversibler Befehl ausgeführt wird.
- Damit der DirectoryEntryCommandProcessor korrekt funktioniert, muss sichergestellt sein, dass es nur einen davon gibt!
- Wie können wir das erreichen? Nächstes Entwurfsmuster:

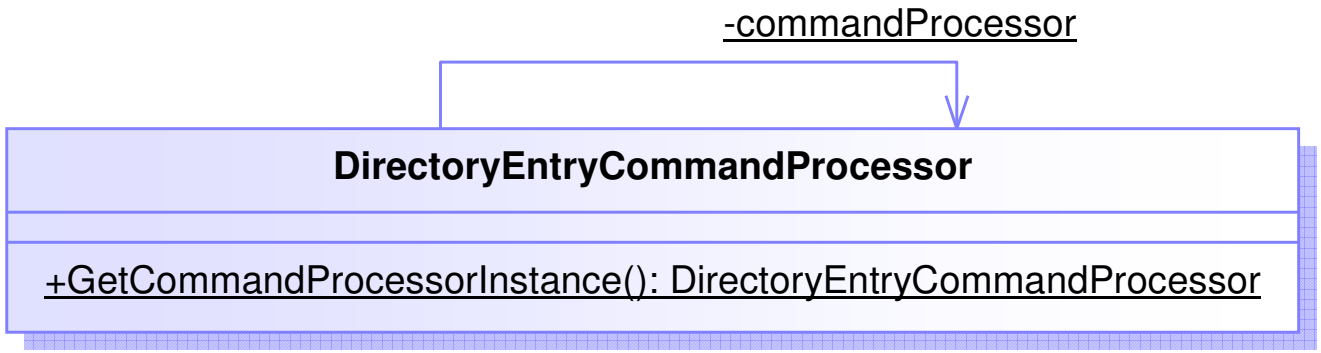
Singleton

- Definition [GoF]:

  „Sichere ab, daß eine Klasse genau ein Exemplar besitzt, und stelle einen globalen Zugriffspunkt darauf bereit.“



Singleton (forts.)



- Hast du gesehen, was bei diesem Diagramm anders ist als vorher?
  - commandProcessor und GetCommandProcessorInstance() gehören zur Klasse, nicht zum Objekt!

Singleton (forts.)

- Aber wie kann man Klassenattribute und –Methoden in LotusScript implementieren? Gar nicht!
- Stattdessen verwende ich eine globale Variable und Funktion:

```
Private commandProcessor As DirectoryEntryCommandProcessor
Public Function GetCommandProcessorInstance As _
                                        DirectoryEntryCommandProcessor
        If commandProcessor Is Nothing Then
                Set commandProcessor = _
                        New DirectoryEntryCommandProcessor()
        End If
        Set GetCommandProcessorInstance = commandProcessor
End Function
```

Wo sind wir jetzt?

Motivation

Grundlagen

Einige wichtige Entwurfsmuster im Detail

Weitere Entwurfsmuster in aller Kürze

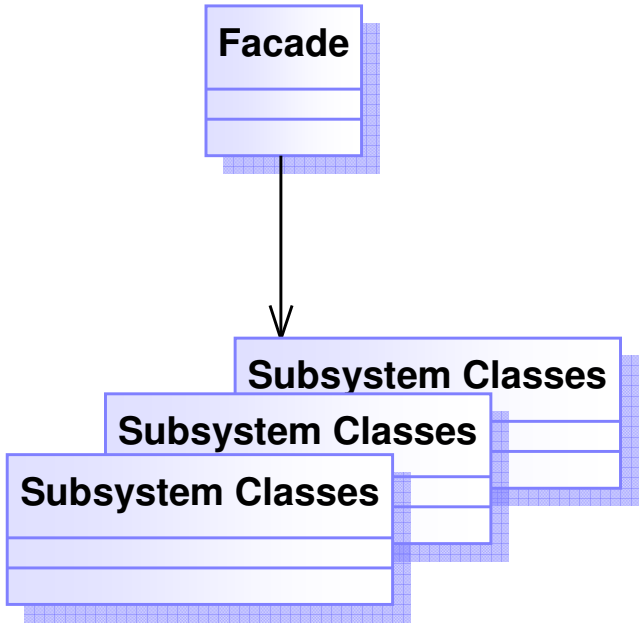
OOP in LotusScript – Einschränkungen und Lösungen

Quellen

Fassade – Façade

- Definition [GoF]:

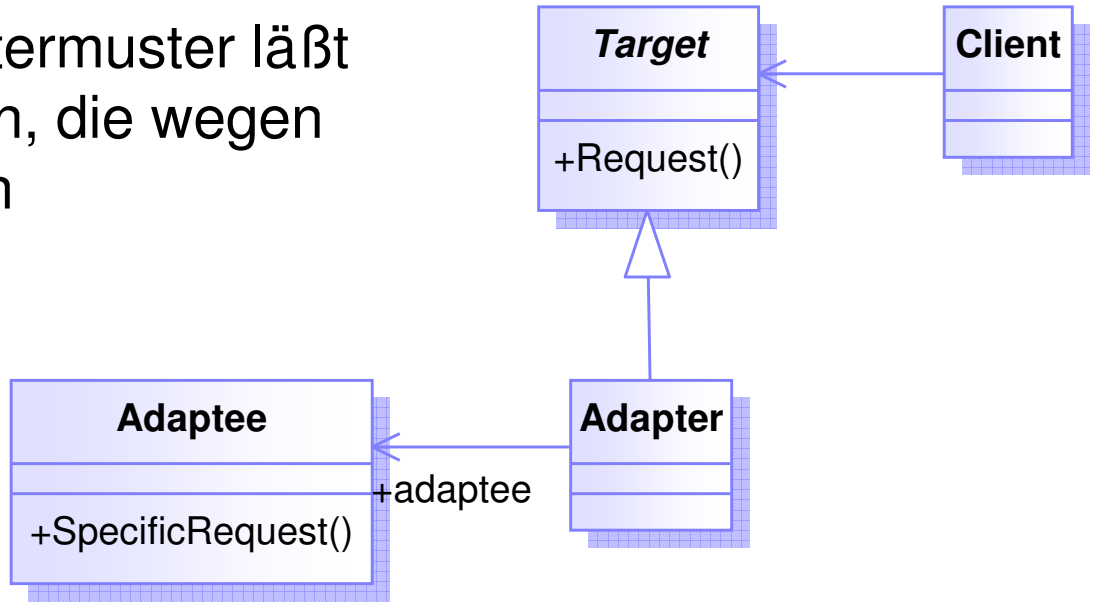
  „Biete eine einheitliche Schnittstelle zu einer Menge von Schnittstellen eines Subsystems. Die Fassadenklasse definiert eine abstrakte Schnittstelle, welche die Benutzung des Subsystems vereinfacht.“



Adapter

- Definition [GoF]:

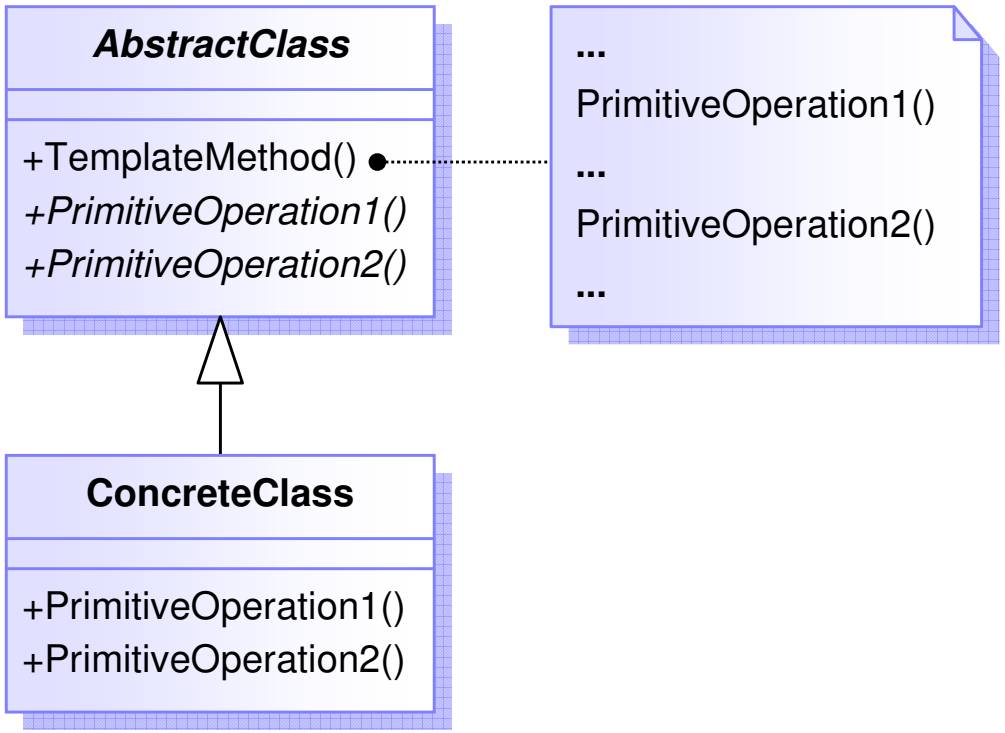
  „Passe die Schnittstelle einer Klasse an eine andere von ihren Klienten erwartete Schnittstelle an. Das Adaptermuster läßt Klassen zusammenarbeiten, die wegen inkompatibler Schnittstellen ansonsten nicht dazu in der Lage wären.“



Schablonenmethode – Template Method

- Definition [GoF]:

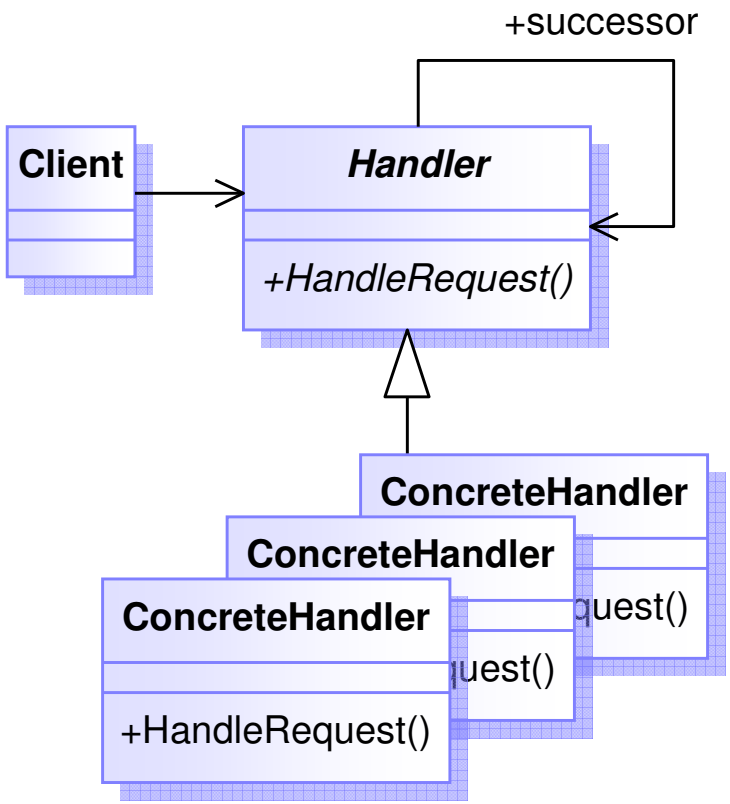
  „Definiere das Skelett eines Algorithmus in einer Operation und delegiere einzelne Schritte an Unterklassen. Die Verwendung einer Schablonenmethode ermöglicht es Unter-klassen, bestimmte Schritte eines Algorithmus zu überschreiben, ohne seine Struktur zu verändern.“



Zuständigkeitskette – Chain of Responsibility

- Definition [GoF]:

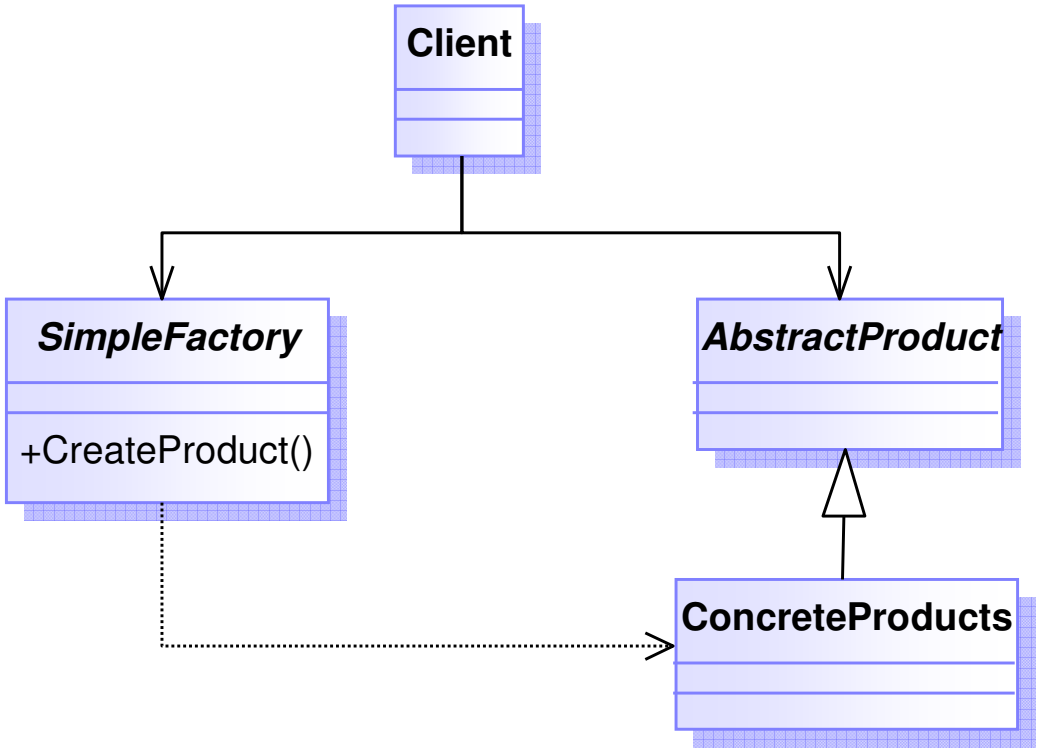
  „Vermeide die Kopplung des Auslösers einer Anfrage mit seinem Empfänger, indem mehr als ein Objekt die Möglichkeit erhält, die Aufgabe zu erledigen. Verkette die empfangenden Objekte und leite die Anfrage an der Kette entlang, bis ein Objekt sie erledigt.“



Fabrik – Simple Factory

- Die Fabrik ist eigentlich kein „richtiges“ Entwurfsmuster.
- Definition [HFDP]:

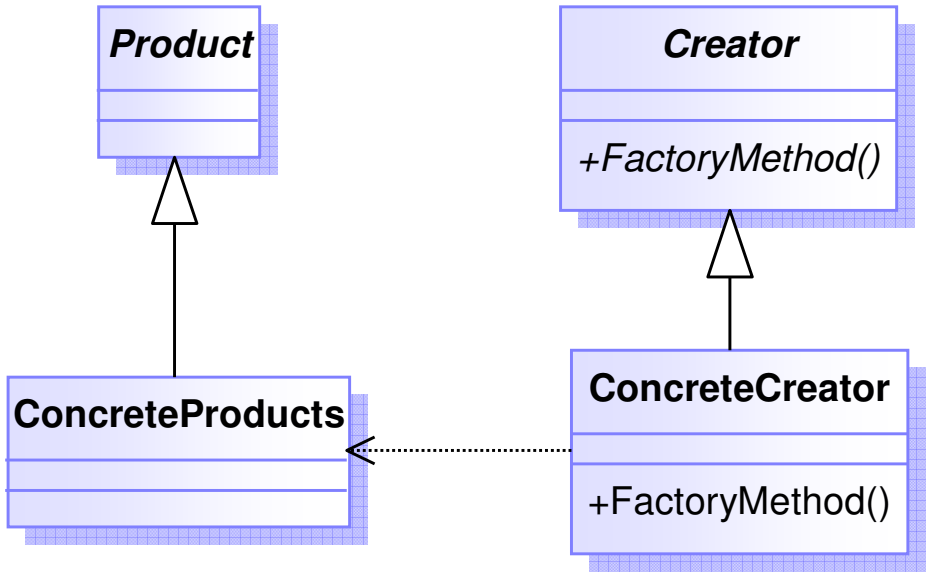
„Fabrik ist ein einfacher Weg um die Benutzer einer Klasse von der konkreten Klasse selbst zu entkuppeln.“



Fabrikmethode – Factory Method

- Definition [GoF]:

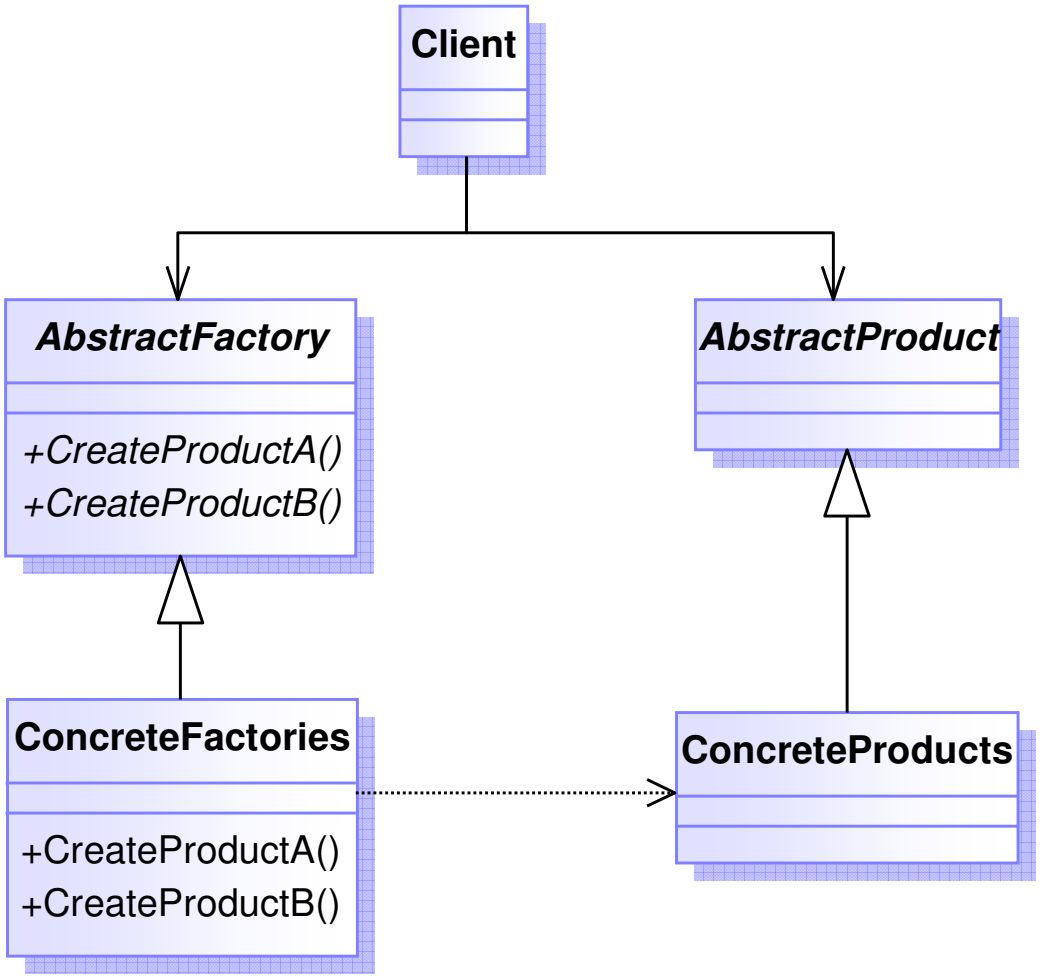
  „Definiere eine Klassenschnittstelle mit Operationen zum Erzeugen eines Objekts, aber lasse Unterklassen entscheiden, von welcher Klasse es ist. Fabrikmethoden ermöglichen es einer Klasse, die Erzeugung von Objekten an Unterklassen zu delegieren.“



Abstrakte Fabrik (Abstract Factory)

- Definition [GoF]:

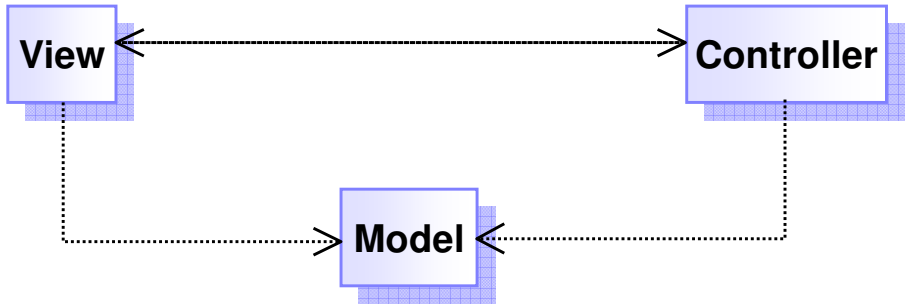
  „Biete eine Schnittstelle zum Erzeugen von Familien verwandter oder voneinander unabhängiger Objekte, ohne ihre konkreten Klassen zu nennen.“



Model View Controller (MVC)

- Definition [POSA]:

  „Das Model-View-Controller-Entwurfsmuster (MVC) unterteilt eine interaktive Anwendung in drei Komponenten. Das Modell (Model) beinhaltet die Kernfunktionalität und die Daten. Die Darstellung (View) zeigt Informationen dem Benutzer an. Der Controller behandelt Benutzereingaben. Darstellung und Controller zusammen bilden die Benutzerschnittstelle. Ein Übertragungs-mechanismus für Änderungen sorgt für die Konsistenz zwischen der Benutzerschnittstelle und dem Modell.“



Model View Controller (MVC)

- MVC ist ein zusammengesetztes Muster
- Modell (Model)
  - hält die Daten,
  - prüft Geschäftsbedingungen usw.
- Darstellung (View)
  - eine (meist) visuelle Repräsentation des Modells
- Controller
  - verbindet Darstellung und Modell,
  - reagiert auf Benutzereingaben und Modell-Daten-Änderungen und
  - überträgt Änderungen

Wo sind wir jetzt?

Motivation

Grundlagen

Einige wichtige Entwurfsmuster im Detail

Weitere Entwurfsmuster in aller Kürze

OOP in LotusScript – Einschränkungen und Lösungen

Quellen

OOP in LotusScript – Einschränkungen

- Keine Schnittstellen (interfaces)
- Keine abstrakten Methoden und Klassen
- Keine Mehrfach- oder Schnittstellenvererbung
  - wobei ich das letztere vorziehen würde, wie in Java
- Keine Pakete oder Namensräume
- Kein Überladen von Methoden (gleicher Name, verschiedene Parametersignaturen)
  - so wäre auch mehr als ein Konstruktor möglich
- Keine Klassen-Attribute oder -Methoden („static“ in Java)
- Kein „static“-Code, der garantiert einmalig beim Laden der Klasse ausgeführt wird.

OOP in LotusScript – Einschränkungen (forts.)

- Keine inneren Klassen
- Keine Möglichkeit, etwas vor dem Konstruktor der Oberklasse zu tun
- Nichts ähnliches wir die Java Reflection-API (naja, nicht wirklich OOP)
- Und eine schreckliche Entwicklungsumgebung für OOP
  (alles im Deklarationsabschnitt)

OOP in LotusScript – Einschränkungen (forts.)

- Keine dieser Einschränkungen gilt für Java, also warum verwende ich nicht Java überall und immer für die Notes-Anwendungsentwicklung?
- Es gibt keine UI-Klassen in Java!
- Ich möchte dieselben Modell-Klassen für das Front- und Back-end benutzen und für das Front-end geht nur LotusScript.
- Ich „glaube“, LS2J müsste langsam sein (ich habe es aber nicht wirklich getestet); es wäre aber auf alle Fälle recht umständlich.
- Es gibt keine UI-Klassen in Java!

OOP in LotusScript – Lösungen

- „static“-Code, der einmalig beim Laden der Klasse ausgeführt wird
  - Definiere die Klasse in einer eignenen Script-Bibliothek
  - Benutze die globale Sub Initialize

- Keine Klassen-Attribute oder -Methoden („static“ in Java)
  - Definiere die Klasse in einer eigenen Script-Bibliothek
  - Bibliotheks-globale Variablen sind fast wie Klassen-Attribute.
  - Subs und Functions sind wie Klassen-Methoden.
  - Wie gesehen kann das Singleton so implementiert werden.
  - Ist so kein „richtiges“ Singleton, ist aber „gut genug“.
  - Aber so verliert man Vererbung.

OOP in LotusScript – Lösungen (forts.)

- Abstrakte Methoden
  - Werfe einen Fehler in der „abstrakten“ Methode
  - Überschreibe diese Methode in der konkreten Unterklasse
  - So erhält man zumindest Laufzeit-Fehler, wenn schon nicht Fehler während des Kompilierens

```
Public Sub ReadDirectory(…) ' in DirectoryReader
        Error 32000, "Abstract method; " &_
                "has to be implemented in a subclass!"
End Sub

Public Sub ReadDirectory(…) ' in LocalDirectoryReader
        ' do some real work
End Sub
```

OOP in LotusScript – Lösungen (forts.)

- Abstrakte Klassen
  - Werfe einen Fehler im Konstruktor (Sub New) der abstrakten Klasse
  - Test auf Ungleichheit von TypeName und Klassenname
  - So erhält man zumindest Laufzeit-Fehler, wenn schon nicht Fehler während des Kompilierens

```
Public Sub New() ' in DirectoryReader
        If Typename(Me) = "DIRECTORYREADER" Then
                Error 32001, |Abstract class: | &_
                |you cannot create objects with this class ("| &_
                Typename(Me) & |") directly, only with | &_
                |concrete subclasses!|
        End If
End Sub
```

Wo sind wir jetzt?

Motivation

Grundlagen

Einige wichtige Entwurfsmuster im Detail

Weitere Entwurfsmuster in aller Kürze

OOP in LotusScript – Einschränkungen und Lösungen

Quellen

Bücher

- [GoF]: Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: „Entwurfsmuster – Elemente wiederverwendbarer objektorientierter Software“, Addison-Wesley, Bonn, 1996, ISBN 3-89319-950-0
- Original-Titel: “Design Patterns – Elements of Reusable Object-Oriented Software”, Addison-Wesley 1995, ISBN 0-201-63361-2
- [HFDP]: Eric & Elisabeth Freeman: “Head First Design Patterns”, O’Reilly 2004, ISBN 0-596-00712-4
- [POSA]: F. Buschmann, R. Meunier, H. Rohnert, P.Sommerlad, M. Stal: “Pattern-Oriented Software Architecture – A System of Patterns”, Wiley 1996, ISBN 0-471-95869-7
- [PK]: Karl Eilebrecht, Gernot Starke: „Patterns kompakt – Entwurfsmuster für effektive Software-Entwicklung“, Spektrum 2004, ISBN 3-8274-1443-1

Online-Bücher

- James W. Cooper: “The Design Patterns Java Compagnion”
  http://www.patterndepot.com/put/8/JavaPatterns.htm
- Bruce Eckel: “Thinking in Patterns”
  http://www.mindview.net/Books/TIPatterns/

Online-Quellen

- [WP]: Wikipedia
  http://en.wikipedia.org/wiki/Design_pattern_%28computer_science%29
- [PPR]: Portland Pattern Repository's Wiki
  http://c2.com/cgi/wiki?WelcomeVisitors
- Hillside.net – Your Patterns Library
  http://hillside.net/patterns/
- Design pattern in simple examples
  http://www.go4expert.com/forums/showthread.php?t=5127
- Design Patterns in C# and VB.NET - Gang of Four (GOF)
  http://www.dofactory.com/Patterns/Patterns.aspx

Zusammenfassung

- Probleme wiederholen sich
- Du bist **nicht** der einzige Entwickler auf der Welt.
- „Ein Entwurfsmuster ist eine Lösung eines Problems in einem Kontext.“ – und meistens eine ziemlich gute!
- Beachte die OO-Entwurfsprinzipien, denn sie sind das Herz und die Seele jedes Entwurfsmusters.
- “Patterns are tools not rules – they need to be tweaked and adapted to your problem.”, Erich Gamma
- Wie immer gilt: Man sollte dieses „Werkzeug“ nicht überbeanspruchen!
- Entwurfsmuster werden entdeckt, nicht erfunden.
  Geh los und finde welche!

Zum guten Schluss...

- Fragen?
  - jetzt stellen oder...
  - später:

    E-Mail: tbahn@assono.de

    Tel.: 04307/900-401
  - Folien und Beispiele unter

    http://www.assono.de/blog/d6plinks/design-patterns-dnug