NOTES & DOMINO ENTWICKLERCAMP

Entwurfsmuster — oder "Das Wissen der Anderen"

8. März 2010

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

Organisatorisches

Motivation und Grundlagen

Erstes Beispiel: DropHandler und DropperRegistry

Zweites Beispiel: Dynamisches Laden

Drittes Beispiel: Fehlerbehandlung

Einige weitere Entwurfsmuster

Abschluss

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 (OCP)
- seit 1999 mit IBM Lotus Notes/Domino (IBM Certified Advanced Application Developer – Lotus Notes/Domino R4 - 8)
- auch Domino-Administrator (IBM Certified Advanced System Administrator – Lotus Notes/Domino 6 - 8)
- Mein Schwerpunkt liegt auf Anwendungen mit Schnittstellen zu anderen Systemen, z. B. RDBMS, SAP R/3, und interaktiven Web-Anwendungen

Organisatorisches

- bitte zum Schluss Bewertungsbögen ausfüllen
- Zwischenfragen erwünscht!
- bitte Handys aus- oder stummschalten
- bitte zum Schluss Bewertungsbögen ausfüllen

Organisatorisches

Motivation und Grundlagen

Erstes Beispiel: DropHandler und DropperRegistry

Zweites Beispiel: Dynamisches Laden

Drittes Beispiel: Fehlerbehandlung

Einige weitere Entwurfsmuster

Abschluss

OOP = Wunderwaffe?

- Objekt-orientierte 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!

Einschub: Was kann man mit OOP machen, aber nicht ohne?

1. In einer Unterklasse kann man Methoden der Oberklasse überschreiben und damit das „Verhalten" verändern.
2. Man kann eine Klasse mehrfach instanziieren, also mehrere Objekte von einer Klasse erstellen.
3. Man kann Objekte in Variablen speichern oder als Parameter übergeben.
4. Information Hiding/Kapselung: Code steht direkt bei den Daten; private Attribute und Methoden

- Man kann natürlich alles auch irgendwie ohne Klassen und Objekte implementieren, aber nicht so einfach, verständlich und elegant.

Und wo ist das Problem?

- Anforderungen ändern sich (nahezu unvermeidlich)
- [Martin]: Design verrottet!
- Warum?
  - viele Abhängigkeiten (können brechen)
  - Änderungen von bestehendem Code (man kann Fehler einbauen)
  - Copy-and-Paste-Wiederverwendung (jede Änderung überall ausführen, alle Stellen wiederfinden!)
- Was dagegen tun?
  - (direkte) Abhängigkeiten minimieren
  - stabilen Code von variablen Anteilen trennen
  - Wiederverwendung von Code durch Vererbung

Was sind Entwurfsmuster?

- 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 Patterns**)

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 über beanspruchen!

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…

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

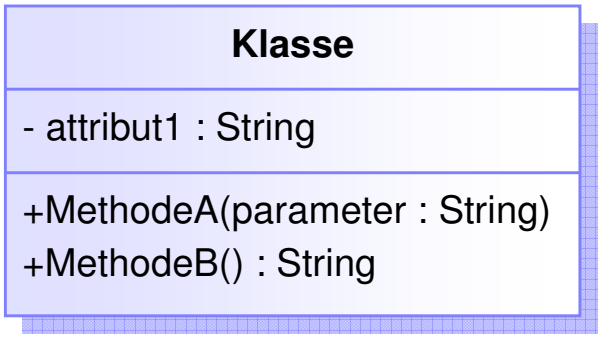
- Entwurfsmuster werden häufig folgenden Klassen zugeordnet:
  - Erzeugungsmuster
    - Erzeugungsmuster beschäftigen sich mit der Erzeugung von Objekten und entkoppeln den Benutzer eines Objekts von dessen Erzeugung.
  - Verhaltensmuster
    - Verhaltensmuster beschäftigen sich mit der Interaktion zwischen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

Exkurs : UML

- UML = Unified Modelling Language
- standardisierte „Bildsprache“ für objektorientierte Modellierung
- soll von Software-Architekten, Programmierern, Fachbenutzern usw. verstanden werden, die Kommunikation vereinfachen und präzisieren
- Es gibt viele verschiedene Diagrammarten, u. a.
  - Klassendiagramme
  - Use-Case-Diagramme
  - Sequenzdiagramme
  - Zustandsdiagramme
- Ich zeige einige Grundlagen zu Klassendiagrammen, die ich später benutze, um die Entwurfsmuster grafisch darzustellen

Exkurs: UML (forts.)

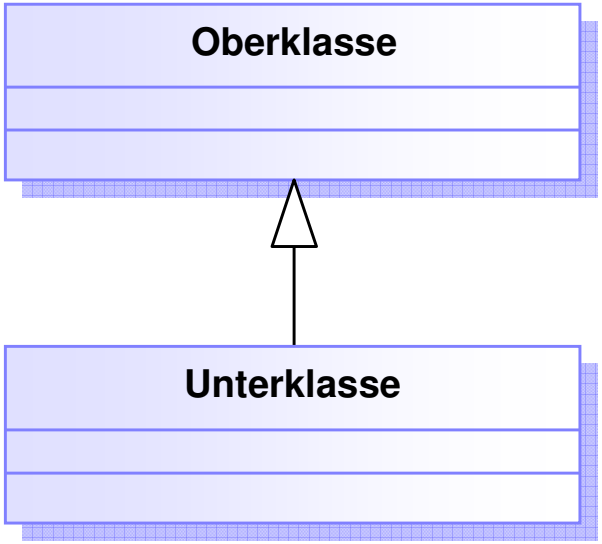
- Klassen, Attribute, Methoden, öffentliche und private Sichtbarkeiten




```
Class Klasse

    Private attribut1 As String

    Public Sub MethodeA(parameter As String)
        ' ...
    End Sub ' Klasse.MethodeA

    Public Function MethodeB() As String
        ' ...
    End Function ' Klasse.MethodeB

    ' ...
End Class ' Klasse
```

Exkurs: UML (forts.)

- Vererbung

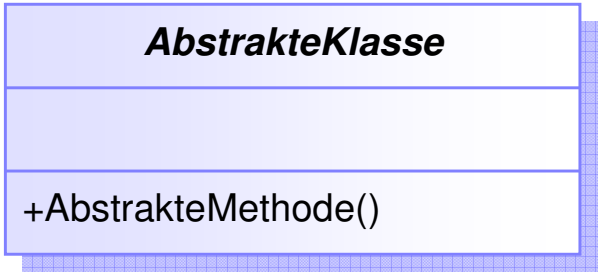



```
Class Oberklasse

End Class ' Oberklasse
```

```
Class Unterklasse As Oberklasse

End Class ' Unterklasse
```

Exkurs: UML (forts.)

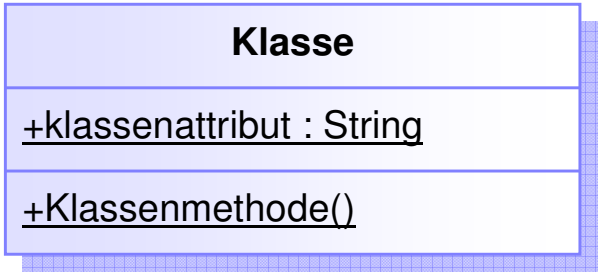
- Abstrakte Klassen und Methoden in LotusScript (Workaround)




```
Class AbstrakteKlasse

    Public Sub New()
        If Typename(Me) = Ucase("AbstrakteKlasse") Then
            Error 1, |Abstract class: … | & Typename(Me)
        End If
    End Sub ' AbstrakteKlasse.New

    Public Sub AbstrakteMethode()
        Error 2, "Abstract method: ..."
    End Sub ' AbstrakteKlasse.AbstrakteMethode

    ' ...
End Class ' AbstrakteKlasse
```

Exkurs: UML (forts.)

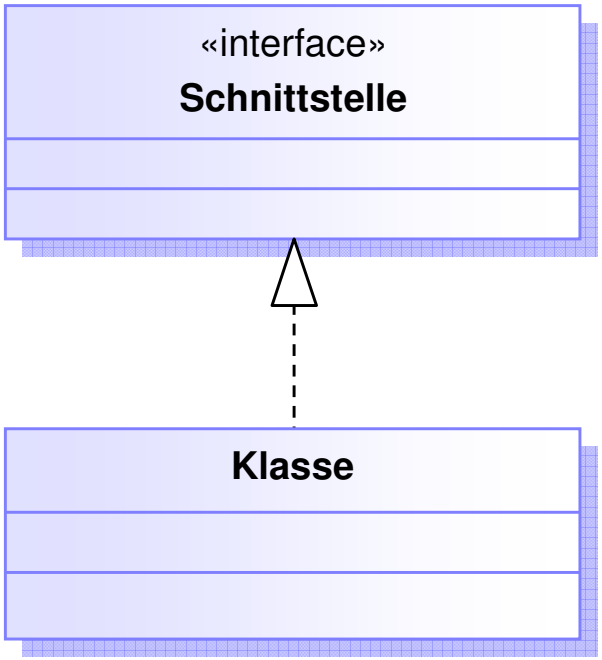
- Klassenattribute und –methoden in LotusScript (Workaround)




```
Private klasseAttribut As String
Private Sub KlasseMethode() ' ...

Class Klasse

    Public Function GetKlassenattribut As String
        GetKlassenattribut = klasseAttribut
    End Function ‘ Klasse.GetKlassenattribut

    Public Sub SetKlassenattribut(attribut as String)
        klasseAttribut = attribut
    End Sub ' Klasse.SetKlassenattribut

    Public Sub Klassenmethode()
        Call KlasseMethode()
    End Sub ' Klasse.Klassenmethode
End Class ' Klasse
```

Exkurs: UML (forts.)

- Schnittstellen (Interfaces): gibt es nicht in LotusScript

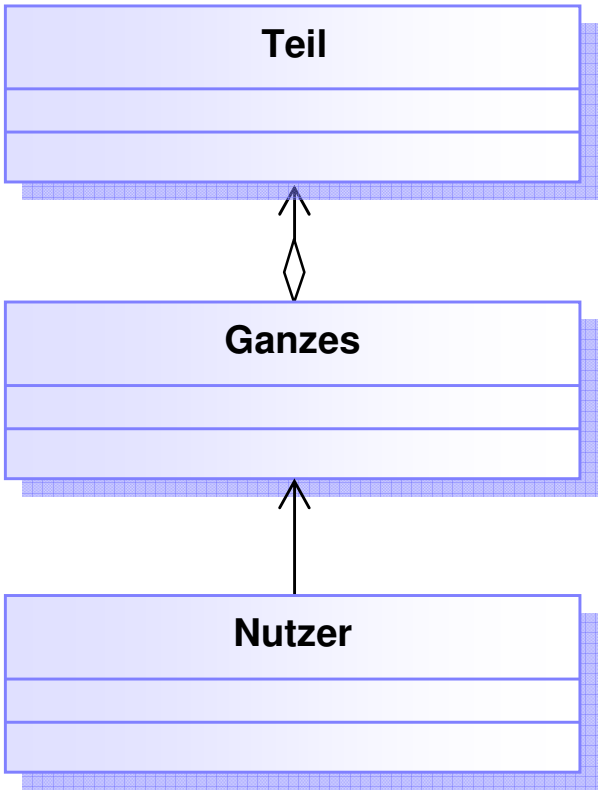



```
Class Schnittstelle ' Klasse statt Schnittstelle?!?

End Class ' Schnittstelle
```

```
' Vererbung statt Implementierung?!?
Class Klasse As Schnittstelle

End Class ' Klasse
```

Exkurs: UML (forts.)

- Assoziation (hat-ein-Beziehung) und Aggregation (Teil-Ganzes-Beziehung)




```
Class Teil

End Class ' Teil

' Aggregation

Class Ganzes
      Dim teile() As Teil
End Class ' Ganzes

‘ Assoziation

Class Nutzer
      Dim meinGanzes As Ganzes
End Class ' Nutzer
```

Organisatorisches

Motivation und Grundlagen

Erstes Beispiel: DropHandler und DropperRegistry

Zweites Beispiel: Dynamisches Laden

Drittes Beispiel: Fehlerbehandlung

Einige weitere Entwurfsmuster

Abschluss

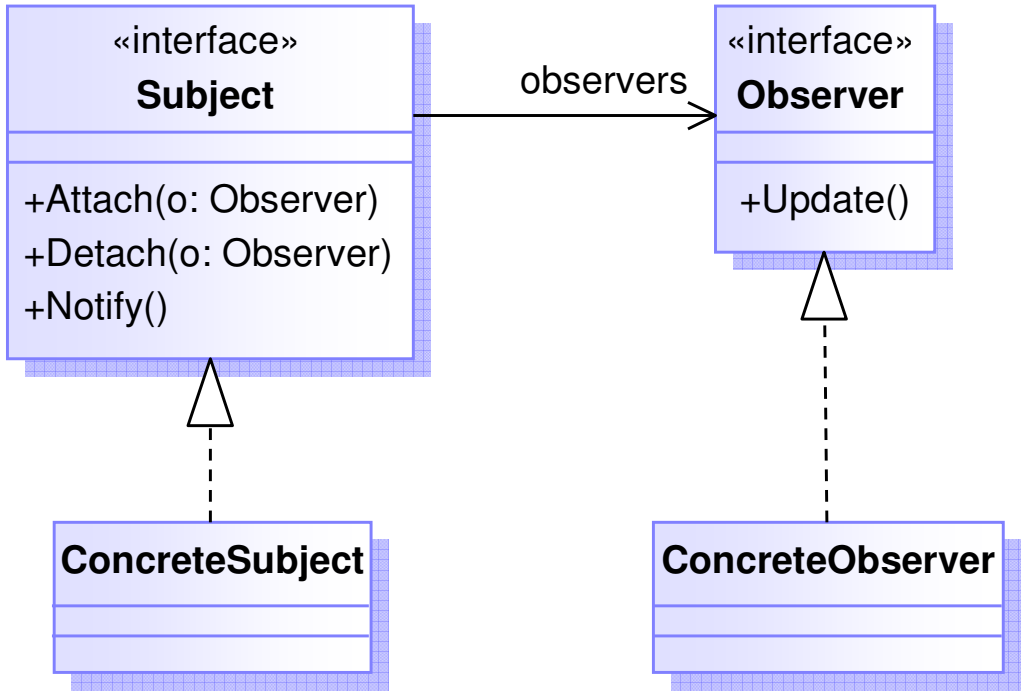
Design Patterns in Notes

- Ein Beispiel – erst einmal ohne OOP
- Nachrichten abonnieren:
  - jeder Benutzer erstellt Profil mit seinen Interessen
  - neue Nachricht zu einem Thema
    - Agent durchläuft alle Profile und sucht nach Abonnenten, die sich für das Thema der neuen Nachricht interessieren
    - versendet Benachrichtigungen an die gefundenen Benutzer
- Das ist ein Entwurfsmuster! (nämlich: Beobachter – Observer)

Beobachter – Observer

- 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.)

- Ein anderes Beispiel, diesmal Klassen in LotusScript

- Source-Code-Review (Datenbank-Skript):
  - DropperRegistry (Subject) und
  - DropHandler (Observer)

- DropHandler.New registriert sich selbst bei der DropperRegistry (Subject.Attach)
- PostDragDrop ruft DropperRegistry.HandleDrop auf
- HandleDrop ruft für jedes Dokument und jeden registrierten DropHandler dessen HandleDocumentDrop-Methode auf (Observer.Update)

Beobachter – Observer (forts.)

- Verwendung von Beobachter – Observer
  - in Java: sehr, sehr viel
    - z. B. Listeners in Swing und JavaBeans
  - in Domino: Extension Manager API
    - z. B. Virenscanner:
      Benachrichtigung bei neuen Mails in der mail.box
  - in Notes-Anwendungen: z. B. Newsletter-Versand
  - LotusScript: selten bis nie
    - keine Nebenläufigkeit
    - und wenn doch, dann voneinander isoliert
    - Reaktionen auf Benutzerinteraktionen: meist Formelsprache oder einfach “Felder bei Schlüsselwortänderung aktualisieren”

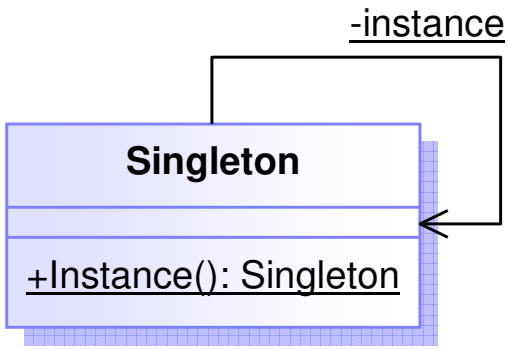
Genau eine einzige DropperRegistry-Instanz…

- Es muss genau **eine einzige DropperRegistry-Instanz** geben, damit das Verfahren funktioniert.
- Diese muss von jedem DropHandler-Konstruktor und von PostDragDrop **leicht erreichbar** sein.
- Schön wäre es außerdem, wenn die einzige Instanz **automatisch erzeugt** würde. Dann kann man nämlich das Erstellen des Objekts nicht aus Versehen vergessen.
- Dafür gibt es auch ein Entwurfsmuster: **Singleton**

Singleton

- Definition [GoF]:

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



Singleton (forts.)

- Aber wie kann man Klassenattribute und –Methoden in LotusScript implementieren? Gar nicht!

- Source-Code-Review (Datenbank-Skript):
  - dropperRegistry (globale Variable)
  - GetDropperRegistry() (globale Funktion)

- dropperRegistryIndirectInstantiation:
  verhindere direktes Aufrufen des Konstruktors

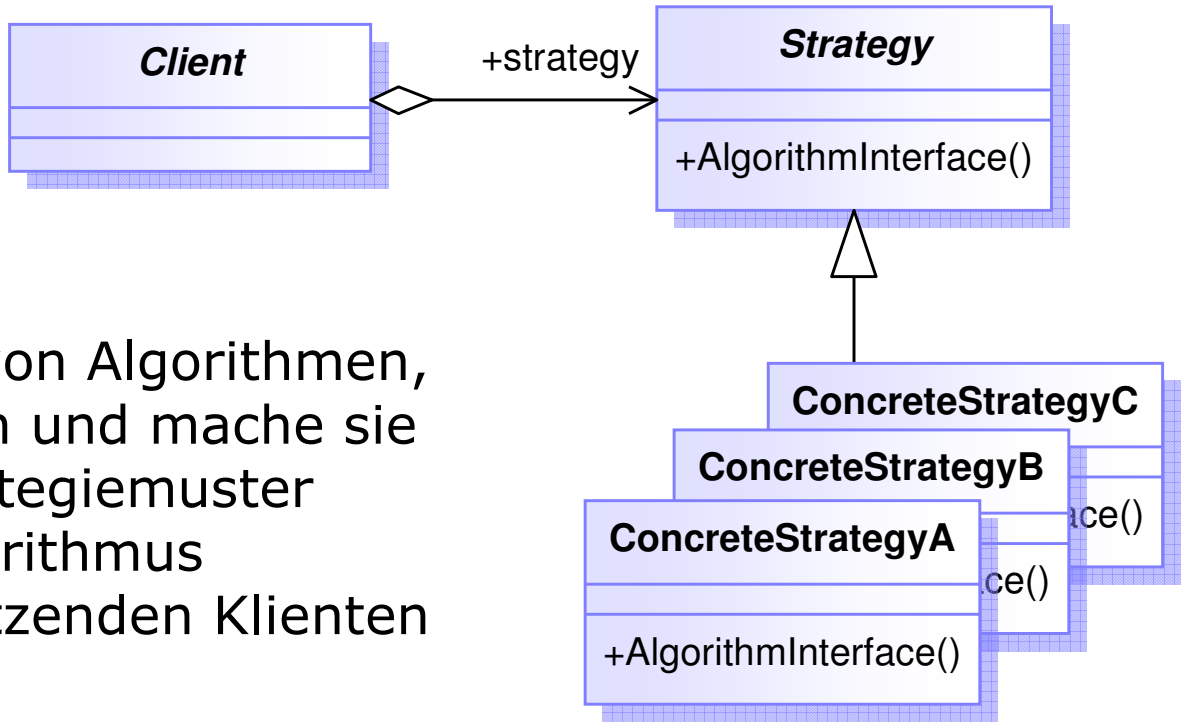
Und da ist noch mehr…

- Die DropperRegistry verwaltet ein Array von DropHandler-Objekten.
- Aber die DropHandler-Klasse selbst „weiß" gar nicht, was sie tun soll, wenn ein Dokument gedroppt wird:

- Source-Code-Review (Datenbank-Skript):
  - DropHandler.HandleDocumentDrop(): wirft Fehler, sonst nichts
  - LoggingDropHandler und RemovingDropHandler überschreiben diese Methode sinnvoll

- Nur die Unterklassen „wissen", was zu tun ist.
- Das ist das nächste Entwurfsmuster: Strategie – Strategy

Strategie – Strategy



- 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.“

Organisatorisches

Motivation und Grundlagen

Erstes Beispiel: DropHandler und DropperRegistry

Zweites Beispiel: Dynamisches Laden

Drittes Beispiel: Fehlerbehandlung

Einige weitere Entwurfsmuster

Abschluss

Dynamisches Laden von Script-Bibliotheken

- Statisches Laden von Script-Bibliotheken mit Use "*Scriptlibrary name*"
- Ladezeit steigt leider nicht linear mit der Anzahl geladener Bibliotheken, sondern (leicht) exponential
- Lösung: dynamisches Laden von Script-Bibliotheken je nach Bedarf
- Damit werden die Script-Bibliotheken wieder super-schnell geladen.
- [Redbook] Appendix B-2 – Dynamic Script Library Loading, S. 222ff.
- Je nach Umgebung (Notes-Version, Plattform, Benutzerrechten usw.) können verschiedene Klassen dynamisch geladen werden.

Dynamisches Laden von Script-Bibliotheken (forts.)

- Source-Code-Review (Common Notes/Domino utils)
  - CreateFactory(): globale Funktion
    Im Execute wird die Script-Bibliothek scriptLibraryName geladen und ein neues Objekt der Klasse factoryClassName erstellt.
  - newFactory As AbstractFactory: notwendige globale Variable
    Eigentlich läuft der LotusScript-Code im Execute isoliert, er hat aber Zugriff auf öffentliche globale Variablen und Routinen
  - AbstractFactory: definiert abstrakte Methoden, mit denen Instanzen von anderen Klassen der dynamisch geladenen Script-Bibliothek erzeugt werden können!
  - factoriesCache List As AbstractFactory: Cache für Effizienz
  - CreateInstance*(): globale Funktionen für größere Bequemlichkeit

Dynamisches Laden von Script-Bibliotheken (forts.)

- Source-Code-Review (Logged deletion utils)
  - GetCascadedDeletionManager(): globale Funktion
  - erzeugt eine CascadedDeletionManagerFactory aus der Script-Bibliothek "Relations utils"
  - Diese Bibliothek wird nicht statisch geladen: siehe (Options)

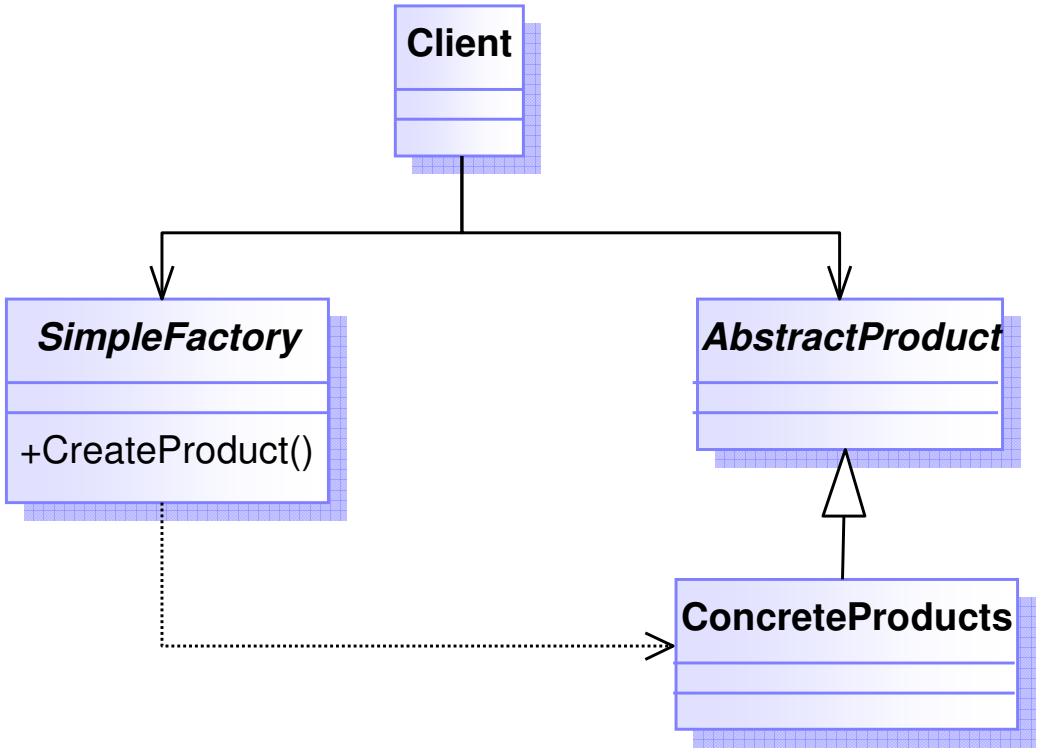
Dynamisches Laden von Script-Bibliotheken (forts.)

- Source-Code-Review (Relations utils)
  - CascadedDeletionManagerFactory As AbstractFactory
  - CascadedDeletionManagerFactory.CreateInstanceWithArgs: erzeugt CascadedDeletionManager-Instanzen
  - CascadedDeletionManager.DeletedDependentDocuments: löscht Dokumente für bestimmte Relationen
  - Relation.DeleteDependentDocuments: ruft DeleteDocument-Prozedur aus "Logged deletion utils" auf
  - Call ExecuteSubWithArgs (in "Common Notes/Domino utils"):
    - kleiner Bruder der CreateFactory-Funktion
    - auch hier wird Script-Bibliothek dynamisch geladen

Fabrik – Simple Factory

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

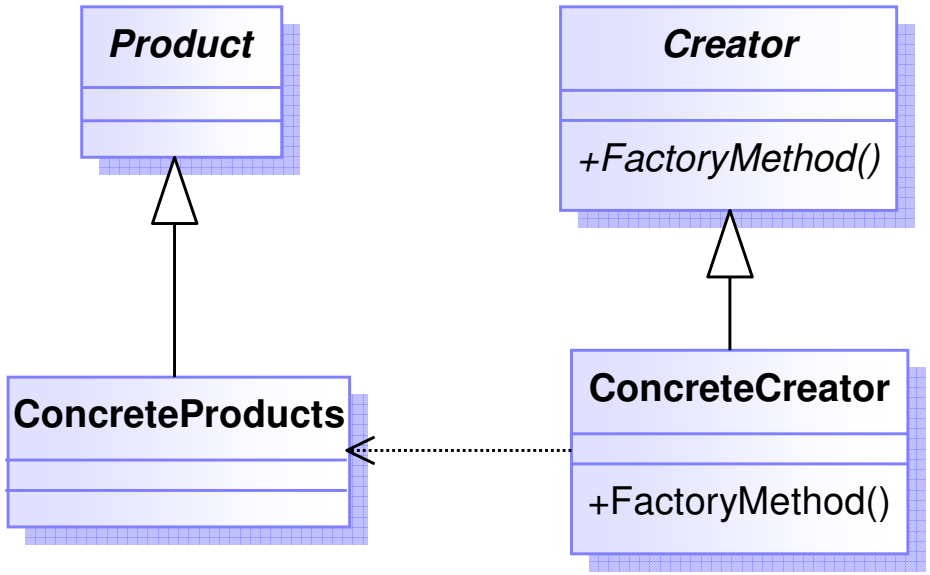
  „Fabrik ist ein einfacher Weg um die Benutzer einer Klasse von der konkreten Klasse selbst zu entkuppeln.“



Fabrikmethode – Factory Method

- Definition [GoF]:

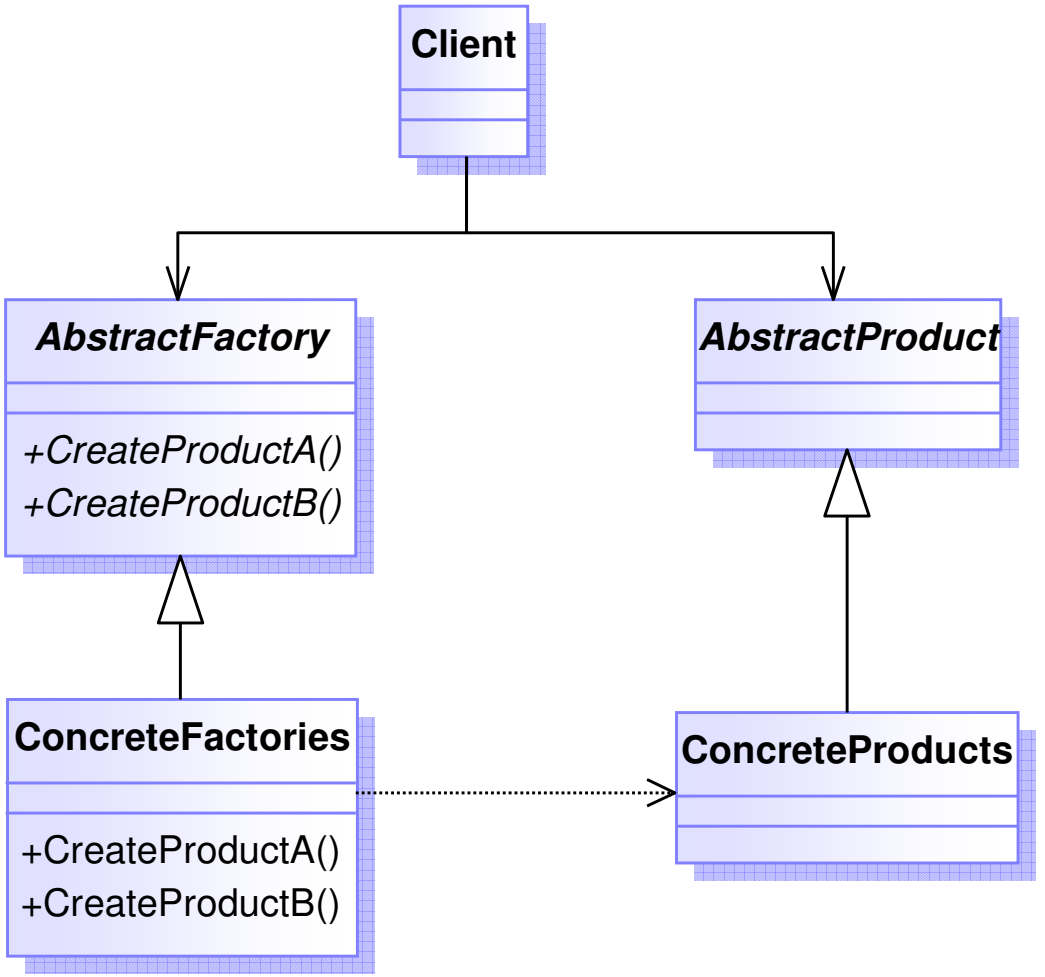
  „Definiere eine Klassenschnittstelle mit Operationen zum Erzeugen eines Objekts, aber lasse Unterklassen entscheiden, von welcher Klasse es ist. Fabrikmethoden ermöglichen es einer Klasse, die Erzeugung von Objekten an Unterklassen zu delegieren."



Abstrakte Fabrik – Abstract Factory

- Definition [GoF]:

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



Dynamisches Laden von Script-Bibliotheken (forts.)

- AbstractFactory ist eine Abstrakte Fabrik – Abstract Factory
- In anderen Script-Bibliotheken kann es konkrete Fabriken geben, wie die CascadedDeletionManagerFactory aus der Script-Bibliothek "Relations utils"
- Wenn die konkreten Fabriken die CreateInstanceOf*()-Methoden überschreiben, können von der einen Fabrik verschiedene „Produkte“ erzeugt werden.
- Die CreateInstance*()-Methoden von AbstractFactory sind Fabrikmethoden – Factory Methods

Organisatorisches

Motivation und Grundlagen

Erstes Beispiel: DropHandler und DropperRegistry

Zweites Beispiel: Dynamisches Laden

Drittes Beispiel: Fehlerbehandlung

Einige weitere Entwurfsmuster

Abschluss

Fehlerbehandlung – mehr als nur protokollieren

- Fehlerbehandlung wird leider häufig vernachlässigt
  - muss einfach zu verwenden sein
  - sollte Details zum Fehlerkontext automatisch bestimmen
  - sollte flexibel erweiterbar sein
- Wir haben da mal was vorbereitet…

- Source-Code-Review (Error handling utils)
  - ErrorDetails: ermittelt selbständig Details zum Fehler
  - ErrorContext: kann Kontextinformationen zu einem Fehler (Text oder Dokument) speichern
  - CodePosition: Speicher für Position im Quellcode
  - TemplateBuildDetails: Speicher für Versionsinformationen zur verwendeten Schablone

Fehlerbehandlung – mehr als nur protokollieren (forts.)

- Source-Code-Review (Error handling utils; forts.)
  - ErrorHandler: kann Fehler „behandeln“
  - LoggingHandler As ErrorHandler: protokolliert Fehler
  - MessageboxHandler As ErrorHandler:
    gibt Fehler in Messagebox aus
  - IgnoreErrorHandler As ErrorHandler:
    „behandelt“ Fehler mit Ignoranz ☺
  - EMailNotification As ErrorHandler:
    schickt Fehlerbenachrichtigung per E-Mail

Fehlerbehandlung – mehr als nur protokollieren (forts.)

- Source-Code-Review (Error handling utils; forts.)
  - ErrorHandler im Detail:
    - IsResponsibleFor(errorDetails):
      Ist der ErrorHandler „verantwortlich" für die Behandlung eines bestimmten Fehlers? Nur wenn ein ErrorHandler hier True zurückgibt, wird er aufgefordert, den Fehler zu „behandeln".
    - CheckResponsibility(errorDetails): abstrakte Methode
    - HandleError(errorDetails): „behandelt" den Fehler
    - HandleErrorWithContext(errorDetails, errorContext): „behandelt" den Fehler im gegebenen Kontext
    - DoHandleError(errorDetails): abstrakte Methode

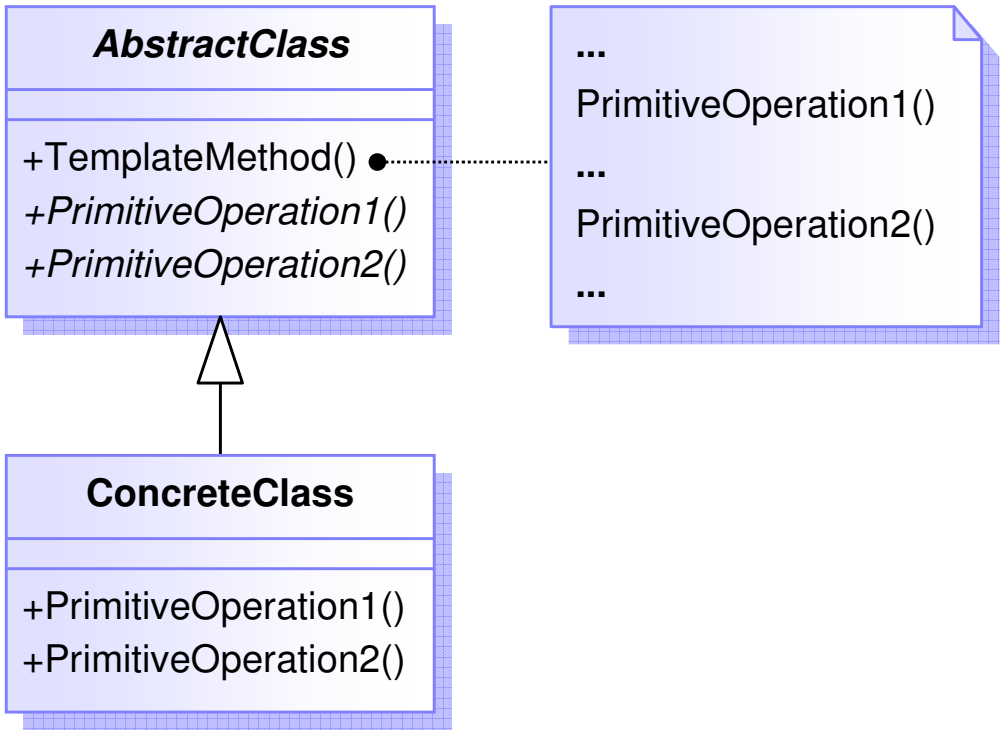
Fehlerbehandlung – mehr als nur protokollieren (forts.)

- CheckResponsibility()-Methode:
  wird wieder erst in Unterklassen von ErrorHandler sinnvoll überschrieben → Strategie – Strategy
- Sie wird aber schon in ErrorHandler selbst aufgerufen und genutzt
  - IsResponsibleFor() ruft CheckResponsibility() auf
  - IsResponsibleFor() ist eine Art „Schablone“, die erst von der konkreten Unterklasse ausgefüllt wird.

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."



Fehlerbehandlung – mehr als nur protokollieren (forts.)

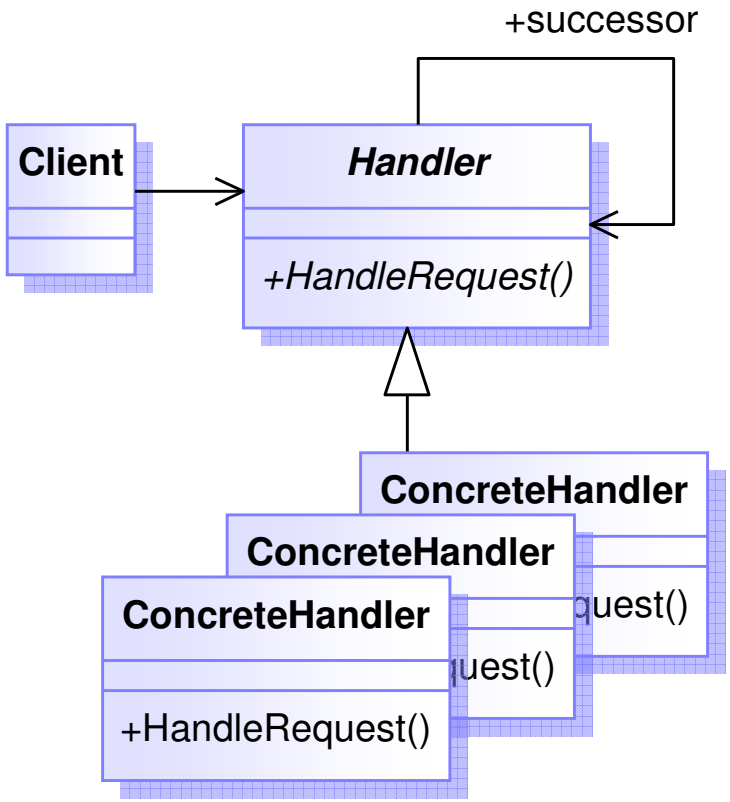
- Alle ErrorHandler werden in einer Prioritäts-Queue verwaltet:

- Source-Code-Review (Error handling utils; noch einmal)
  - errorHandlersArray As Variant
  - RegisterErrorHandler(): ErrorHandler zur Queue hinzufügen
  - RemoveErrorHandler(): und wieder entfernen
  - HandleError() durchläuft registrierte ErrorHandler nach Priorität
    - Ist IsResponsibleFor() wahr, wird HandleError() bzw. HandleErrorWithContext() aufgerufen.
    - Ist das Ergebnis dieses Aufrufs wahr, wurde also der Fehler erfolgreich behandelt, wird die Schleife vorzeitig verlassen

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."



Zuständigkeitskette – Chain of Responsibility (forts.)

- Die Fehlerbehandlung ist fast eine Zuständigkeitskette – Chain of Responsibility, aber nur fast, da die ErrorHandler nicht selbst ihren Nachfolger verwalten.

Organisatorisches

Motivation und Grundlagen

Erstes Beispiel: DropHandler und DropperRegistry

Zweites Beispiel: Dynamisches Laden

Drittes Beispiel: Fehlerbehandlung

Einige weitere Entwurfsmuster

Abschluss

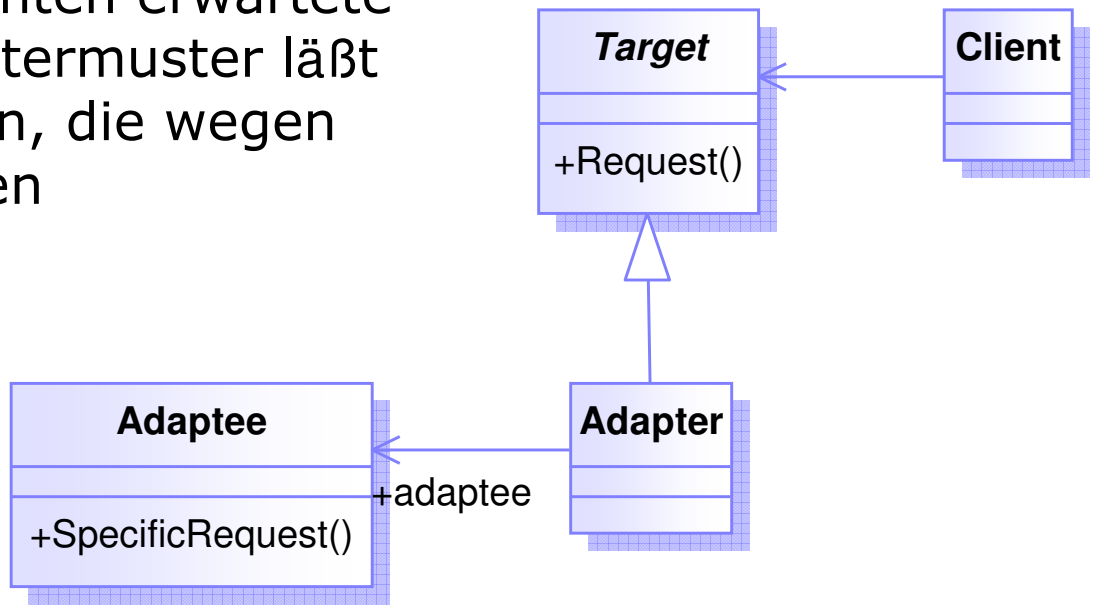
Einige weitere Entwurfsmuster

- In alphabetischer Reihenfolge (und nur, wenn noch Zeit übrig ist)
  - Adapter
  - Befehl – Command
  - Befehlsprozessor – Command Processor
  - Dekorierer – Decorator
  - Fassade – Façade
  - Iterator
  - Kompositum – Composite
  - Model View Controller (MVC)
  - Proxy

Adapter

- Definition [GoF]:

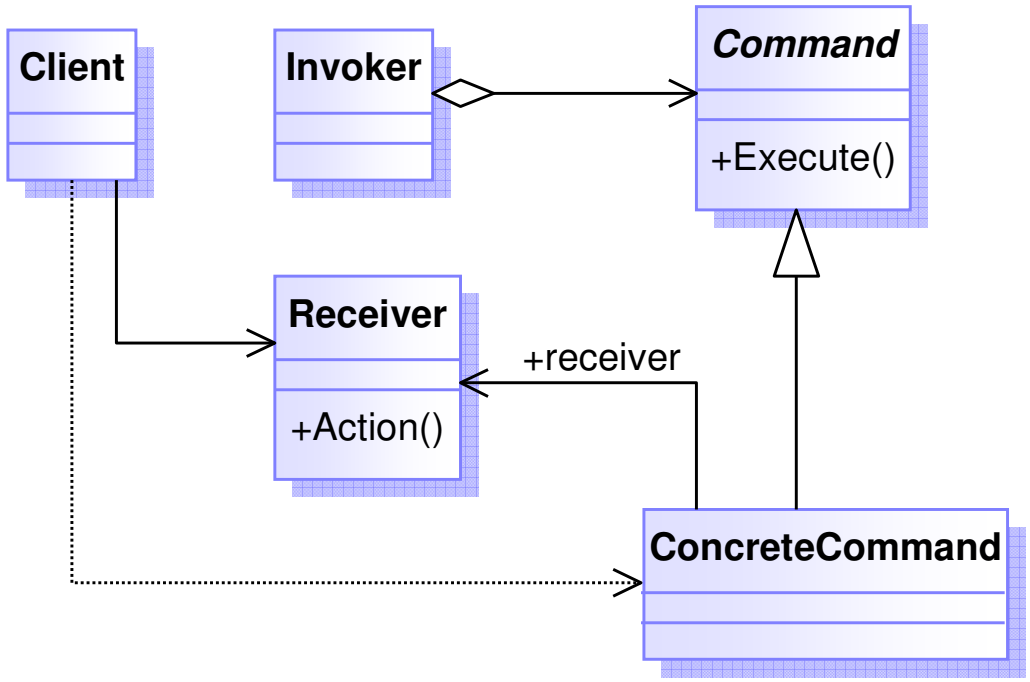
  „Passe die Schnittstelle einer Klasse an eine andere von ihren Klienten erwartete Schnittstelle an. Das Adaptermuster läßt Klassen zusammenarbeiten, die wegen inkompatibler Schnittstellen ansonsten nicht dazu in der Lage wären.“



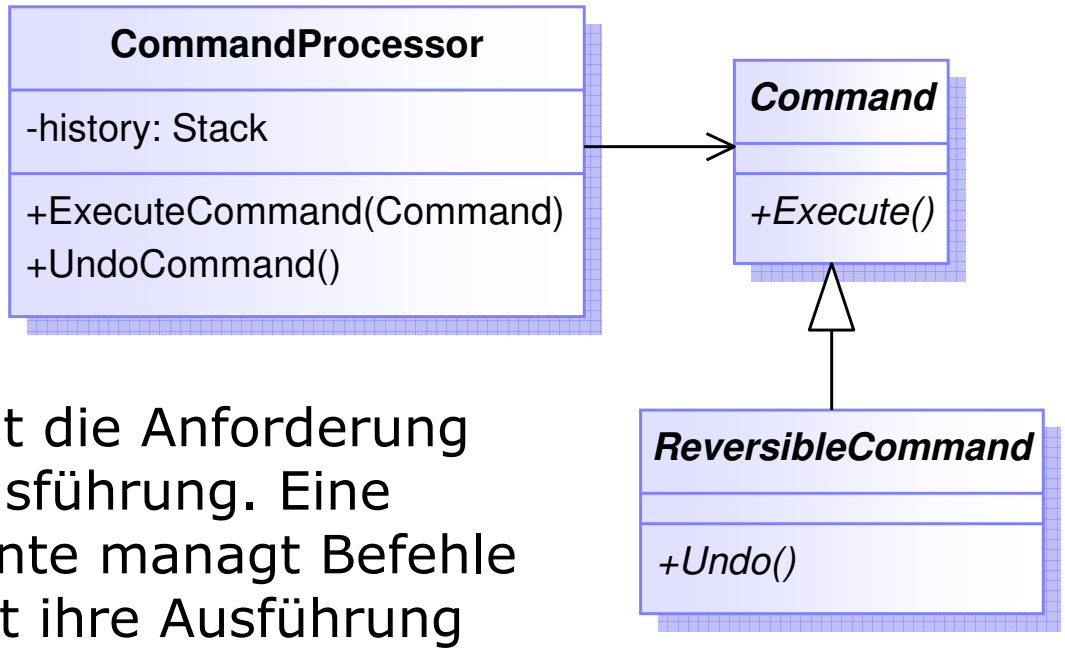
Befehl – Command

- 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."



Befehlsprozessor – Command Processor



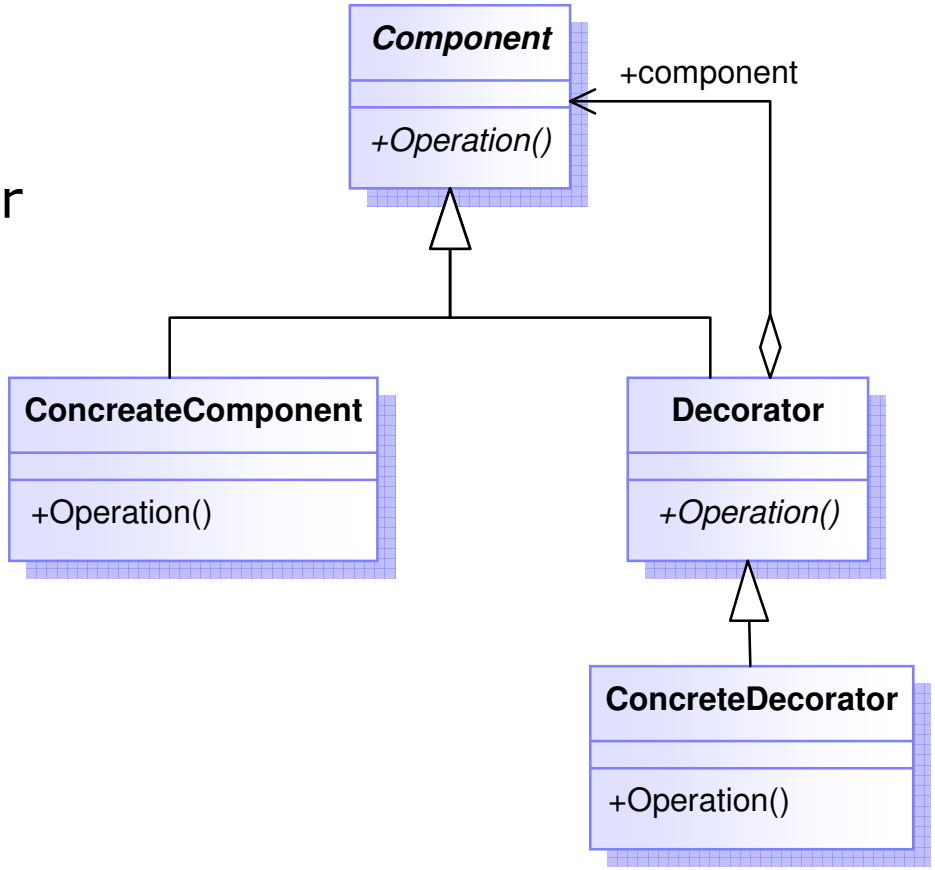
- Definition [POSA]:

  „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."

Dekorierer – Decorator

- Definition [GoF]:

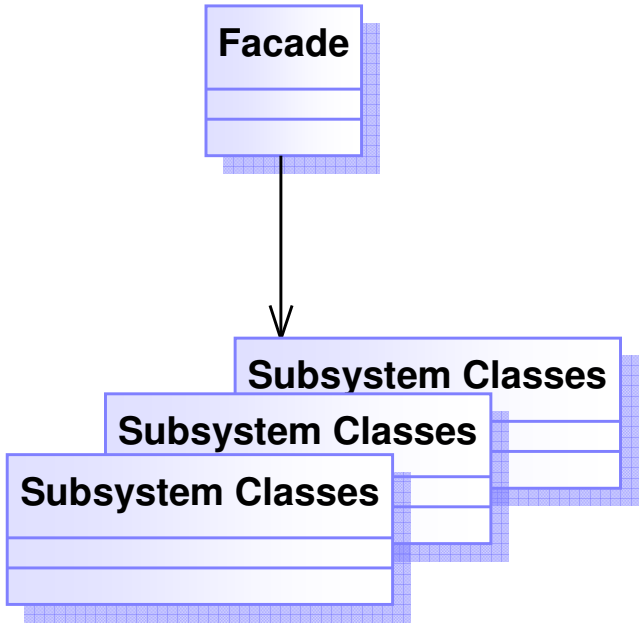
  „Erweitere ein Objekt dynamisch um Zuständigkeiten. Dekorierer bieten ein flexible Alternative zur Unterklassenbildung, um die Funktionalität einer Klasse zu erweitern.“



Fassade – Façade

- Definition [GoF]:

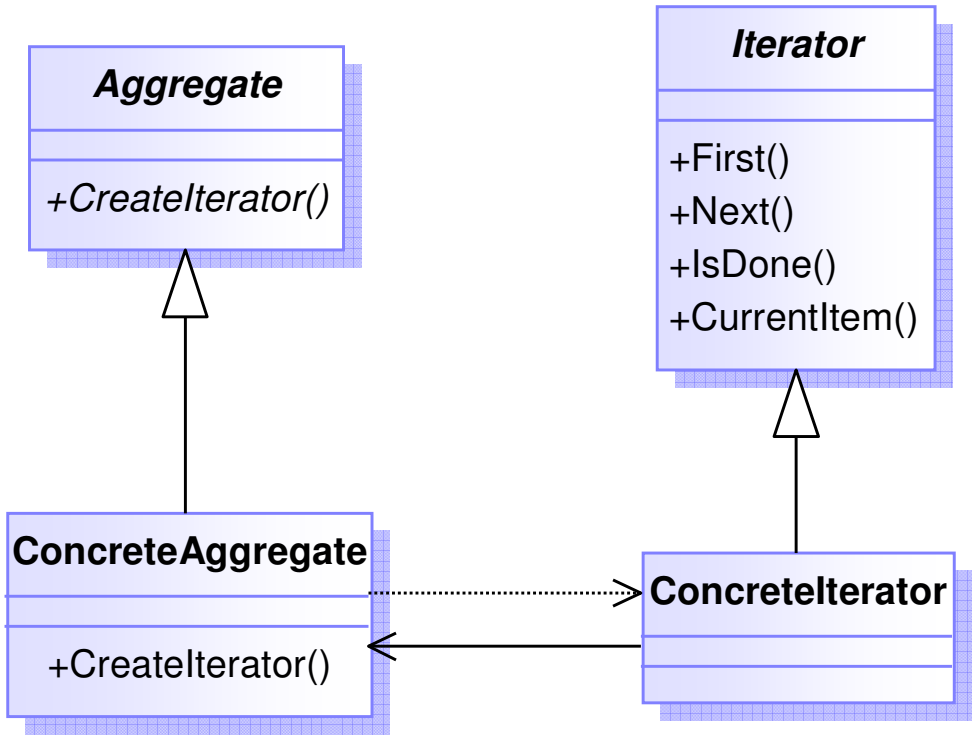
  „Biete eine einheitliche Schnittstelle zu einer Menge von Schnittstellen eines Subsystems. Die Fassadenklasse definiert eine abstrakte Schnittstelle, welche die Benutzung des Subsystems vereinfacht.“



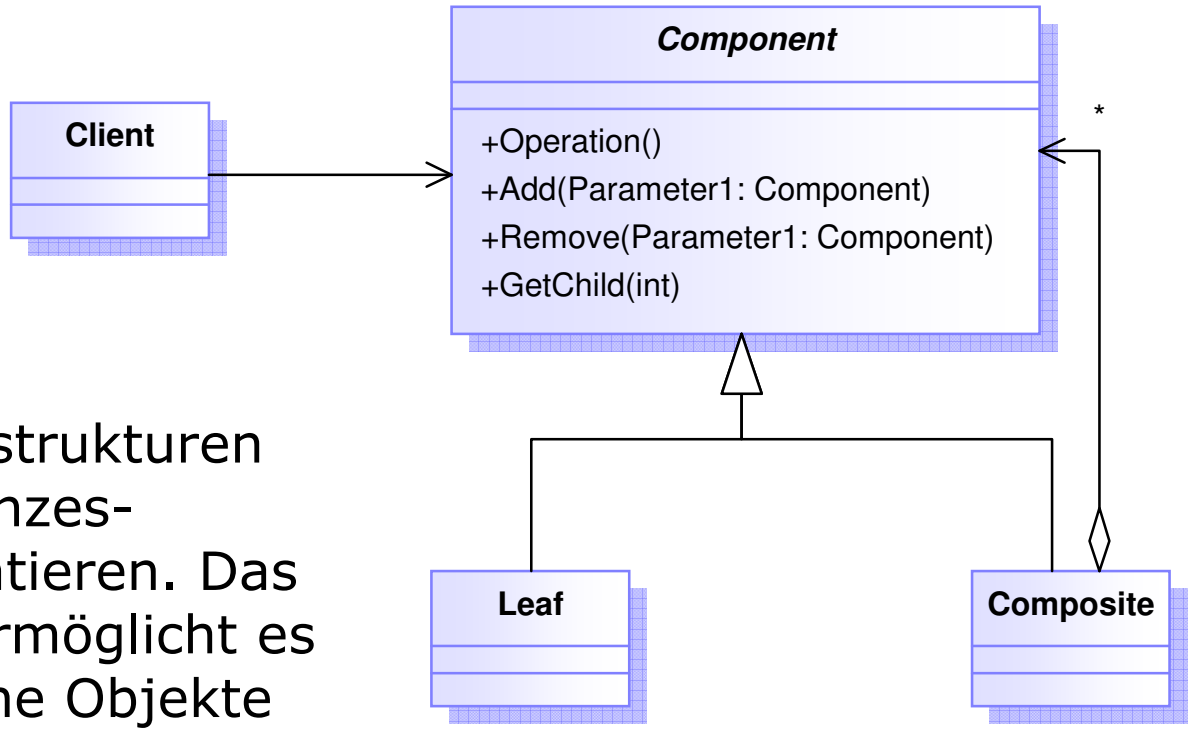
Iterator

- Definition [GoF]:

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



Kompositum – Composite



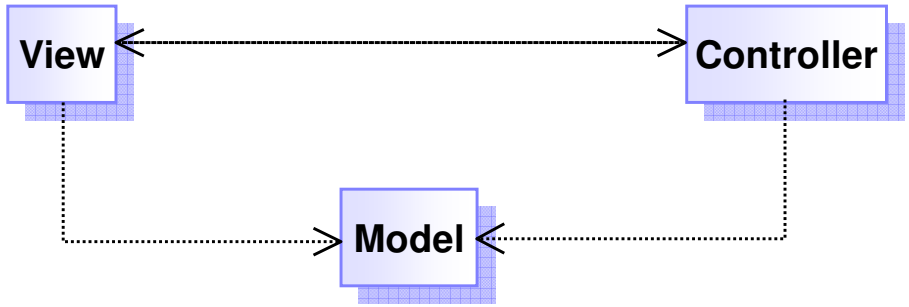
- Definition [GoF]:

  „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."

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."



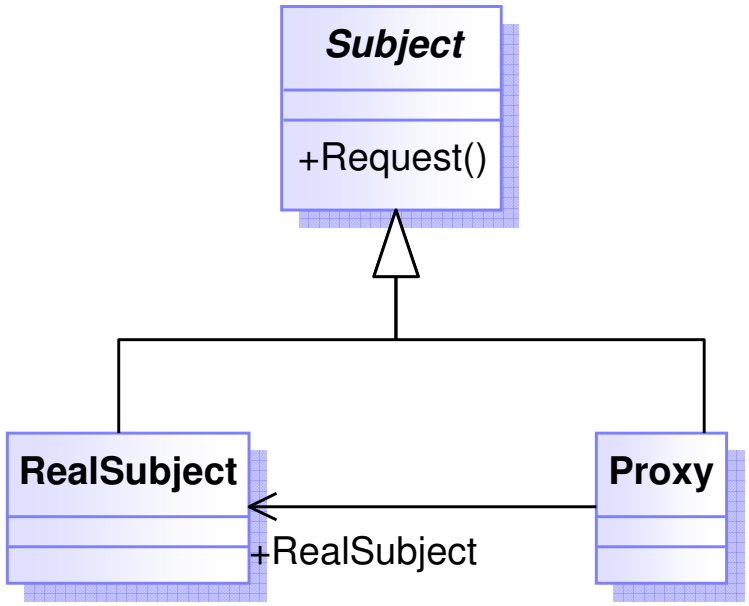
Model View Controller (MVC)

- 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

Proxy

- Definition [GoF]:

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



Organisatorisches

Motivation und Grundlagen

Erstes Beispiel: DropHandler und DropperRegistry

Zweites Beispiel: Dynamisches Laden

Drittes Beispiel: Fehlerbehandlung

Einige weitere Entwurfsmuster

Abschluss

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!
- “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!

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
- Thomas Erler, Michael Ricken: „UML 2 GE-PACKT – die praktische Referenz“, mitp 2005, ISBN 3-8266-1484-4

Online-Bücher und -Artikel

- [Martin]: Robert C. Martin: „Design Principles and Design Patterns“ http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf
- 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/
- [Redbook]: „Performance Considerations for Domino Applications“ http://www.redbooks.ibm.com/redbooks/pdfs/sg245602.pdf

Weitere 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
- assonos blog
  http://www.assono.de/blog

  insbesondere
  http://www.assono.de/blog/archive?openview&type=Category&key=OOP

Hilfsmittel und Werkzeuge

- LotusScript.doc
  - Dokumentation von LotusScript-Code ähnlich wie JavaDoc
  - http://www.lsdoc.org
- Class Navigator
  - Noteshound-„Werkzeugkasten“
  - Class Browser!
  - http://www.noteshound.com
- Teamstudio Script Browser
  - kostenlos
  - http://www.teamstudio.com/support/scriptbrowser.html

Zum guten Schluss…

- **Fragen?**
  - jetzt stellen oder später
    - E-Mail: tbahn@assono.de
    - Telefon: 04307 900-401

- Folien und Beispiel-Datenbank unter http://www.assono.de/blog/d6plinks/EntwicklerCamp-2010-Design-Patterns

Anhänge

OO-Entwurfsprinzipien

Objektorientierte Programmierung (OOP)

OOP in LotusScript – Einschränkungen und Lösungen

OO-Entwurfsprinzipien

- KISS – Keep It Simple, Stupid
  - Vermeide unnötige Komplexität und 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, so dass du Code nur an jeweils einer Stelle ändern oder korrigieren musst.

OO-Entwurfsprinzipien (forts.)

- Prinzip der einzelnen Verantwortlichkeit (Separation of Concerns)
  - Jede Klasse sollte nur genau eine Verantwortlichkeit haben – oder zumindest so wenige wie möglich.
- Offen-Geschlossen-Prinzip (Open-Closed Principle)
  - Klassen sollte offen sein für Erweiterungen, aber geschlossen gegenüber Veränderungen
- Kapsele, was sich ändert
  - So beeinflussen Änderungen nicht den stabilen Teil deines Codes

OO-Entwurfsprinzipien (forts.)

- Bevorzuge Komposition gegenüber Vererbung
  - dann kann man das Verhalten eines zur Laufzeit ändern, indem man in einem Attribut ein Objekt einer anderen Klasse speichert, nicht nur beim Kompilieren, wenn man Methoden in Unterklassen überschreiben kann
- Strebe nach lose gekoppelten Entwürfen (Loose Coupling)
  - Veränderungen wirken dann nur lokal
- Mache die Methoden und Attribute so privat wie möglich und so öffentlich wie nötig
  - Die öffentliche Schnittstelle ist ein Vertrag. Alles, was nur intern bekannt ist, darf nach belieben geändert werden.

Anhänge

OO-Entwurfsprinzipien

Objektorientierte Programmierung (OOP)

OOP in LotusScript – Einschränkungen und Lösungen

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".

Anhänge

OO-Entwurfsprinzipien

Objektorientierte Programmierung (OOP)

OOP in LotusScript – Einschränkungen und Lösungen

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 anonyme 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, private Variablen sind fast wie Klassen-Attribute.
  - private 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) = Ucase("DirectoryReader“) Then
                Error 32001, |Abstract class: | &_
                |you cannot create objects with this class ("| &_
                Typename(Me) & |") directly, only with | &_
                |concrete subclasses!|
        End If
End Sub
```