OOSE: Objektorientierte Software-Entwicklung

Durchführung

Dipl.-Inform. Ralf Reißing

Abteilung Software Engineering

Raum 2.154

Sprechstunde nach Vereinbarung

http://www.informatik.uni-stuttgart.de/ifi/se/people/reissing.html

Kapitel 1: Einführung

- Organisatorisches
  - Voraussetzungen
  - Termine
  - Prüfung
- Inhalte
- Objektorientierte Software-Entwicklung

Einführung

Organisatorisches (1)

Voraussetzungen

- notwendig:
  - Konzepte der Objektorientierung: Kapselung, Vererbung, Polymorphismus
  - Grundkenntnisse des Software Engineering
- vorteilhaft:
  - C++, Java
  - UML
  - Erfahrung in der objektorientierten Software-Entwicklung
  - Besuch der KAOS-Vorlesung

Einführung

Organisatorisches (2)

Termine

Vorlesung (2V) dienstags, 13:15-14:45 Uhr, 1.039 (ab 28.04.1998)

Skript

- leider noch nicht verfügbar, da Vorlesung im Aufbau begriffen
- Kopien der Folien der Vorlesung werden im Semesterapparat abgelegt

Prüfung

- nur als Wahlfach prüfbar (nicht in der Vertiefungslinie SE!)
- mündliche Prüfung, etwa 30 min

Einführung

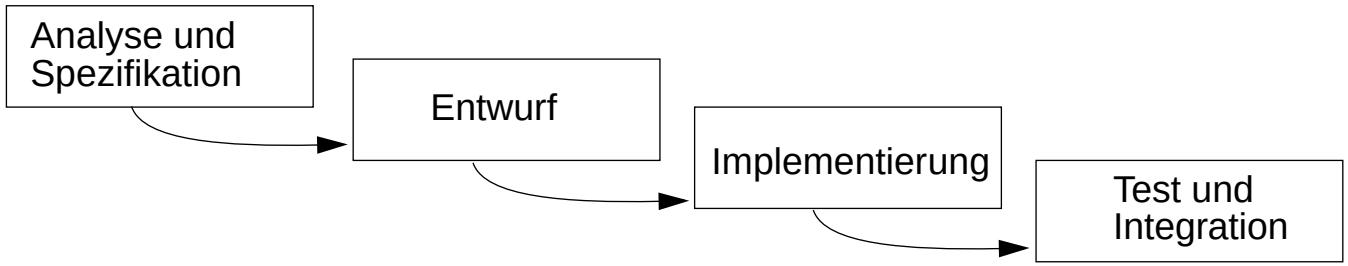
Inhalte

1. Einführung
2. Entwurf
3. Muster
4. Architekturmuster
5. Entwurfsmuster
6. Idiome
7. Bibliotheken, Frameworks
8. Entwurfsheuristiken
9. Werkzeug-Material-Metapher

Objektorientierte Software-Entwicklung

Überblick (1)

- konventionelle Entwicklung (Wasserfallmodell)



- objektorientierte Entwicklung
  - iterative, inkrementelle Entwicklung
  - Bau von Prototypen
  - in jeder Iteration werden die Phasen des Wasserfallmodells durchlaufen
  - kein Strukturbruch: OOA, OOD, OOP

Objektorientierte Software-Entwicklung

Überblick (2)

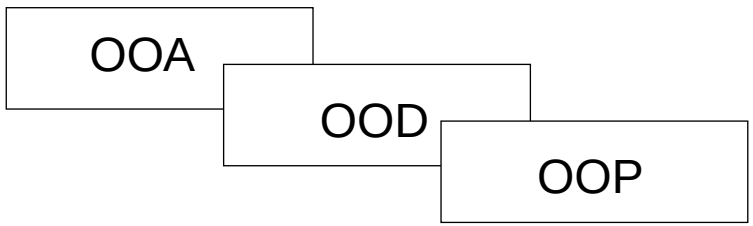
- Objektorientierung ist mehr als Programmieren in einer objektorientierten Sprache!
- Objektorientierung ist ein durchgängiger Ansatz für die Software-Entwicklung
  - Verwenden der objektorientierten Modellierungskonzepte
    - abgeschlossene, autonome Einheiten mit Struktur und Verhalten -> Objekte, Klassen
    - Generalisierung/Spezialisierung -> Vererbung
  - in Analyse (OOA), Entwurf (OOD) und Implementierung (OOP)

Objektorientierte Software-Entwicklung

Überblick (3)

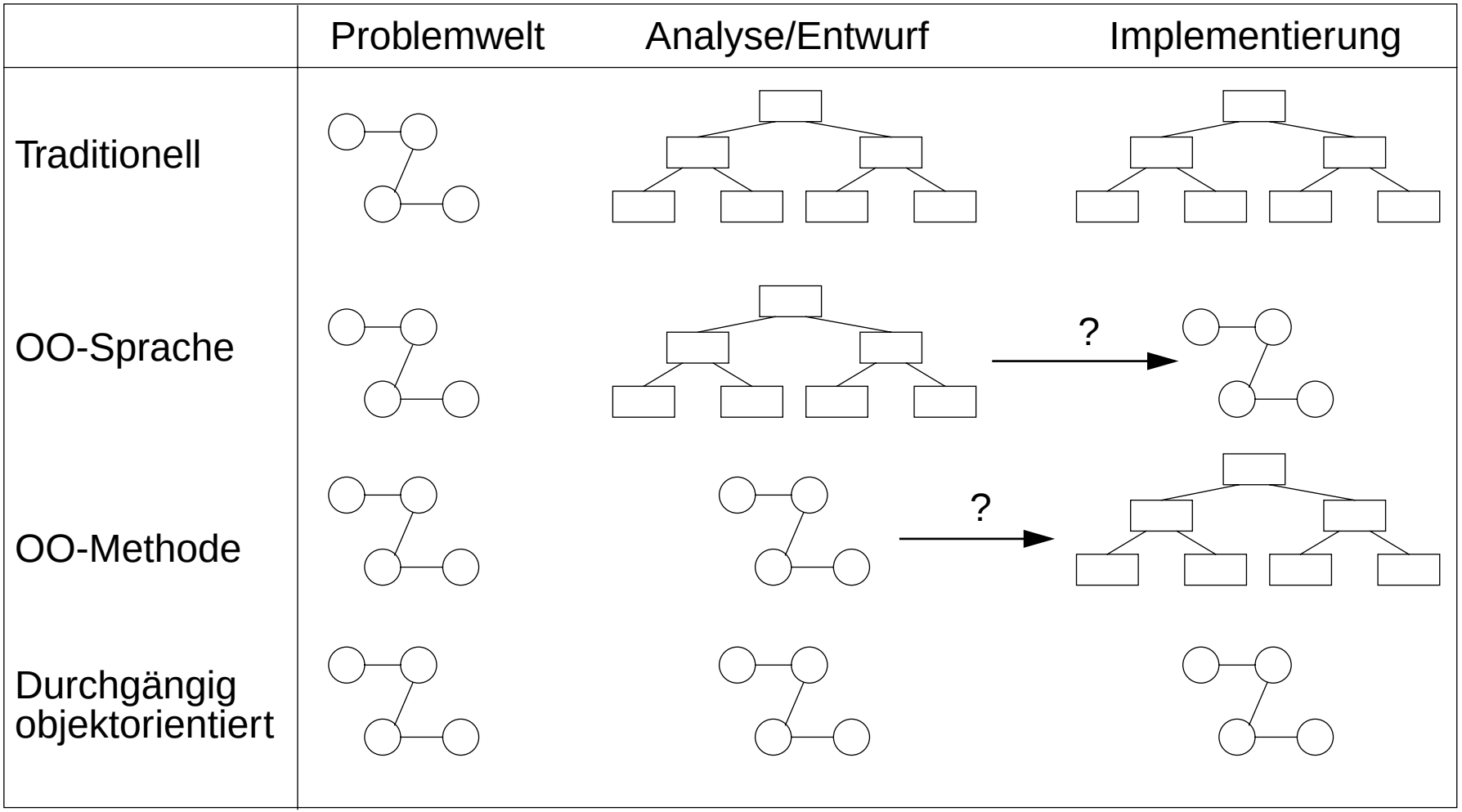
- Brüche zwischen den Ergebnissen werden geringer

- Grenzen zwischen den Tätigkeiten verschwimmen



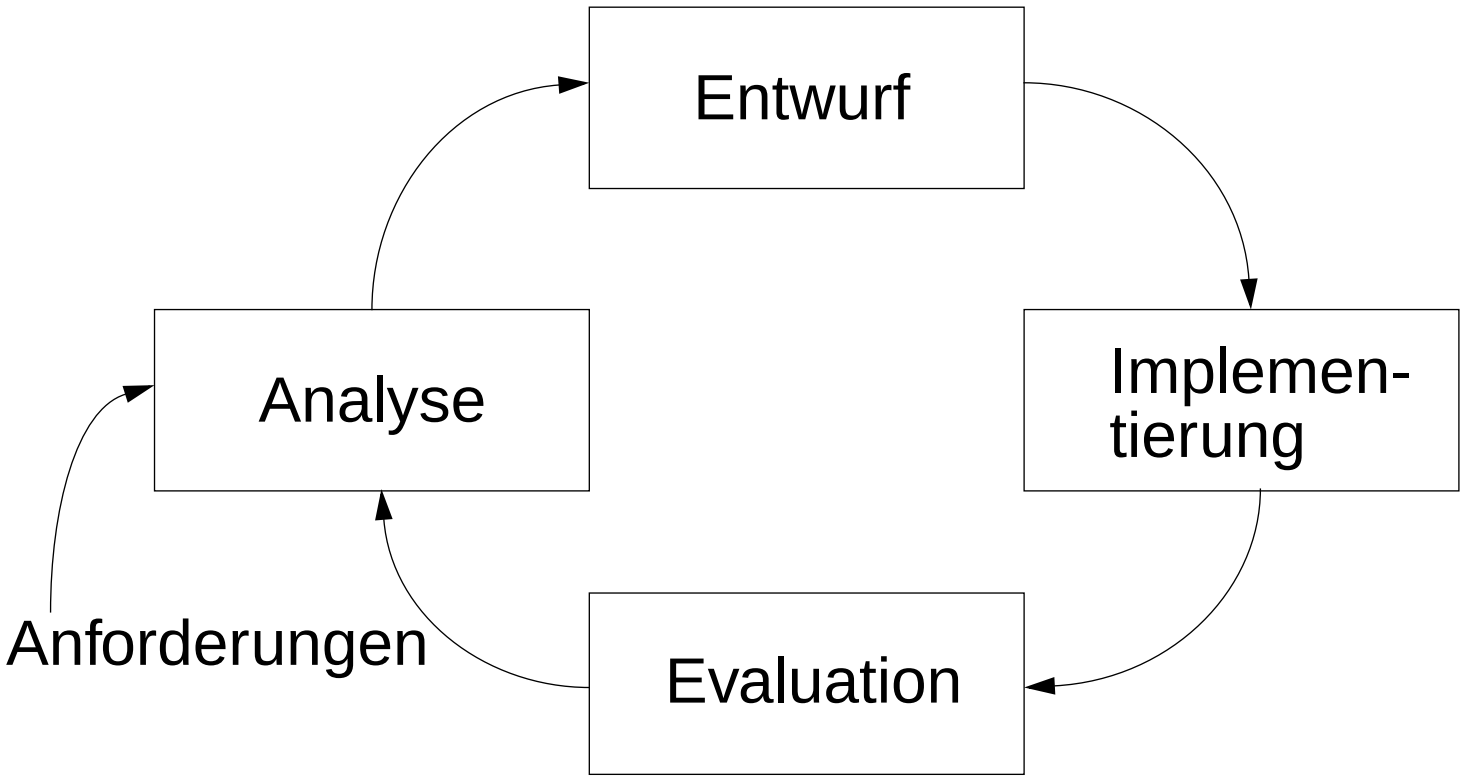
Objektorientierte Software-Entwicklung

Durchgängigkeit der Konzepte



Objektorientierte Software-Entwicklung

Iterative Entwicklung (1)



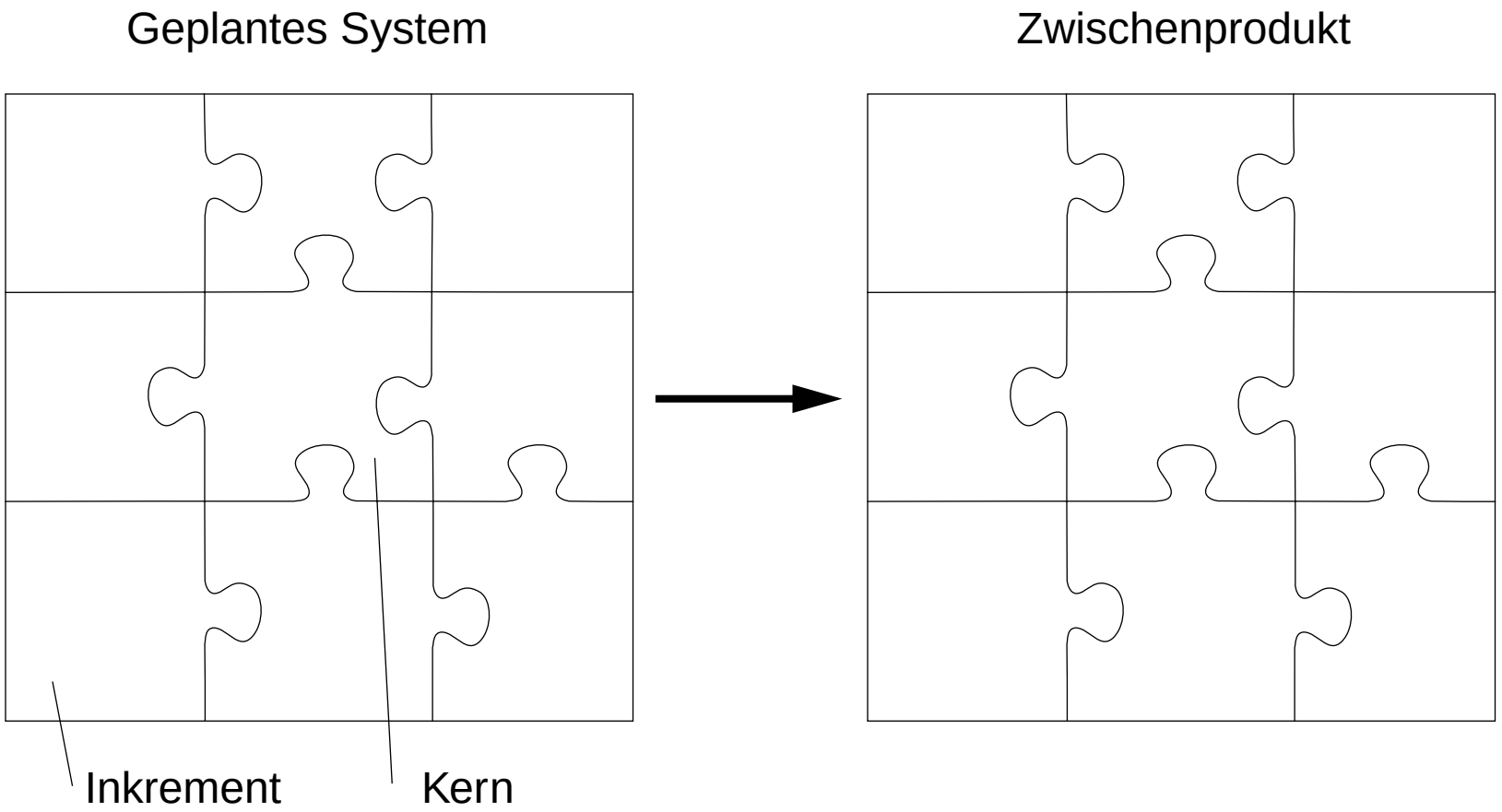
Objektorientierte Software-Entwicklung

Iterative Entwicklung (2)

- Kontrolliertes stetiges Überarbeiten einer Software mit dem Ziel, Fehler zu beheben und Verbesserungen einzuarbeiten
  „we get things wrong before we get them right“
- Löst Probleme mit dem Wasserfallmodell:
  - Anforderungen lassen sich nicht vollständig und konsistent beschreiben
  - Einsatz eines Systems verändert die daran gestellten Anforderungen
  - Wartungsaktivitäten nicht außerhalb der Entwicklung, sondern in die Entwicklung integriert

Objektorientierte Software-Entwicklung

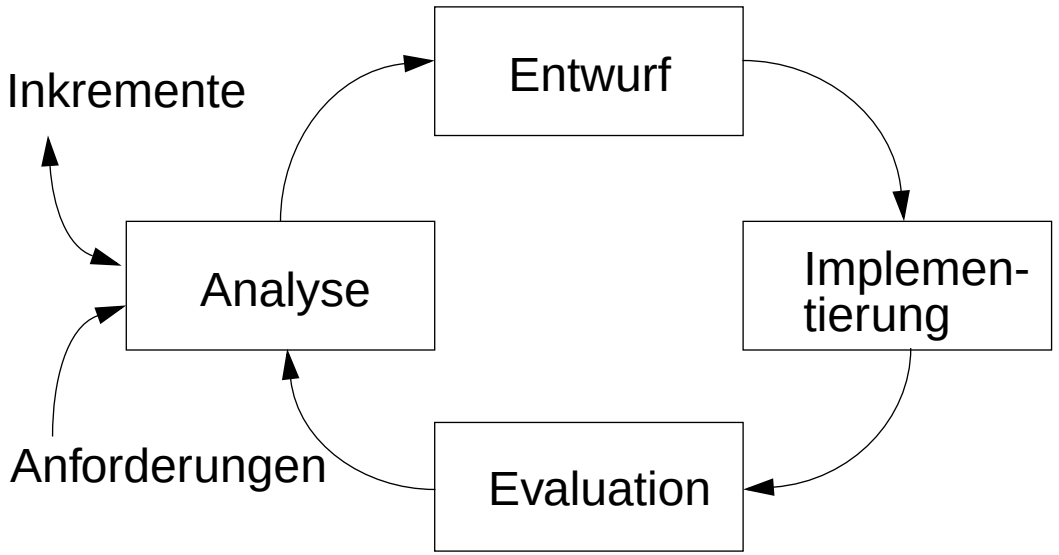
Inkrementelle Entwicklung (1)



Objektorientierte Software-Entwicklung

Inkrementelle Entwicklung (2)

- Aufteilung des Systems in Teile (Inkremente)
- Inkremente werden nach und nach implementiert
- Inkrementelle Entwicklung ist grundsätzlich iterativ!
- Kurze Entwicklungszeiten
- Frühere Evaluation durch Anwender möglich



Objektorientierte Software-Entwicklung

Objektorientierte Analyse (OOA)

Vorgehen

1. Objekte und Klassen der Anwendungswelt identifizieren
2. Verantwortlichkeiten identifizieren und den Klassen zuordnen
3. Zusammenarbeit zwischen den Klassen identifizieren
4. Vererbungshierarchie definieren
5. Subsysteme definieren

Ergebnisse

- Konzeptuelles Modell der Realität
  - Statisches Modell (OOA-Klassenmodell)
  - Dynamisches Modell (z.B. Use Cases, Sequenzdiagramme)

☞ wird ausführlich in den Vorlesung KAOS behandelt

Objektorientierte Software-Entwicklung

Objektorientierter Entwurf (OOD)

Vorgehen

1. Festlegung der Architektur des Systems
2. Überarbeiten des OOA-Klassenmodells, z.B.
   - ☆ Hinzufügen/Verschmelzen/Ändern von Klassen und Subsystemen
   - ☆ Änderungen an der Vererbungsstrukturen
   - ☆ Hinzufügen/Entfernen von Beziehungen

Ergebnisse

- ❑ Modell der Implementierung
  - ❍ Systemarchitektur, u.a. das OOD-Klassenmodell
  - ❍ Dynamisches Modell
- ☞ Schwerpunkt dieser Vorlesung

Objektorientierte Software-Entwicklung

Objektorientierte Programmierung (OOP)

Vorgehen

Umsetzung des OOD-Modells in einer objektorientierten Programmiersprache

Dabei Klassentests und schrittweise Integration des Systems

Ergebnisse

Getestete Implementierungen für die Klassen des OOD-Modells

Integriertes System

☞ wird als bekannt vorausgesetzt

Wiederverwendung

Anspruch und Wirklichkeit bei OO

- Als das OO-Paradigma neu aufkam, hieß es, es löse das Wiederverwendungsproblem:
  - Klassen sind wiederverwendbare Einheiten, da abgeschlossen und selbständig (ADT)
  - Genügt eine Klasse den Anforderungen nicht vollständig, kann sie erweitert werden (Vererbung)
  - ☞ Wiederverwendung sei daher kein Problem
- Aber:
  - nur Wiederverwendung im kleinen (wenn überhaupt)
  - organisatorische Probleme sind immer noch ungelöst
  - Wiederverwendungskultur kommt nicht von alleine
- Neue Hoffnungsträger: Komponenten, Frameworks, Muster, ...

Kapitel 2:
Entwurf

- Bedeutung und Einbettung in den Software Life Cycle
- Entwurfsqualität
- Entwurfsprinzipien
- Exkurs: Entwurfsraum

Entwurf

Begriff

design *(IEEE Standard 610.12-1990 bzw. Taylor, 1959)*

(1) The process of defining the architecture, components, interfaces, and other characteristics of a system or component.
(2) The result of the process in (1)

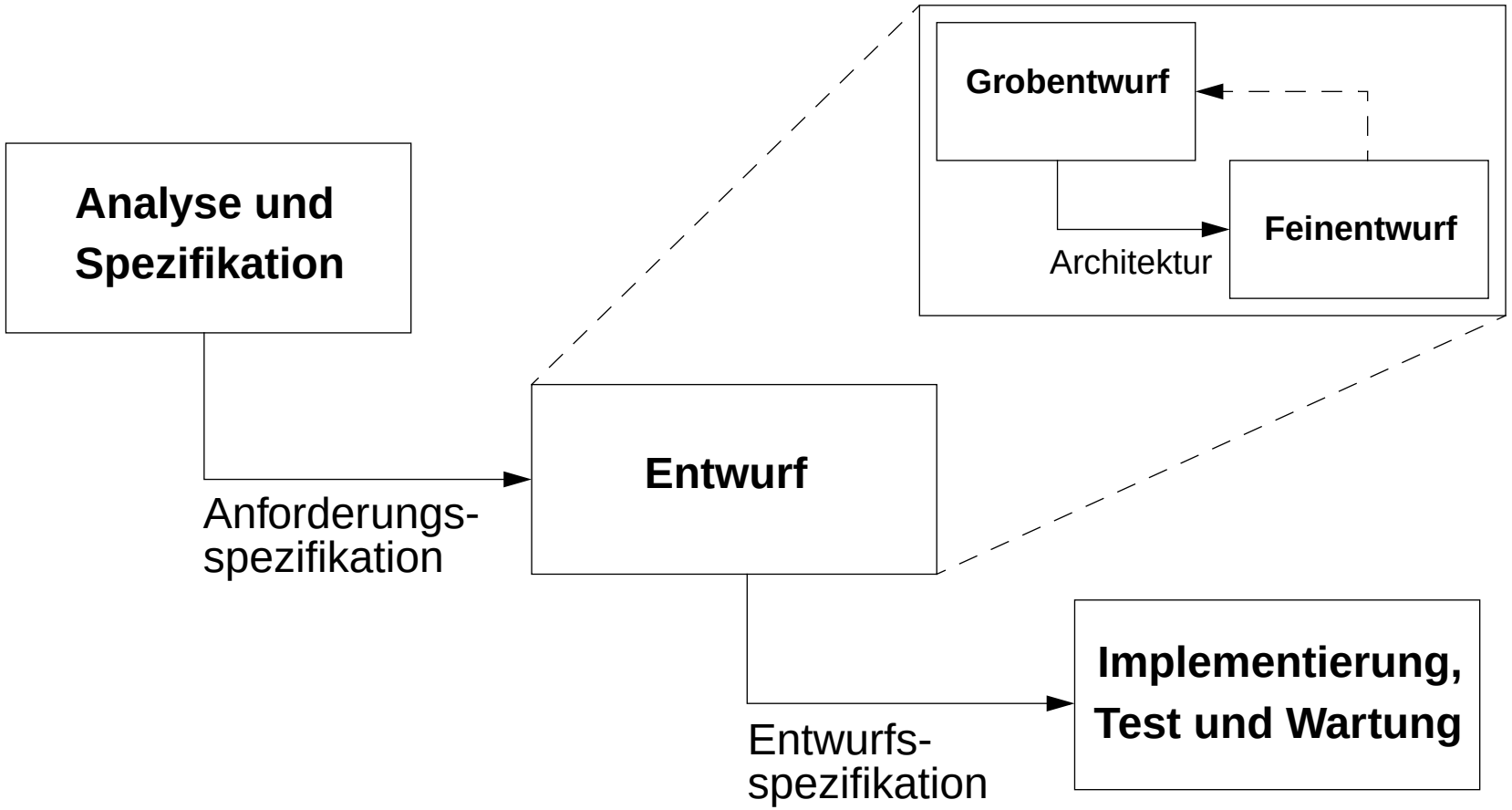
The process of applying various techniques and principles for the purpose of defining a device, a process or a system in sufficient detail to permit its physical realization.

design description *(IEEE Standard 610.12-1990)*

A document that describes the design of a system or component. Typical contents include system or component **architecture**, **control logic**, **data structures**, **input/output-formats**, **interface descriptions**, and **algorithms**.

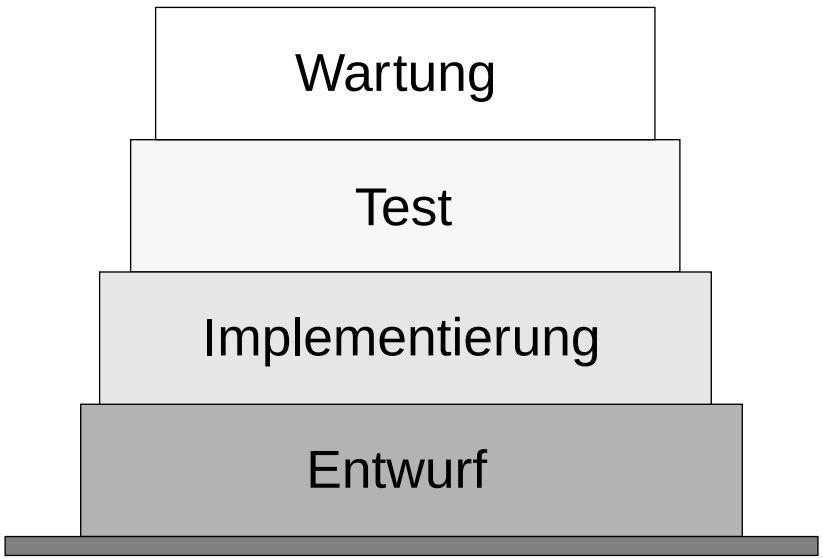
Entwurf

Einbettung in den Software-Life-Cycle

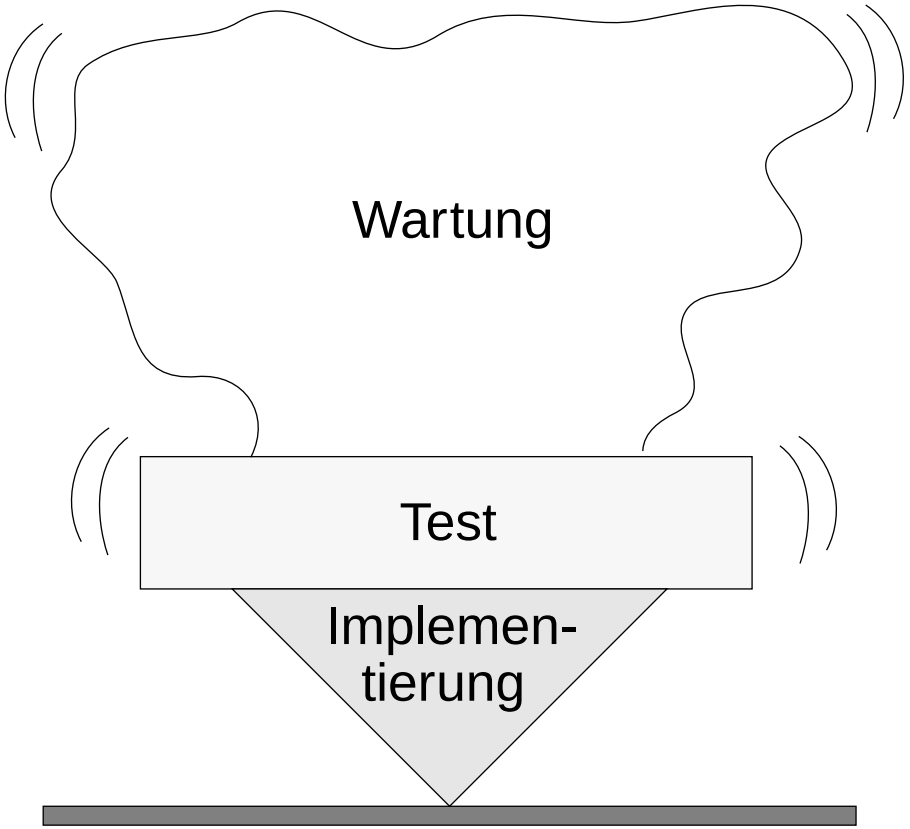


Bedeutung des Entwurfs

nach Pressman, 1992, S. 317



mit Entwurf



ohne Entwurf

Entwurf

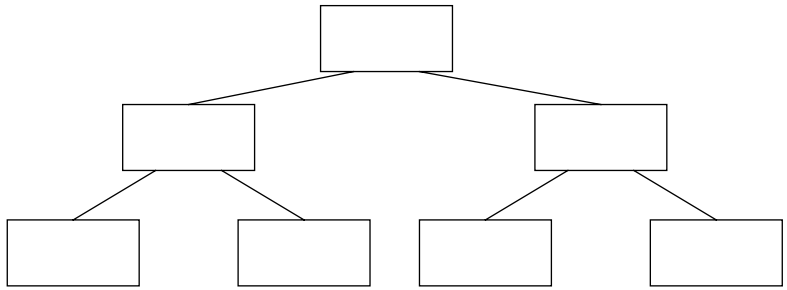
Phasen

- Grobentwurf (Architekturentwurf)
  - System hierarchisch in Subsysteme zergliedern
  - Komponenten identifizieren und den Subsystemen zuordnen
  - Beziehungen der Komponenten (Konnektoren) identifizieren
  - Zusammenspiel der Komponenten etablieren
  - Ergebnis: Architekturbeschreibung
- Feinentwurf (Komponentenentwurf)
  - Die Schnittstellen der Komponenten werden genau festgelegt
  - Aufruf- und Verwendungsbeziehungen zwischen Komponenten werden genau festgelegt
  - Ergebnis: Spezifikationen der Komponenten

Entwurf

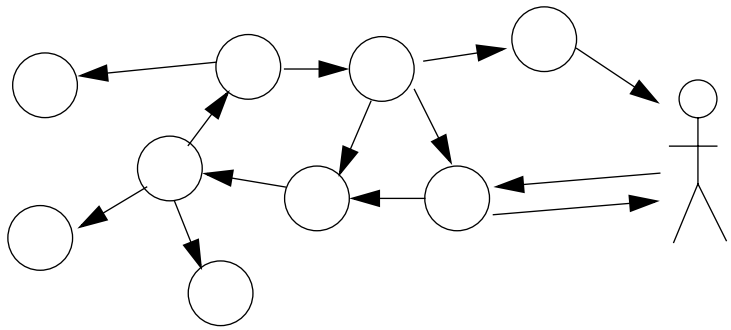
Strukturen (OOD)

- Statische Struktur
  - Klassen
  - Subsysteme
  - Vererbungsbeziehungen
  - Assoziationen (z.B. verwendet)



  - ☞ **Programm**

- Dynamische Struktur
  - Objekte
  - Zugriffe (Attribute)
  - Nachrichten (Methodenaufrufe)



  - ☞ **Ablauf des Programms**

Entwurf

Logischer vs. phsikalischer Entwurf

logischer Entwurf

Abstraktion von

- verwendeter Plattform (Rechner, Betriebssystem, Netzwerke)
- verwendeter anderer Systeme (z.B. Datenbank, Broker)

physikalischer Entwurf

Festlegen aller bisher offengelassenen Realisierungsentscheidungen auf der Grundlage von

- Vorgaben durch Kunden und Management
- Kostenaspekten, z.B.
  - Anschaffungs- und Betriebskosten
  - Einarbeitungszeit der Entwickler und Benutzer
  - Effizienzüberlegungen (Zeit und Platz)

Entwurf

Qualitätskriterien

Entwicklungssicht

- übersichtlich
- verständlich
- änderbar (flexibel)
  - wartbar
  - erweiterbar
  - restrukturierbar
  - portabel
- leicht implementierbar
- wiederverwendbar

Ausführungssicht

- interoperabel
- effizient
- verläßlich
- prüfbar (testbar)

Änderbarkeit

Warum ist Änderbarkeit so wichtig?

Lehman‘s First Law of Software Evolution

A program that is used and that as an implementation of its specification reflects some reality, undergoes continual change or becomes progressively less useful.

☞ Das System wird sich im Laufe der Zeit ändern (oder weggeworfen)

Bersoff‘s First Law of System Engineering

No matter where you are in the system life cycle, the system will change and the desire to change it will persist thoughout the life cycle.

☞ Das System wird sich auch schon während der Entwicklung ändern

Bertholt Brecht

Von allen Dingen das beständigste ist der Wandel.

Entwurf

Grundlegende Prinzipien (1)

- Abstraktion
- Teile und herrsche
- Modularisierung
- Kapselung
- Information Hiding
- Kopplung und Zusammenhalt
- ausreichend, vollständig, einfach (primitiv)

Entwurf

Grundlegende Prinzipien (2)

- Trennung der Zuständigkeiten (Separation of Concerns)
- Trennung zwischen Vorgehensweise und Implementierung
- Trennung zwischen Schnittstelle und Implementierung
- Open Closed Principle
- Liskov Substitution Principle

Abstraktion

- abstraction *(IEEE Standard 610.12-1990)*:
  A view of an object that focuses on the the information relevant to a particular purpose and ignores the remainder of the information.
- Abstraktion wird allgemein angewendet, um mit Komplexität umgehen zu können.
- Beispiele für Abstraktion
  - Kapselung
  - Generalisierung (z.B. abstrakte Oberklassen)
  - virtuelle Maschinen

Teile und herrsche

- Bewährtes Prinzip zur Beherrschung von Weltreichen
- Prinzip des Top-Down-Entwurfs
  - zerlege ein System oder eine Komponente in kleinere, relativ unabhängige Teile (Komponenten)
  - die Komponenten können separat entworfen werden
  - Abstraktion einer Komponente auf ihre Schnittstelle
  - Vorgehen erzeugt Hierarchien/Baumstrukturen von Komponenten

Modularisierung

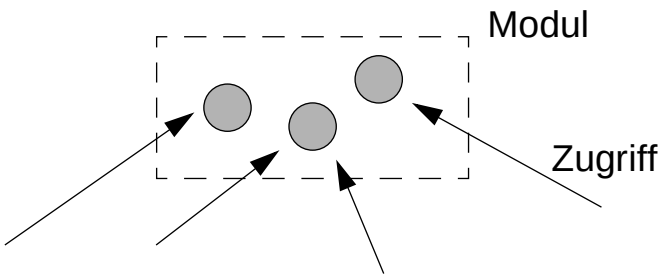
- modularization *(IEEE Standard 610.12-1990)*:
  The process of breaking a system into components to faciliate design and develoment.
- Aufteilung des Systems in sinnvolle Subsysteme und Module
- Ein Modul ist ein Behälter für Funktionen oder Zuständigkeiten des Systems

Kapselung

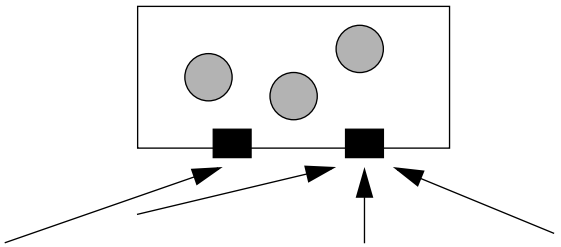
- encapsulation *(IEEE Standard 610.12-1990)*:
  A software development technique that consists of isolating a system function or a set of data and operations on those data within a module and providing precise specifications for the module.
- Zusammengehörige Bestandteile einer Abstraktion werden zu einem Ganzen zusammengefaßt und von anderen abgegrenzt

Information Hiding

- Alle Entwurfsentscheidungen werden in Modulen verborgen
- Besonders solche, die sich wahrscheinlich ändern werden
- Zugriff auf Interna des Moduls nur über wohldefinierte Schnittstellen
- Der Verwender darf nur erfahren, was er wissen muß, um ein Modul zu verwenden
- Beispiele: Datenstrukturen einer Datenkapsel, Sortieralgorithmus, Parameter der Programm-konfiguration



ohne Information Hiding:
beliebiger Zugriff



mit Information Hiding:
Zugriff über Schnittstellen

Kopplung und Zusammenhalt

Übersicht

(hier nach Ludewig, J.: vorläufiges Skript zur Vorlesung Software Engineering)

Kopplung

Maß für die Stärke der Verbindung (und damit Abhängigkeit) von Programmkomponenten untereinander
Ziel: möglichst niedrige Kopplung (von Komponenten)

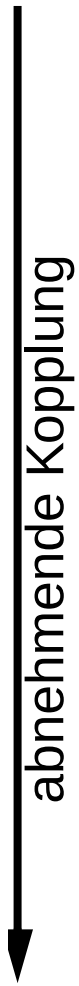
Zusammenhalt

Maß für die Stärke der Zusammengehörigkeit von Bestandteilen einer Programmkomponente
Ziel: möglichst hoher Zusammenhalt (von Teilen einer Komponente)

Kopplung und Zusammenhalt

Stufen der Kopplung

1. Einbruch: Modifikation des Codes einer anderen Komponente (vor allem in Assembler möglich)
2. volle Öffnung: Zugriff auf alle Daten, z. B. Datenhaltung in globalen Variablen
3. Fremdsteuerung: Aufrufer teilt über Steuerparameter mit, wie sich Aufgerufener zu verhalten hat.
4. selektive Öffnung: bestimmte Daten sind zugänglich, z. B. `Done` in `InOut` (Modula-2)
5. Parameter: Programmteile sind als Prozeduren formuliert, die nur über Parameter miteinander kommunizieren
6. Funktionen: nur Wertparameter und Funktionsresultate
7. keine Kopplung: keine Verbindung zwischen unabhängigen Programmteilen



Kopplung und Zusammenhalt

Stufen des Zusammenhalts

1. kein Zusammenhalt: rein zufällige Zusammenstellung (z.B. je 100 Zeilen des Programms)
2. Ähnlichkeit: ähnlicher Zweck, z. B. Fehlerbehandlungsprozedurensammlung
3. zeitlich: Ausführung zur gleichen Zeit, z. B. Initialisierungsanweisungen
4. arbeitet mit denselben Daten: z. B. Datumsoperationen eines Kalenderpakets
5. Kommunikation über Zwischenergebnisse: der eine Teil verwendet, was der andere erzeugt (typisch für funktionalen Entwurf).
6. einziger Zweck: z.B. Iteratoroperationen einer Liste
7. einziges Datum: ADT oder Datenkapsel

Ausreichend, Vollständig, Einfach

- Jede Komponente des Systems soll hinreichend, vollständig und einfach sein.
- **ausreichend**: Komponente umfaßt alle Eigenschaften einer Abstraktion, die für eine sinnvolle und effizienten Verwendung der Komponente notwendig sind
  (minimale Schnittstelle, macht Verwendung sinnvoll)
- **vollständig**: Komponente umfaßt alle relevanten Eigenschaften einer Abstraktion
  (generelle Schnittstelle, erlaubt Wiederverwendung)
- **einfach** (primitiv): Alle Operationen einer Komponenten können leicht implementiert werden,
  d.h. komplexe Operationen werden auf die Kombination einfacherer Operationen zurückgeführt

Trennung der Zuständigkeiten

(Separation of Concerns)

- Die Aufteilung des Systems sollte anhand von Zuständigkeiten vorgenommen werden
- Komponenten, die an der gleichen Aufgabe beteiligt sind, werden gruppiert, und von denen abgegrenzt, die für andere Aufgaben zuständig sind -> Subsysteme
- Bei Komponenten, die für mehrere Aufgaben zuständig sind, werden die Bestandteile intern nach Zuständigkeiten gruppiert

Trennung zwischen Vorgehensweise und Implementierung

- Eine Komponente soll entweder für die Vorgehensweise oder die Implementierung zuständig sein, nicht für beides
- Komponente für Vorgehensweise
  - trifft Entscheidungen anhand des Kontextes
  - interpretiert Ergebnisse
  - koordiniert andere Komponenten
  - ändert sich wahrscheinlich häufig
- Komponente für Implementierung
  - führt einen Algorithmus auf vorliegenden Daten aus
  - benötigte Informationen werden beim Aufruf mitgegeben
  - benötigt zur Ausführung keine weiteren Informationen
  - ist damit einfacher wiederzuverwenden

Trennung zwischen Schnittstelle und Implementierung

- Eine Komponente soll aus Schnittstelle und Implementierung bestehen
- Schnittstelle:
  - definiert die Funktionalität und die Verwendung
  - ist den Verwendern zugänglich
- Implementierung:
  - tatsächlicher Code für die Funktionalität der Komponente
  - weitere, nur intern benötigte Funktionalität
- Trennung schützt die Verwender vor Implementierungsdetails
- Die Implementierung kann geändert oder ausgetauscht werden, ohne daß der Verwender davon betroffen ist

Open Closed Principle

Bertrand Meyer „Object-oriented Software Construction“:

> Modules should be both open and closed.

geschlossen: das Modul kann gefahrlos verwendet werden, da sich seine Schnittstelle nicht mehr ändert

offen: das Modul kann problemlos erweitert werden

- Die beiden Forderungen sind im traditionellen imperativen Programmierparadigma unvereinbar
- Im objektorientierten Paradigma kann es mit Hilfe der Vererbung gelöst werden:
  - die Schnittstelle wird in eine abstrakte Klasse A umgewandelt
  - je nach Bedarf können Implementierungen in Form von Subklassen von A erzeugt werden. Die Subklassen können auch die Funktionalität von A erweitern

Liskov Substitution Principle

Barbara Liskov „Data abstraction and hierarchy“, 1988 (vereinfacht):

> Falls ein Verwender erwartet, mit einem Objekt der Klasse T zusammenzuarbeiten, dann wird er ebenso gut mit einem Objekt der Klasse S zusammenarbeiten, wenn S eine Subklasse von T ist.

- Forderung an alle Subklassen S von T, für die durch T gegebene Schnittstelle dieselbe Semantik anzubieten
- Erweiterungen von T sind problemlos möglich
- Einschränkungen von T können Schwierigkeiten bereiten

```
class Document { ... print(); ... }
class Form extends Document { ... print(); ... }
Document d = new Document(); d.print();
Document d = new Form(); d.print();
```

Entwurfsprinizipien nach Davis (1)

aus „201 Principles of Software Development“

- Transition from requirements to design is not easy
- Evaluate alternatives
- Design without documentation is *not* design
- Encapsulate
- Don‘t reinvent the wheel
- Keep it simple
- Use coupling and cohesion

Entwurfsprinizipien nach Davis (2)

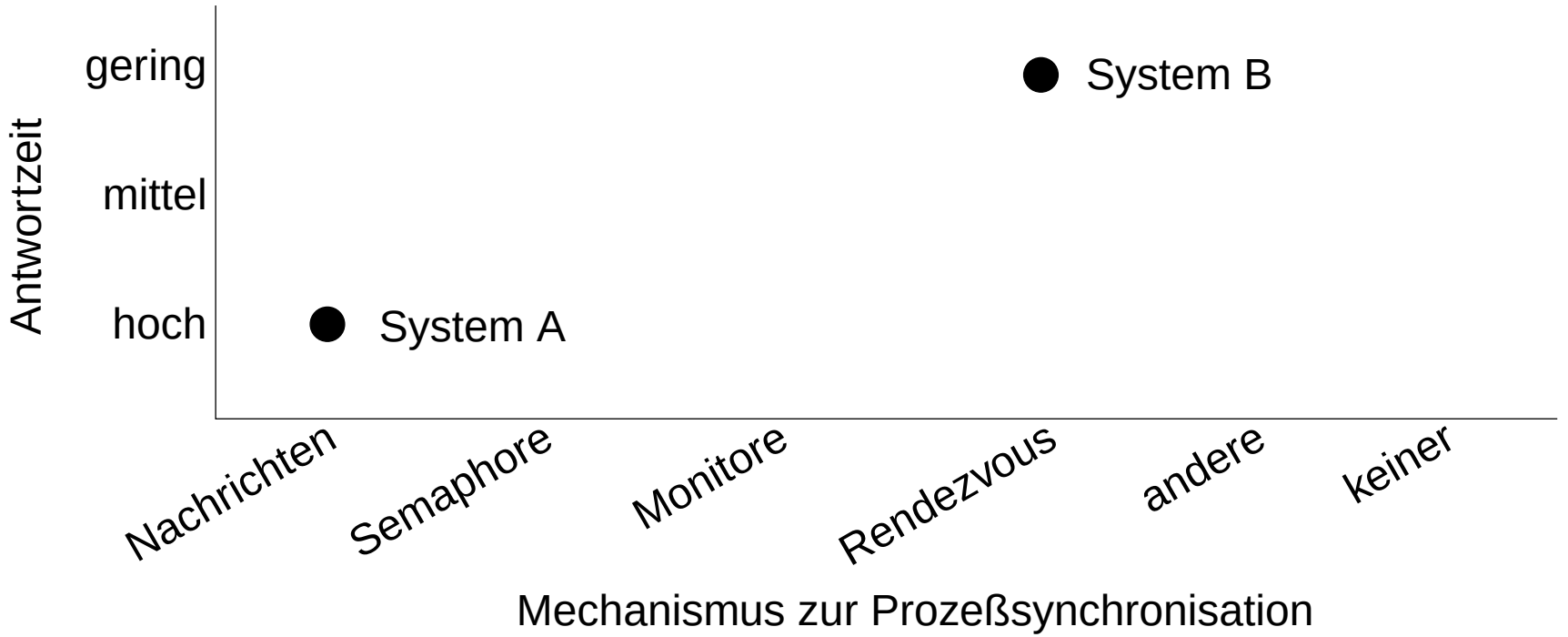
aus „201 Principles of Software Development“

- Design for change
- Design for maintenance
- Build generality into software
- Build flexibility into software
- Module specifications provide all the information the user needs and nothing more
- Design is multidimensional
- Great designs come from great designers

Entwurfsraum

Begriff

- Klassifiziert mögliche Systemarchitekturen anhand eines multidimensionalen Skalensystems (spannt einen Raum auf)
- Erleichtert das Auffinden von Alternativen und die Auswahl einer Alternative



Entwurfsraum

Verwendung

- Die gewählten Skalen reflektieren
  - Anforderungen
  - Bewertungskriterien (hinsichtlich Funktion oder Leistung)
  - Struktur (oder andere Entwurfsentscheidungen)
- Unterteilung z.B. in funktionalen Entwurfsraum und strukturellen Entwurfsraum möglich
- Durch Untersuchung von Korrelationen zwischen Kriterien und Vergabe von Nutzwerten für Kriterien ist auch eine Quantifizierung möglich (Nutzwertanalyse)

Kapitel 3:
Muster

- Begriff und Bedeutung
- Geschichte der (Entwurfs-)Muster
- Bestandteile und Beschreibung von Mustern
- Beispiel eines Entwurfsmusters: Observer Pattern

Wiederverwendung von Wissen in der Softwareentwicklung

- Komponenten (Bausteine)
- Bibliotheken (Komponentensammlungen)
- Rahmenwerke (wiederverwendbare Entwürfe)
- Programmgeneratoren
- Muster (Problemlösungen)
- Anti-Muster (Problemlösungen, die nicht funktionieren)
- Prinzipien und Heuristiken (Richtlinien, Daumenregeln)
- Handbücher (Anleitungen)

Muster

allgemeiner Begriff

„Pattern“ im Websters Dictionary

1: a form or model proposed for imitation, see exemplar

2: something designed or used as a model for making things; e.g. a dressmaker‘s pattern

3: a model for making a mold into which molten metal is poured to form a casting

4: an artistic, musical, literary, or mechanical design or form

5: a natural or chance configuration; e.g. frost pattern

„Muster“ im Brockhaus, 1993

1) Vorlage, nach der etwas hergestellt wird

2) etwas ins seiner Art Vollkommenes, Vorbild

Muster

Theorie des Lernens

- Ein Mensch lernt, indem
  - er Beispiele beobachtet bzw. Erfahrungen macht,
  - die Beobachtungen generalisiert, abstrahiert und
  - daraus allgemeine Regeln (Muster) ableitet.
  - Diese werden dann erprobt und wenn nötig modifiziert
- Das gilt auch für Problemlösungen:
  - erprobte Lösungen werden immer wieder eingesetzt, (allerdings oft auch dann, wenn sie unangebracht sind)
  - damit reduziert sich der Aufwand der Problemlösung
- Ein großer Teil der Kultur beruht auf erprobten Lösungen, z.B. Sprache, Schrift, Technik. Die praktische Ausbildung eines Menschen umfaßt daher vor allem die Weitergabe erprobter Lösungen.

Sammlungen von Mustern

Katalog vs. Mustersprache

Katalog

eine beliebige Sammlung von Mustern, die in der Regel nach demselben Schema beschrieben sind

Mustersprache (pattern language)

- ein System von Mustern,
  - die sich mit demselben Gebiet befassen
  - die sich gegenseitig ergänzen
  - deren Beziehungen und Kombinationsmöglichkeiten explizit aufgezeigt werden
- es können durch Kombination der Muster größere Probleme gelöst werden, als das mit einzelnen Mustern möglich ist
- auch zur Herleitung eines Entwurfs verwendbar

Muster in der Softwareentwicklung

Geschichte (1)

Muster in der Architektur: Christopher Alexander

- In seinem Buch „A Pattern Language“ (1977) beschreibt Alexander ein System von Mustern zur Architektur auf den Ebenen
  - Landschaftsplanung
  - Stadtplanung
  - einzelne Gebäude
  - einzelne Räume
- „Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.“ (Alexander, 1977, S. x)
- Definition Muster: **Lösung** eines **Problems** in einem **Kontext**

Muster in der Softwareentwicklung

Geschichte (2)

Muster in der Softwareentwicklung

- Kent und Cunningham bezogen von Alexander ihre Inspiration für Entwurfsmuster in der Software-Entwicklung (etwa 1987)
- Ab etwa 1990 gibt es Workshops über Software-Architektur, bei denen auch über Entwurfsmuster diskutiert wird.
- Gamma, Vlissides, Johnson und Helm veröffentlichen 1995 das Buch „Design Patterns“ und machen damit Entwurfsmuster populär.
- Seither steigt die Anzahl der Publikationen exponentiell.
- Seit 1994 gibt es die Konferenz PLoP (Pattern Languages of Program Design); später kamen die Konferenzen EuroPLoP (1996) und ChiliPLoP (1997) dazu.

Muster in der Softwareentwicklung

Begriff „Muster“ (Riehle, 1997)

- Ein Muster ist eine in einem bestimmten Kontext erkennbare Form.
- Es dient als Vorlage zum Erkennen, Vergleichen und Erzeugen von Musterexemplaren.
- Ein Muster ist die Essenz aus Erfahrung und Analyse immerwiederkehrender Situationen.
- Es besitzt eine innere Struktur und Dynamik.

Muster in der Softwareentwicklung

Arten

- Verbreitet:
  - Analysemuster
  - Architekturmuster
  - Entwurfsmuster
  - Idiome
- Noch eher selten:
  - Prozeßmuster, Vorgehensmuster (z.B. für Test, Review)
  - Organisationsmuster
- Neue Idee:

  Antipatterns: beschreiben häufig gewählte *falsche* Lösungen für bestimmte Probleme und machen alternativen Lösungsvorschlag

Muster in der Softwareentwicklung

Begriff „Design Pattern“ (Gamma et al., 1995)

A description of an object-oriented design technique which **names**, **abstracts** and **identifies** aspects of a **design structure** that are **useful for creating an object-oriented design**.

The design pattern identifies **classes** and **instances**, their **roles**, **collaborations** and **responsibilities**. Each design pattern focuses on a particular object-oriented design **problem** or issue. It describes **when** it applies, whether it can be applied in the presence of other design constraints, and the **consequences** and **trade-offs** of its use.

Muster in der Softwareentwicklung

Eigenschaften von Entwurfsmustern

- Ein Entwurfsmuster dokumentiert vorhandene, erprobte Entwurfserfahrung; diese kann dadurch wiederverwendet werden
- Die durch Entwurfsmuster beschriebenen Abstraktionen liegen über der Stufe einzelner Klassen (Mikroarchitektur)
- Entwurfsmuster können flexibel kombiniert werden
- Entwurfsmuster helfen bei der Erstellung und der Dokumentation komplexer und heterogener Software-Architekturen
- Entwurfsmuster unterstützen Entwurfseigenschaften wie Änderbarkeit und Wiederverwendbarkeit
- Entwurfsmuster bilden ein Vokabular für den Entwurf und erleichtern daher die Kommunikation
- Entwurfsmuster können auch beim Reengineering vorhandener Software als Analysemittel dienen

Muster in der Softwareentwicklung

Bestandteile einer Beschreibung

- **Name**
- **Problem** (problem)
- **Kontext** (context): Bereich, in der das Problem auftritt
- **Einflußfaktoren** (forces), die berücksichtigt werden müssen
- **Lösung** (solution): Struktur und Dynamik
- Einschränkungen (constraints)
- Implementierung (implementation)
- Beispiele, Varianten, bekannte Verwendungen, verwandte Muster, ...

Leider gibt es bisher kein einheitliches Format zur Musterbeschreibung

Entwurfsmuster

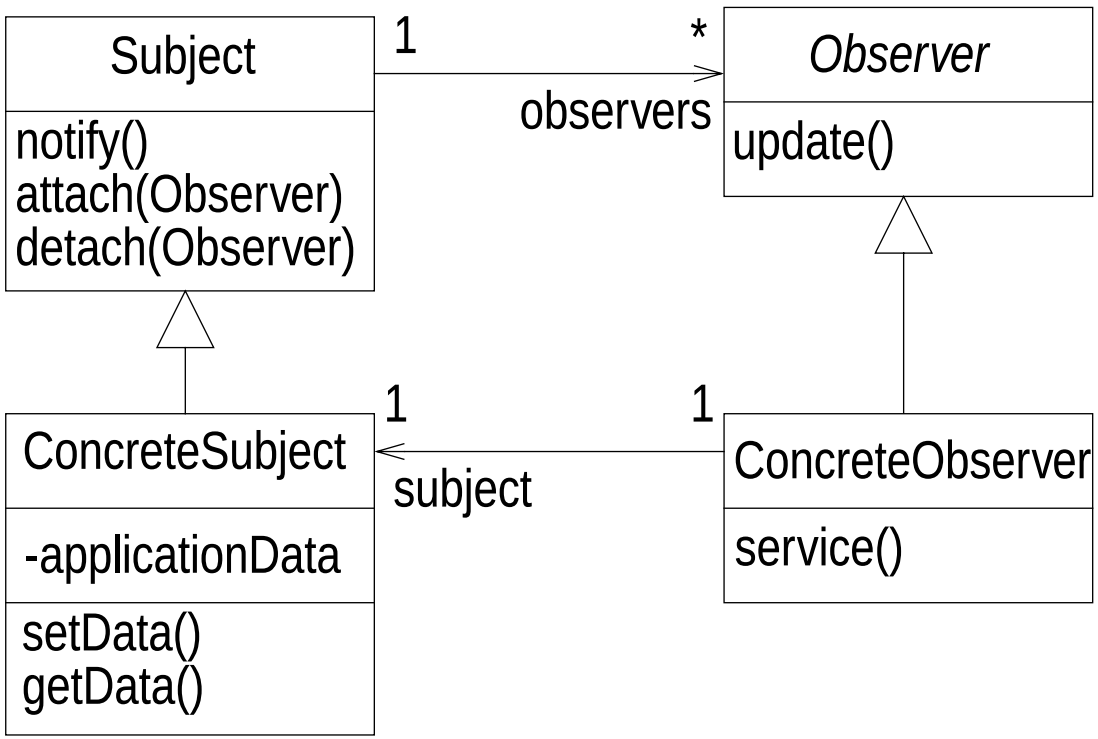
Beispiel Observer Pattern (1)

- **Problem**: Die Konsistenz zwischen Objekten soll sichergestellt werden, z.B. zwischen Daten (Modell) und ihren Darstellungen (Views)
- **Kontext**: Der Zustand eines oder mehrerer Objekte (Beobachter) ist vom Zustand eines anderen Objekts (Subjekt) abhängig. Es ist wichtig, jederzeit die Konsistenz der Objekte zu gewährleisten: alle Änderungen des Subjekts sollen dem Beobachter bekannt werden. Ein Subjekt kann mehrere Beobachter haben
- **Einflußfaktoren**:
  - Die Objekte sollen trotz der Forderung nach Erhaltung der Konsistenz weitgehend entkoppelt bleiben, d.h. sie sollen möglichst wenig voneinander wissen
  - Die Beobachter eines Subjekts sind a priori nicht bekannt; ihre Anzahl kann sich dynamisch ändern.

Entwurfsmuster

Beispiel Observer Pattern (2)

- **Lösung**: Änderungen des Subjekts werden an die Beobachter propagiert. Beobachter registrieren sich dazu beim Subjekt.



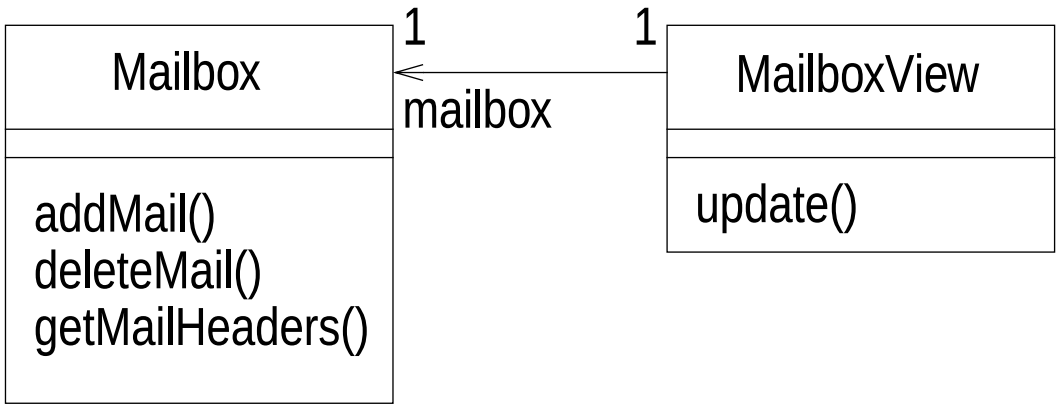
Entwurfsmuster

Beispiel Observer Pattern (3)

- **Implementierung**:

  Die Bibliothek `java.util` bietet eine Klasse `Observable` als Superklasse für Subject an. Beobachterklassen müssen das Interface `Observer` implementieren (`update`-Methode).

  Beispielimplementierung: rudimentäre Mailbox mit zugehöriger GUI



  Der Code ist auf der OOSE-Seite im WWW verfügbar.

Entwurfsmuster

Beispiel Observer Pattern (4)

```
class Mailbox extends Observable {
private Vector mails;
public Mailbox() { ... }
public void addMail(String header) {
  if( (header != null ) && !mails.contains(header) ) {
    mails.addElement(header);
    this.setChanged(); // Zustand hat sich geaendert
    this.notifyObservers(); // Beobachter benachr.
  }
}
public void deleteMail(String header) { ... }
public Enumeration getMailHeaders() { ... }
}
```

Entwurfsmuster

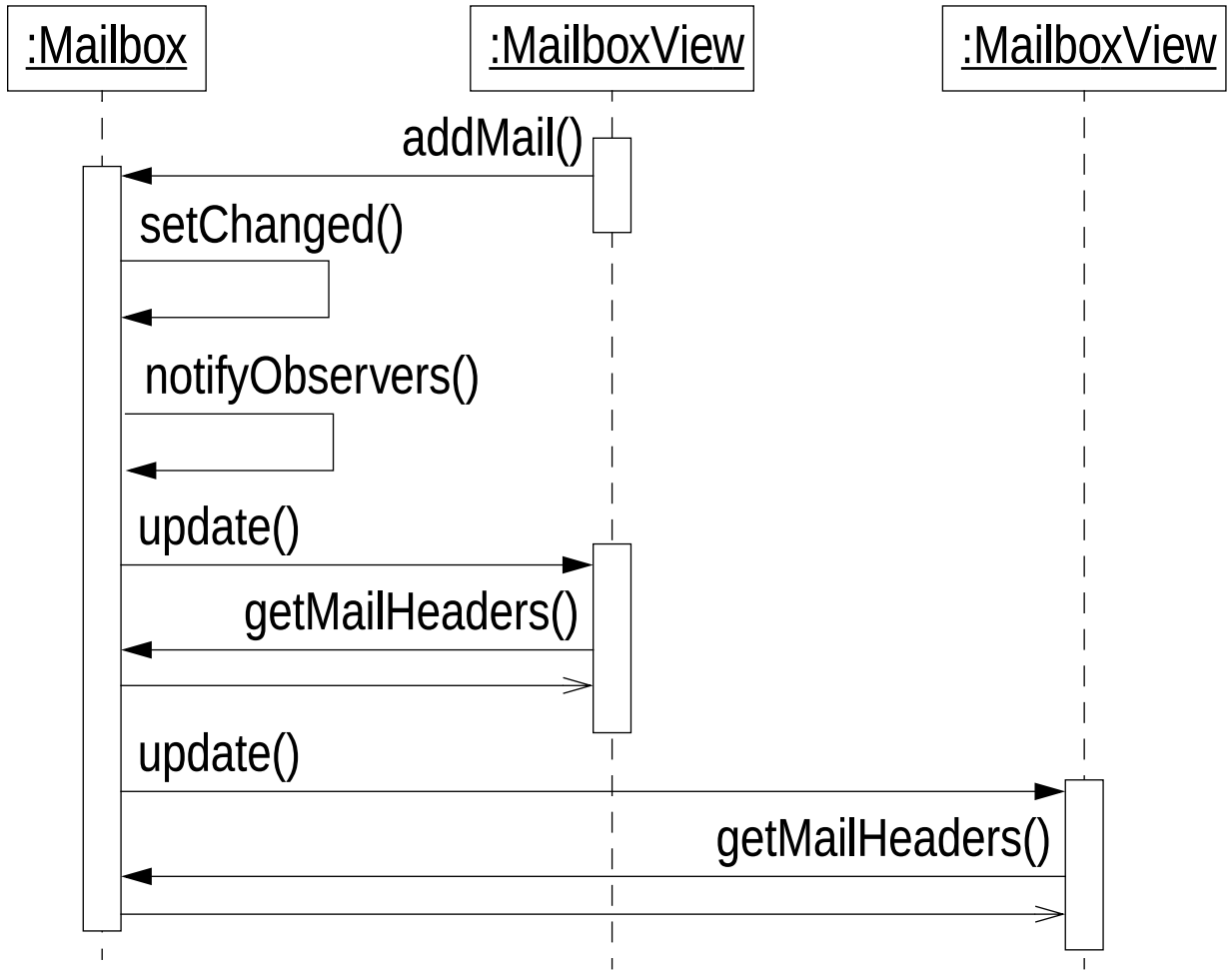
Beispiel Observer Pattern (5)

```
class MailboxView implements Observer {
private Mailbox mailbox;
public MailboxView(Mailbox box) {
   mailbox  = box;
   mailbox.addObserver(this); // als Beobachter reg.
   ...
}
public void update(Observable o, Object arg) {
   this.fillList(((Mailbox) o).getMailHeaders());
}
}
```

Entwurfsmuster

Beispiel Observer Pattern (6)

Sequenzdiagramm bei zwei Views



Entwurfsmuster

Beispiel Observer Pattern (7)

- **Varianten**:
  - Ansammeln von Notifikationen
  - Getypte Notifikationen (Events) wie in Smalltalk
  - Push- bzw.Pull-Modell
  - ...
- **Bekannte Verwendungen**: zuerst verwendet in der Umsetzung des Model-View-Controller-Konzepts in Smalltalk
- auch bekannt als Publisher/Subscriber (Coad, 1997)

Entwurfsmuster

Beispiel Observer Pattern (8)

- **Beispiel**: Event-Behandlung im AWT von Java 1.1
  - Benutzeraktionen (Tastatur, Maus) lösen Events aus
  - Komponenten (z.B. Button) schicken Events an alle ihre Beobachter (sogenannte Listener)
  - Listener registrieren sich dazu bei den Komponenten
  - Je nach Komponente muß der Listener ein bestimmtes Interface implementieren
  - Je nach Art des Events wird eine Methode des Interfaces aufgerufen und der Event als Parameter mitgegeben

Entwurfsmuster

Beispiel Observer Pattern (9)

```
// Applet mit Button "Click me", piepst beim Klicken
public class Beeper extends Applet
                    implements ActionListener {
   Button button;
   public void init() {
      setLayout(new BorderLayout());
      button = new Button("Click Me");
      add("Center", button);
      button.addActionListener(this);    // registrieren
   }
   public void actionPerformed(ActionEvent e) {
      Toolkit.getDefaultToolkit().beep();
   }
}
```

Erfahrungen mit Mustern

- Muster beruhen auf Erfahrung, sie sind häufig nur mit entsprechendem Hintergrundwissen und eigener Erfahrung verständlich und nachvollziehbar.
- Es ist schwierig festzustellen, ob ein Muster anwendbar ist und angewendet werden sollte. Die Problembeschreibung (Kontext, Einflußfaktoren) ist hierfür sehr wichtig.
- Es gibt noch keine Untersuchungen, inwieweit die Anwendung von Mustern die Produktivität und die Qualität in der Software-entwicklung beeinflußt.

Kapitel 4: Architekturmuster

- Architekturbegriff in der Software-Entwicklung
- Architekturstile nach Shaw/Garlan
- Architekturmuster nach Buschmann et al.
- Pragmatischer Ansatz: „Big Ball of Mud“

Software-Architektur

Begriffe

- **Architektur** (architecture) *(IEEE Standard 610.12-1990)*:
  The organizational structure of a system or component.
  - beschreibt den Aufbau der Realisierung eines Systems
  - besteht aus Komponenten und Konnektoren
- **Komponente**: abgeschlossener Teil einer Architektur, der zur Berechnung und/oder Datenhaltung dient. Besitzt eine definierte Schnittstelle nach außen. Z.B. Subsystem, Modul
- **Konnektor**: Beziehung/Verbindung zwischen Komponenten

- **Meta-Architektur**: eine Architekturphilosophie
- **Makro-Architektur**: ein Architekturmuster
- **Mikro-Architektur**: ein Entwurfsmuster

Architekturstile

Begriff

- architectural style *(Shaw, Garlan, 1996)*:
  defines a family of systems in terms of a **pattern of structural organization**.
  defines a vocabulary of **component** and **connector types**, and a set of **constraints** on how they can be combined.
- Shaw und Garlan unterscheiden
  - Architekturmuster (z.B. Client-Server, Model-View-Controller), die allgemeiner Natur sind, und
  - Referenzmodelle (z.B. ISO-OSI 7-Schichtenmodell), die in der Regel für bestimmte Anwendungsfelder gedacht sind
- Architekturstile können als Grundlage für konkrete Architekturen verwendet werden.

Architekturstile

Architekturentwurf mit Architekturstilen

- die Wahl eines Architekturstils legt den Entwerfer auf eine bestimmte Sichtweise fest
- Architekturstile treten häufig kombiniert auf, z.B. auf verschiedenen Ebenen.
- ein Architekturstil (oder eine Kombination davon) liefert noch keine vollständige Architektur, sondern nur ein Architekturgerüst

Architekturstile

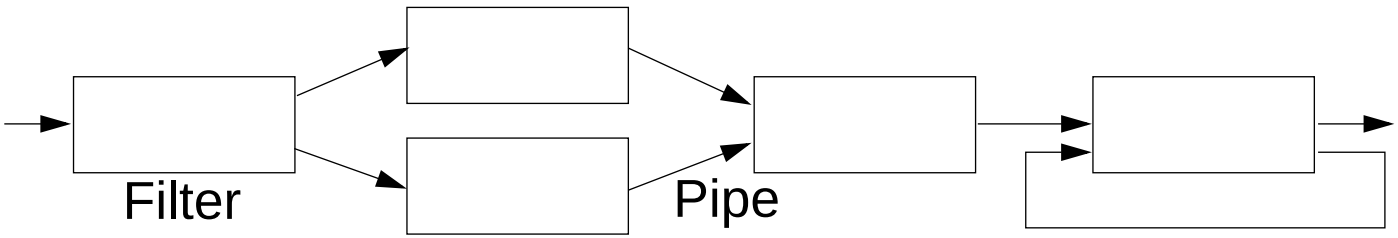
Übersicht häufig verwendeter Stile

- Datenflußsysteme
  - sequentielle Batch-Dateien
  - **Pipes und Filter**
- Aufruf-und-Rückkehr-Systeme
  - Hauptprogramm und Unterprogramme
  - **objektorientiertes System**
  - **hierarchische Schichten**
- Unabhängige Komponenten
  - kommunizier. Prozesse
  - verteilte Systeme, z.B. Client/Server
  - **ereignisgesteuertes System**
- Zustandsübergangssysteme
- Virtuelle Maschinen
  - **Interpreter**
  - regelbasiertes System
- Datenzentriertes System (Repository)
  - Datenbank
  - Hypertext-System
  - **Schwarzes Brett** (Blackboard)

Pipes und Filter

Allgemeines

- Filter (Komponenten):
  - haben Menge von Eingaben und Ausgaben
  - lesen Eingabe aus/schreiben Ausgaben in Streams
  - realisieren eine Transformation von Eingaben in Ausgaben
  - arbeiten autonom, d.h. von anderen Filtern unabhängig
- Pipes (Konnektoren):
  - „Rohrleitungen“ für die Streams
  - verbinden Ausgabe von Filtern mit der Eingabe von Filtern



Pipes und Filter

Beispiele

- Unix-Shell: Unix-Werkzeuge werden mit Hilfe von Pipes verkettet, um gemeinschaftlich eine Aufgabe zu lösen. Über die Pipes fließen Byte-Datenströme.

  ```
  cat File.java | grep '^ *//' | wc -l
  ```

- Traditionelle Compilerarchitektur: Scanner, Parser, Semantische Analyse, Code-Generierung.
- Spezialfälle:
  - Pipelines: lineare Verkettung, d.h. keine Zyklen
  - Bounded Pipes: Pipes mit begrenzter Speicherkapazität (z.B. Puffer)
  - Getypte Pipes: Die fließenden Daten müssen von einem bestimmten Typ sein (z.B. unter Unix)

Pipes und Filter

Vor- und Nachteile

▲ Ein-/Ausgabeverhalten des Systems kann aus dem Zusammenspiel der Filter heraus verstanden werden
-> Klassische funktionale Dekomposition eines Systems

▲ Wiederverwendung von Filtern ist einfach

▲ Leicht zu modifizieren oder zu erweitern: Filter austauschen oder zusätzliche einfügen

▲ Filter können parallel ausgeführt werden

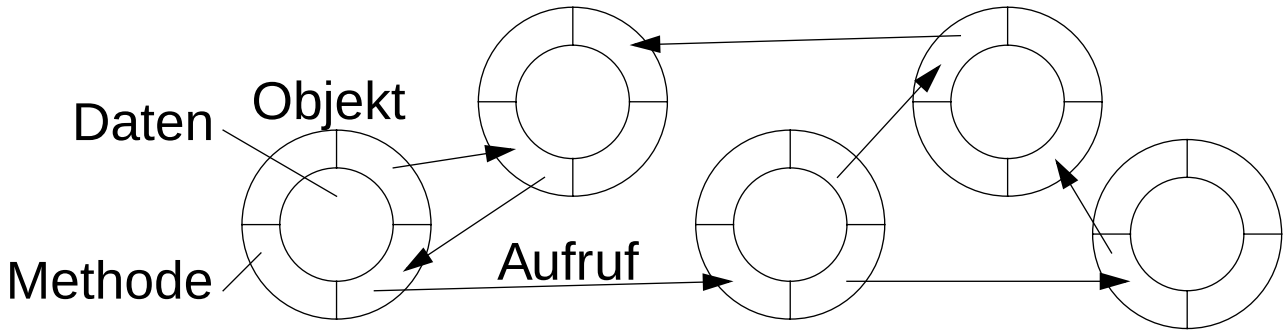
▼ Für interaktive Programme eher ungeeignet

▼ Primitives Datenformat für Pipes (wie z.B. Bytes unter Unix) erzwingt dauernde Umwandlungen von Datenformaten
-> Performance-Verlust, höhere Komplexität der Filter

Objekt-orientiertes System

Allgemeines

- Objekte (Komponenten):
  - Exemplare einer Klasse oder eines abstrakten Datentyps
  - Kapseln Daten und zugehörige Operationen (Methoden)
  - Repräsentation der Daten ist geheim
  - Konsistenz wird vom Objekt selbst sichergestellt
- Methodenaufrufe (Konnektoren):
  - Parameter
  - ggf. Rückgabewert



Objekt-orientiertes System

Vor- und Nachteile

▲ interne Repräsentation und Implementierung eines Objekts kann geändert werden, ohne daß Verwender davon betroffen wäre

▲ „natürliche“ Zerlegung eines Problems in Sammlung interagierender Agenten möglich

▼ Identität eines Objekts muß bekannt sein, um seine Methoden aufrufen zu können -> hohe Vernetzung der Objekte untereinander

▼ Das Zusammenspiel der Objekte zur Laufzeit ist anhand des Codes schwierig zu durchschauen, daher lange Einarbeitungszeiten nötig

Hierarchische Schichten

Allgemeines

- Schichten (Komponenten):
  - benutzen Dienste der direkt untergeordneten Schicht
  - bieten Dienste für die direkt übergeordnete Schicht an
- Konnektoren:
  - Protokolle der einzelnen Schichten
- Varianten:
  - es darf auf alle tiefer liegenden Schichten direkt zugegriffen werden. Erhöht die Effizienz, aber auch die Kopplung

Hierarchische Schichten

Beispiel ISO-OSI-7-Schichten-Modell

Application	Protokolle der Anwendungen
Presentation	Struktur und Semantik von Daten
Session	Dialogsteuerung und Synchronisation
Transport	Zerlegung in Pakete, Sicherstellen der Zustellung
Network	Routing
Data Link	Fehlererkennung und -korrektur
Physical	Bitübertragung

Hierarchische Schichten

Beispiel Betriebssystem

- Das System wird realisiert als aufeinander aufbauende virtuelle Maschinen
- Jede virtuelle Maschine bietet Dienste an und verwendet die Dienste untergeordneter virtueller Maschinen
- Hardware simuliert ebenfalls eine virtuelle Maschine!
- Betriebssystem wird portabler, wenn die Hardware über spezielle Abstraktionsschicht angesprochen wird

Hierarchische Schichten

Beispiel Informationssystem

Typische Schichtenarchitektur für Informationssysteme

Interaktionsschicht	Benutzungsoberfläche
Werkzeugschicht	Applikationsfunktionalität
Materialschicht	Applikationsdatenhaltung
Datenhaltungsschicht	Persistente Datenhaltung (DB, ...)

Hierarchische Schichten

Vor- und Nachteile

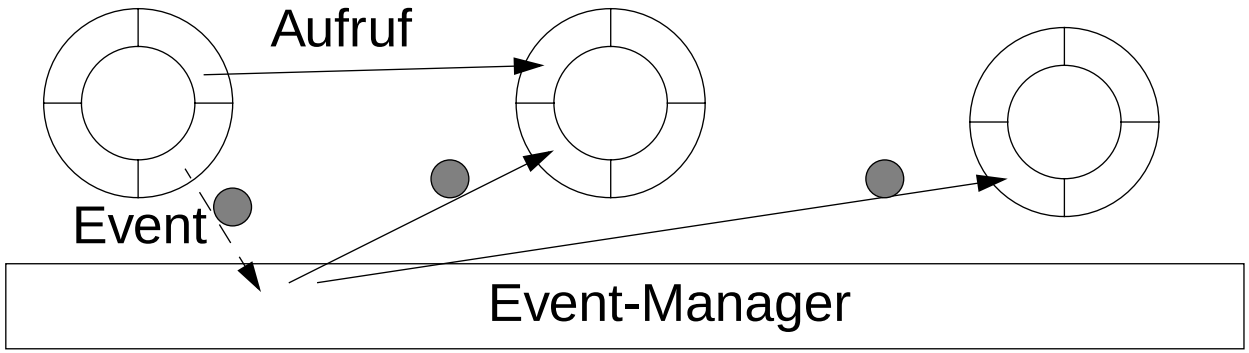
- ▲ System kann aus Schichten mit zunehmender Abstraktion aufgebaut werden
- ▲ Änderungen an einer Schicht betreffen höchstens die beiden benachbarten Schichten
- ▲ Implementierung einer Schicht kann ausgetauscht werden, solange das Protokoll beibehalten wird
- ▲ Auf eine Schicht können verschiedene andere Schichten aufgesetzt werden (z.B. FTP, Telnet, HTTP auf TCP/IP), d.h. die Schicht und darunter liegende werden wiederverwendet

- ▼ Identifikation und Abgrenzen von Schichten ist schwierig
- ▼ Performance-Verluste durch Durchreichen von Information durch verschiedene Schichten

Ereignisgesteuertes System

Allgemeines

- impliziter Aufruf von Operationen durch Verschicken von Events
- Module (Komponenten):
  - besitzt Operationen
  - registriert Operationen, die bei einem bestimmten Event aufgerufen werden sollen, beim System
- Konnektoren:
  - Aufrufe von Operationen
  - Versenden von Events



Ereignisgesteuertes System

Beispiele

- Kopplung einer Benutzungsoberfläche an den funktionalen Kern eines Systems, z.B.
  - Desktops für Betriebssysteme
  - im AWT von Java
- Kopplung verschiedener Werkzeuge, z.B.
  - Debugger verschickt Breakpoint-Events, die von Editoren, Variablenmonitoren, etc. verarbeitet werden
- Datenbankmanagementsysteme (zur Sicherstellung von Konsistenzbedingungen)

Ereignisgesteuertes System

Vor- und Nachteile

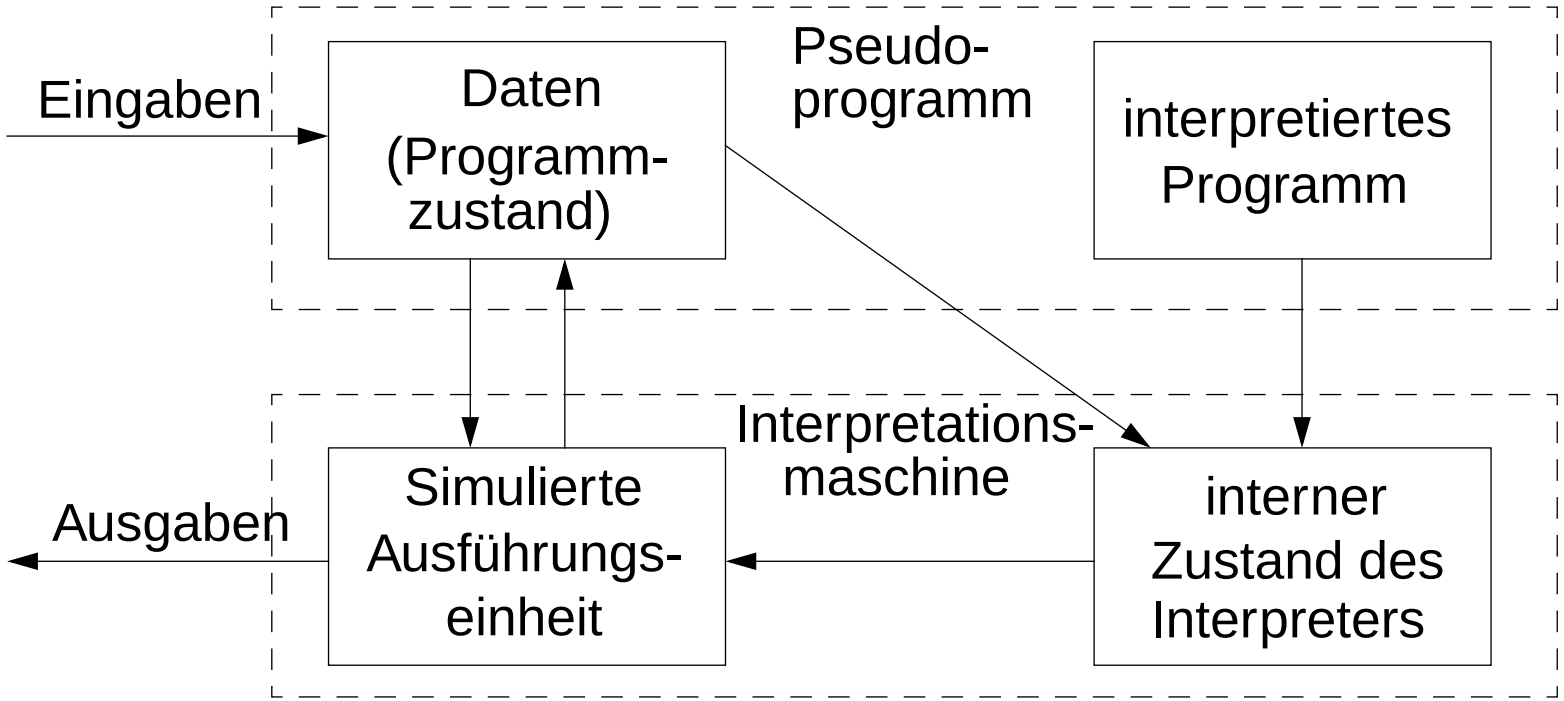
- ▲ Starke Entkopplung der Komponenten
- ▲ Es können jederzeit neue Komponenten ins System eingebracht werden und alte ersetzt werden
- ▲ Die Wiederverwendung von Komponenten wird vereinfacht

- ▼ Es ist nicht garantiert, daß überhaupt eine Komponente auf einen Event reagiert
- ▼ Die Reihenfolge der Ausführung registrierter Operationen bei einem Event ist nichtdeterministisch; daher wird außerdem expliziter Aufruf angeboten
- ▼ Ereignisgesteuerte Systeme sind schwierig zu prüfen (Test, Verifikation), da die Semantik eines Events vom (dynamischen) Zustand des Event-Managers abhängt.

Interpreter

Allgemeines

- in Software realisierte virtuelle Maschine



- Kopplung über schreibende/lesende Datenzugriffe (Konnektoren)

Interpreter

Beispiel Java Virtual Machine (JVM)

- Stackmaschine (Null-Adress-Maschine) für Java Bytecode
  - entworfen für optimale Plattformunabhängigkeit
  - kein Ausnutzen von Registern oder anderer Eigenheiten der Ausführungsplattform, daher Performance-Einbußen gegenüber Maschinencode
  - Assembler-Sprache mit OO-Unterstützung
- Java-Compiler generiert plattformunabhängigen Bytecode für die JVM
- Interpreter auf der Zielplattform führt Bytecode aus

andere Anwendungen von Interpreter:

- regelbasierte Systeme, LISP, Prolog, ...

Interpreter

Vor- und Nachteile

▲ flexible Möglichkeit, die Lücke zwischen der vom Programm erwarteten Ausführungsmaschine und der durch die Hardware realisierten zu schließen

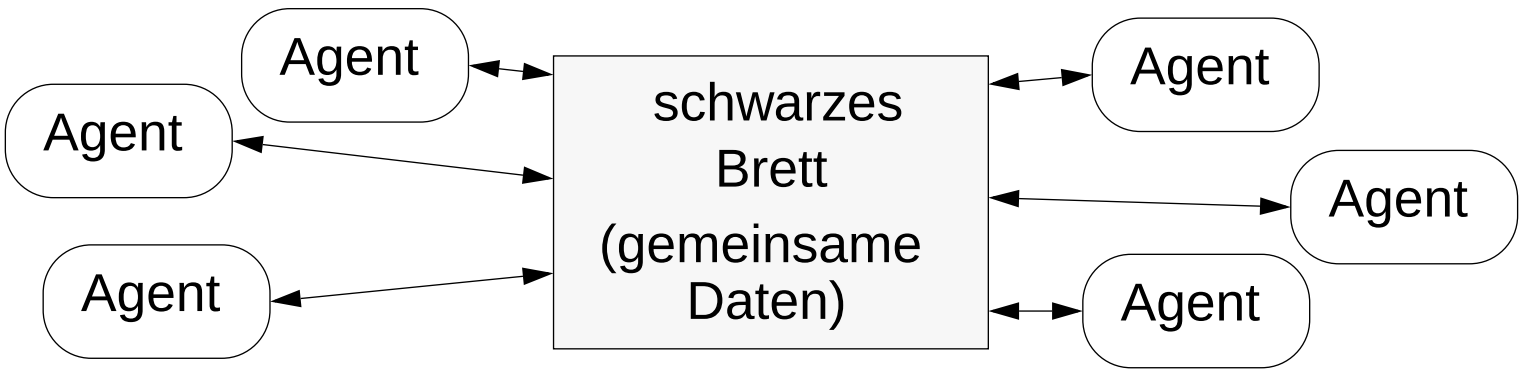
▲ Programme sind leicht portierbar, da Interpreter abstrakte Maschine anbieten kann (nur Interpreter muß portiert werden)

▼ Interpretations-Overhead durch Umsetzung von virtuellen Befehlen in Maschinenbefehle zur Laufzeit

Schwarzes Brett

Allgemeines

- Komponenten:
  - Schwarzes Brett (Blackboard) als zentrale Datenstruktur
  - „Agenten“ (knowledge sources), die auf dem schwarzen Brett operieren
    - bringen neues Wissen ein
    - verarbeiten vorhandenes Wissen
  - Steuerung: aktiviert Agenten
- Konnektoren: Zugriffe von Agenten auf das schwarze Brett



Schwarzes Brett

Beispiele

- Systeme zur Lösung von Problemen, für die kein deterministisches Verfahren bekannt oder effektiv anwendbar ist, z.B.
  - Systeme zur Spracherkennung
  - Systeme zur Mustererkennung
  - Steuerung für mobile Roboter

Varianten:

- Datenbank als zentrale Datenstruktur,
  Batches als Agenten (verarbeiten vornehmlich Eingaben)
- Compiler (schwarzes Brett: Symboltabelle, Syntaxbaum)
- Programmierumgebungen (schwarzes Brett: Programmdateien)

Schwarzes Brett

Vor- und Nachteile

▲ Agenten sind völlig entkoppelt

▲ Agenten können auch zur Laufzeit hinzugefügt und ausgetauscht werden, ohne daß andere Agenten betroffen sind

▲ Agenten können parallel ausgeführt werden

▼ Programmverhalten ist hochgradig nichtdeterministisch und daher schwer prüfbar

Architekturmuster

Übersicht der Muster aus Buschmann et al. (1996)

- From Mud to Structure
  - Layers
  - Pipes and Filters
  - Blackboard
- Distributed Systems
  - Broker
- Interactive Systems
  - **Model-View-Controller**
  - **Presentation-Abstraction-Control**
- Adaptable Systems
  - **Microkernel**
  - Reflection

Model-View-Controller (MVC)

Problem (1)

Kontext

Interaktive Programme mit einer flexiblen Benutzungsschnittstelle.

Problem

Benutzungsschnittstellen werden besonders häufig geändert, z.B.

- Erweiterung von Menüs durch neue Funktionen
- Völlig neue Benutzungsschnittstelle, z.B. Text-UI $\rightarrow$ GUI

Wenn die Benutzungsschnittstelle stark mit dem funktionalen Kern verflochten ist, sind Änderungen mühsam und fehlerträchtig.

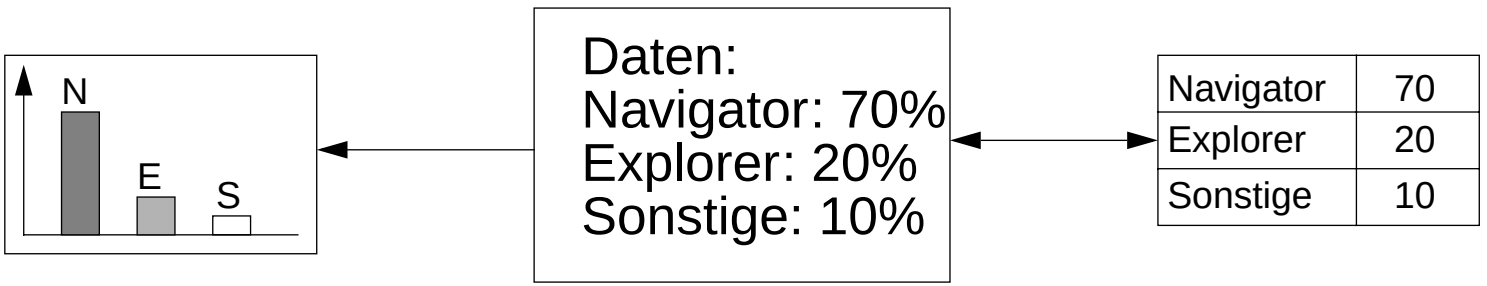
Besonders schwierig wird es, wenn ein System in mehreren Varianten mit verschiedenen Benutzungsoberflächen angeboten werden soll.

Model-View-Controller (MVC)

Problem (2)

Einflußfaktoren

- Dieselbe Information wird in verschiedenen Fenstern auf verschiedene Arten dargestellt
- Die Anzeige und das Verhalten des Programms muß Änderungen der Daten (d.h. des Zustands) sofort reflektieren
- Änderungen an der Benutzungsstelle sollten jederzeit einfach möglich sein, auch zur Laufzeit
- Die Portierung oder die Änderung des Look-and-Feel sollten den funktionalen Kern des Programms nicht betreffen



Model-View-Controller (MVC)

Lösung (1)

Idee

Trennung von Eingabe, Verarbeitung und Ausgabe in unabhängige Komponenten:

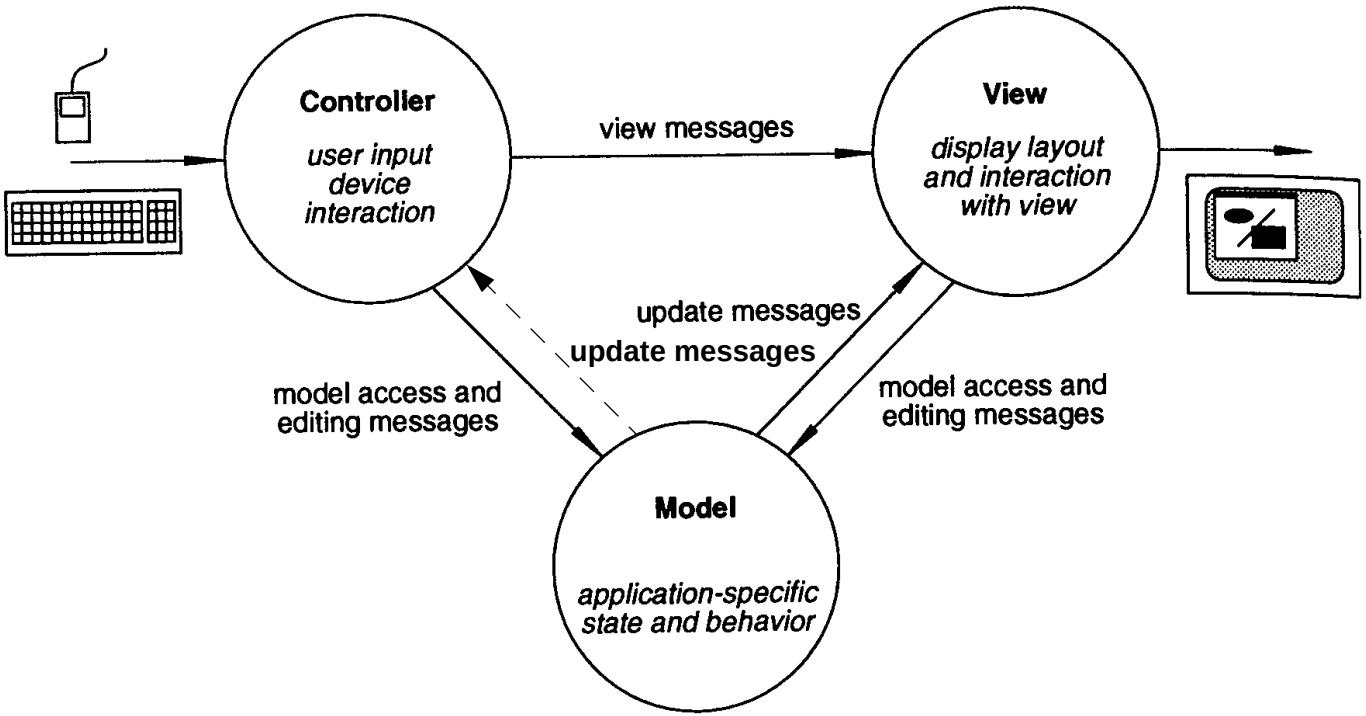
- **Model**: kapselt Daten und Funktionalität des Programms, die unabhängig von Ein- und Ausgabe sind. Teilt seinen Views alle Änderungen mit (Change-Propagation).
- **View**: stellt eine Sicht auf die Daten des Programms (das Model) für den Benutzer dar. Es kann mehrere Views für ein Model geben.
- **Controller**: verarbeiten die Eingaben des Benutzers (in Form von Events). Diese werden in Nachrichten an Model und Views umgesetzt. Zu jeder View gehört genau ein Controller.

Views und Controller registrieren sich beim Model, um über Änderungen informiert zu werden (Observer Pattern)

Model-View-Controller (MVC)

Lösung (2)

Interaktion der Komponenten

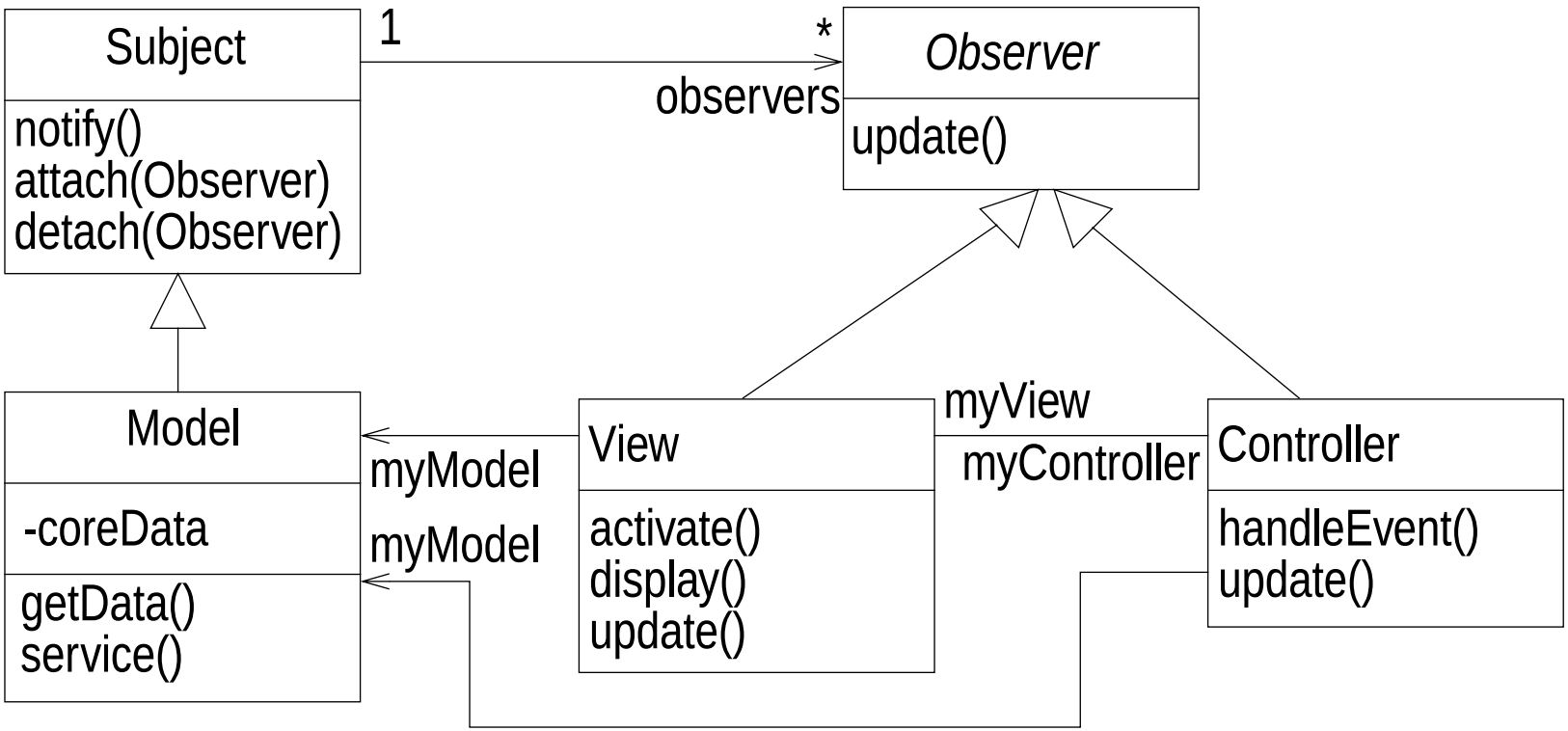


Quelle: Pree (1995), S. 70

Model-View-Controller (MVC)

Lösung (3)

Klassendiagramm



Model-View-Controller (MVC)

Bemerkungen

- Der Controller muß dafür sorgen, daß je nach Zustand des Programms gewisse Eingabemöglichkeiten erlaubt sind oder nicht. Daher kann es notwendig sein, sich als Beobachter des Models zu registrieren.
- Der Controller einer View kann zur Laufzeit ersetzt werden, z.B. um die Model-Änderungsmöglichkeiten in einer View abzuschalten.
- Es können zur Laufzeit beliebig viele Views auf ein Model existieren. Werden diese interaktiv erstellt, ist es sinnvoll, einen View-Manager zu haben.
- Häufig werden zur Vereinfachung View und Controller zu einer Komponente zusammengefaßt. Dann ergibt sich das Document-View Pattern.

Model-View-Controller (MVC)

Vor- und Nachteile

▲ Ein Model kann verschiedene Views haben, diese können problemlos ausgetauscht werden, auch zur Laufzeit

▲ Das Verhalten des Programms kann durch Austausch der Controller problemlos geändert werden, auch zur Laufzeit

▲ Durch Change-Propagation reflektieren die Views und Controller immer den aktuellen Zustand des Programms

▼ Höhere Komplexität des Entwurfs

▼ Effizienzverluste durch viele Update-Nachrichten und folgende Inspektionen des Models durch View/Controller

▼ Hohe Kopplung zwischen View bzw. Controller und Model

▼ Hohe Kopplung zwischen View und Controller

Presentation-Abstraction-Control (PAC)

Problem

Kontext

Entwicklung eines interaktiven Programms, das aus Agenten besteht

Problem

Wie kann ein interaktives Programm strukturiert werden, das konzeptionell aus Agenten (Werkzeugen) aufgebaut ist?

Einflußfaktoren

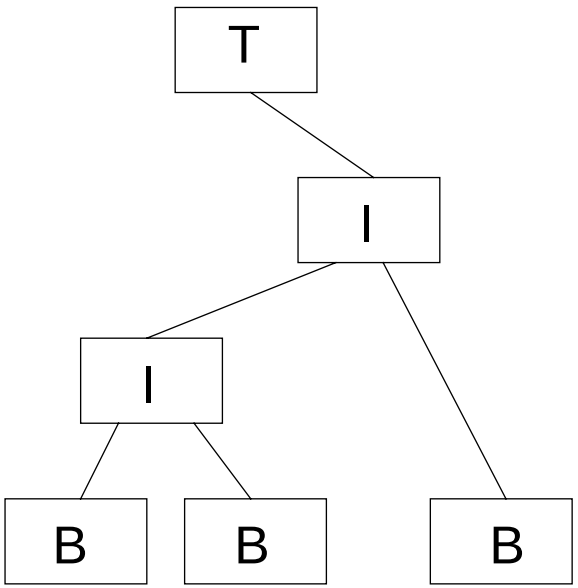
- Agenten verwalten in der Regel ihren Zustand selbst
- Agenten haben eine eigene Benutzungsoberfläche
- Benutzungsoberflächen neigen zu häufigen Änderungen
- Agenten müssen kooperieren können
- Das System muß um neue Agenten erweiterbar sein

Presentation-Abstraction-Control (PAC)

Lösung

Das System wird in drei verschiedene Arten von Agenten zerlegt:

- **Top-level Agent**: funktionaler Kern des Systems. Die meisten anderen Agenten operieren auf diesem Kern.
- **Bottom-Level Agenten**: repräsentieren ein in sich geschlossenes semantisches Konzept, mit dem der Benutzer des Systems arbeitet, z.B. Klassen-Browser
- **Intermediate-Level Agenten**: dienen der Vermittlung zwischen Agenten. Sie fassen Agenten tieferer Stufen zusammen (Komposition) oder koordinieren sie (z.B. durch Change Propagation).



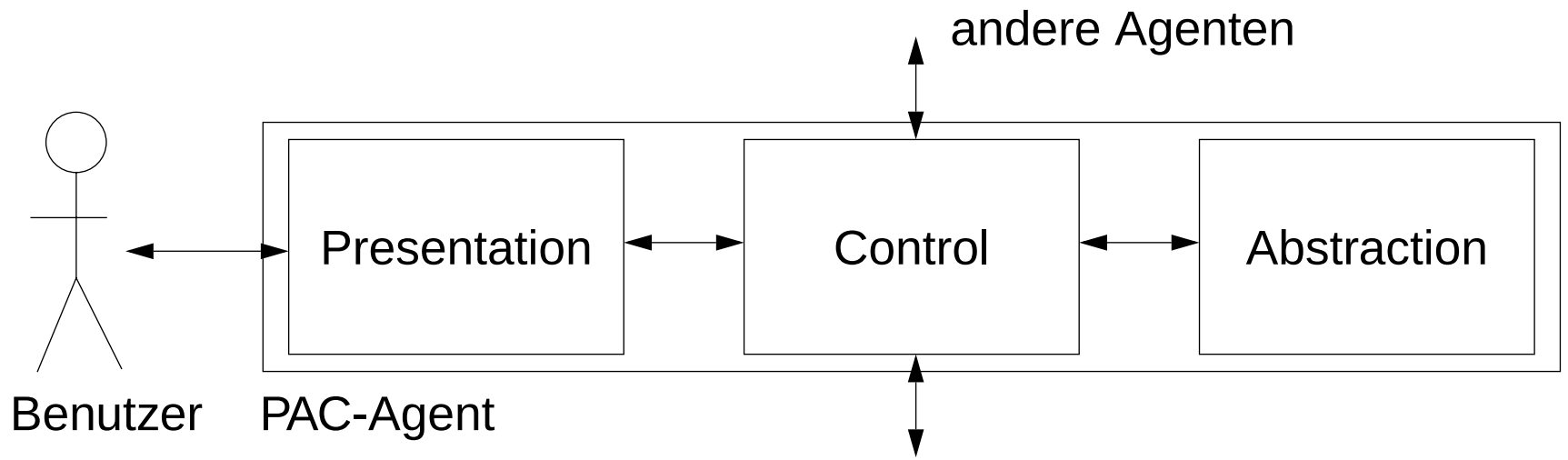
Der Benutzer interagiert vor allem mit Bottom-Level Agenten.

Presentation-Abstraction-Control (PAC)

PAC-Agenten

Jeder Agent zerfällt in drei Bestandteile:

- **Presentation**: die Benutzungsschnittstelle, d.h. das sichtbare Verhalten des Agenten
- **Abstraction**: der funktionale Kern mit der Zustandsverwaltung
- **Control**: koppelt Presentation und Abstraction und realisiert die Kommunikation mit anderen Agenten.



Presentation-Abstraction-Control (PAC)

Top-Level Agent

Es gibt immer nur einen Top-Level Agent.

Presentation: umfaßt Teile der Benutzungsschnittstelle allgemeiner Art (kann auch ganz fehlen)

Abstraction: verwaltet das globale Modell, kapselt häufig auch die Anbindung an externe Systeme wie z.B. Datenbanken

Control:

- erlaubt anderen Agenten, auf das globale Modell zuzugreifen
- koordiniert die Agentenhierarchie
- verwaltet Informationen für die Interaktion, z.B. welche Operationen erlaubt sind oder eine Historie für Undo.

Presentation-Abstraction-Control (PAC)

Bottom-Level Agent

Repräsentiert ein semantisches Konzept der Anwendungswelt, z.B. eine Mailbox. Das Konzept ist atomar, d.h. die kleinste sinnvollen Einheit, die ein Benutzer manipulieren kann.

Presentation: präsentiert eine spezifische Sicht auf das semantische Konzept. Kapselt dazu nötige Daten, z.B. Bildschirmkoordinaten, Größe.

Abstraction: verwaltet agenten-spezifische Daten, z.B. Zustandsinformationen. Realisiert den funktionalen Kern des Agenten.

Control: stellt Konsistenz zwischen Presentation und Abstraction sicher. Realisiert die Kommunikation mit Agenten der höheren Stufe. Events (z.B. open) werden an die Presentation, Daten an die Abstraction weitergeleitet.

Bottom-Level Agenten können auch Systemdienste realisieren.

Presentation-Abstraction-Control (PAC)

Intermediate-Level Agent

Zwei verschiedene Rollen: Komposition und Koordination

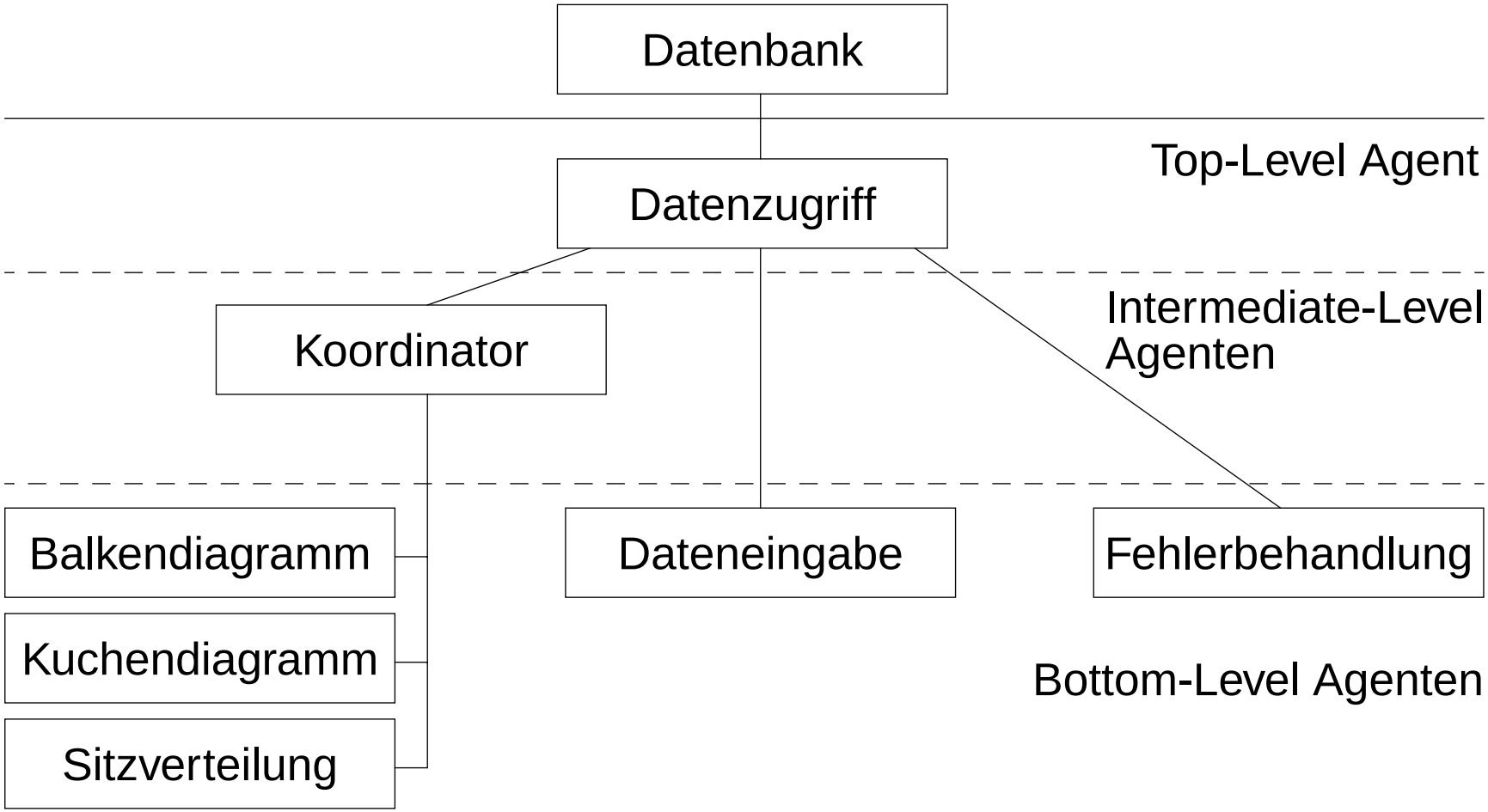
Presentation: Benutzungsschnittstelle des Agenten, sofern vorhanden

Abstraction: verwaltet den agenten-spezifischen Zustand

Control: stellt Konsistenz zwischen Presentation und Abstraction sicher. Realisiert die Kommunikation mit Agenten der höheren und der niedrigeren Stufe.

Presentation-Abstraction-Control (PAC)

Beispiel Parlamentswahlauswertungssystem



Presentation-Abstraction-Control (PAC)

Beispiel Netscape Communicator

sichtbare Agenten (bottom-level?):

- HTML-Browser
- Anzeige HTML-Code der Seite
- Anzeige von Seiteninformationen (Struktur, Metadaten, etc.)
- HTML-Editor
- Preferences-Dialog
- Adreßbuch
- Bookmark-Verwalter
- Anzeige der Nachrichten in einer Mailbox
- Message-Center (Übersicht über Mailfolder und Newsgroups)
- verschiedene Einstellungs- und Auswahldialoge

Presentation-Abstraction-Control (PAC)

Vor- und Nachteile

- ▲ Prinzip der Trennung der Zuständigkeiten konsequent angewendet
- ▲ System kann leicht geändert und erweitert werden
- ▲ Agenten sind wiederverwendbar und portabel
- ▲ Agenten können auf verschiedene Rechner oder Prozessoren eines Rechners verteilt werden
- ▲ Mehrbenutzerfähig, wenn die Benutzer synchronisiert werden
- ▼ Hohe Komplexität des Systems; daher Abstraktion nicht zu niedrig ansetzen (z.B. graphische Objekte eines Zeichenprogramms nicht als vollwertige Bottom-Level Agenten modellieren)
  -> PAC anwenden, wenn semantische Einheiten hinreichend groß
- ▼ Effizienzverluste durch Kommunikation und ständige Datenübertragungen; Caching kann das etwas ausgleichen

Microkernel

Problem (1)

Kontext

Es sollen mehrere Programme entwickelt werden, die Programmierschnittstellen verwenden, die auf eine ähnliche Kernfunktionalität aufbauen

Problem

Wie sollte Software für einen Anwendungsbereich, in dem es eine Menge von ähnlichen Standards und Technologien gibt, aufgebaut sein? z.B. für Anwendungsplattformen wie Betriebssysteme (z.B. UNIX) und graphische Benutzungsoberflächen (z.B. X-Windows).

Microkernel

Problem (2)

Einflußfaktoren

- Die Plattform muß mit einer fortwährende Evolution von Soft- und Hardware zurechtkommen
- Die Plattform soll portabel, erweiterbar und an neue Technologien anpaßbar sein.
- Es soll möglich sein, auf der Plattform andere, ähnliche Plattformen zu simulieren (Emulation)
- Der funktionale Kern der Plattform soll in eine Komponente mit minimalem Bedarf an Speicher und Rechenleistung ausgelagert werden

Microkernel

Lösung

Das System wird in fünf verschiedene Komponenten zerlegt:

Microkernel: Kapselt die grundlegenden Dienste der Plattform. Verwaltet systemweite Ressourcen (Prozesse, Dateien, Peripheriegeräte, usw.) und stellt Prozeßkommunikation zur Verfügung. Die konkrete Hardware wird weitgehend abstrahiert.

Interne Server: aus dem Microkernel ausgelagerte zusätzliche Dienste, z.B. Treiber. Dürfen nur vom Microkernel verwendet werden.

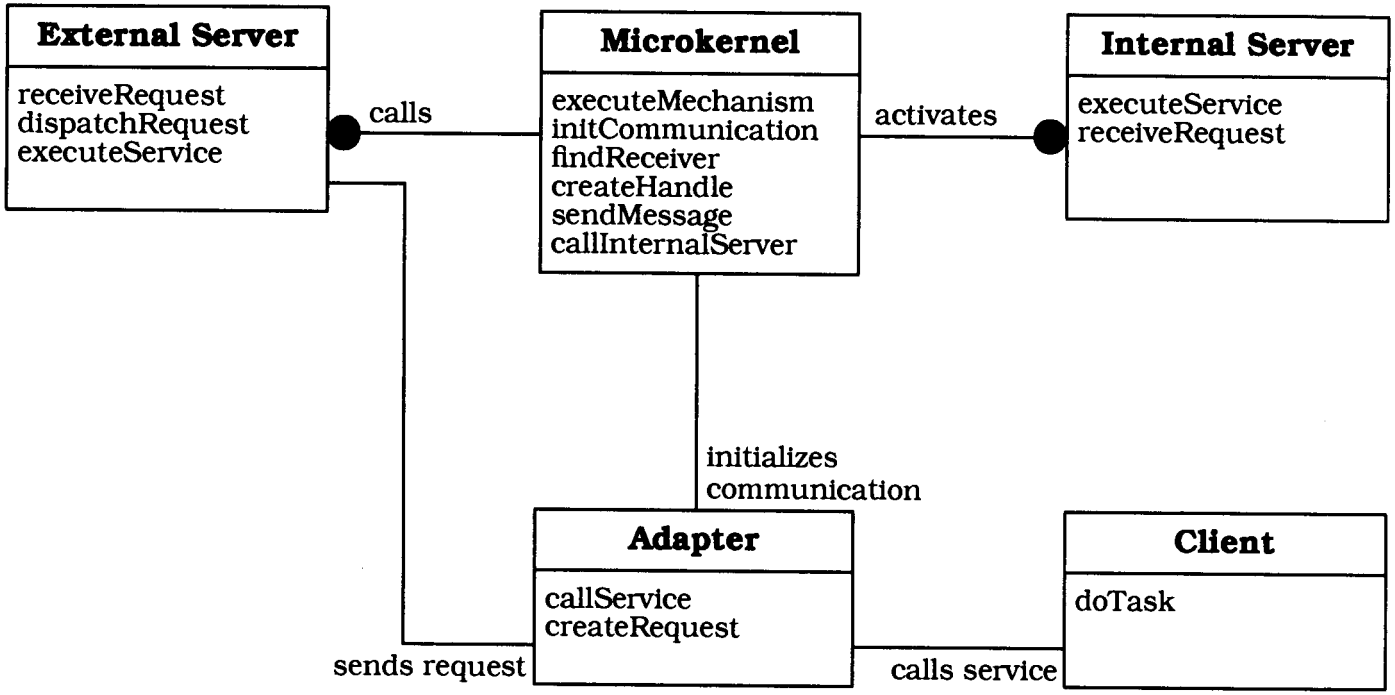
Externe Server: Stellen eine bestimmte Sicht auf den Microkernel zur Verfügung. Laufen als Prozesse auf dem Microkernel.

Adapter: Emulatoren, die die API einer Plattform auf externe Server abbilden. Dienen zur Entkopplung von Clients und externen Servern.

Clients: Benutzen externe Server über einen Adapter.

Microkernel

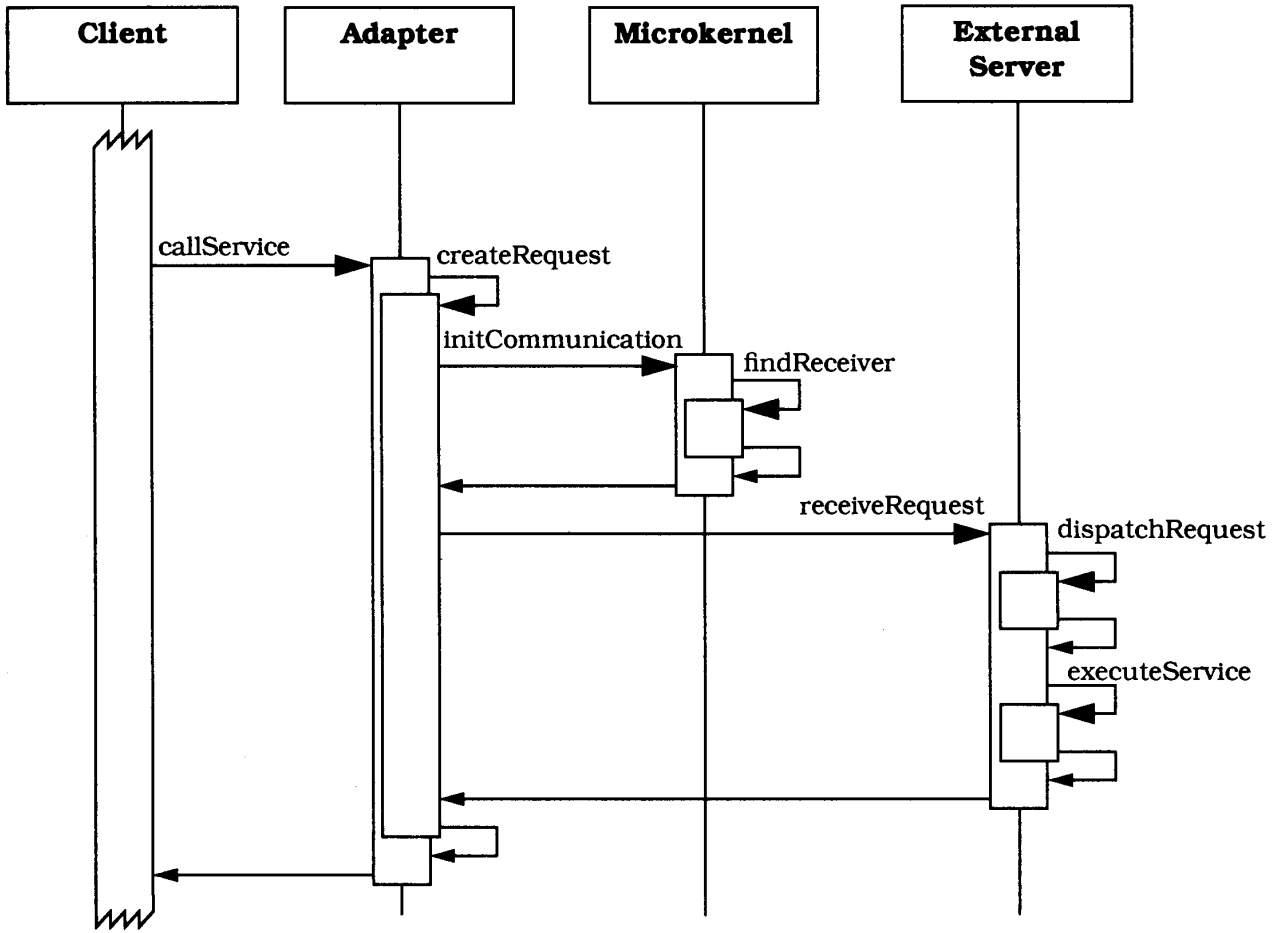
Klassendiagramm



Quelle: Buschmann et al. (1996), S. 178

Microkernel

Sequenzdiagramm: Client ruft API-Funktion auf



Microkernel

Vor- und Nachteile

- ▲ Hohe Portabilität
- ▲ Flexibilität und Erweiterbarkeit
- ▲ Trennung von Mechanismen (realisiert im Microkernel, z.B. Prozeß) von Verfahrensweise (realisiert in den externen Servern, z.B. Prozeß-Scheduling)

- ▼ Leistungseinbußen gegenüber einem monolithischen System
- ▼ Hohe Komplexität von Entwurf und Implementierung

Big Ball of Mud

Problem (1)

Foote und Yoder (1997) beschreiben in ihrem Paper ein Muster für die de-facto Standardarchitektur in der Softwareentwicklung.

Kontext

Es soll ein Softwaresystem unter den üblichen Bedingungen der Praxis entwickelt werden. D.h. es herrscht hoher Zeitdruck, das Budget ist klein, das Management will so schnell wie möglich ein laufendes Programm sehen, die Anforderungen sind unklar oder unzureichend dokumentiert, ...

Problem

Wie kann ein Softwaresystem mit brauchbarer Architektur unter den genannten Bedingungen entwickelt werden?

Big Ball of Mud

Problem (2)

Einflußfaktoren

- es steht nicht genug Zeit zur Verfügung, wohldurchdachte Architektur- und Implementierungsentscheidungen zu fällen
- es mangelt an Erfahrung im Entwurf und im Anwendungsbereich
- um eine geeignete Architektur festzulegen, braucht man viel Zeit und Wissen, das man häufig zu Beginn des Entwurfs noch gar nicht hat.
- die Komplexität eines Softwaresystems spiegelt die Komplexität dessen wider, was es modelliert.
- es kommt sicher zu unerwarteten Änderungen, für die die Architektur nicht ausgelegt ist
- Investitionen in eine gute Architektur scheinen (zunächst) unnötig, insbesondere dann, wenn das System schon läuft

Big Ball of Mud

Lösung

Zunächst werden Architekturüberlegungen vernachlässigt. Erstes Ziel ist es, ein lauffähiges System zu erstellen, das nach und nach erweitert wird. Man erhält dadurch mit ziemlicher Sicherheit einen großen Matschklumpen (Big Ball of Mud).

Die bei Entwicklung (und Wartung) gemachten Erfahrungen werden dazu verwendet, dem bestehenden System nach und nach eine durchdachte Architektur zu geben. Sollte das unmöglich sein, werden Teile rekonstruiert.

„Form follows function“: Der Fokus liegt in der ersten Phase auf Eigenschaften und Funktionalität, erst dann auf Architektur und Performance.

Kent Beck (1997): „Make it work. Make it right. Make it fast.“

Stewart Brand (1994): „Maintenance is learning“

Big Ball of Mud

Beteiligte Muster (1)

Throwaway Code (Ursache)

Es wird ein schnelle Lösung für ein Problem, ein Prototyp oder eine Machbarkeitsstudie benötigt

☞ Erstelle einfachen, zweckdienlichen Code zum Wegwerfen, der ausschließlich das vorhandene Problem löst

Eigentlich sollte der Code später durch „richtigen“ ersetzt werden, das geschieht jedoch selten.

Big Ball of Mud

Beteiligte Muster (2)

Piecemeal Growth (Ursache und Strategie)

Zu Beginn der Entwicklung entworfene Architekturen sind oft unbeweglich, fehlgeleitet und sehr schnell veraltet.

Die Anforderungen der Benutzer ändern sich mit der Zeit, wenn das Programm verwendet wird. Wenn die Software erfolgreich ist, wird sie auch andere Benutzer ansprechen als ursprünglich geplant.

- ☞ Kümmere dich schrittweise um Änderungen, die spätere Änderungen und Wachstum erleichtern. Änderungen sollten so klein und so lokal wie möglich durchgeführt werden.
- ☞ iterative, inkrementelle, evolutionäre Entwicklung

Wenn ein System erweitert wird, geht das in der Regel auf Kosten der Architektur.

Big Ball of Mud

Beteiligte Muster (3)

Keep it working (Ursache und Strategie)

Es hat sich ein hoher Wartungsbedarf angesammelt. Allerdings ist eine Generalüberholung unangebracht, da das System diese nicht überleben könnte.

- ☞ Führe unbedingt nötige Wartungsarbeiten (in kleinen Schritten) durch, aber halte das System unbedingt am Laufen.
  Beispiel: Microsofts Daily Build
- ☞ Führe diejenigen Änderungen bevorzugt aus, die die Änderbarkeit des Systems am wenigsten untergraben

Wenn immer nur kleine Änderungen gemacht werden, kann beim Auftreten eines neuen Fehlers die Ursache schneller eingekreist werden. Notfalls können die Änderungen schnell rückgängig gemacht werden.

Big Ball of Mud

Beteiligte Muster (4)

Sweeping it under the rug (Lösungsstrategie)

Wuchernder, verstrickter, planloser Code ist schwer zu verstehen, zu reparieren oder zu erweitern. Mit der Zeit wird es immer schlimmer, wenn man es nicht in den Griff bekommt.

- ☞ Wenn du das Durcheinander nicht wegbekommst, trenne es zumindest vom Rest ab. Das konzentriert die Unordnung auf einen eingegrenzten Bereich und versteckt sie (hinter einer Fassade).
- ☞ Bei Bedarf kann dieser Bereich überarbeitet oder neu geschrieben werden.

Beispiel: OO-Schnittstelle zu Legacy Systems

Big Ball of Mud

Beteiligte Muster (5)

Reconstruction (Lösungsstrategie)

Der Code ist so degeneriert, daß er nicht mehr verstanden und schon gar nicht repariert werden kann.

Die Kosten für ein „Weiter wie bisher“ beginnen die Kosten für einen Neuanfang zu übersteigen.

- ☞ Wirf den alten Code weg und fang neu an.
- ☞ Analysiere den alten Code sorgfältig, um herauszufinden, was daran gut und schlecht war.

Die bisher gesammelten Erfahrungen mit dem Code werden zu einer viel besseren Architektur führen.

Unter Umständen kann es sich lohnen, noch einen Schritt weiterzugehen und ein Framework zu entwickeln.

Kapitel 5: Entwurfsmuster

- Entwurfsmuster nach Gamma et al. (1995), auch bekannt als „the GoF book“ (Gang of Four)

Entwurfsmuster nach Gamma et al. (1995)

Übersicht

	Creational	Structural	Behavioral
Class	**Factory Method**	Adapter (class)	Interpreter **Template Method**
Object	Abstract Factory Builder Prototype **Singleton**	**Adapter** (object) Bridge **Composite** Decorator Facade Flyweight Proxy	Chain of Responsibility Command **Iterator** Mediator Memento **Observer** State **Strategy** Visitor

Entwurfsmuster nach Gamma et al. (1995)

Klassifikation

- Ebene
  - **Class**: auf Klassenebene, etabliert durch Vererbung, statisch
  - **Object**: auf Objektebene, etabliert durch Assoziation, dynamisch
- Zweck
  - **Creational**: zur Erzeugung von Objekten
  - **Structural**: Aufbau/Strukturierung komplexer Objekte
  - **Behavioral**: Zusammenspiel und Verantwortlichkeiten von Objekten zur Lösung komplexer Aufgaben.

Template Method

Problem

- Ein Algorithmus, zu dem es mehreren Varianten gibt, soll realisiert werden
- Es soll möglich sein, einfach neue Varianten hinzuzufügen
- Es soll möglichst wenig Code dupliziert werden

Idee

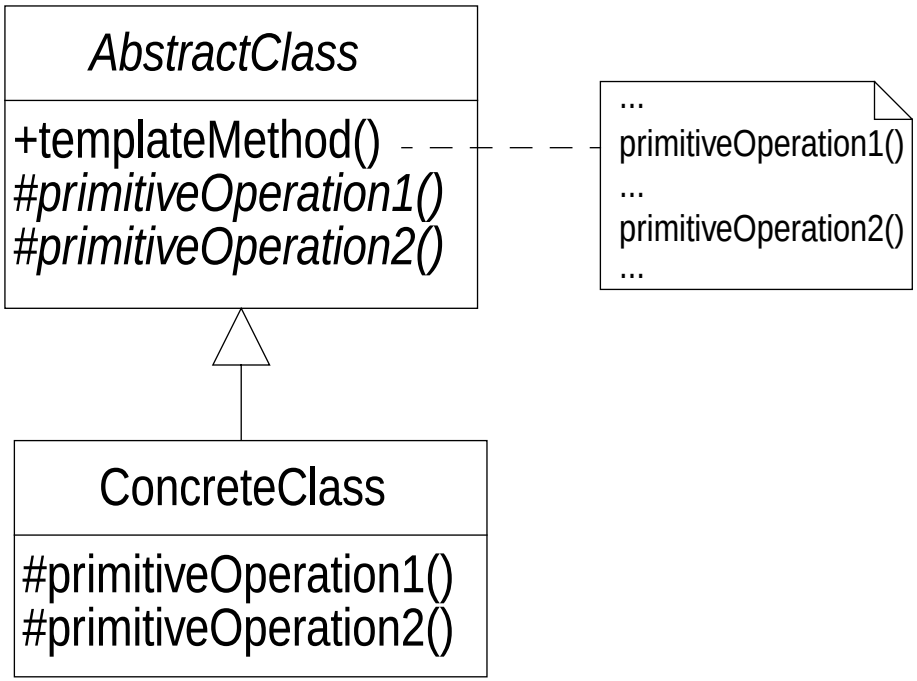
- allen Varianten gemeinsamen Code in ein Algorithmusgerüst packen
- die varianten Teile in neue Methoden auslagern, die vom Gerüst aufgerufen werden.

Dazu müssen allerdings alle Varianten einem gemeinsamen Schema entsprechen.

Template Method

Struktur

- das Algorithmusgerüst wird in der konkreten Methode `templateMethod()` realisiert
- die variablen Teile sind in die abstrakten Methoden `primitiveOperationX()` ausgelagert und werden von `templateMethod()` aufgerufen

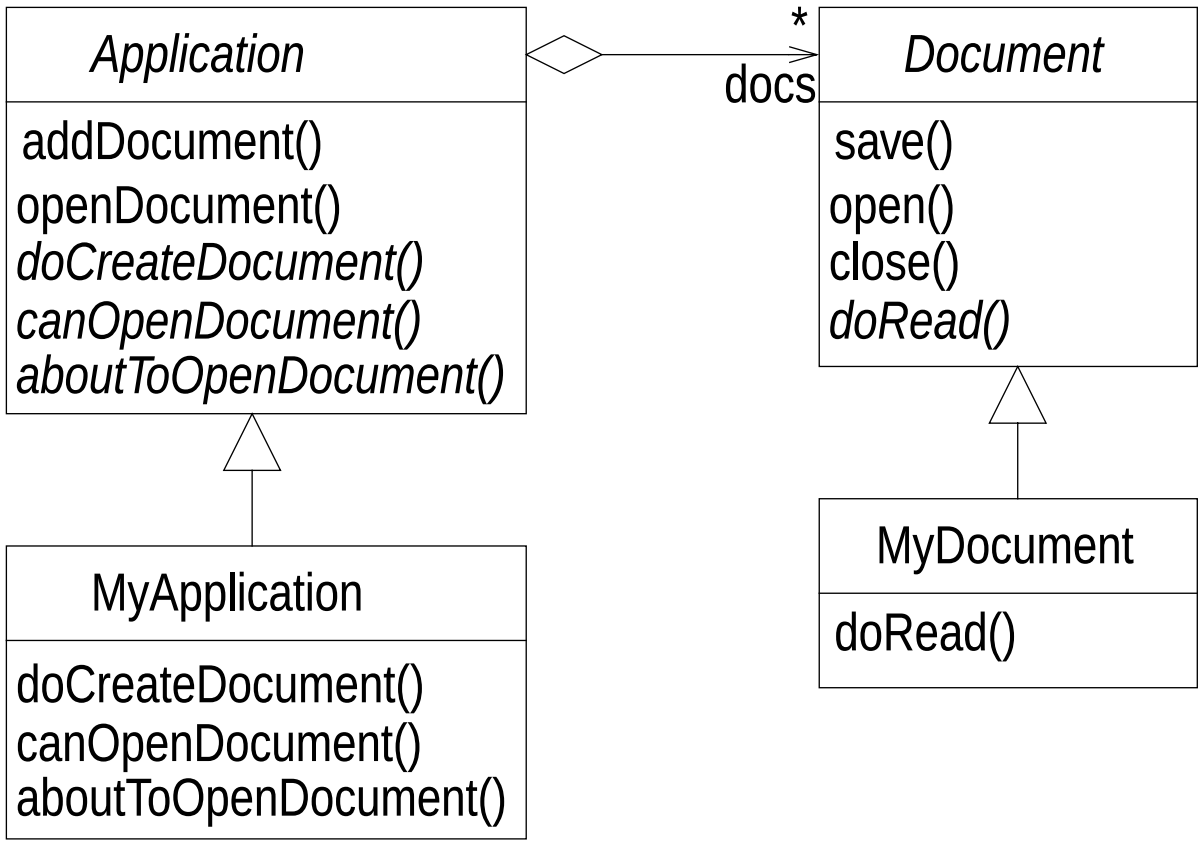


- eine konkrete Realisierung erbt von AbstractClass und implementiert alle primitiven Methoden.
- In der Regel sollten die primitiven Methoden von außen nicht aufgerufen werden können, daher sind sie protected

Template Method

Beispiel (1)

- Eine Anwendung kann beliebig viele Dokumente haben.
- Die Template Method `openDocument` dient dazu, ein anwendungsspezifisches Dokument zu öffnen



Template Method

Beispiel (2)

Implementierung von openDocument in Application

```
openDocument(String name) {
   if (!canOpenDocument(name))
      return; // Dokument kann nicht geöffnet werden
   Document doc = this.doCreateDocument(name);
   if (doc != null) {
      this.addDocument(doc);
      this.aboutToOpenDocument(doc);
      doc.open();
      doc.doRead();
   }
}
```

Template Method

Bemerkungen

- eines der wichtigsten Muster zur Realisierung von Frameworks
- Umkehrung der Steuerung: „Don‘t call us, we‘ll call you“
  d.h. Superklasse ruft Operationen der Subklasse auf
- primitive Operationen müssen nicht abstrakt sein, sie können auch eine sinnvolle Default-Implementierung beinhalten (z.B. NOP)
- es sollte so wenig wie möglich primitive Operationen geben, die redefiniert werden müssen (Narrow Inheritance Interface Principle)
- der Algorithmus sollte möglichst allgemein parametrisiert werden können
- Template Methode final deklarieren, um Redefinition zu verhindern
- Eine sinnvolle Konvention ist es, primitive Methoden mit dem Präfix do zu benennen (z.B. `doCreateDocument`, `doRead`)
- mögliche Alternative: generisches Paket (z.B. in Ada)

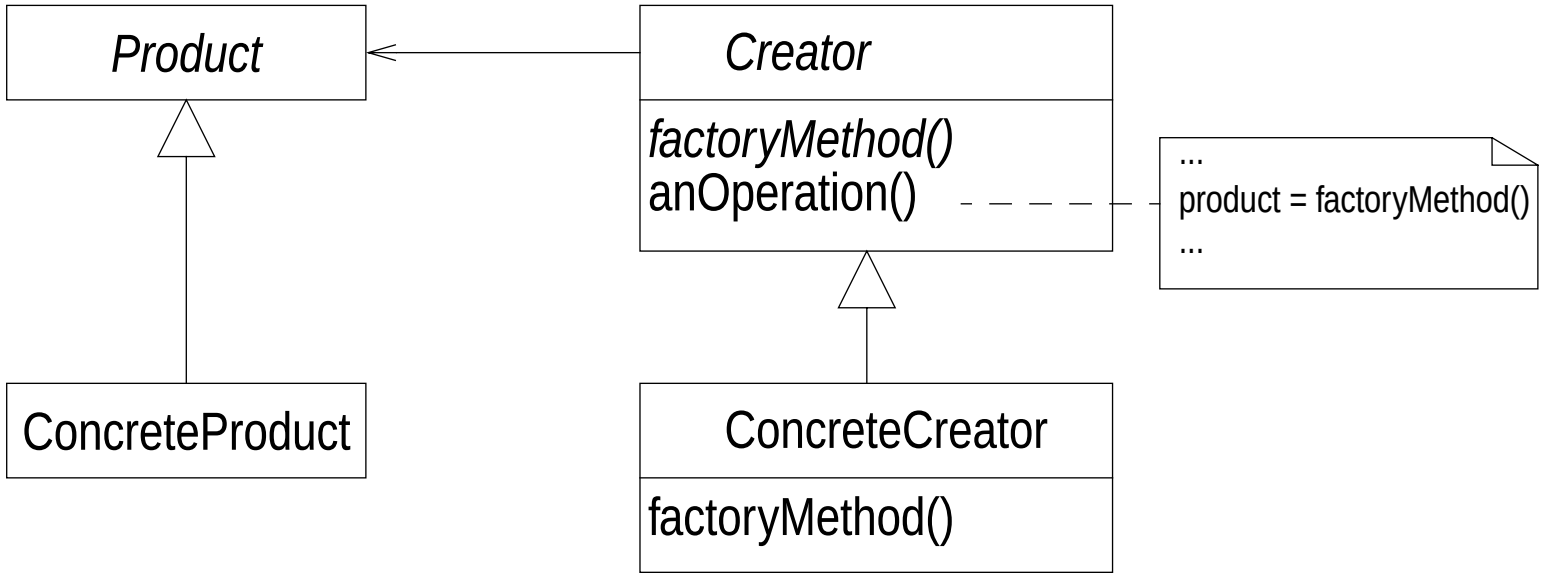
Factory Method

Problem

- In Frameworks wird vor allem mit abstrakten Klassen gearbeitet. Dennoch muß es diesen möglich sein, konkrete Objekte zu instantiieren.
- Deren Typ ist aber bei der Entwicklung des Frameworks nicht bekannt, erst bei der Verwendung
- Daher definiert eine abstrakte Klasse eine abstrakte Methode (Factory Method), die ein Objekt von einer allgemeinen Oberklasse liefert.
- Diese ist dann von konkreten Unterklassen so zu redefinieren, daß ein Objekt vom gewünschten Typ erzeugt wird. D.h. für die konkrete Instantiierung sind die Unterklassen verantwortlich.
- z.B. `doCreateDocument()` aus dem Template Method-Beispiel

Factory Method

Struktur



- Creator verwendet `factoryMethod()`, um ein Objekt vom Typ Product zu erzeugen.
- `factoryMethod()` ist zunächst abstrakt und muß redefiniert werden.

Factory Method

Bemerkungen

- Falls sinnvoll, kann in Creator eine Default-Implementierung der Factory Method angegeben werden
- Factory Methods können auch parametrisiert werden, um verschiedene Produkte zu liefern
  - spart spezialisierte Factory Methods ein
  - Unterklassen können neue Varianten zulassen oder vorhandene umwidmen
- Alternative zur Factory Method: die zu instantiierende Klasse als Parameter einer generischen Klasse Creator angeben und dadurch konkrete Kreatoren erzeugen

Factory Method

Vor- und Nachteile

- ▲ Framework kann unabhängig von anwendungsspezifischen Klassen geschrieben werden
- ▲ Framework-Code hängt nur von der Schnittstelle von Product ab
- ▲ höhere Flexibilität als bei der expliziten Instantiierung einer bestimmten Klasse

- ▼ Um ein konkretes Objekt instantiieren zu können, muß auf jeden Fall eine Unterklasse von Creator gebildet werden.

Adapter

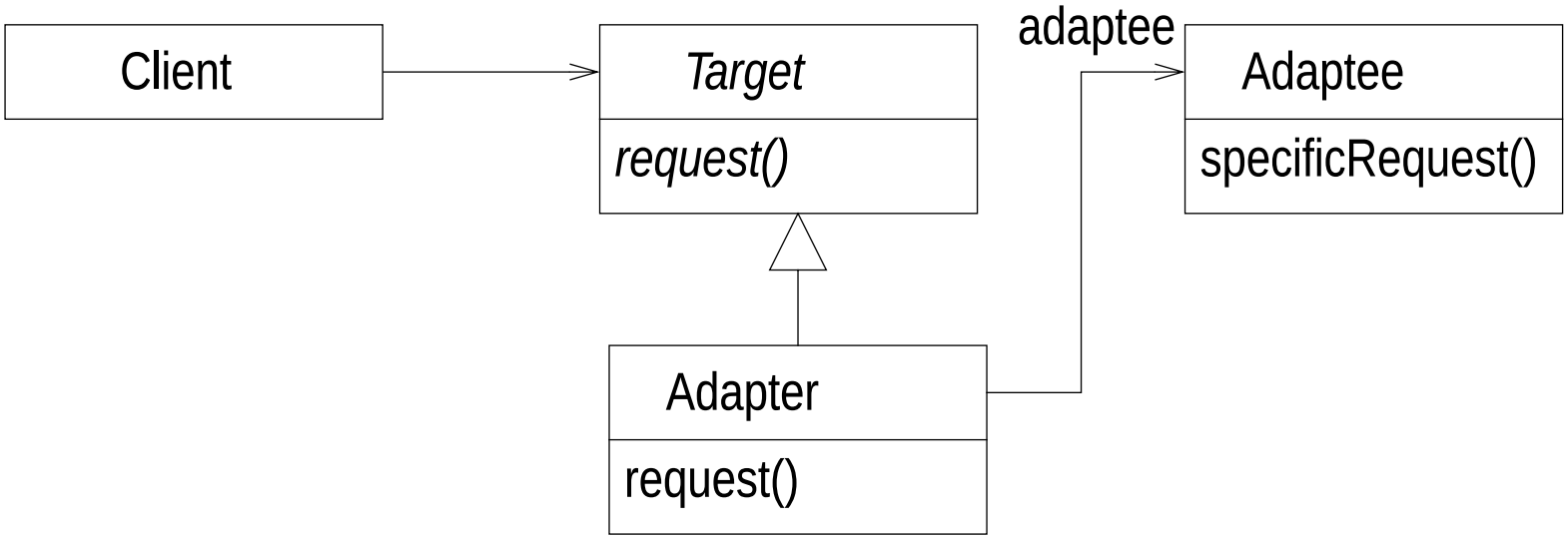
Problem

- eine Klasse A soll eine andere Klasse B verwenden, aber deren Schnittstelle (soll vom Typ C sein) paßt nicht
- das passiert häufig, wenn man verschiedene Klassenbibliotheken kombinieren oder Klassen wiederverwenden will

- „Lösung“ 0: Code von B von Hand ändern (falls Source verfügbar)
- Lösung 1: bilde eine Subklasse BC von B und C und redefiniere die Methoden von C so, daß sie die von B verwenden.
  Problem: Methoden-Durcheinander, umfangreiche Schnittstelle, Mehrfachvererbung nötig
- Lösung 1a (nur C++): BC erbt von B privat (zur Implementierung)
- Lösung 2: Verpacke die Klasse B in einem Adapter, der die gewünschte Schnittstelle zur Verfügung stellt

Adapter

Struktur (Objektvariante)



- Adapter benutzt Adaptee (uses-Relation)
- Client ruft `request()` von Adapter auf
- Adapter setzt den Aufruf von `request()` um in einen Aufruf von `adaptee.specificRequest()`.

Adapter

Beispiel (1)

gewünschte Schnittstelle:

```
abstract class Dictionary {
   addEntry(String key, String descr);
   hasEntry(String key);
   deleteEntry(String key);
}
```

Vorhandene Klasse:

```
class Abbreviations {
   addAbbrev(String abbr, String descr);
   hasAbbrev(String abbr);
   removeAbbrev(String abbr);
}
```

Adapter

Beispiel (2)

```
class AbbreviationAdapter extends Dictionary {
   Abbreviations abbrevs;
   AbbreviationAdapter(Abbreviations a) {
      abbrevs = a;
   }
   addEntry(String key, String descr) {
      abbrevs.addAbbrev(key,descr);
   }
   hasEntry(String key); {
      return abbrevs.hasAbbrev(key);
   }
   deleteEntry(String key);{
      abbrevs.removeAbbrev(key);
   }
}
```

Adapter

Beispiel (3)

- Sortierte Listenklasse L mit Elementen vom Typ Sortable (hat Methode `isSmaller()`).
- Es soll eine sortierte Liste von Objekten der Klasse Order erstellt werden, die keine Vergleichsoperation hat. Sortiert wird nach Auftragsnummer, abfragbar mit `getOrderID()`

```
class OrderAdapter extends Sortable {
  private Order o;
  public OrderAdapter(Order theOrder) {o = theOrder;}
  public boolean isSmaller(Sortable x) {
    return o.getOrderID()
      < ((OrderAdapter) x).o.getOrderID();
  }
  // getOrder() + weitere Operationen von Sortable ...
}
```

Adapter

Bemerkungen

- adaptiert wird durch die Lösung nicht nur Adaptee selbst, sondern auch alle Subklassen von Adaptee
- der Adapter kann
  - lediglich Methodenaufrufe umsetzen, wenn die Benennung nicht stimmt
  - bei Aufrufen Parameter weglassen, hinzufügen oder konvertieren
  - fehlende Methoden zusätzlich implementieren, d.h. die von Adaptee zur Verfügung gestellte Funktionalität erweitern
- Muster auch bekannt als „Wrapper“

Adapter

Vor- und Nachteile

- ▲ Adapter erlaubt die Verwendung einer Klasse mit unpassender Schnittstelle ohne Code-Änderungen
- ▲ kann die Funktionalität von Adaptee erweitern
- ▲ es können auch Objekte von Unterklassen von Adaptee bei der Adaption verwendet werden

- ▼ aber zur Nutzung zusätzlicher Funktionalität dieser Unterklassen muß der Code des Adapters geändert werden

Singleton

Problem

- Es soll sichergestellt werden, daß es zu einer Klasse höchstens eine Instanz gibt.
- Diese Instanz soll global verfügbar sein, d.h. es muß eine Möglichkeit für alle Objekte geben, diese Instanz anzusprechen.
- Beispiele: Drucker-Spooler, Dateisystem, Window-Manager

- Lösung 1: Instanz in globaler Variablen ablegen.
  Problem: verhindert nicht die mehrfache Instantiierung.
- Lösung 2: deklariere alle Eigenschaften der Instanz als Klasseneigenschaften.
  Problem: keine Redefinition möglich

Singleton

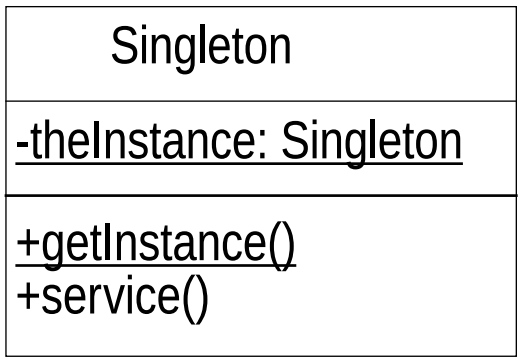
Lösung

Idee

die Klasse ist selbst dafür verantwortlich, daß es von ihr nur ein Exemplar gibt. Sie stellt auch diese Instanz zur Verfügung.

Struktur

- die Instanz wird in der privaten Klassenvariablen `theInstance` gespeichert
- sie kann mit der Klassenmethode `getInstance()` angefordert werden. (Aufruf `Singleton.getInstance()`)



Singleton

Implementierung

```
class Singleton {
   static private Singleton theInstance = null;
   static public Singleton getInstance() {
     if (theInstance == null)
       theInstance = new Singleton();
     return theInstance;
   }
   /* Es darf keinen public-Konstruktor geben (also
     auch keinen Default-Konstruktor). Damit Unter-
     klassen ihn trotzdem benutzen können, wird er
     protected deklariert */
   protected Singleton() { ... }
   ... // Instanzvariablen und -methoden
}
```

Singleton

Bemerkungen (1)

- Redefinition der Singleton-Klasse ist möglich, allerdings muß dazu auch `getInstance()` überschrieben werden (alternativ: im Singleton Factory Method zur Instantiierung verwenden)

```
class NewSingleton extends Singleton {
   static public Singleton getInstance() {
     if (theInstance == null)
       theInstance = new NewSingleton();
     return theInstance;
   }
   protected NewSingleton() {
     super(); ...
   }
   ... // Instanzvariablen und -methoden
}
```

Singleton

Bemerkungen (2)

- alternativ kann in `getInstance()` anhand globaler Information (z.B. Properties, Umgebungsvariablen) oder Parametern entschieden werden, welche Klasse zu instantiieren ist.
- Wenn ein Programm viele Singletons hat, kann eine Registry eingerichtet werden. Von dieser können die Singletons über einen Schlüssel (z.B. Klassenname) angefordert werden. Dazu registriert sich jede Singleton-Instanz (im Konstruktor) bei der Registry. Problem: Konstruktor muß von irgendwem aufgerufen werden.

Singleton

Vor- und Nachteile

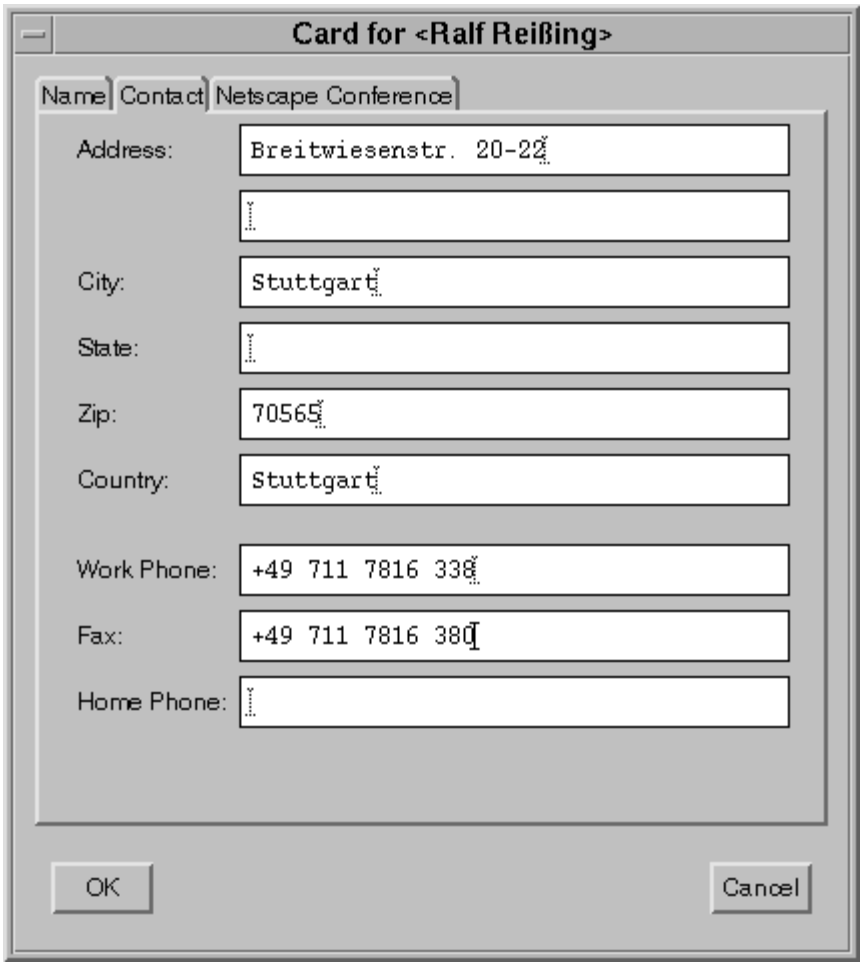
- Überwachter Zugang zu einer einzigen Instanz
- Instanz braucht erst bei Bedarf erzeugt werden
- Keine globale Variable notwendig
- Die Singleton-Klasse kann redefiniert werden
- Kann verallgemeinert werden für n Instanzen
- Flexibler als die Realisierung durch Klassenoperationen

Strategy

Motivation (1)

Aufgabe

In einem Eingabeformular (z.B. eines Adreßbuchs) muß in einer Reihe von Textfeldern die Eingabe auf korrekte Syntax geprüft werden.
(z.B. ganze Zahl, Postleitzahl, E-Mail-Adresse, Telefonnummer).



Strategy

Motivation (2)

- Lösung 1: monolithisch
  - beim Bestätigen der Eingaben wird in einer „submit“-Methode geprüft, ob die Textfelder korrekt ausgefüllt sind.
  - Problem: schlechte Änderbarkeit bei Änderung des Formulars oder der Syntax, Code-Duplikation
- Lösung 2: „Faktorisierung“
  - die Prüfroutinen für einzelne Eingabeformate werden in spezielle Methoden ausgelagert, die von der „submit“-Methode aufgerufen werden.
  - Problem: schlechte Änderbarkeit bei Änderung des Formulars

Strategy

Motivation (3)

- Lösung 3: Spezialisierung von TextField
  - Spezielle Unterklassen von TextField einführen, die die Prüffunktion beinhalten (z.B. EMailTextField). Das Formularobjekt stößt die Prüfung der einzelnen Textfelder an.
  - Problem: Explosion der Klassen
- Lösung 4: Strategy Pattern
  - Es werden spezielle Prüfklassen eingeführt. Diese prüfen einen Text auf korrekte Syntax. Ein TextField wird mit einer Prüfklasse assoziiert und delegiert die Prüfung an diese.
  - Da das Prüfobjekt zur Laufzeit ausgetauscht werden kann, können auch Textfelder geprüft werden, deren Semantik sich dynamisch ändert (z.B. Stichwort, Signatur, ISBN bei einem Formular zur Bibliotheksrecherche)

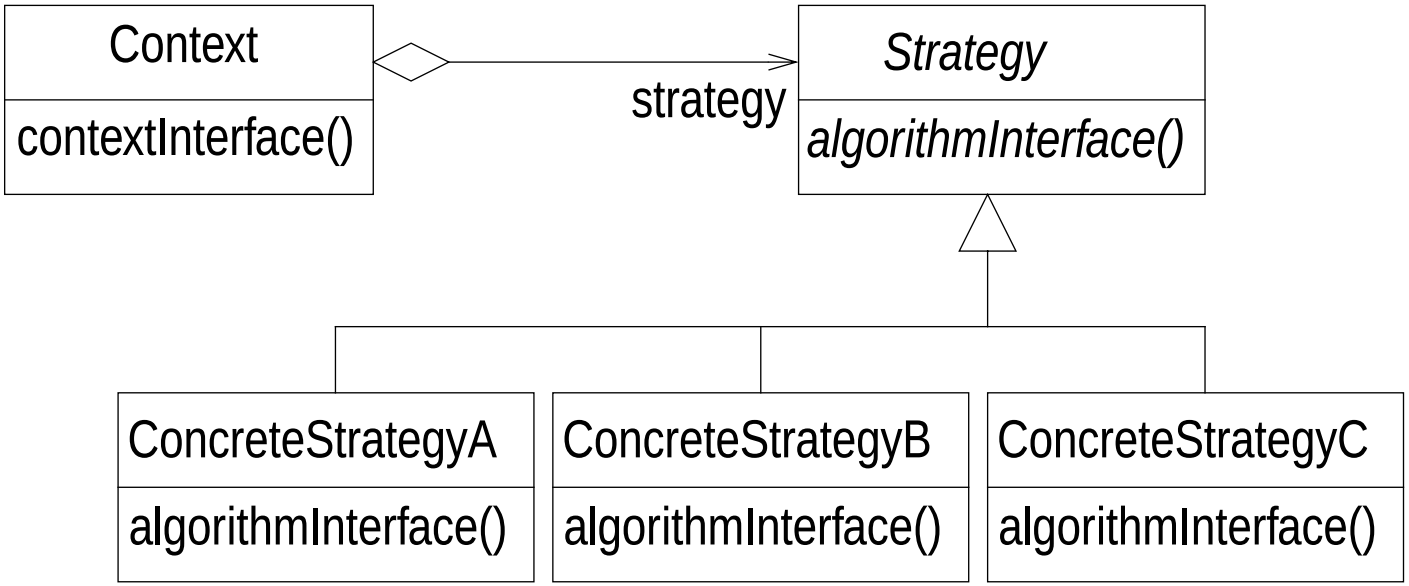
Strategy

Anwendbarkeit

- eine Menge verwandter Klassen unterscheidet sich nur in ihrem Verhalten. Dann kann eine Klasse mit verschiedenen Verhaltensweisen konfiguriert werden.
- es werden verschiedene Varianten eines Algorithmus benötigt. Diese werden in separate Klassen ausgelagert.
- die von einem Algorithmus (intern) benötigten Daten sollen dem Verwender verborgen bleiben. Dazu wird er in eine Klasse verpackt.
- eine Klasse hat je nach Zustand ein unterschiedliches Verhalten, das durch umfangreiche bedingte Anweisungen realisiert wird. Diese lassen sich gruppieren und in Verhaltensklassen auslagern.
- das Verhalten einer Klasse soll sich zur Laufzeit (von außen steuerbar) ändern

Strategy

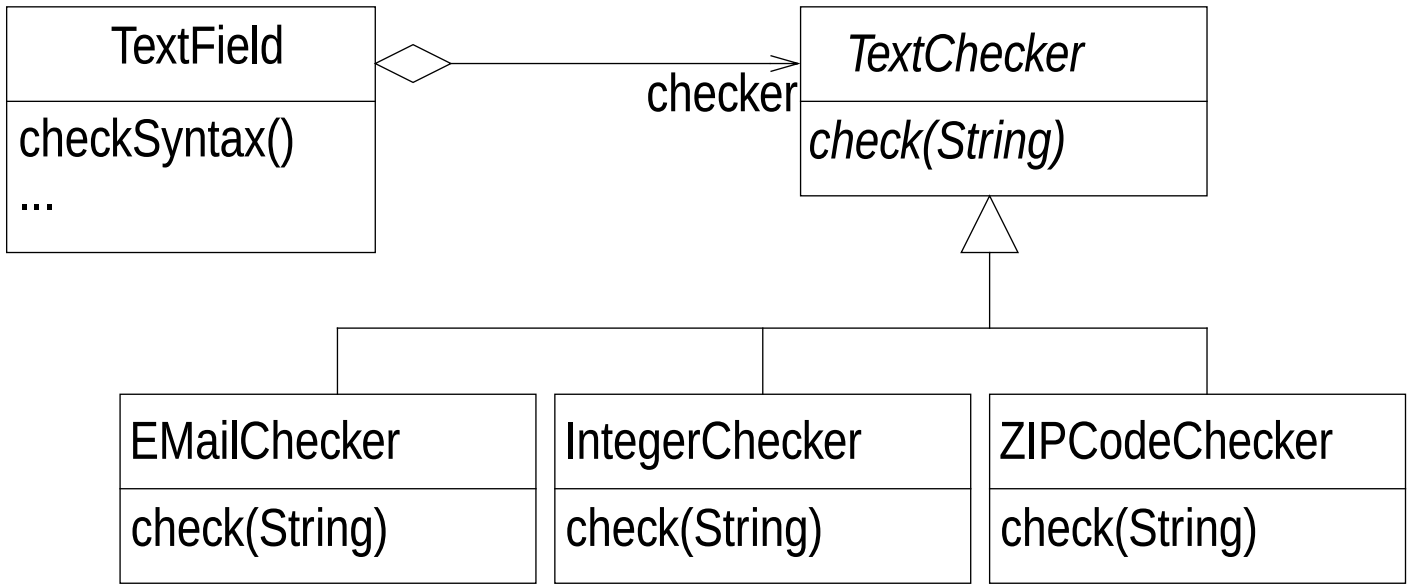
Struktur



- `contextInterface()` delegiert spezifische Aufgaben an `strategy.algorithmInterface()`
- dazu wird entweder das Objekt selbst (`this`) oder nur die erforderlichen Daten übergeben (geringere Kopplung!)

Strategy

Anwendungsbeispiel TextFieldChecker



- `checkSyntax()` von `TextField` liest den eingegebenen Text aus und übergibt ihn zur Prüfung an `checker.check()`
- `TextChecker`-Klassen lassen sich auch in anderen Bereichen wiederverwenden

Strategy

Weitere Anwendungsbeispiele

- Java-Bibliotheken
  - Layout-Manager von Komponenten
  - EventListener
- Comparator, z.B. für Elemente sortierter Listen
- Textformatierungsalgorithmen (z.B. einfach, TeX)
- Routing-Algorithmen für Schaltnetz-Layout
- Speicherallokationsstrategien für Standardkomponenten in einer Klassenbibliothek (unmanaged, managed, controlled)

Strategy

Vor- und Nachteile

- ▲ Familien verwandter Algorithmen als Strategy-Hierarchie darstellbar
- ▲ Alternative zu vielen Subklassen mit unterschiedlichem Verhalten
- ▲ erlaubt Trennung zwischen Vorgehensweise und Implementierung
- ▲ es können unterschiedliche Implementierungen desselben Verhaltens angeboten werden (z.B. zeit- oder platzoptimiert)
- ▲ Verwender können eine Klasse mit Hilfe von Strategies konfigurieren
- ▼ Dazu müssen sie allerdings über die Strategies und ihre Semantik Bescheid wissen (d.h. Offenlegen eines Teils der Implementierung)
- ▼ Kommunikations-Overhead zwischen Kontext und Strategy
- ▼ Noch mehr Objekte im System

Iterator

Problem

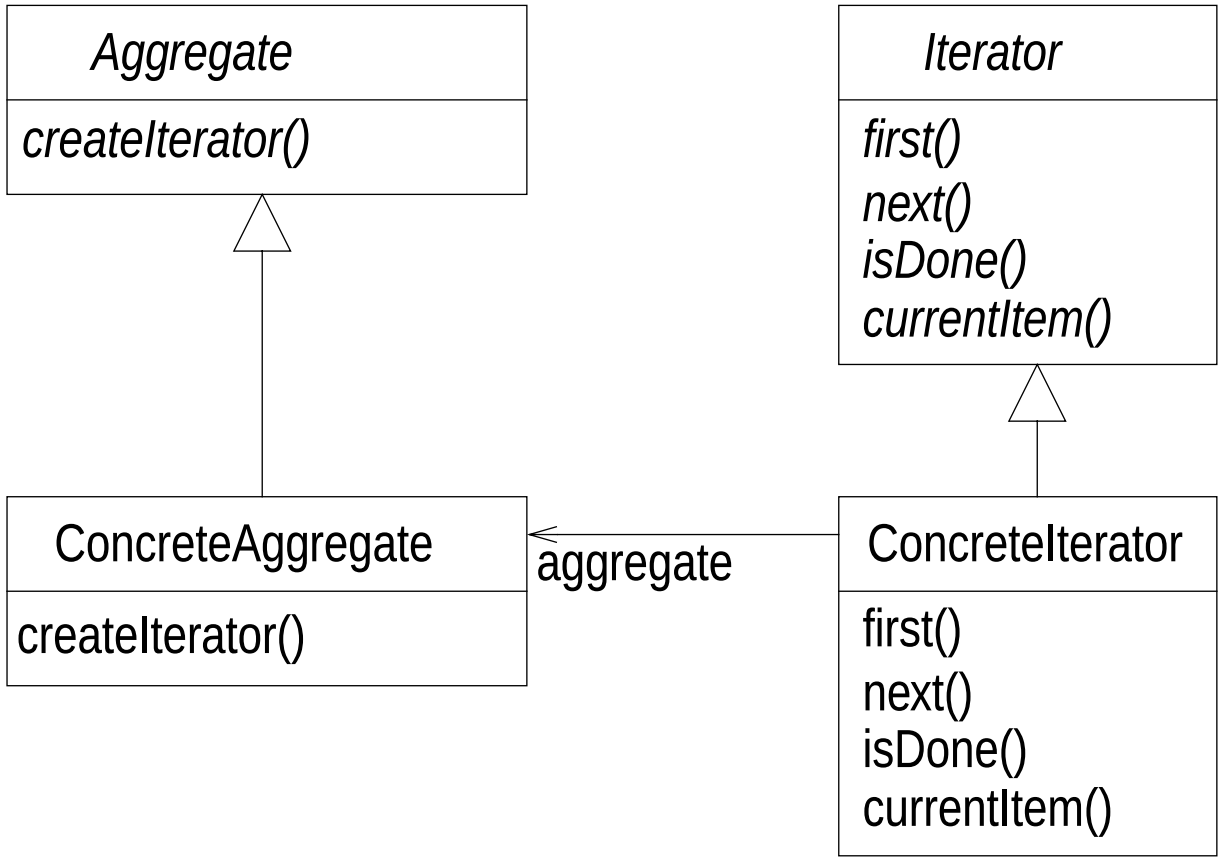
- es soll über die Elemente einer komplexen Datenstruktur (z.B. Liste, Baum) iteriert werden
- es soll möglich sein, mehrere Iterationen gleichzeitig durchführen zu können
- die interne Struktur der Datenstruktur soll trotzdem verborgen bleiben

- Lösung 1: Erweitere die Schnittstelle der Datenstruktur um Methoden zur Iteration (Cursor-Mechanismus)
- Lösung 2: Trenne die Datenstruktur von ihrem Iterator

Iterator

Struktur

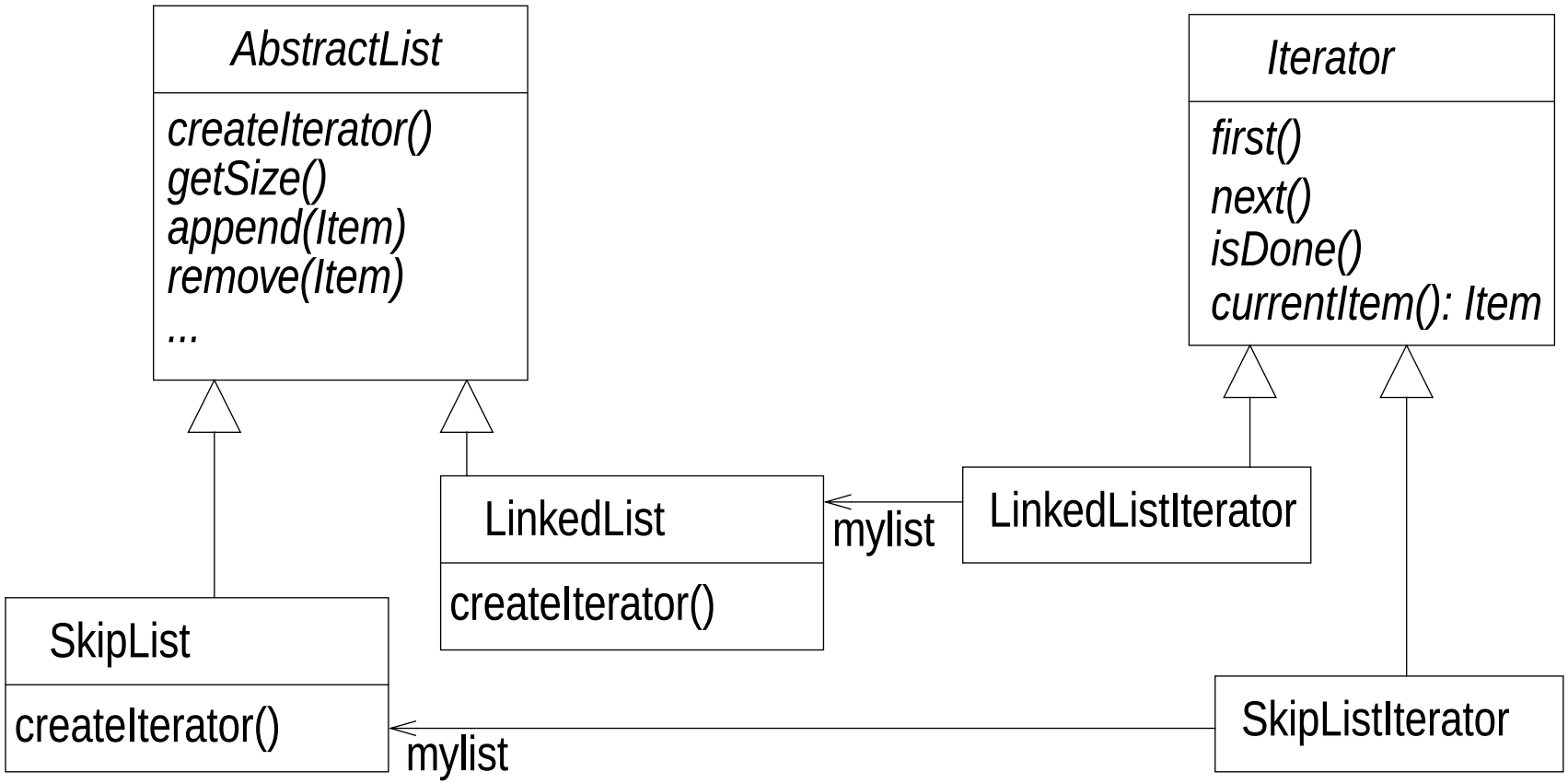
- Datenstruktur (Aggregate) stellt eine Factory Method für Iteratoren zur Verfügung
- konkreter Iterator ist mit seiner Datenstruktur assoziiert



Iterator

Polymorphe Iteration

Verschiedene Implementierungen einer Datenstruktur haben die gleiche Schnittstelle (inklusive zugehörigem Iterator)



Iterator

Verwendung

```
List list = new List();
Iterator iter = list.createIterator();
iter.first();
while (!iter.isDone()) {
   Item item = iter.currentItem();
   // tue etwas mit item, z.B.
   if (item.getColor() == GREEN) {
      item.setColor(RED);
   }
   iter.next();
}

// typische C++ Aufschreibung (nach Gamma et al., 1995)
for(iter.first(); !iter.isDone(); iter.next()) { ... }
```

Iterator

Bemerkungen

- Bei getypten Sprachen spielt der Rückgabetyp von `currentItem()` eine Rolle.
  - läßt sich am elegantesten durch generische Typparameter lösen, mit denen Datenstruktur und Iterator instantiiert werden, z.B. `List<Order>`, `ListIterator<Order>`
  - alternativ kann mit einer gemeinsamen Oberklasse der Elementtypen gearbeitet werden (notfalls Object)
- Iteratoren können verschiedene Iterationsstrategien auf derselben Datenstruktur implementieren (z.B. inorder, preorder, postorder)

Iterator

Varianten (1)

Kontrolle über die Iteration

- externer Iterator: die Steuerung der Iteration liegt beim Verwender, der Iterator stellt Navigationsoperationen auf der Datenstruktur zur Verfügung → flexibel, fehlerträchtig
- interner Iterator: führt (automatisch) eine gegebene Operation auf allen Elementen der Datenstruktur aus, Steuerung der Iteration wird vor Verwender verborgen → höhere Abstraktion, unflexibel

Iterator

Realisierung interner Iteratoren (1)

- Operation als Parameter vom Interface-Typ IteratorAction (Strategy Pattern)

```
interface IteratorAction {
  public void /*boolean*/ handleItem(Item i);
}
class InternalIterator {
  public void doIteration(IteratorAction action);
}
```

- `handleItem()` kann boolschen Wert zurückgeben, um Iteration abbrechen zu können.
- alternativ InternalIterator als generische Klasse realisieren, die mit einer IteratorAction instantiiert wird.

▼ für jede Aktion muß neue Klasse IteratorAction implementieren

Iterator

Realisierung interner Iteratoren (2)

Alternative: `doIteration` als Template Method realisieren, `handleItem` und `resetIterator` als primitive Methoden dazu

```
abstract class InternalIterator {
  public void doIteration() {
    resetIterator();
    // über alle Elemente x iterieren, dabei jeweils
      handleItem(x); // aufrufen
  }
  protected void resetIterator() { } // Default-Impl.
  abstract protected void handleItem(Item i);
}
```

- für jede Aktion muß eine eigene Unterklasse von InternalIterator geschrieben werden.

Iterator

Varianten (2)

Grundlage der Iteration

- Iterator arbeitet mit Kopie der Datenstruktur, damit er von Änderungen nicht betroffen wird
  - deep copy: Elemente werden ebenfalls kopiert
  - shallow copy: Es werden nur Referenzen auf die Elemente kopiert

  Problem: Änderungsoperationen bei der Datenstruktur ändern nicht das Original, Elemente des Originals ändern sich nur bei shallow copy
- Iterator arbeitet auf dem Original der Datenstruktur
  - ohne Änderungsschutz $\rightarrow$ Risiko des Verwenders
  - robuster Iterator $\rightarrow$ u.U. aufwendig zu implementieren

Iterator

Varianten (3)

Definition des Iterationsalgorithmus

- durch Iterator
  aber: Iteratoren brauchen (lesenden) Zugriff auf die interne Repräsentation der Datenstruktur (C++: friend, Java: gleiches Package oder innere Klasse). Die benötigte Sicht kann auch durch eine spezielle („private“) Schnittstelle realisiert werden.
- durch die Datenstruktur
  Cursor-Konzept: Iterator speichert nur momentane Iterationsposition und bittet Datenstruktur um Auskünfte und Änderungen. Dazu bietet Datenstruktur spezielle Operationen an.

Null-Iterator

Iterator für Blattknoten einer Composite-Struktur, bei dem `isDone()` immer `false` liefert. Erlaubt uniforme Behandlung aller Knoten.

Iterator

Iteratorlösung im Java Development Kit

- Interface java.util.Enumeration:

```
public interface Enumeration {
  boolean hasMoreElements();
  Object nextElement();
}
```

- Datenstrukturen geben zur Iteration ein Enumeration-Objekt zurück, z.B. Vector, Hashtable

```
Vector vec;
Enumeration enum = vec.elements();
while(enum.hasMoreElements()) {
  // Cast nötig, da nextElement Objects zurückgibt
  Item element = (Item) nextElement();
  // ... tue etwas mit element
}
```

Iterator

Vor- und Nachteile

▲ Trennung der Zuständigkeiten zwischen Datenstruktur und Iterator, d.h. Schnittstelle und Implementierung der Datenstruktur vereinfacht

▲ mehrere Iteratoren gleichzeitig auf derselben Datenstruktur möglich

▲ polymorphe Iteratoren erlauben Abstraktion von der konkreten Implementierung der Datenstruktur

▼ Iteratoren benötigen Zugriff auf internen Zustand der Datenstruktur (Durchbrechen der Kapselung, enge Kopplung)

▼ mehr Objekte (und Klassen) im System

Composite

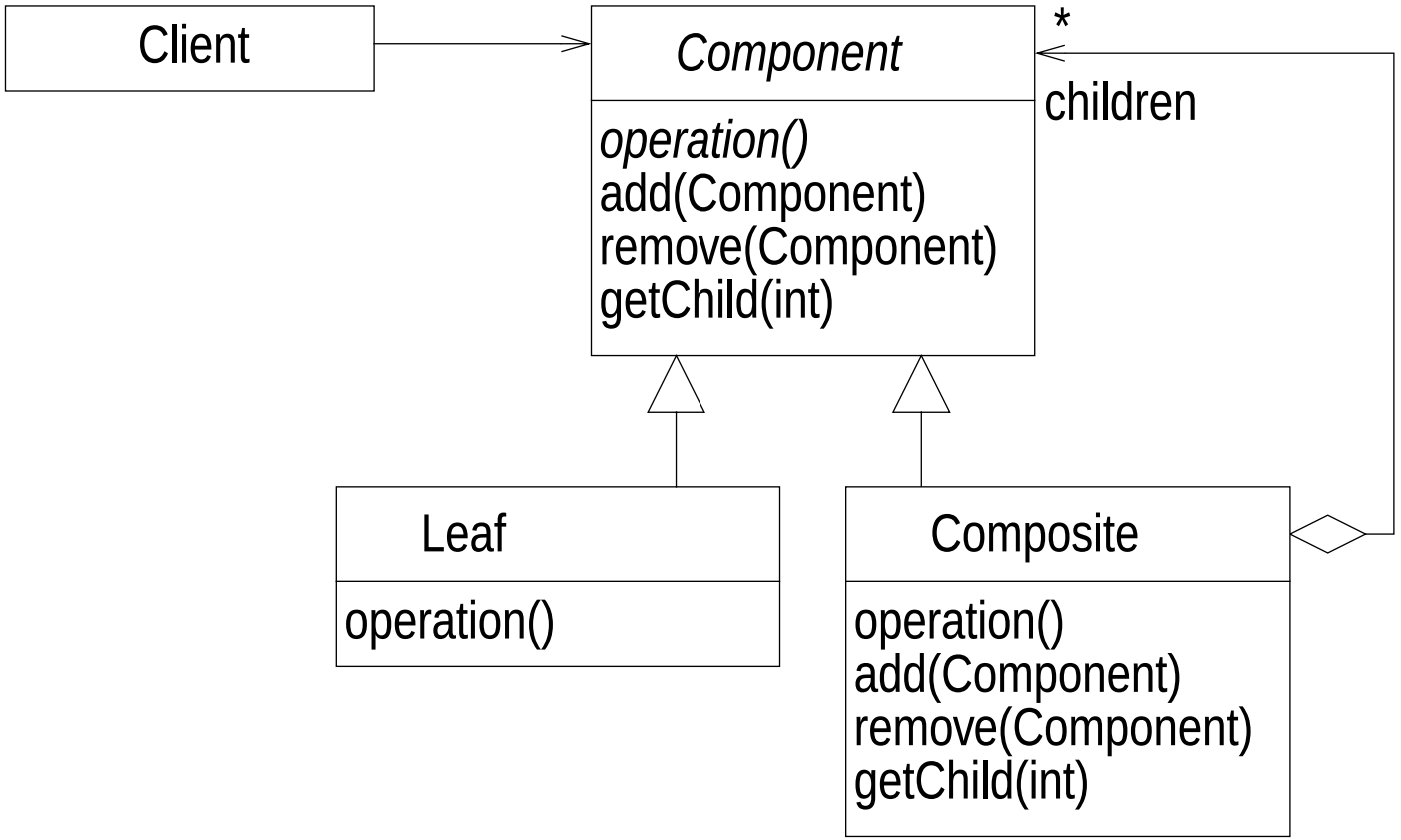
Problem

- es sollen hierarchische Teil-Ganzes-Beziehungen modelliert werden
- einzelne Objekte (Primitive) und zusammengesetzte Objekte (Container) sollen für den Verwender die gleiche Schnittstelle haben
- Beispiele: gruppierbare graphische Objekte, Parse-Baum, GUI-Elemente

- Lösung: gemeinsame Oberklasse für Primitive und Container, die beide Eigenschaften in sich vereinigt

Composite

Struktur



Composite

Bemerkungen (1)

- Composite reicht normale Methodenaufrufe (`operation()`) in der Regel an alle Kinder durch, kann aber auch noch zusätzliche Operationen durchführen
- die Aufnahme von Composite-Operationen in Component ist praktisch (einheitl. Schnittstelle), aber auch problematisch, da ihr Aufruf auf einen Leaf nicht sinnvoll ist.
  - nötig ist zumindest eine Default-Implementierung dieser Operationen (z.B. Exception auslösen), ansonsten müßte jede Leaf-Klasse das selbst tun
  - ein Verschieben in Composite kann sinnvoll sein (höhere Sicherheit).

  In beiden Fällen muß der Verwender darauf achten, nur bei Composite-Objekten `add()` etc. aufzurufen. Nur bedarf es beim zweiten Fall dazu eines expliziten Casts.

Composite

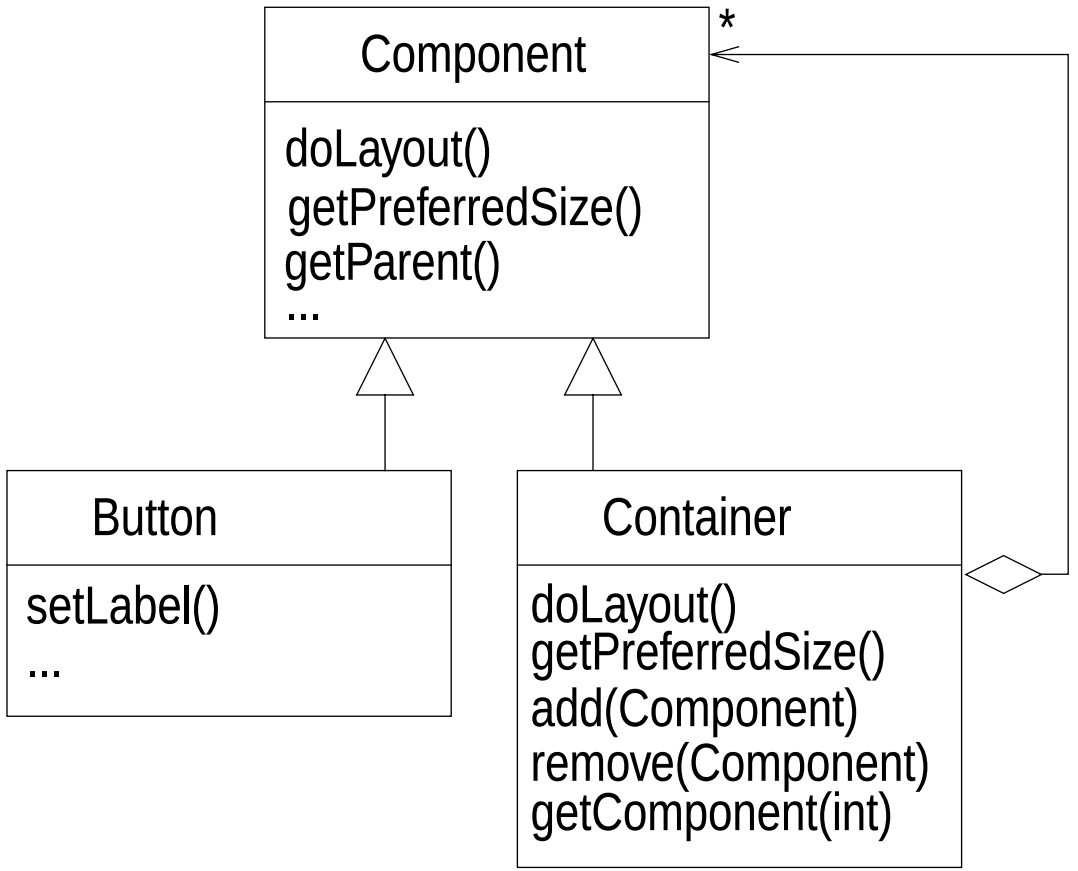
Bemerkungen (2)

- Component kann um `getParent()` erweitert werden, um den übergeordneten Composite referenzieren zu können.
  Problem: wer enthält den obersten Composite?
- statt mittels `getChild()` die Kinder einzeln verfügbar zu machen, kann nach außen auch ein externer und/oder interner Iterator über die Kinder zur Verfügung gestellt werden.

Composite

Beispiel java.awt.Component

- Klassenhierarchie für GUI-Elemente (Button, TextField, Frame, ...)
- nur gemeinsame Operationen werden in Component deklariert
- einfache GUI-Elemente (Button, ...) werden direkt von Component abgeleitet
- Container (Panel, Frame, ...) werden von Container abgeleitet



Composite

Vor- und Nachteile

- ▲ gleiche Schnittstelle für einfache und zusammengesetzte Objekte
- ▲ ermöglicht einfachere Verwendung
- ▲ Struktur kann leicht um neue Klassen erweitert werden

- ▼ Composite-Operationen in der Component-Klasse problematisch
- ▼ Restriktionen bei der Zusammensetzung lassen sich schwierig realisieren (z.B. Einschränkung auf bestimmte Typen läßt sich nur zur Laufzeit prüfen)

Weitere Muster

Kurzbeschreibungen (1)

Abstract Factory: stellt eine Schnittstelle zur Erzeugung von Familien verwandter oder abhängiger Objekte zur Verfügung, ohne ihre konkrete Klasse anzugeben.
Beispiel: WidgetFactory für GUI-Elemente

Bridge: entkoppelt eine Abstraktion von ihrer Implementierung, so daß die beiden unabhängig voneinander variieren können.
Beispiel: Peers für GUI-Elemente im Java AWT 1.0

Builder: trennt die Erstellung eines komplexen Objekts von seiner Repräsentation, so daß derselbe Erstellungsprozeß verschiedene Repräsentationen erzeugen kann.
Beispiel: Textformat-Konverter-Backends

Weitere Muster

Kurzbeschreibungen (2)

Chain of Responsibility: vermeidet die Kopplung vom Sender einer Anfrage an den Empfänger, indem einer Reihe von Objekten die Chance gegeben wird, die Anfrage zu bearbeiten. Die Empfängerobjekte werden verkettet und die Anfrage wird die Kette entlang weitergegeben, bis ein Objekt sie behandelt.
Beispiel: Event-Handling in einer Hierarchie von Objekten, z.B. in einer graphischen Benutzungsoberfläche

Command: kapselt einen Befehl als Objekt. Dadurch können Verwender mit verschiedenen Befehlen parametrisiert werden. Außerdem können Befehle protokolliert und rückgängig gemacht werden.
Beispiel: Operationen einer Textverarbeitung

Weitere Muster

Kurzbeschreibungen (3)

Decorator: erweitert ein Objekt zur Laufzeit um zusätzliche Funktionalität. Flexible Alternative zur Vererbung.
Beispiel: Verarbeitung von Ein-/Ausgabeströmen in java.io

Facade: stellt eine einheitliche Schnittstelle zu einem Subsystem zur Verfügung. Durch die höhere Abstraktionsebene der Schnittstelle kann das Subsystem einfacher verwendet werden.
Beispiel: Objekt, das ein Subsystem zum Einlesen einer Datei kapselt

Flyweight: unterstützt die effiziente Verwendung einer großen Menge feingranularer Objekte durch gemeinsame Nutzung (Sharing).
Beispiel: Objekte für die (attributierten) Zeichen in einer Textverarbeitung

Weitere Muster

Kurzbeschreibungen (4)

Interpreter: definiert eine Repräsentation der Grammatik einer gegebenen Sprache und stellt einen Interpreter zur Verfügung, der Sätze der Sprache in dieser Repräsentation interpretieren kann.
Beispiel: Suche nach regulären Ausdrücken

Mediator: definiert ein Objekt, das die Art und Weise der Interaktion einer Menge von Objekten kapselt. Dadurch sind diese Objekte nur lose miteinander gekoppelt. Außerdem kann die Art und Weise der Interaktion leichter geändert werden.
Beispiel: Dialoge mit kompliziertem Zusammenspiel der Elemente

Memento: erlaubt es, den internen Zustand eines Objekts zu externalisieren (und zu speichern), so daß das Objekt später wiederhergestellt werden kann.
Beispiel: Kontext-Informationen für Undo

Weitere Muster

Kurzbeschreibungen (5)

Prototype: Objekte einer bestimmten Art werden durch Kopieren (Klonen) eines prototypischen Objekts erzeugt, das zu diesem Zweck angegeben wurde.
Beispiel: Prototyp-Objekte für die Zeichenelemente eines graphischen Editors

Proxy: Stellvertreter für ein anderes Objekt, um den Zugang zu diesem erleichtern oder kontrollieren zu können.
Beispiel: Platzhalter für Bilder in einer Textverarbeitung

State: erlaubt einem Objekt, sein Verhalten zu ändern, wenn sich sein interner Zustand ändert. Das Objekt ändert dabei scheinbar seine Klassenzugehörigkeit.
Beispiel: Netzwerkverbindungen mit Zuständen wie wartend, etabliert, geschlossen

Weitere Muster

Kurzbeschreibungen (6)

Visitor: repräsentiert eine Operation, die auf den Elementen einer Objektstruktur durchgeführt werden soll. Die Operation kann auf diese Weise definiert werden, ohne daß die Elementklassen dafür erweitert werden müßten.
Beispiel: Operationen zur semantischen Analyse/Optimierung/Code-Generierung auf einem Syntaxbaum

Entwurfsmuster nach Gamma et al. (1995)

Zusammenhänge

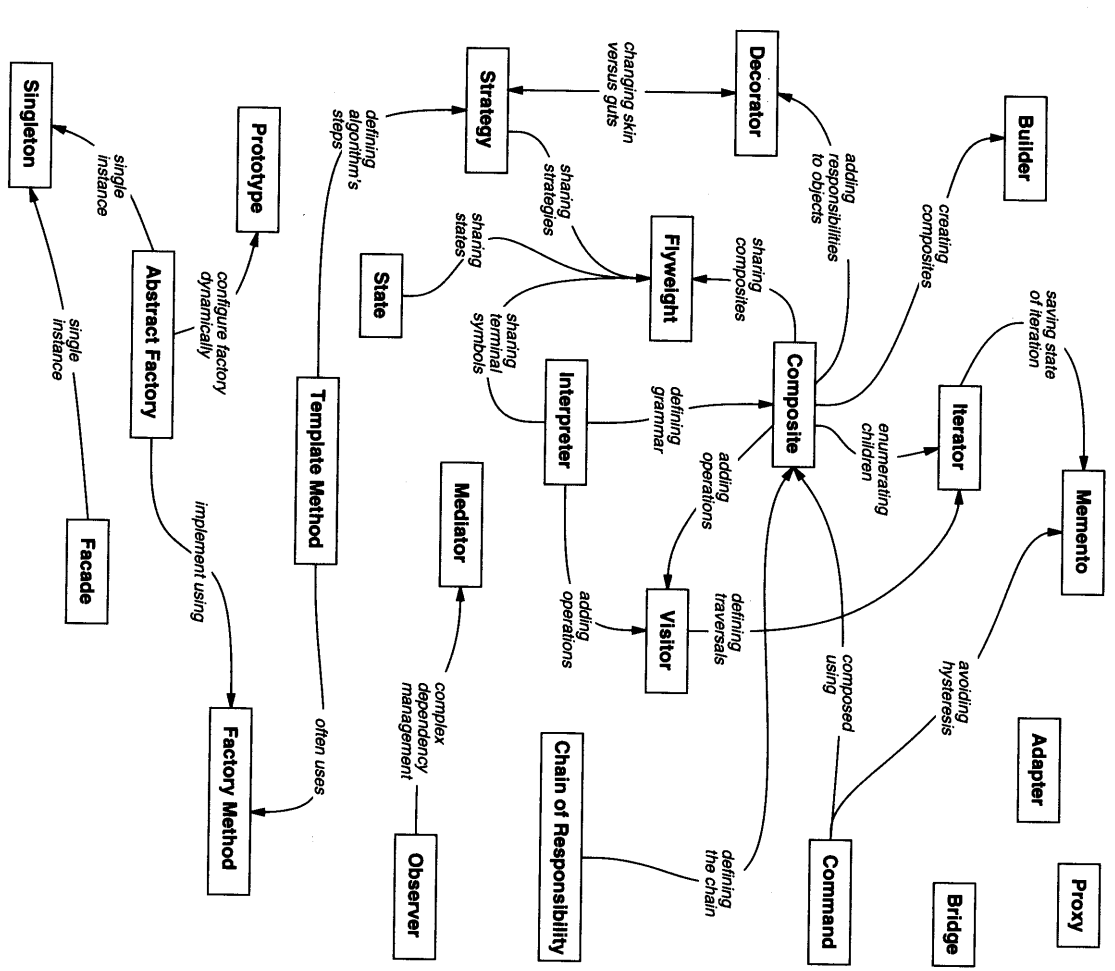
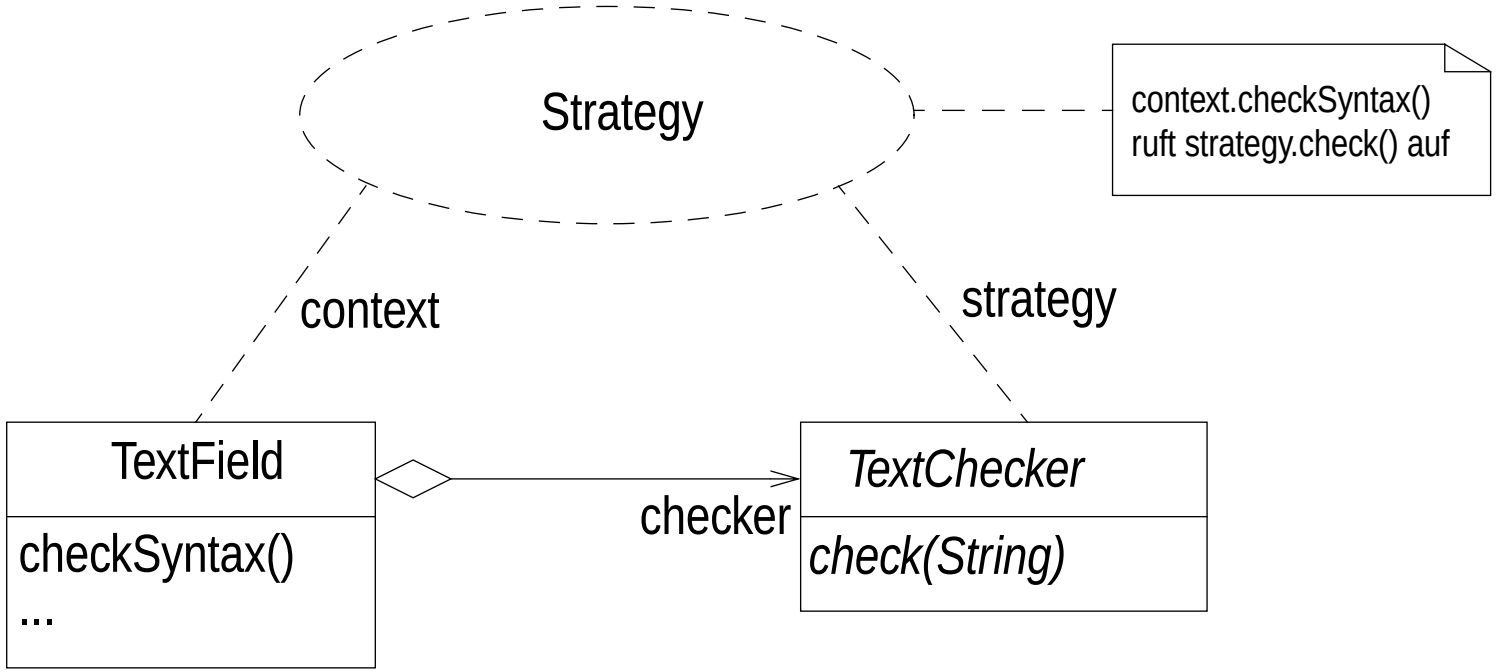


Figure 1.1: Design pattern relationships

Entwurfsmuster

UML-Darstellung



Kapitel 6:
Idiome

- Begriff
- Beispiele für Idiome

Idiome

Begriff

Buschmann et al. (1996):

An idiom is a **low-level** pattern **specific to a programming language**. An idiom describes how to **implement** particular aspects of components or the relationship between them using the features of the given language.

Eigenschaften

- niedrige Abstraktionsebene (Feinentwurf und Implementierung)
- programmiersprachenspezifisch
- paradigmenspezifisch

Idiome

Beispiele

- Programmierkonventionen, z.B. Formatierung, Kommentierung und Namensgebung
- typische Code-Strukturen, z.B. minimale Klassenschnittstelle
- typische Lösungsansätze für durch die Sprache nicht direkt unterstützte Probleme, z.B. Speicherverwaltung
- konkrete Implementierung eines Entwurfsmusters

Beispiele für Idiome

Übersicht

- aus Beck (1997) (für Smalltalk)
  - Indented Control Flow
- aus Coplien (1992) (für C++)
  - Orthodoxe kanonische Form (für Klassen)
  - Counted Pointer
  - Counted Body
- aus Buschmann et al. (1996)
  - Singleton in C++
  - Singleton in Smalltalk

Indented Control Flow

Idiom zur Formatierung

Wie sollen Aufrufe von Methoden in Smalltalk eingerückt werden?

- Aufrufe mit keinem oder einem Argument:

```
foo isNil
2 + 3
a < b ifTrue: [ ... ]
```

- Aufrufe mit mehr als einem Argument:

```
a < b
   ifTrue: [ ... ]
   ifFalse: [ ... ]
```

Orthodoxe kanonische Form

Definition

Eine Klasse hat immer mindestens die folgenden Bestandteile:

- Default-Konstruktor

```
String()
```

- Copy-Konstruktor

```
String(const String& s)
```

- Zuweisungsoperator

```
String& operator=(const String& s)
```

- Destruktor

```
~String()
```

Grund: Konstruktor und Destruktor dienen der Ressourcenverwaltung. Default-Copy-Konstruktor und Default-Zuweisung haben attributweise shallow copy-Semantik – die ist manchmal die falsche.

Orthodoxe kanonische Form

Implementierungen

```
String::String() {
   rep = new char[1]; rep[0] = '\0';
}
String::String(const String& s) {
   rep = new char[s.length()+1]; ::strcpy(rep, s.rep);
}
String& String::operator=(const String& s) {
   if (rep != s.rep) { // Sonderfall s == *this
      delete[] rep;
      rep = new char[s.lenght()+1]; ::strcpy(rep,s.rep);
   }
   return *this;
}
String::~String() { delete[] rep; }
```

Orthodoxe kanonische Form

Anwendbarkeit

Idiom *muß* angewendet werden, wenn

- Zuweisung erlaubt sein soll oder Objekte als Wertparameter übergeben werden sollen

und

- das Objekt Zeiger auf Objekte (mit Reference Count) enthält oder der Destruktor eine delete-Operation auf einem Attribut durchführt.

Idiom *sollte* für alle nichttrivialen Klassen angewendet werden. Dann kann bei allen Klassen davon ausgegangen werden, daß ihr Speicher-Management richtig funktioniert.

Counted Pointer

Problem

C++ stellt keine automatische Speicherverwaltung (Garbage Collection) zur Verfügung. Diese muß von Programmierer explizit definiert werden, wenn Objekte oder Attributwerte dynamisch allokiert werden. Die entstehenden Referenzen stellen ein Problem dar.

Einflußfaktoren

- es ist für Objekte unangemessen, nur als Wertparameter übergeben zu werden
- Objekte können von mehreren Verwendern gleichzeitig referenziert werden
- dangling references sollen vermieden werden
- nicht mehr benötigte Objekte (d.h. sie werden nicht mehr referenziert) sollen automatisch deallokiert werden
- das Objekt soll sich selbst um die Speicherverwaltung kümmern

Counted Pointer

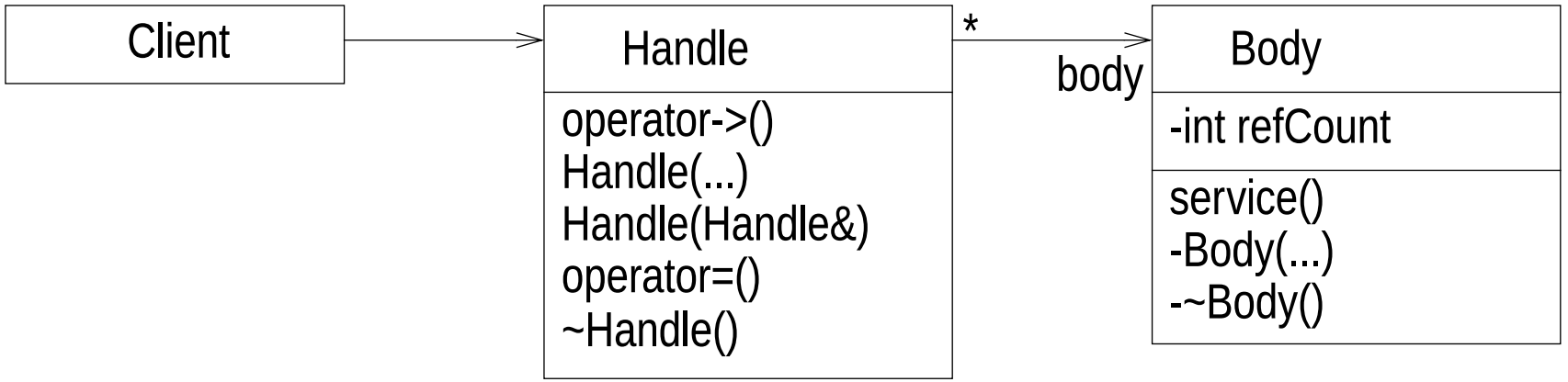
Lösung

Es wird ein Reference Count-Mechanismus aufgebaut:

- die eigentliche Klasse Body wird um einen Referenzenzähler erweitert
- die Klasse Handle kapselt Instanzen von Body.
  - stellt die einzige Zugriffsmöglichkeit dar
  - Handle enthält eine Referenz auf ein Body-Objekt
  - Handle-Objekte können nun als Wertparameter im Programm verwendet werden (automatische Allokation/Deallokation)
  - `operator->` von Handle wird redefiniert (macht Indirektion über Handle-Objekt transparent: Handles können wie Zeiger auf Body benutzt werden: „Smart Pointers“)

Counted Pointer

Struktur



- Handle folgt der orthodoxen kanonischen Form
- Konstruktor und Destruktor von Body sind privat, für Handle durch friend-Deklaration aber zugänglich

Counted Pointer

Implementierung von Handle

```
class Handle {
   Body* body;
public:
   Handle(...) {
      body = new Body(...); body->refCount = 1; }
   Handle(const Handle& h) {
      body = h.body; body->refCount++; }
   Handle& operator=(const Handle& h) {
      h.body->refCount++;
      if (--body->refCount <= 0) delete body;
      body = h.body; return *this; }
   ~Handle() {
      if (--body->refCount <= 0) delete body; }
   Body* operator->() { return body; }
}
```

Counted Pointer

Beispiel

```
void doSomething(Handle x) { ... }
void main {
  Handle h(...); // refCount = 1
  { Handle g(h); // Copy-Konstruktor, refCount = 2
    h->service();
    g->service();
  } // g wird deallokiert, da nicht mehr gültig
  h->service(); // weiterhin möglich, refCount = 1
  Handle f = h; // operator=(), refCount = 2
  f->service();
  delete f; // Destruktor, refCount = 1;
  doSomething(h); // h wird kopiert, refCount = 2
} // h wird deallokiert, refCount = 0,
  // d.h. h.body wird deallokiert
```

Counted Pointer

Varianten

- Counted Body
  - aus Effizienzgründen (um Kopien zu vermeiden) werden große Objekte gemeinsam genutzt, so lange das möglich ist.
  - Trotzdem sieht es für den Verwender so aus, als hätte er eine eigene Kopie, da bei Modifikationen eine echte Kopie gemacht wird (wenn es mehr als eine Referenz gibt)
  - Um die gewünschte Transparenz zu bekommen, wird nicht `operator->()` in Handle überladen, sondern die gesamte Schnittstelle von Body nach Handle kopiert (Decorator Pattern)
- falls die Klasse Body nicht verändert werden kann:
  - in Adapter mit `refCount` verpacken, oder
  - `refCount` in neues Objekt verpacken, das parallel zu Body von Handle verwaltet wird

Singleton

C++

```
class Singleton {
   static Singleton* theInstance = null;
   Singleton() { ... }
public:
   static public Singleton* getInstance() {
     if (!theInstance)
        theInstance = new Singleton();
     return theInstance;
   }
   ... // Instanzvariablen und -methoden
}
```

Singleton

Smalltalk

Klasse Singleton mit Instanzvariable `TheInstance`.

Implementierung der Methoden:

```
new
   self error: 'cannot create new object'
getInstance
   TheInstance isNil ifTrue: [TheInstance := super new]
   ^ TheInstance
```

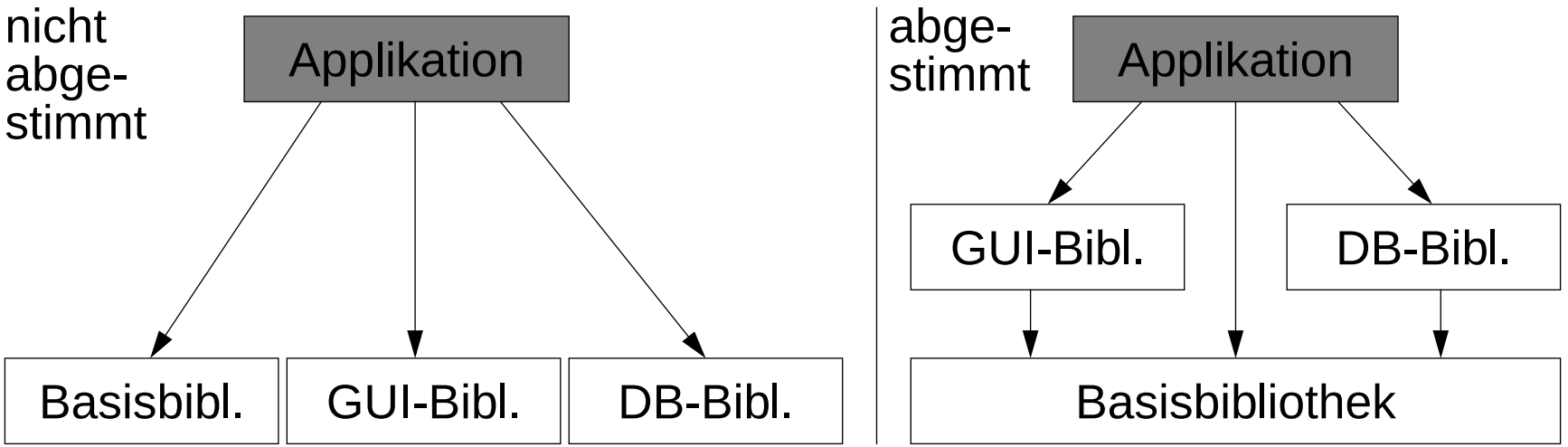
Kapitel 7: Bibliotheken/Frameworks

- Begriffe (Klassifikation nach Kilberth et al., 1994)
  - Bausteinsammlung
  - Framework
- Beispiel eines Frameworks: ET++
- Metamuster nach Pree (1995)

Begriffe

Bausteinsammlung

- enthält Klassen, die anwendungsunabhängige Leistungen erbringen (Bausteine)
- besteht aus einem oder mehreren Clustern (semant. Gruppierung)
- generelle Klassen: Basisbibliothek (z.B. Booch Components, MFC)
- spezielle Klassen: z.B. GUI-Bibliothek, DB-Bibliothek
- Schwerpunkt: Wiederverwendung von Code auf Klassenebene



Begriffe

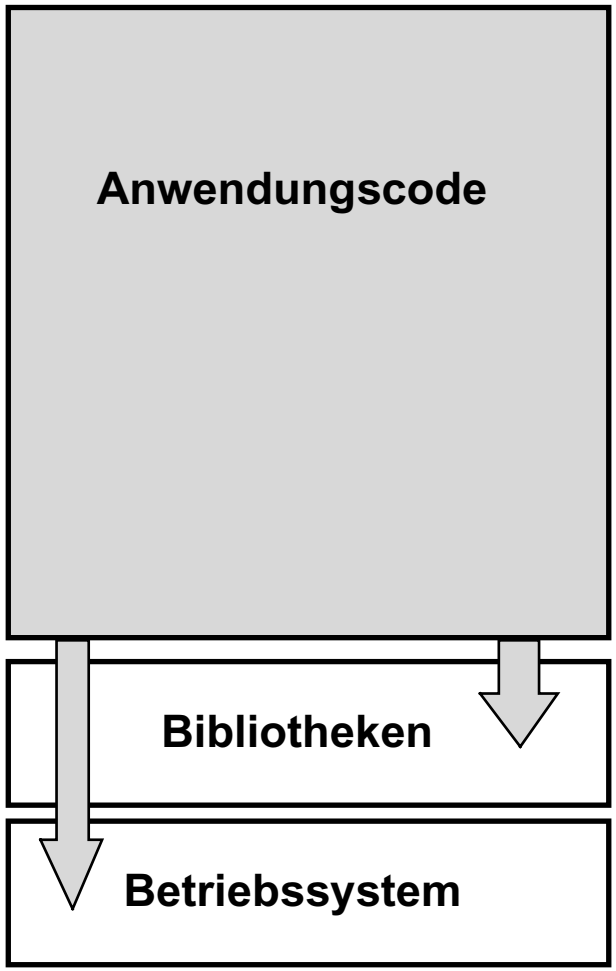
Framework (Rahmenwerk)

- unvollständiges Software-System (oder -Subsystem), das zu instantiieren ist (Halbfabrikat)
- definiert eine bestimmte Architektur für eine Familie von Systemen (Makro-Architektur, frozen spots)
- stellt grundlegende Bausteine zur Verfügung (teilweise in Form abstrakter Klassen = Schnittstellen)
- definiert die Stellen, an denen Anpassungen für das konkrete System gemacht werden sollen (hot spots, hooks)
- Schwerpunkt: Wiederverwendung des Entwurfs
- Application Framework: Framework für Systeme eines bestimmten Anwendungsbereiches (z.B. Bankensofware: ProFIS)

Kontrollfluß in Systemen

Prozedurale Betrachtung

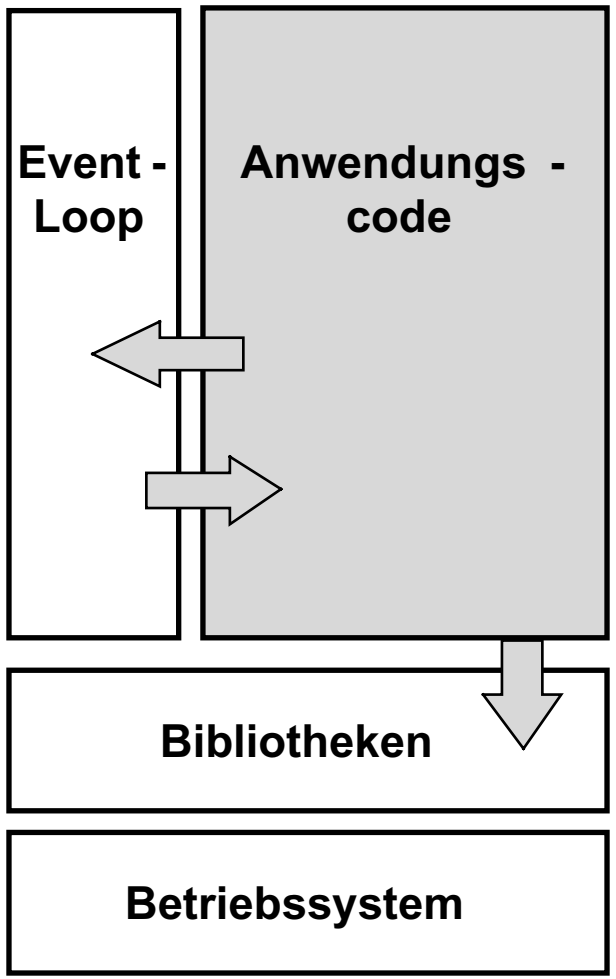
- gesamter Kontrollfluß ist im Anwendungscode
- Operationen der darunterliegenden Schichten werden aus der Kontrollstruktur des Anwendungscode heraus aufgerufen
- Programmierer ist „Herr des Kontrollflusses“



Kontrollfluß in Systemen

Event Loop

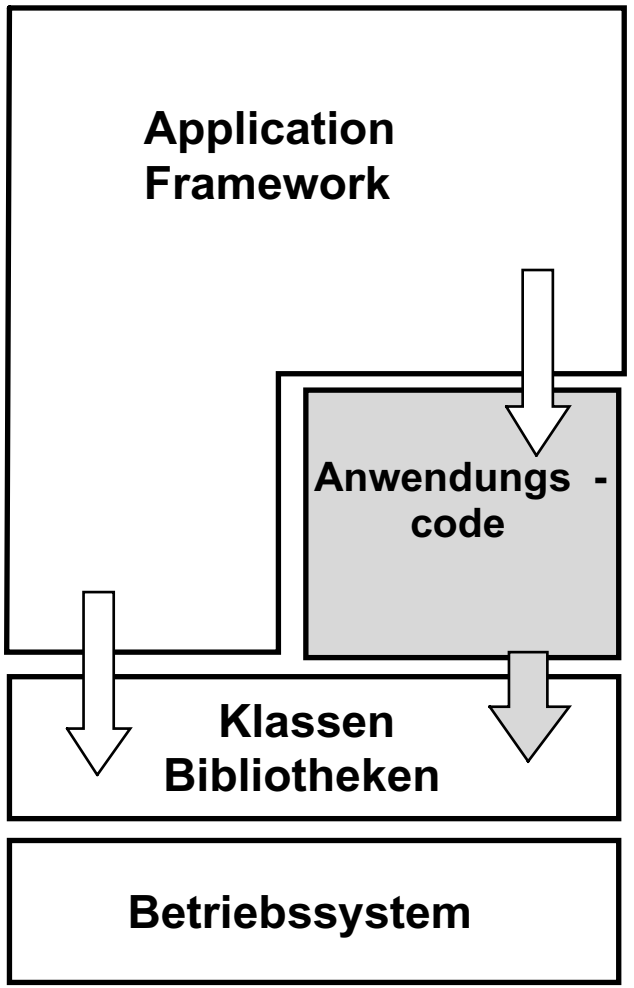
- interaktiv bedienbare Systeme führen zu einen neuen Strategie für den Kontrollfluß
- Event-Loop wird von der Entwicklungsplattform bereitgestellt
- kontrolliert die Interaktion mit dem Benutzer und ruft spezifische Teile des Anwendungscodes auf (Callbacks)
- z.B. Macintosh Toolbox, ET



Kontrollfluß in Systemen

Framework

- der zentrale und überwiegende Teil des Kontrollflusses ist im Framework implementiert
- Framework ruft den anwendungs-spezifischen Code aus seinem Kontrollfluß auf
- Programmierer ist nicht mehr „Herr des Kontrollflusses“
- Programmierer muß die zentralen Mechanismen des Frameworks und die hot spots kennen



Klasse - Bausteinsammlung - Framework

Abgrenzung

Klasse

- „einfach“ zu entwerfen und zu implementieren, da für genau einen Zweck gedacht (nicht wiederverwendbar!)

Bausteinsammlung

- Entwurf und Implementierung ist nicht trivial, da für viele Anwendungssituationen gedacht
- entsteht iterativ und evolutionär

Framework

- verlangt hohes Wissen in Entwurf und Anwendungsgebiet
- mehrere Applikationen der Applikationsklasse sollten erstellt worden sein
- entsteht iterativ und evolutionär

Framework

Entstehung

- erst verwenden, dann wiederverwenden,
  d.h. Framework aus bestehenden Applikationen ableiten
- aus bestehenden Applikationen werden Gemeinsamkeiten herausgelöst und abstrahiert; dabei gehen auch Wartungserfahrungen ein.
- Stellen vorsehen, an denen anwendungsspezifischer Code integriert werden kann (hot spots, hooks).
- Mechanismen vorsehen, die integrierten anwendungsspezifischen Code verwenden
- Evolution durch Verwendung in verschiedenen Projekten:

  „Good frameworks are usually the result of many design iterations and a lot of hard work“ (Wirfs-Brock, Johnson, 1990)
- Framework-Entwurf ist ähnlich schwierig wie Sprachentwurf

Framework

Kriterien für Hot Spots (Pree, 1995, S. 228)

- Welche Aspekte unterscheiden sich von Anwendung zu Anwendung?
- Was ist der gewünschte Grad an Flexibilität?
- Muß sich das flexible Verhalten zur Laufzeit ändern können?

Beispiel

Berechnung der Rechnungssumme für Hotels

- die Parameter der Berechnung können sich ändern (z.B. Preise)
- die Art und Weise der Berechnung ist je nach Anwendung unterschiedlich
- die Art und Weise der Berechnung kann sich zur Laufzeit ändern (z.B. spezielle Angebote)

Framework

Verwendung

1. Eignung des Frameworks für die geplante Applikation prüfen
2. In das Frameworks einarbeiten, dabei wichtig:
   - ❍ Dokumentation, insbesondere der hot spots
   - ❍ Hilfestellungen, z.B.
     - ☆ „Kochbuch“ mit „Rezepten“
     - ☆ Werkzeuge zur Instantiierung (z.B. Wizards)
3. Eigene Klassen (i.d.R. Subklassen von Framework-Klassen) schreiben und in das Framework integrieren
4. (Feedback an Framework-Autoren)

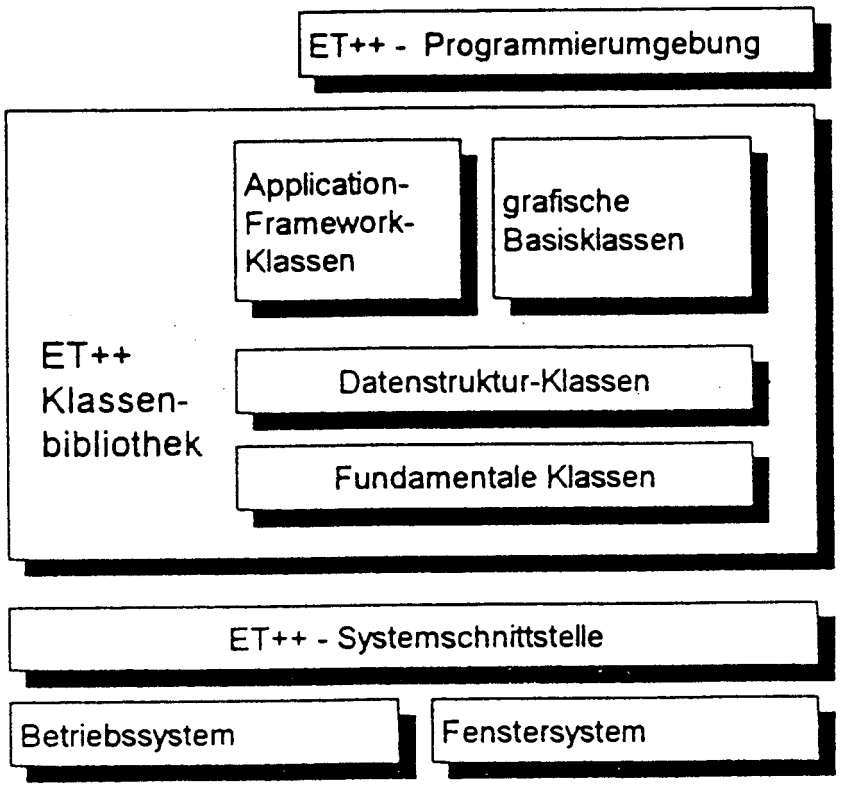
White-box vs. Black-box

- ❑ white-box: Verwender muß neue Klassen definieren (von innen)
- ❑ black-box: Werkzeug generiert neue Klassen (von außen)

ET++

Übersicht

- entwickelt von André Weinand und Erich Gamma 1987-1992 an der ETH Zürich
- Framework für Anwendungen mit GUI
- geschrieben in C++
- Plattformen: UNIX und Mac
- Source Code und Werkzeuge sind kostenlos verfügbar (ftp://ftp.ubilab.ch/pub/ET++)



ET++

Die Basisklasse Object

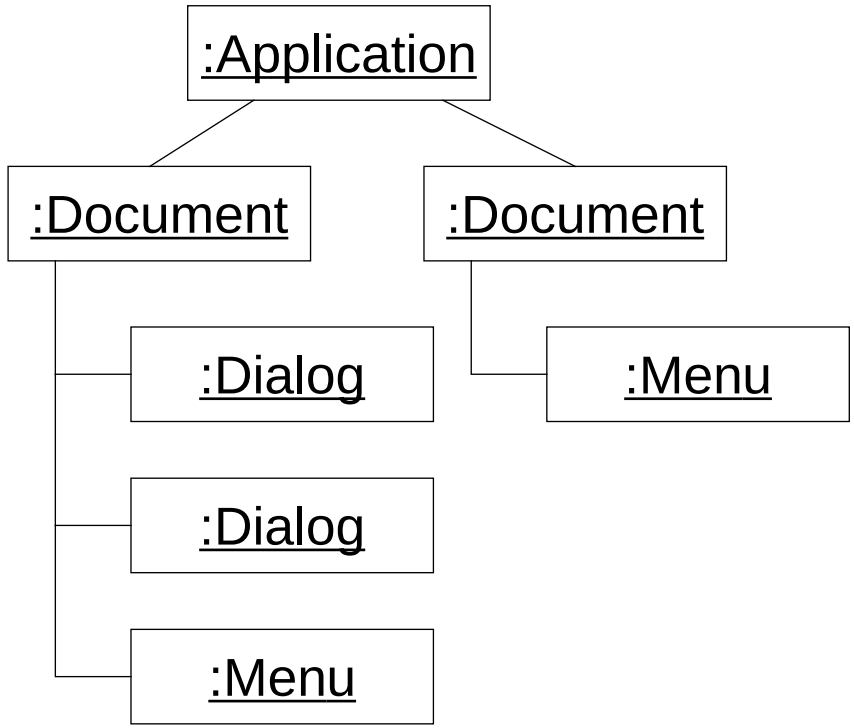
- Oberklasse aller Klassen in ET++
- Observer-Mechanismus (Methoden `Send`, `AddObserver`, `RemoveObserver` und `DoObserve`)
- Reflexion (Meta-Informationen, Methoden `IsKindOf`, `IsA`, `ClassName`)
- Serialisierung (Methoden `PrintOn`, `ReadFrom`)
- Vergleich, Kopie (Methode `DeepClone`)
- Elementtyp von vielen Datenstrukturen (z.B. ObjList)

- VObject ist Basisklasse für alle graphischen Klassen (z.B. GUI-Elemente)

ET++

Die zentralen Framework-Klassen (1)

- **Manager**: verwaltet eine Menge von Managern
- Unterklassen von Manager:
  - **Application**: repräsentiert die Anwendung (i.d.R. nur eine Instanz). Verwaltet eine Menge von Submanagern (Dokumente)
  - **Document**: kapselt eine Datenstruktur (Modell) der Anwendung
  - **Dialog**
  - **Menu**



ET++

Die zentralen Framework-Klassen (2)

- **Data**: repräsentiert Daten aus externen Datenquellen (Datei, Datenbank)
- **View**: stellt ein Dokument für den Benutzer dar. Ein Dokument kann beliebig viele Views haben.
- **Window**: repräsentiert ein Fenster, enthält häufig View(s)

ET++

Bemerkungen

- Command Pattern für Ausführung und Undo/Redo von Befehlen (Verwaltung durch CommandProcessor)
- Event handling (z.B. von Befehlen) mit Chain of Responsibility Pattern
- Composite Pattern für VObjects (CompositeVObject)
- Decorator Pattern für Clipping von VObjects (Clipper)
- jede Menge Factory Methods
  (für Dokumente einer Applikation, für den CommandProcessor, ...)

Metamuster

Begriff

- **Metamuster (Metapattern)**: ein Mittel zur Dokumentation und Kommunikation des Entwurfs von Frameworks, das den Entwurfsmuster-Ansatz unterstützt.
- (unvollständiges) Klassifikationsschema für Entwurfsmuster
- Klassifikation von Metamustern
  - Schnittstelle und Interaktion
  - Komposition
- Zielsetzung: Metamuster sollen die
  - Entwicklung und
  - Dokumentation

  von Frameworks erleichtern

Metamuster der Interaktion

Template- und Hook-Methode

- **Template-Methode**: frozen spot, ruft Hook-Methode auf
- **Hook-Methode**: hot spot, wird von Template-Methode aufgerufen.

Beispiel

- Template Method Pattern: Template Method und primitive Methode

Assoziierte Begriffe

- **Template-Klasse**: eine Klasse mit einer Template-Methode
- **Hook-Klasse**: die Klasse, die die Hook-Methode zu einer Template-Methode enthält

Metamuster der Komposition

Übersicht

Klassifikation nach:

- Statische Beziehung der Template-Klasse zur Hook-Klasse (keine/selbe Klasse/Unterklasse)
- Anzahl der von einem Template-Klassen-Objekt referenzierten Hook-Klassen-Objekte (1 oder N)

Metamuster	Statische Beziehung	ref. Hook-Objekte
Unification	selbe Klasse	–
1:1 Connection	keine	1
1:N Connection	keine	N
1:1 Recursive Unification	selbe Klasse	1
1:N Recursive Unification	selbe Klasse	N
1:1 Recursive Connection	Unterklasse	1
1:N Recursive Connection	Unterklasse	N

Metamuster der Komposition

Unification (1)

Struktur

Repräsentanten

Template Method, Factory Method

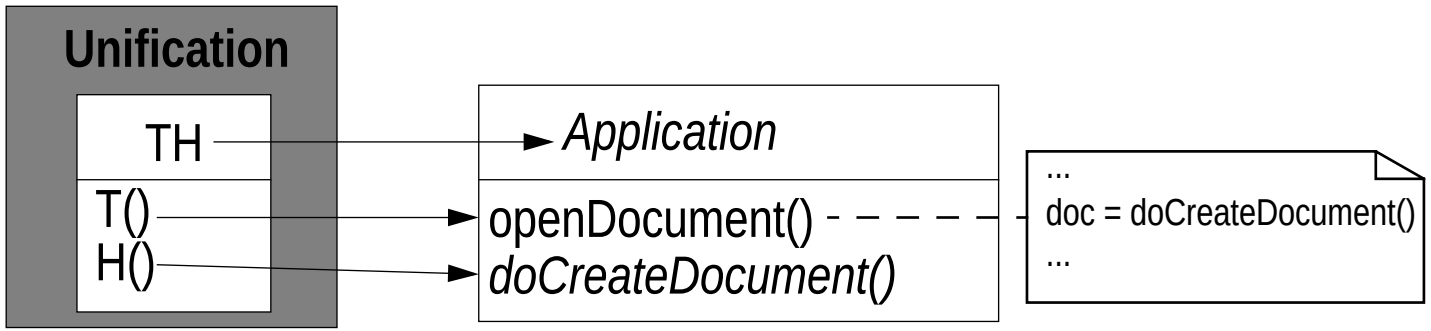
Typische Template-Methode

```
T() { ...
   this.H(); }
... }
```

Metamuster der Komposition

Unification (2)

Anwendung

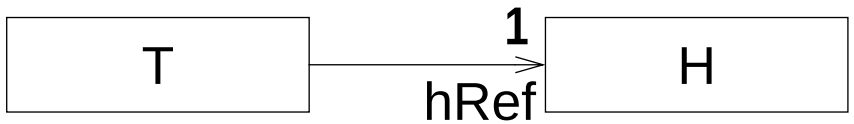


- unflexibel: Verhalten der Template-Methode kann nicht dynamisch, sondern nur durch Redefinition der Hook-Methode verändert werden

Metamuster der Komposition

1:1 Connection (1)

Struktur



Repräsentanten

Stategy, State, Adapter, Abstract Factory, Builder, Prototype, Bridge, Proxy

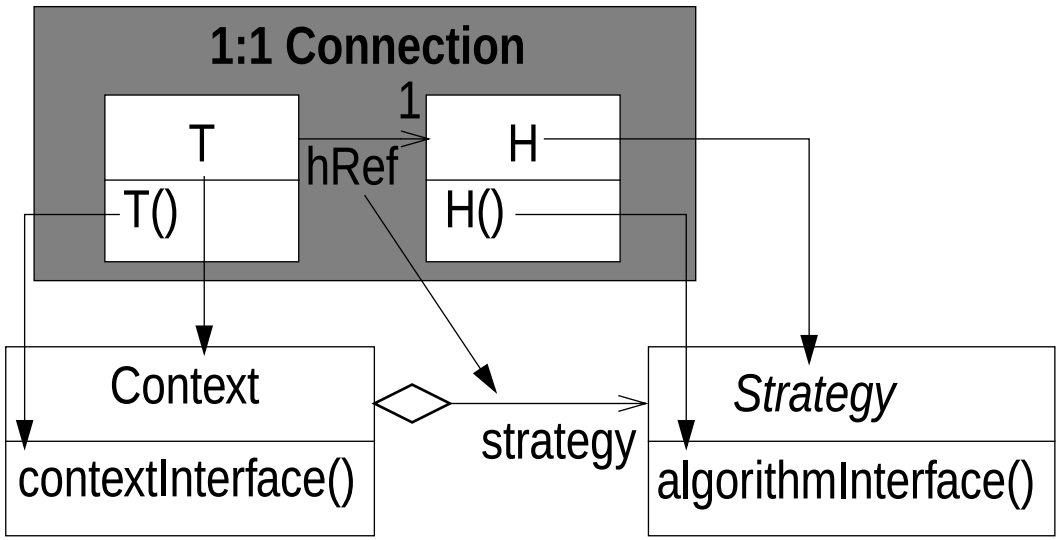
Typische Template-Methode

```
T() { ...
   hRef.H();
... }
```

Metamuster der Komposition

1:1 Connection (2)

Anwendung



Metamuster der Komposition

1:N Connection (1)

Struktur

Repräsentanten

Observer, Flyweight, Master-Slave (Buschmann et al., 1996)

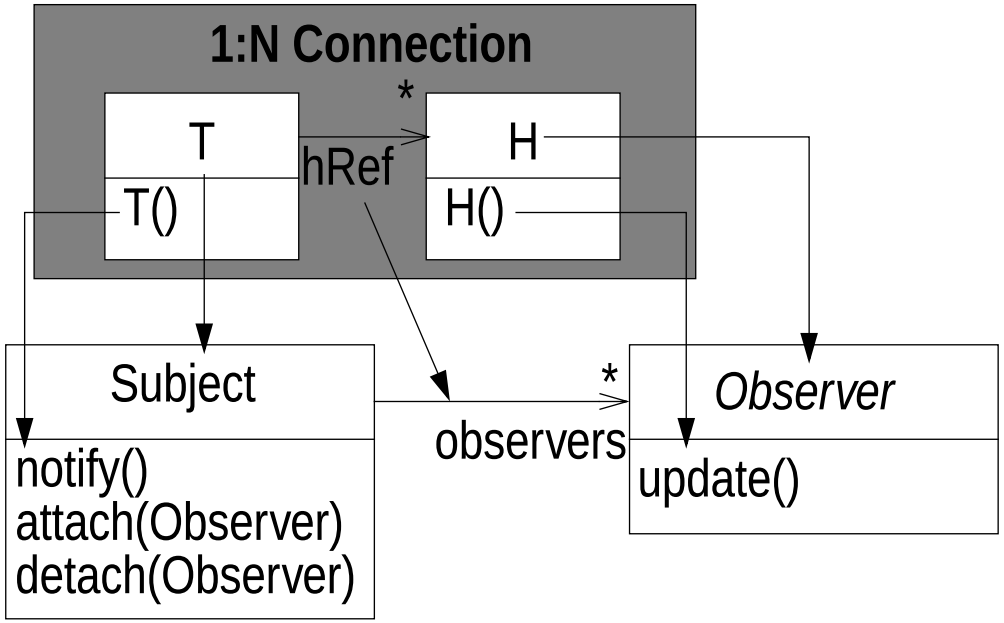
Typische Template-Methode

```
T() { ...
   foreach h in hRef {
     h.H(); } //u.U. bedingter Aufruf
... }
```

Metamuster der Komposition

1:N Connection (2)

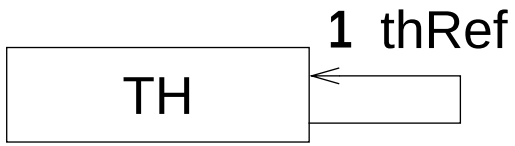
Anwendung



Metamuster der Komposition

1:1 Recursive Unification (1)

Struktur



Repräsentanten

Chain of Responsibility

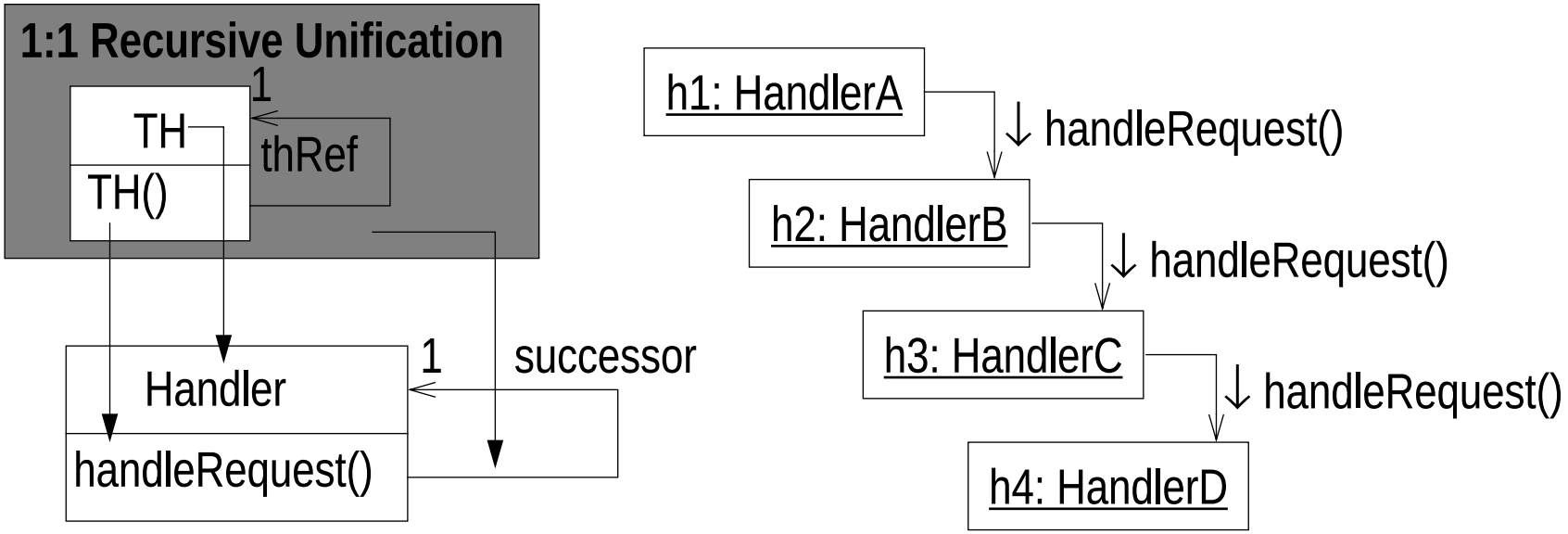
Typische Template-Methode

```
TH() { ...
   if (thRef != null) { thRef.TH(); }
... }
```

Metamuster der Komposition

1:1 Recursive Unification (2)

Anwendung

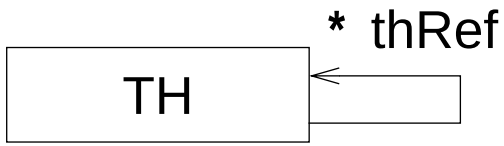


- TH() implementiert die Navigation über eine Kette von Objekten. Subklassen erweitern TH() um die dabei durchzuführenden Aktionen (wichtig: dabei Aufruf der Oberklassenmethode TH() !)

Metamuster der Komposition

1:N Recursive Unification (1)

Struktur



Repräsentanten

Composite, Interpreter

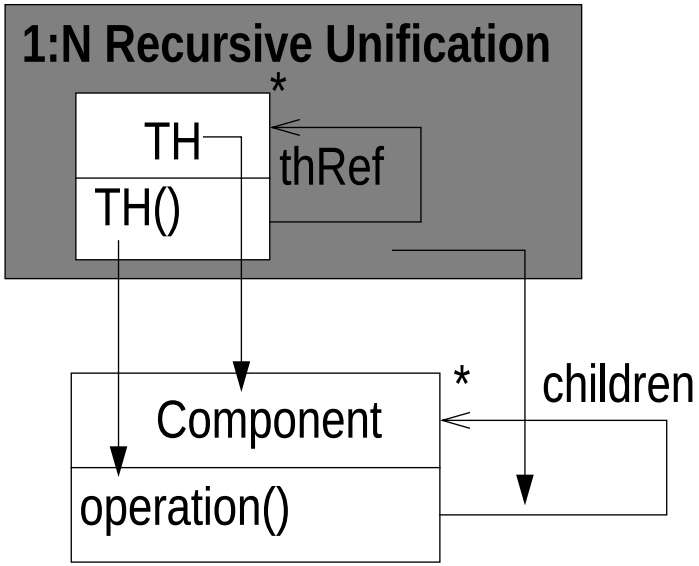
Typische Template-Methode

```
TH() { ...
   foreach th in thRef {
      th.TH();
   }
... }
```

Metamuster der Komposition

1:N Recursive Unification (2)

Anwendung

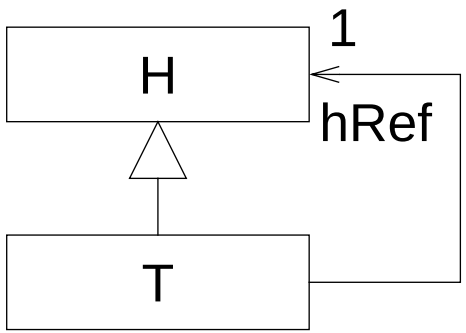


▲ zusammengesetzte und einfache Objekte können nach außen hin gleich behandelt werden (Unifikation)

Metamuster der Komposition

1:1 Recursive Connection (1)

Struktur



Repräsentanten

Decorator

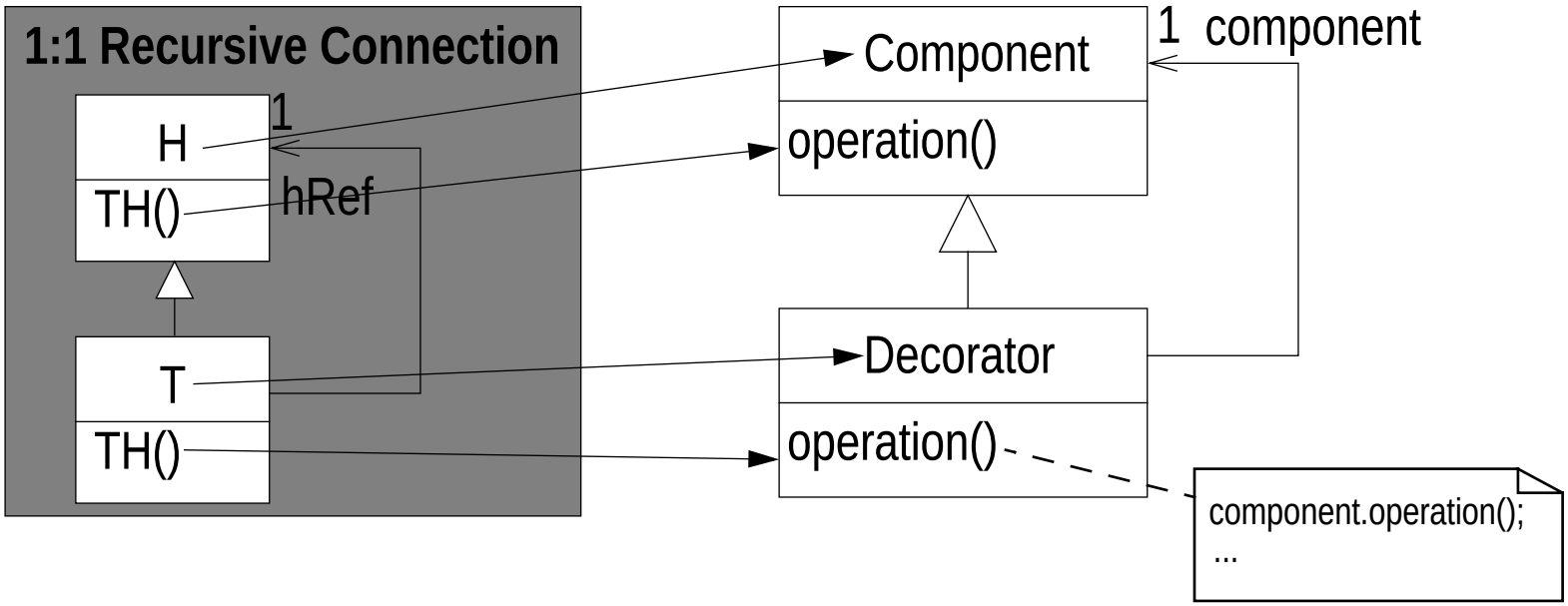
Typische Template-Methode

```
TH() { ...
   if (hRef != null) {
      hRef.TH() };
... }
```

Metamuster der Komposition

1:1 Recursive Connection (1)

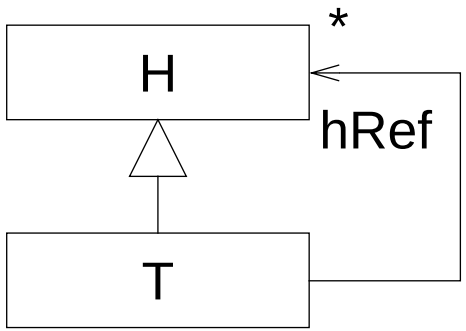
Anwendung



Metamuster der Komposition

1:N Recursive Connection (1)

Struktur



Repräsentanten

Variante von Composite

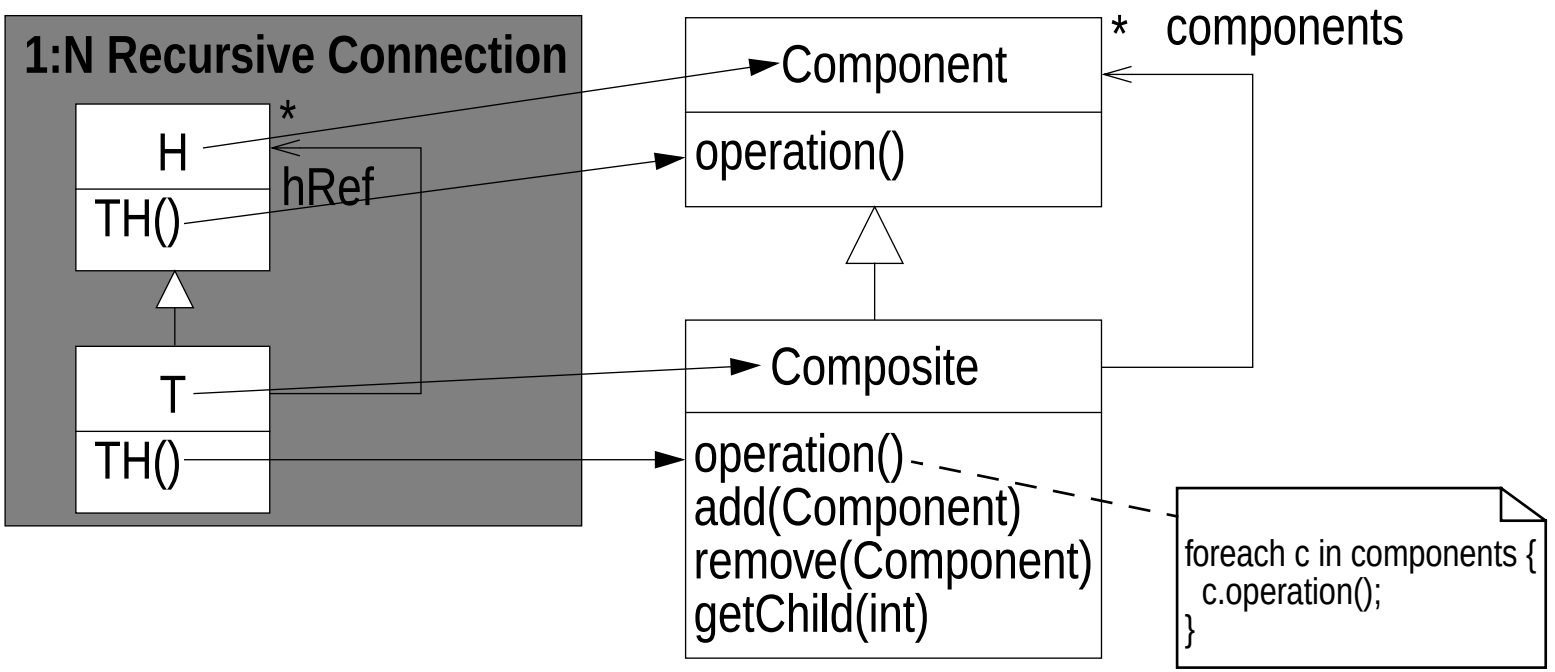
Typische Template-Methode

```
TH() { ...
   foreach h in hRef {
      h.TH(); }
... }
```

Metamuster der Komposition

1:N Recursive Connection (2)

Anwendung



Beispiel Textkonverter

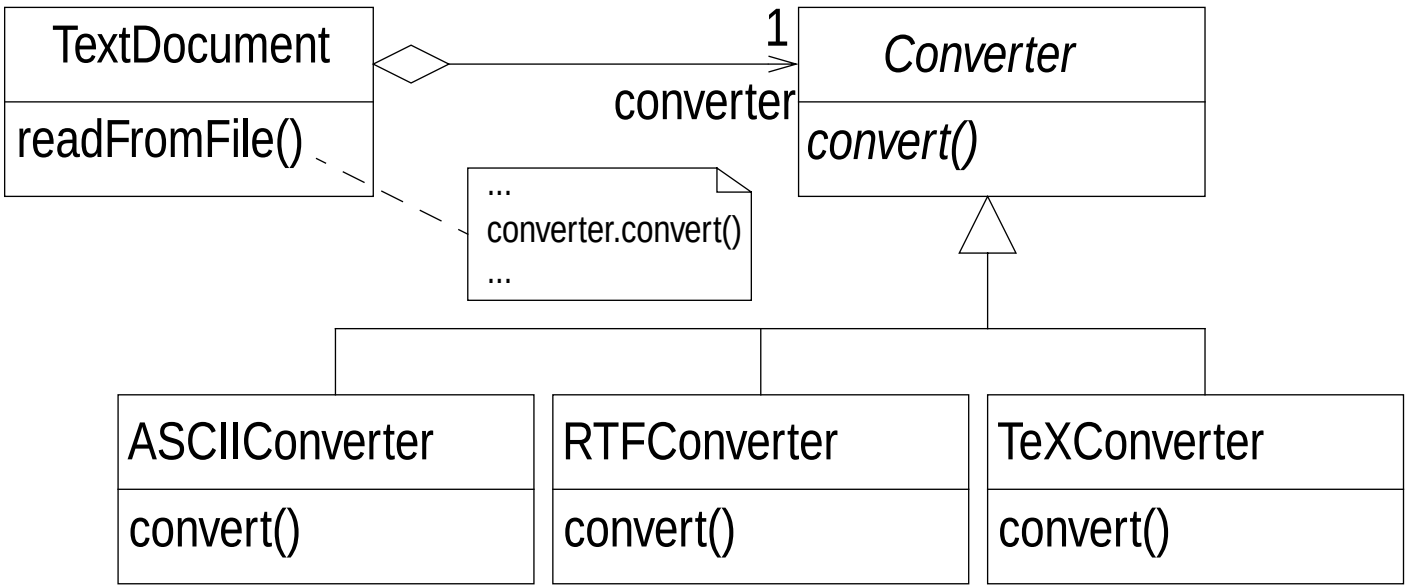
Problem

- Eine Textverarbeitung kann eine Menge von Formaten importieren
- Beim Öffnen einer Datei wird versucht, diese mit Hilfe der bekannten Formate zu lesen
- Es soll möglich sein, neue Formate hinzuzufügen

☞ Konversion wird speziellen Konverter-Objekten übertragen

Beispiel Textkonverter

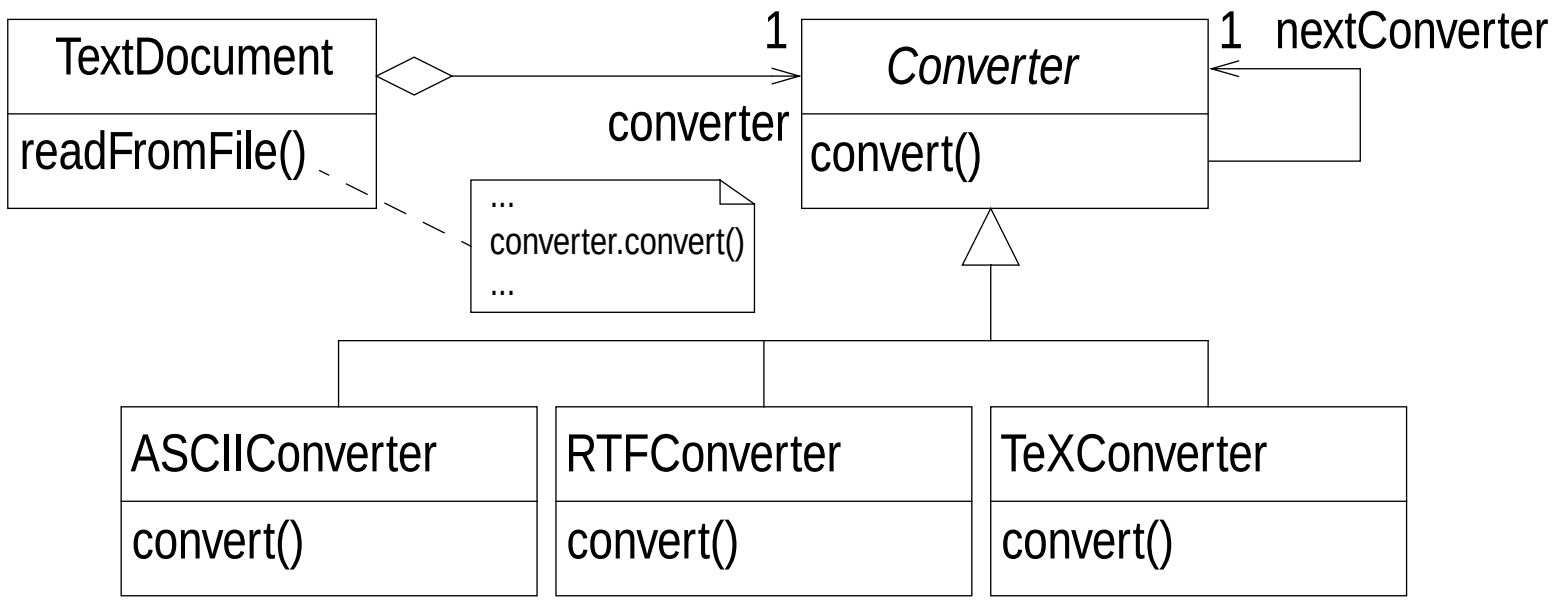
Lösung mit 1:1 Connection



- es kann nur genau ein Konverter verwendet werden

Beispiel Textkonverter

Lösung mit 1:1 Recursive Unification (1)



- Es kann eine Kette von Konvertern gebildet werden. Wenn ein Konverter mit dem Format der Datei nicht zurechtkommt, delegiert er die Konversion an seinen Nachfolger

Beispiel Textkonverter

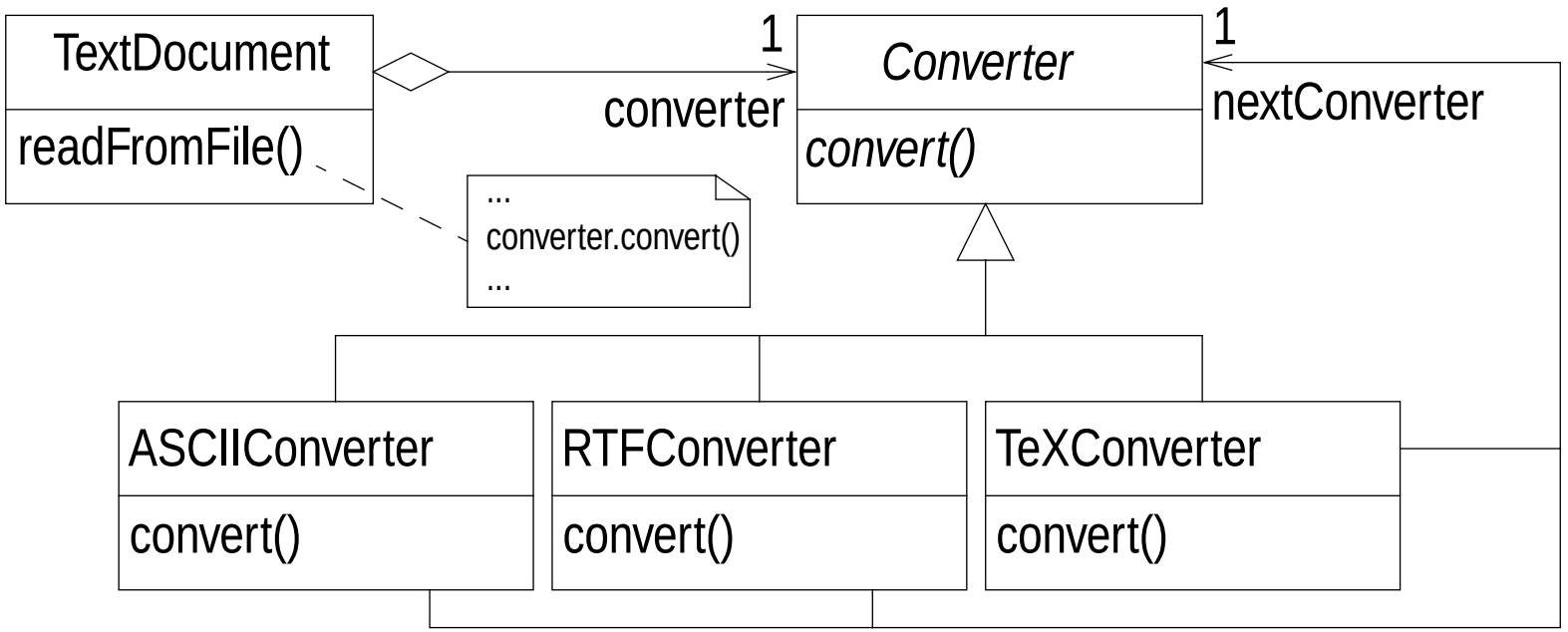
Lösung mit 1:1 Recursive Unification (2)

- ▲ Kette jederzeit erweiterbar, Verwender davon nicht betroffen
- ▼ Reihenfolge der Konverter fest
- ▼ konkrete Konverter müssen in ihrer Implementierung von `convert()` die Oberklassenmethode aufrufen (fehlerträchtig):

```
if (conversionIsPossible())
   // Konvertierung vornehmen ...
   return true;
else
   return super.convert();
   // sorgt dafür, daß der nächste Konverter an
   // die Reihe kommt
```

Beispiel Textkonverter

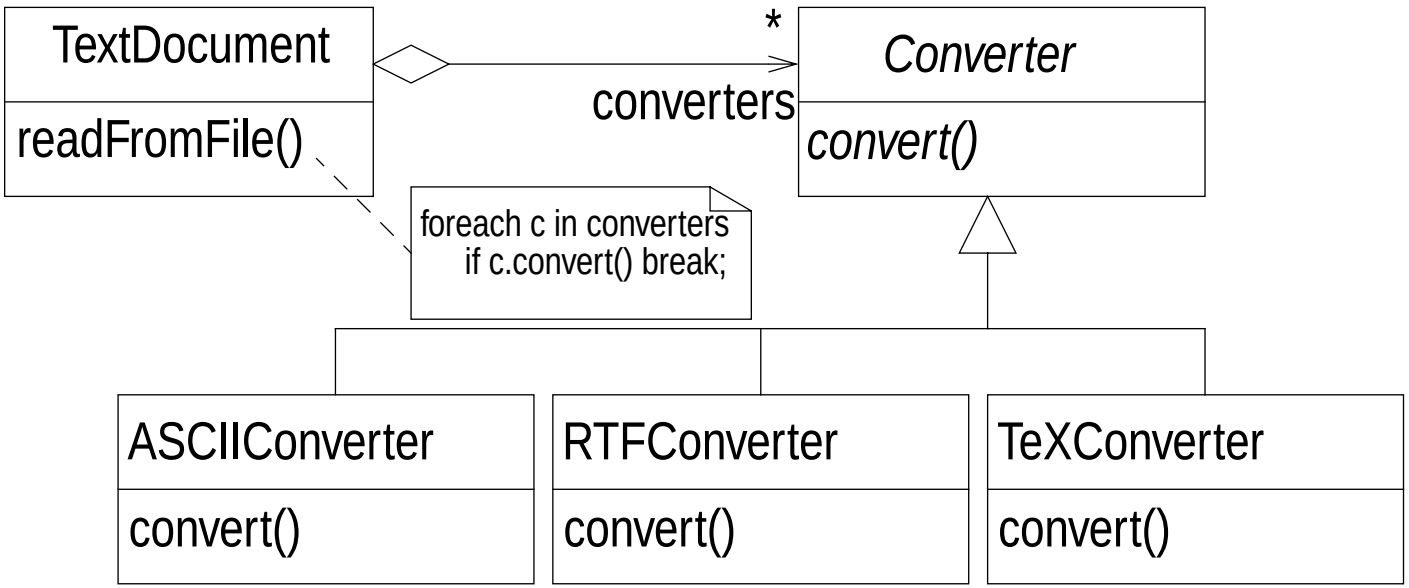
Lösung mit 1:1 Recursive Connection



- jeder Konverter muß die Delegation selbst implementieren (fehlerträchtig)

Beispiel Textkonverter

Lösung mit 1:N Connection



- ▲ TextDocument bestimmt die Auswahl und die Reihenfolge der Konverter
- ▼ TextDocument ist für die Auswahl des Konverters zuständig

Kapitel 8: Entwurfsheuristiken

- Entwurfsheuristiken nach Riel (1996)
- Transformationsmuster nach Riel (1996)

Entwurfsheuristiken

Begriff

Heuristik

- Webster: „involving or serving as an aid to learning, discovery, or problem-solving by experimental and especially trial-and-error methods“
- Langenscheids Fremdwörterlexikon: „erkenntnistheoretisches und methodisches Verfahren zur Gewinnung neuer wiss. Erkenntnisse und zur Problemlösung“
- hier: Richtschnur, Daumenregel
- Heuristik ist kein Prinzip!

Entwurfsheuristik

- Regel, die beim Entwurf eines objektorientierten Systems angewendet werden sollte (sofern dies sinnvoll ist)

Entwurfsheuristiken

Beispiele

Riel (1996) enthält 61 Entwurfsheuristiken, z.B.

- Eine Klasse sollte eine und nur eine Abstraktion erfassen.
- Modelliere die reale Welt, wann immer das möglich ist.
- Eine Klasse muß wissen, was sie enthält, aber sie sollte nie wissen, wer sie enthält.
- Vererbung sollte nur verwendet werden, um eine Spezialisierungshierarchie zu modellieren.
- Basisklassen sollten abstrakt sein.
- Wenn in deinem Entwurf Mehrfachvererbung vorkommt, nehme an, daß du einen Fehler gemacht hast, oder beweise das Gegenteil.
- Verderbe nicht deinen logischen Entwurf durch physische Entwurfskriterien (z.B. Effizienz)

Transformationsmuster

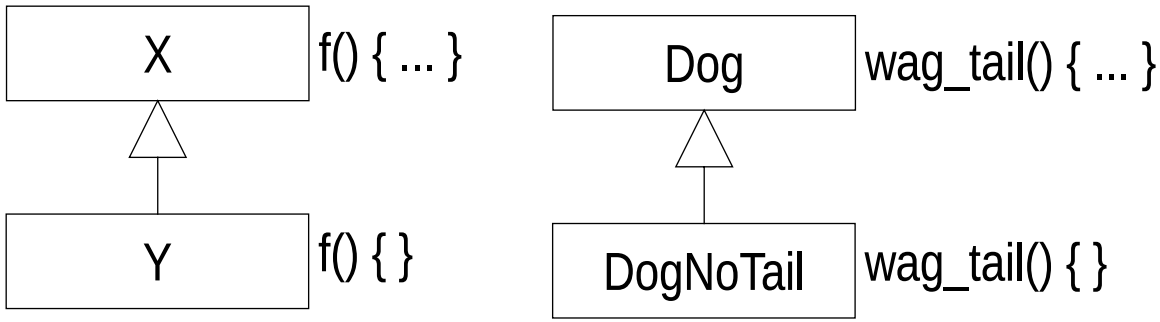
Begriff

- abgeleitet aus Entwurfsheuristiken
- Bestandteile:
  - Quellmuster: eine schlechte Entwurfsstruktur (Problem)
  - Motivierende Heuristik: die Heuristik, gegen die das Quellmuster verstößt
  - Erläuterung (der Heuristik)
  - Zielmuster: bessere Entwurfsstruktur (Lösung)
- eher Antipattern als Entwurfsmuster

Transformationsmuster

Inverted Inheritance Pattern (1)

Quellmuster



Motivierende Heuristik

Redefiniere nicht die Methode einer Basisklasse mit einer NOP-Methode (einer Methode, die nichts tut)

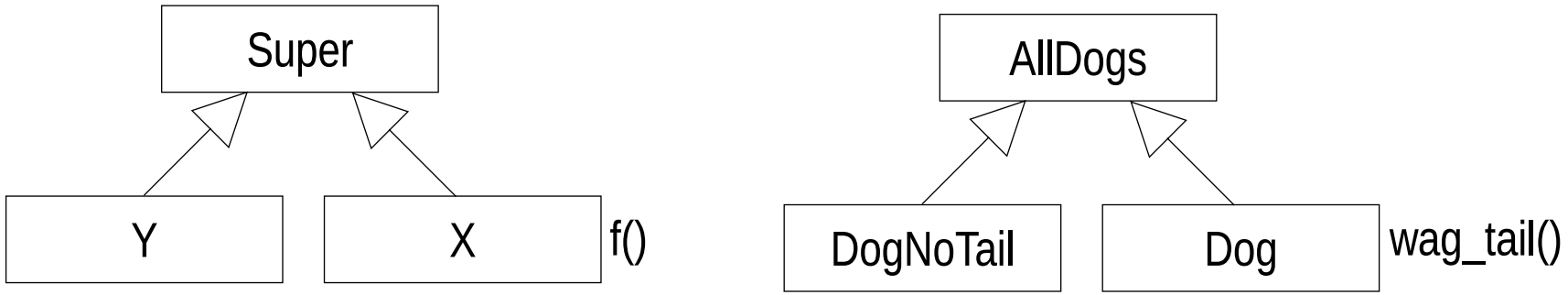
Erläuterung

Aussage eines solchen Entwurfs: (a) Y ist Spezialisierung von X. (b) Y kann etwas *nicht*, was X kann. Widerspruch!

Transformationsmuster

Inverted Inheritance Pattern (2)

Zielmuster

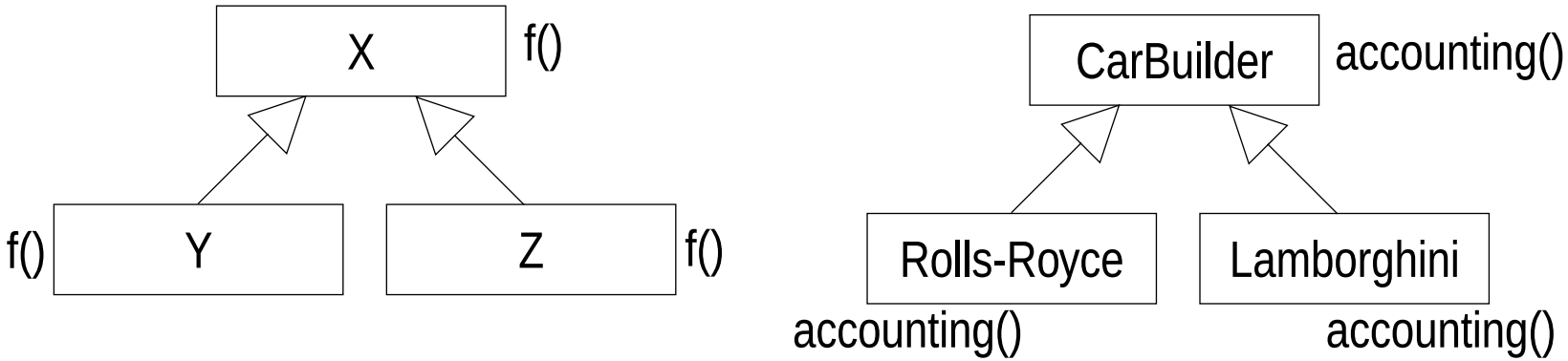


führe Oberklasse ein, die die gemeinsame Abstraktion darstellt.

Transformationsmuster

One-Instance Pattern (1)

Quellmuster



Jede Subklasse hat nur genau eine Instanz!

Motivierende Heuristik

Hüte dich vor abgeleiteten Klassen, die genau eine Instanz haben. Stelle sicher, daß die abgeleiteten Klassen nicht Instanzen der Basisklasse sind.

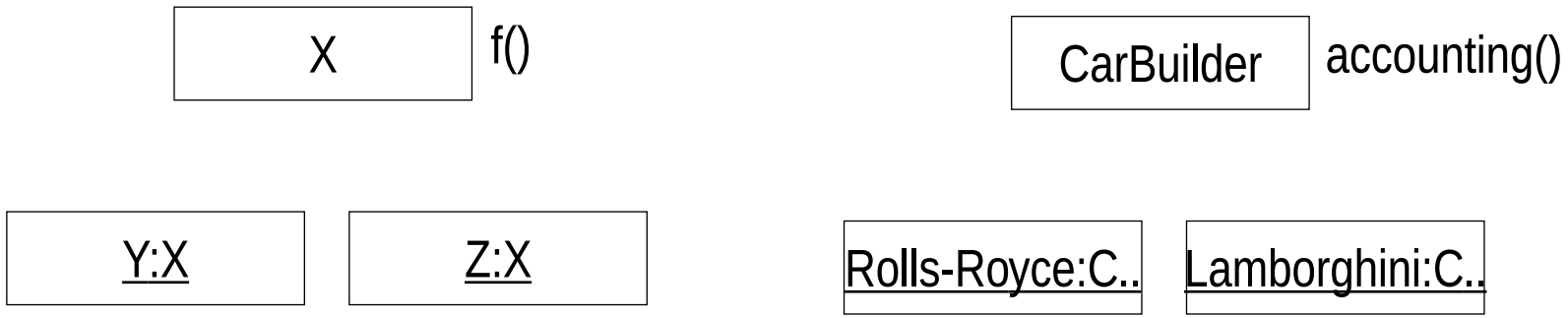
Transformationsmuster

One-Instance Pattern (2)

Erläuterung

Objekte, die sich im Verhalten unterscheiden, werden oft als Klassen modelliert. Das führt zu einer unerwünschten Klassenwucherung.

Zielmuster

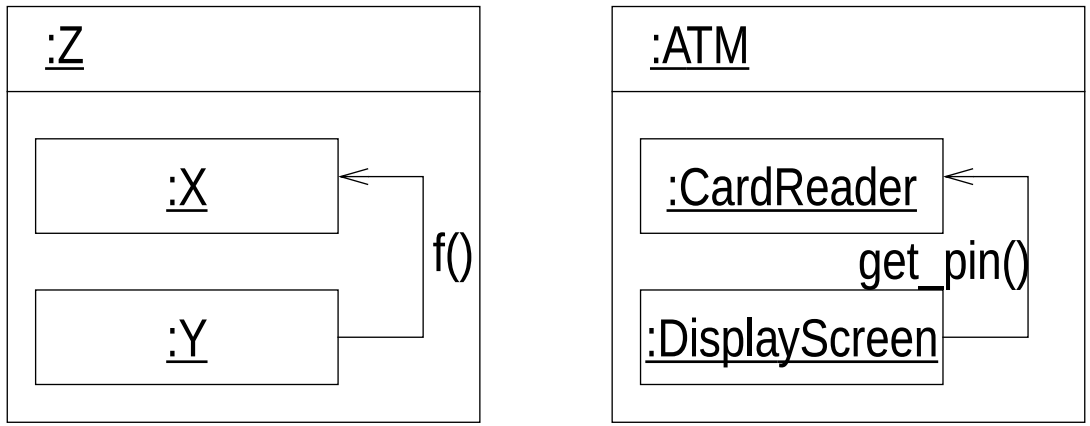


Das unterschiedliche Verhalten wird durch Attribute modelliert, die das Verhalten parametrisieren. Beispiele: interpretierbare Formel; Strategy Pattern

Transformationsmuster

Lexical Scope Pattern (1)

Quellmuster



Motivierende Heuristik

Klassen, die in derselben Klasse enthalten sind, sollten keine Benutzt-Beziehung untereinander haben.

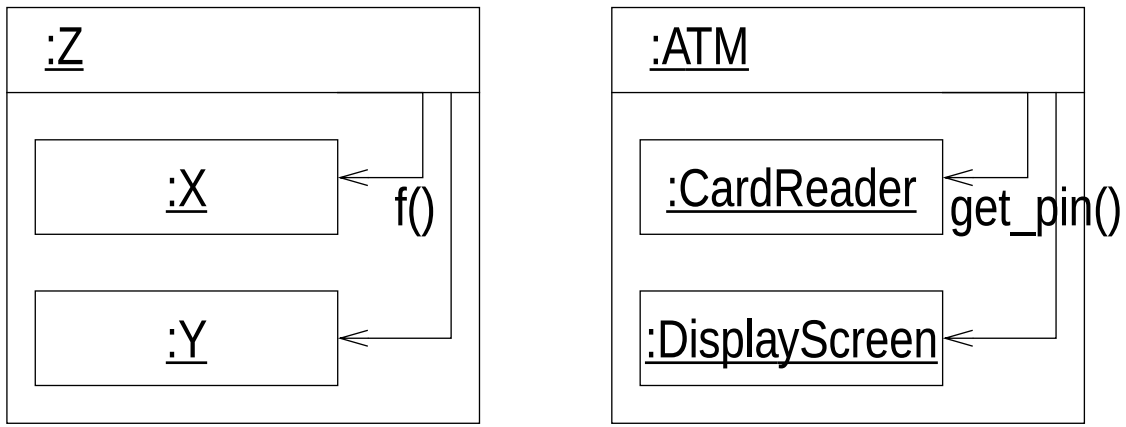
Transformationsmuster

Lexical Scope Pattern (2)

Erläuterung

Die Benutzt-Beziehung zwischen den Klassen ist redundant, da die umschließende Klasse die beiden Klassen bereits benutzt.

Zielmuster

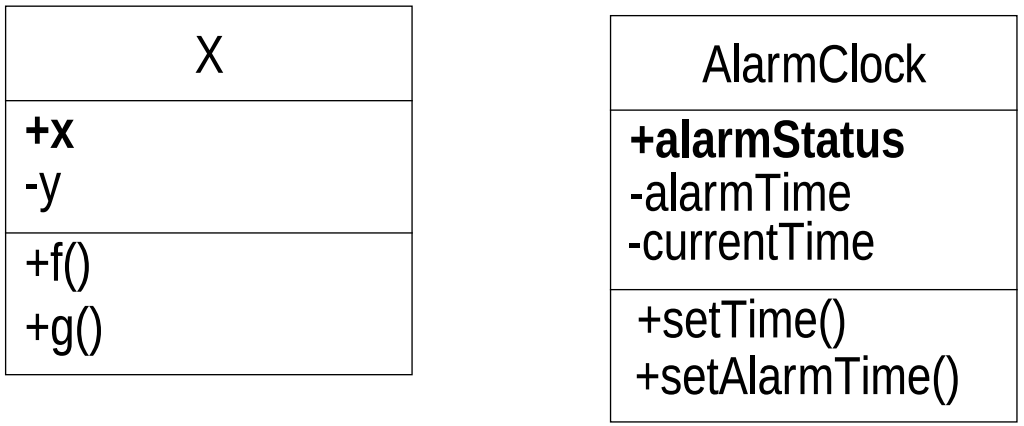


Nur die umschließende Klasse kümmert sich um kombiniertes Verhalten der beiden Klassen und koordiniert sie (Entkopplung!)

Transformationsmuster

Data-Hiding Pattern (1)

Quellmuster



Ein Attribut ist öffentlich zugänglich und wird von Verwendern direkt gelesen und geschrieben.

Motivierende Heuristik

Eine Klasse soll ihre Implementierung (Datenstruktur) völlig verbergen.

Transformationsmuster

Data-Hiding Pattern (2)

Erläuterung

Öffentliche Attribute machen die Verwender von der Implementierung der Klasse abhängig. Der Entwerfer sollte sich fragen:

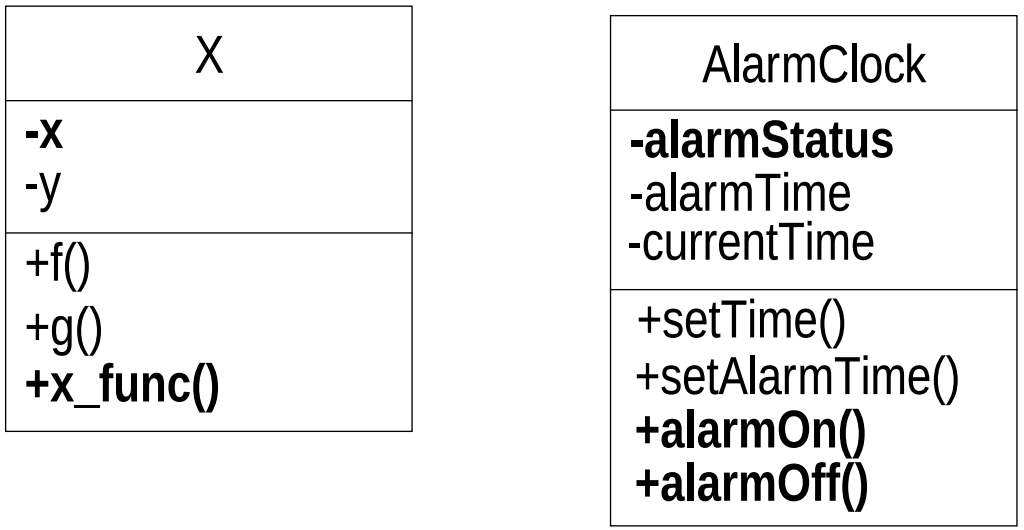
- Warum muß dieses Attribut öffentlich sein?
- Was tun die Verwender mit diesem Attribut?
- Warum macht die Klasse es nicht für sie?

Das öffentliche Attribut sollte privat werden und der Zugang durch neue öffentliche Methoden geregelt werden. Im schlimmsten Fall handelt es sich dabei um get/set-Methoden, im besten Fall um eine völlige Abstraktion der zugrundeliegenden Datenstruktur.

Transformationsmuster

Data-Hiding Pattern (3)

Zielmuster



alternativ: Methode `setAlarmStatus(boolean)`

Ähnliche Überlegungen gelten auch für get/set-Methoden selbst! Unter Umständen sind sie überflüssig oder durch Methoden höherer Abstraktionsstufe zu ersetzen.

Kapitel 9: Werkzeug-Material-Metapher

- Begriffe
  - Leitbild
  - Werkzeug-Material-Metapher
  - Werkzeug
  - Material
  - Aspekt
- Softwareentwicklung mit der Werkzeug-Material-Metapher
- Beispiel AbbreviationExplainer

Begriffe

Leitbild

- ❑ paradigmatisches Prinzip bei der Software-Entwicklung
- ❑ gibt eine Sichtweise auf das zu lösende Problem vor
- ❑ hilft dabei, das Problem zu strukturieren (Analyse)
- ❑ hilft dabei, die Lösung zu strukturieren (Entwurf)
- ☞ liefert einen Rahmen für die Entwicklung

Beispiele:

- ❑ Arbeitsplatz für qualifizierte menschliche Tätigkeiten
- ❑ Direkte Manipulation
- ❑ Werkzeug-Material-Metapher
- ❑ Objektorientierung

Begriffe

Werkzeug-Material-Metapher

- Ziele
  - spezielle Ausrichtung auf interaktive Systeme zur Bearbeitung von Aufgaben (Unterstützung von Sachbearbeitern)
  - die Aufgabenbearbeitung soll „intuitiv“ möglich sein, also der gewohnten Vorgehensweise nachempfunden
- Idee
  - das System wird modelliert als eine Menge autonomer, miteinander kooperierender Werkzeuge (Funktionsträger), mit denen die zugrundeliegenden Informationen (Daten = Material) bearbeitet werden
  - der Benutzer kann diese Werkzeuge frei (z.B. in beliebiger Reihenfolge) verwenden, um sein Aufgaben zu bearbeiten.
  - Arbeitsmittel = Werkzeug, Arbeitsgegenstand = Material

Begriffe

Werkzeug

- Werkzeug = (Modell eines) Arbeitmittels
- wird benutzt, um Material zu bearbeiten, z.B.
  - Drucker: druckt ein Dokument aus
  - Editor: bearbeitet ein Dokument
  - Terminplaner: verwaltet Termine
- kann für unterschiedliche Materialien geeignet sein z.B.
  - Stift: beschreibt Karteikarten, Endlospapier, ...
- manche Werkzeuge der realen Welt können in der Modellierung weggelassen werden, z.B. ein Locher
- Werkzeuge realisieren die interaktiven Komponente einer Anwendung

Begriffe

Material

- Material = (Modell eines) Arbeitsgegenstands
- wird von unterschiedlichen Werkzeugen benutzt
- Beispiele
  - Dokumente, z.B. Formulare, Briefe
  - Termine
- Auch Werkzeuge können als Material verwendet werden, es kommt auf den durchzuführenden Vorgang an!
  - Beispiel Stiftbecher: „verwaltet“ Stifte

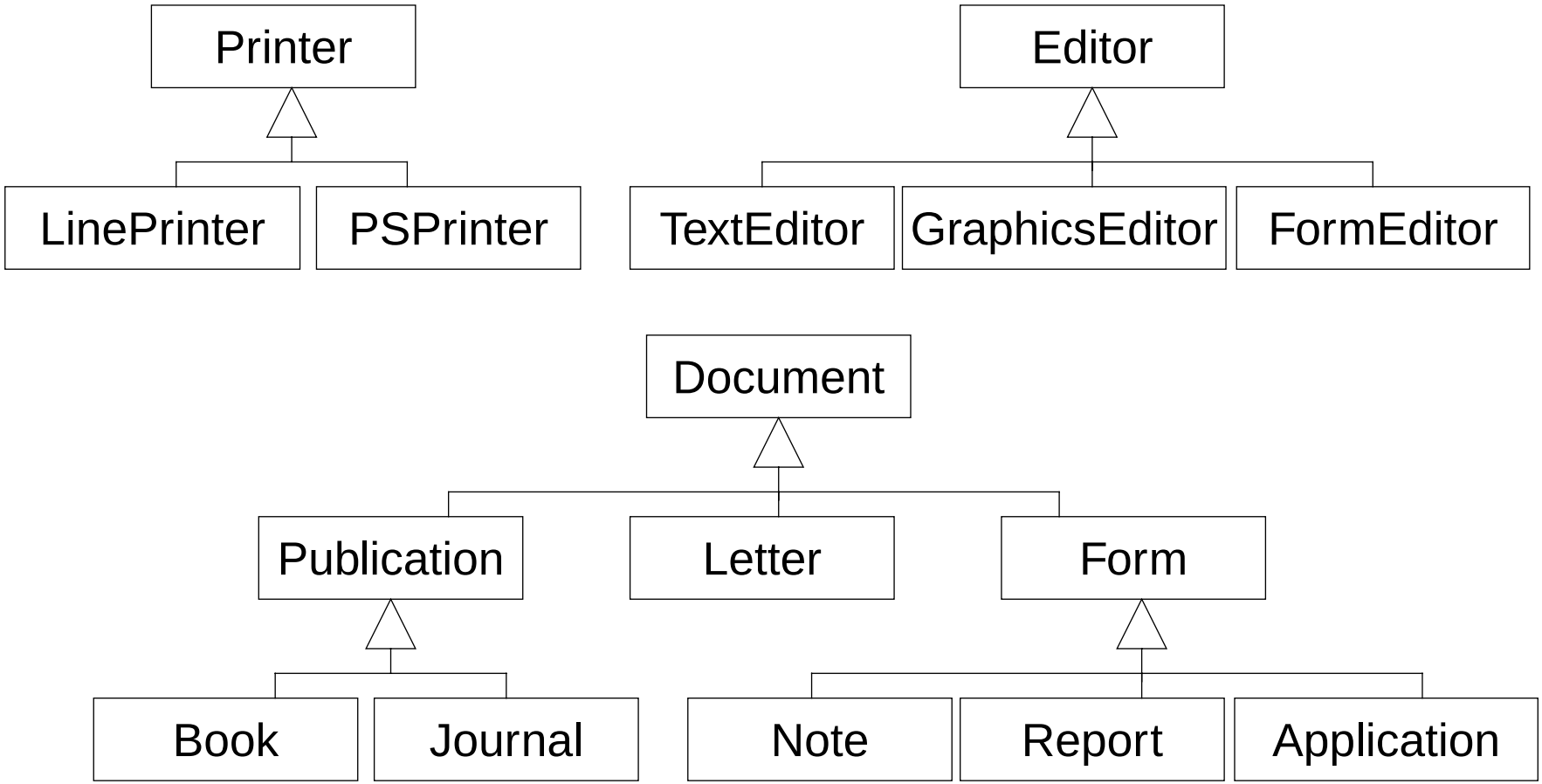
Begriffe

Werkzeug- und Materialklasse

- Werkzeugklasse
  - objektorientiertes Modell eines Werkzeugs
  - stellt Operationen zur Manipulation von Materialien zur Verfügung
  - diese werden auf der Grundlage der Operationen auf den einzelnen Materialien realisiert
  - Vererbung läßt sich einsetzen, um Spezialisierungen darzustellen
- Materialklasse
  - objektorientiertes Modell eines Materials
  - stellt Operationen zur Abfrage und zur Veränderung seines Zustands zur Verfügung

Werkzeug- und Materialklassen

Beispiele



Begriffe

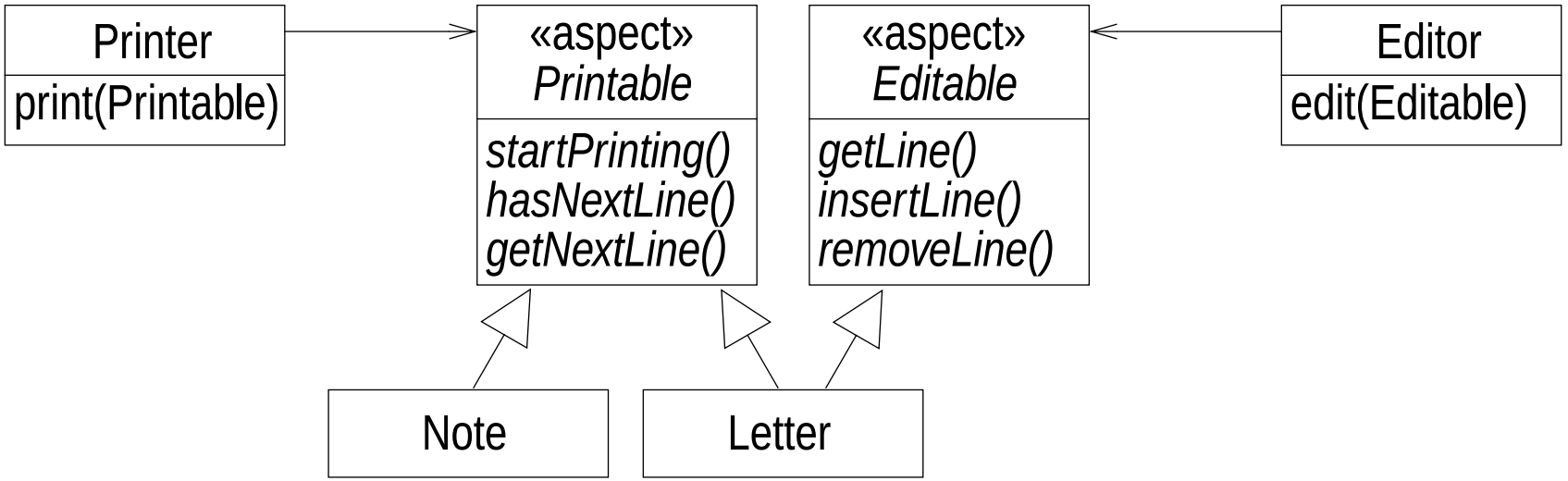
Aspekt

- Damit ein Werkzeug auf ein Material zugreifen kann, muß dieses eine bestimmte Schnittstelle anbieten
- Eine solche Schnittstelle wird als Aspekt bezeichnet
- Ein Werkzeug greift unter einem Aspekt auf ein Material zu
- Bietet ein Material einen Aspekt an, kann es vom zugehörige Werkzeug bearbeitet werden
- Aspekte drücken das Zusammenpassen von Werkzeug und Material aus
- über Aspekte wird das Werkzeug vom konkreten Material entkoppelt

Begriffe

Aspektklasse

- objektorientiertes Modell eines Aspekts
- definiert eine reine Schnittstelle (Menge abstrakter Methoden)
- in der Regel braucht man zur Modellierung Mehrfachvererbung
- in Java können Interfaces als Aspektklassen verwendet werden



Vorgehensweise

Entwicklung mit Werkzeug-Material-Metapher

1. Materialklassen auf der Grundlage der vertrauten Gegenstände der realen Welt entwerfen
2. Werkzeuge entwerfen, die den aktuellen Arbeitserfordernissen entsprechen. Hier sind Analogien zur realen Welt hilfreich (als auch hinderlich)
   - ❍ Werkzeug selbst
   - ❍ (graphische) Repräsentation für den Benutzer
3. den Zusammenhang von Werkzeugen und Materialien in Aspektklassen darstellen: Aspekte ausgehend von den Werkzeugen entwerfen und Materialien zuordnen
4. Aspektklassen als „Framework“ für die Integration neuer Werkzeuge und Materialien verwenden

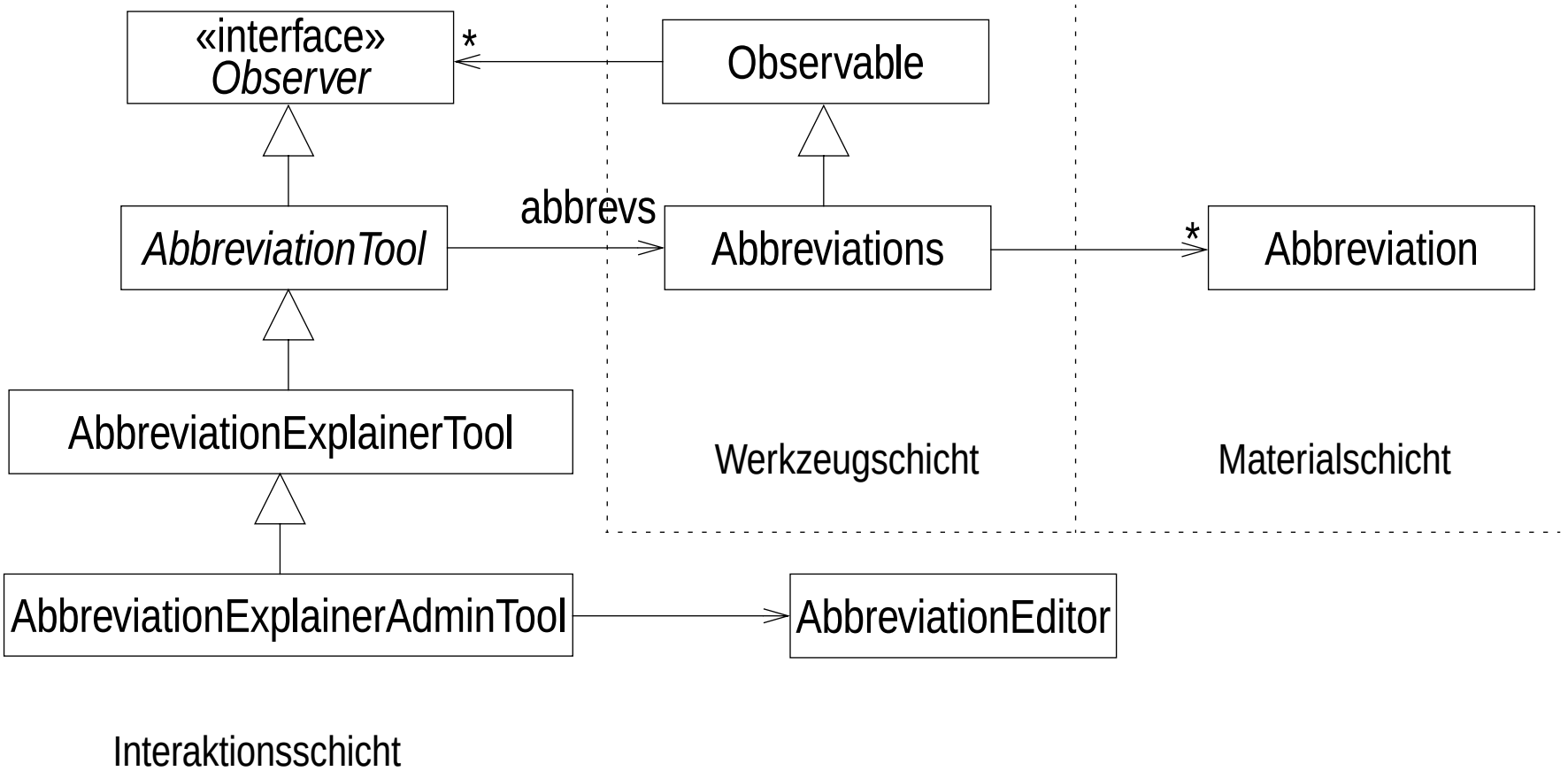
Beispiel AbbreviationExplainer

Anforderungen

- In einer Firma werden sehr viele Abkürzungen verwendet; neue Mitarbeiter haben dadurch größere Schwierigkeiten.
- daher soll ein System erstellt werden, das Abkürzungen mit ihren zugehörigen Erläuterungen verwalten und anzeigen kann.
- es soll möglich sein, neue Abkürzungen hinzuzufügen sowie vorhandene Abkürzungen abzuändern und zu löschen.
- die Anzeige ist für jedermann zugänglich, die Modifikation soll nur einem eingeschränkten Personenkreis zur Verfügung stehen. Daher sollen zwei verschiedene Sichten erstellt werden.
- es sollen beliebig viele Anzeige- und Modifikationssichten gleichzeitig möglich sein, diese sollen immer auf dem aktuellen Bestand arbeiten.

Beispiel AbbreviationExplainer

Übersicht des Entwurfs



Beispiel AbbreviationExplainer

Materialklassen

- Abkürzung (Abbreviation)

```
class Abbreviation extends Object {
   private String abbr;  //Abkuerzung
   private String expl;  //Erlaeuterung
   public Abbreviation( String abbreviation,
                        String explanation);
   public boolean equals(Abbreviation anAbbrev);
   public boolean isSmaller(Abbreviation anAbbrev);
   public String getAbbreviation();
   public String getExplanation();
}
```

Beispiel AbbreviationExplainer

Werkzeugklassen

- Verwaltung der Abkürzungen (Abbreviations)

```
class Abbreviations extends Observable {
   public Abbreviations();
   public void addAbbreviation(String abbr,
                               String expl);
   public void removeAbbreviation(String abbr);
   public boolean hasAbbreviation(String abbr);
   public String getExplanation(String abbr);
   public Enumeration getAbbreviations();
   public int size();
}
```

Beispiel AbbreviationExplainer

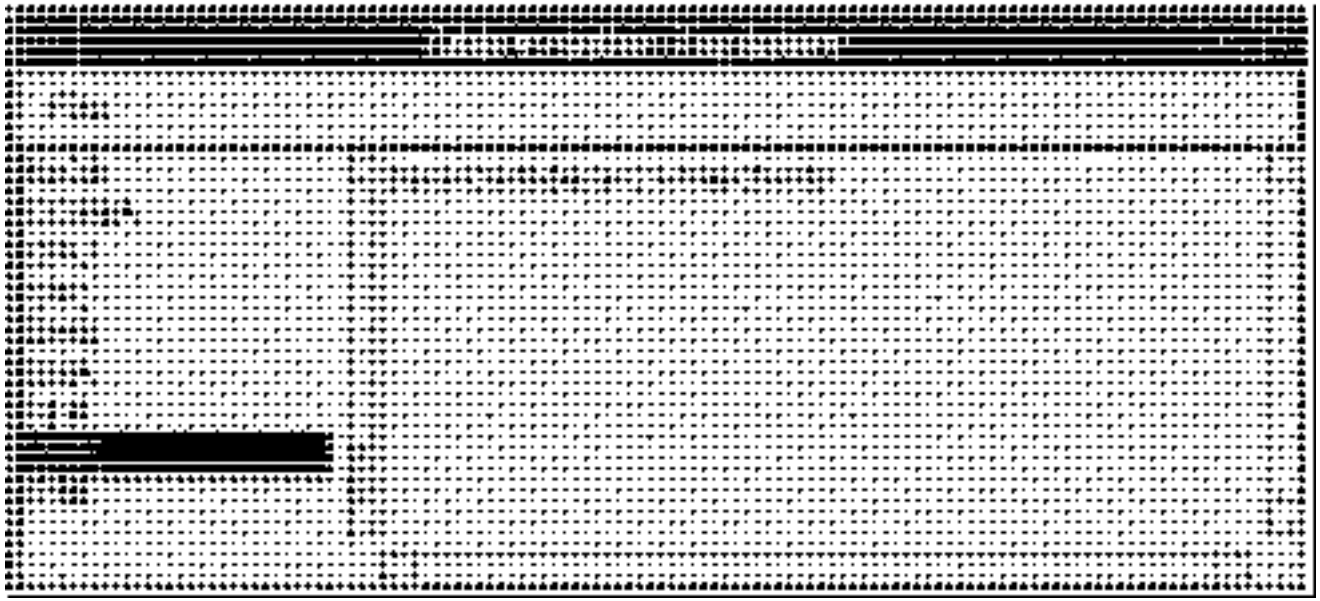
Interaktionsklassen

- Anzeige der Abkürzungen (AbbreviationExplainerTool)
- Modifikation der Abkürzungen (AbbreviationExplainerAdminTool)
- Basisklasse für Werkzeuge über Abbreviations (AbbreviationTool):

```
abstract class AbbreviationTool implements Observer {
   protected Frame frame;     // Fenster des Werkzeugs
   protected Abbreviations abbrevs;
   AbbreviationTool(String title, Abbreviations abbrs);
   public void showTool();
   public void hideTool();
   abstract public void update(Observable o,
                               Object arg);
}
```

Beispiel AbbreviationExplainer

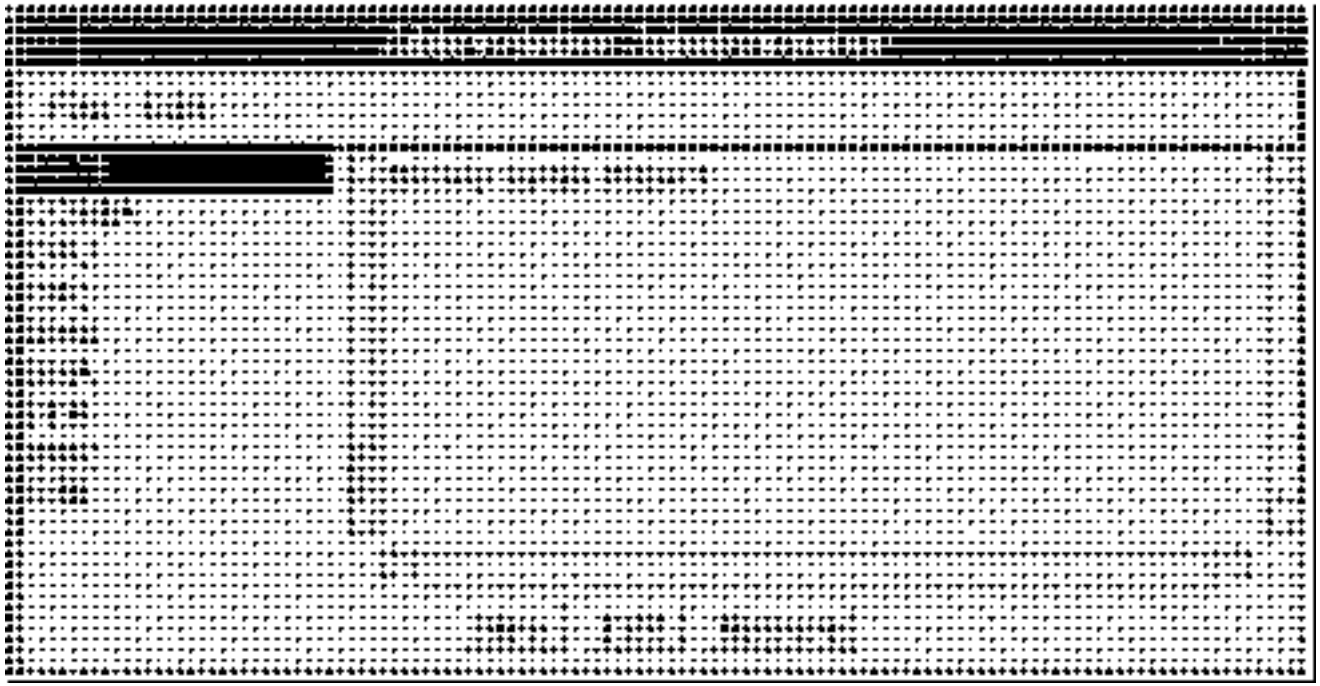
AbbreviationExplainerTool



```
class AbbreviationExplainerTool
                              extends AbbreviationTool {
   private void showExplanation(String abbr);
   public void update(Observable o, Object arg);
}
```

Beispiel AbbreviationExplainer

AbbreviationExplainerAdminTool (1)



- erweitert Anzeigewerkzeug um Menü und Buttons zur Bearbeitung von Abkürzungen (Neu, Ändern, Löschen)

Beispiel AbbreviationExplainer

AbbreviationExplainerAdminTool (2)

```
class AbbreviationExplainerAdminTool
                    extends AbbreviationExplainerTool {
  public void update(Observable o, Object arg);
    // update aktualisiert auch Buttons & Menüs
  private void newAbbrev();
  private void editAbbrev();
  private void removeAbbrev();
}
```

Beispiel AbbreviationExplainer

Ausblick

Was noch getan werden könnte:

- update-Nachrichten verfeinern (add, remove)
  erlaubt effizienteres und genaueres Anpassen der Views
- Abbreviations als SINGLETON realisieren
  garantiert, daß es nur ein Abkürzungsverzeichnis gibt
- Befehle (Exit, New, etc.) mit COMMANDS realisieren
  macht die Ausführung von Befehlen flexibler
- Behandlung der Commands mittels CHAIN OF RESPONSIBILITY
  Kommandos werden vom zuständigen Objekt ausgeführt
- Hilfe-Funktion einbauen (ebenfalls mittels COMMAND und CHAIN OF RESPONSIBILITY)

Beispiel AbbreviationExplainer

Von der Anwendung zum Framework

- Verallgemeinerung: es sollen nicht nur Abkürzungen, sondern beliebige Assoziationen (Abbildung von Schlüssel auf Wert) eines Typs gespeichert werden können
- Aspektklasse Association (key, value) ersetzt Abbreviation
  - abstrakten Basisklasse (bzw. Interfaces) für Assoziationen
  - Abbreviation kann mittels eines ADAPTERS an die Association-Schnittstelle angepaßt werden oder reimplementiert werden
- AssociationManager ersetzt Abbreviations, verwaltet Associations
  - AssociationManager erhält eine FACTORY METHOD zur Erzeugung der gewünschten Association-Subklasse
  - alternativ: PROTOYPE einsetzen
- Die vorhandenen Views werden an AssociationManager angepaßt
- Zur Darstellung von Schlüssel und Wert wird `toString` verwendet

Beispiel AbbreviationExplainer

Aspekt Association

```
interface Association {
   public boolean equals(Association a);
   public boolean isSmaller(Association a);
   public void setKey(Object key);
   public Object getKey();
   public void setValue(Object value);
   public Object getValue();
}

class Abbreviation extends Object
   implements Association { ... } // Abk. + Erläuterung
class Person extends Object
   implements Association { ... } // Name + Adresse
```

Beispiel AbbreviationExplainer

AssociationManager

```
abstract class AssociationManager extends Observable {
   public AssociationManager();
   public void addAssociation( Object key,
                               Object value);
   public void removeAssociation(Association assoc);
   public boolean hasValue(Object key);
   public Object getValueOf(Object key);
   public Enumeration getAssociations();
   public int size();
   abstract protected Association doMakeAssociation(
     Object key, Object value);
     // Factory Method für konkrete Assoziation
}
```

Literatur

Software Engineering

- Pressman, R.: Software Enginering: A Practitioner‘s Approach. 3. europäische Auflage. McGraw-Hill, 1994.
- Bersoff, E.; Henderson, V.; Seigel, S.: Software Configuration Management. Prentice Hall, 1980.
- Lehman, M.: Programs, Life Cycles and Laws of Software Evolution. In: Proceedings of the IEEE, 68(9), 1060-1076, 1980.
- Ludewig, J.: Software Engineering: Vorläufiges, unvollständiges Skript zur Vorlesung Software Engineering an der Fakultät Informatik der Universität Stuttgart. 1997.

Literatur

OO allgemein

- Meyer, B.: Object-Oriented Software Construction. 2. Auflage. Prentice Hall, 1997.
- Oestereich, B.: Objektorientierte Softwareentwicklung: mit der Unified Modeling Language. 3. Auflage. Oldenbourg, 1997
- Kilberth, K.; Gryczan, G.; Züllighoven, H.: Objektorientierte Anwendungsentwicklung: Konzepte, Strategien, Erfahrungen. 2. Auflage. Vieweg, 1994.
- Booch, G.: Object-oriented Analysis and Design with Applications. 2. Auflage. Benjamin/Cummings, 1994.

Software-Architektur

- Shaw, M.; Garlan, D.: Software Architecture: Perspectives on an Emerging Discipline. Prentice Hall, 1996.

Literatur

Muster

- Alexander, C.; Ishikawa, S.; Silverstein, M.: A Pattern Language. Oxford University Press, 1977.
- Alexander, C.: The Timeless Way of Building. Oxford University Press, 1979.
- Lea, D.: Christopher Alexander - An Introduction for Object-Oriented Designers. ACM SIGSOFT Software Engineering Notes 19(1), 1994.

Literatur

Architekturmuster

- Buschmann, F.; Meunier, R.; Rohnert, H.; Sommerlad, P.; Stal, M.: Pattern-Oriented Software Architecture. Wiley, 1996.
- Foote, B.; Yoder, J.: Big Ball of Mud. Technical Report #WUCS-97-34, 1997. Siehe auch http://laputa.isdn.uiuc.edu/mud/mud.html.

Konferenzbände

- Coplien, J.; Schmidt, D. (ed.) (1995): Pattern Languages of Program Design. Addison-Wesley, Reading, MA, 1995.
- Vlissides, J.; Coplien, J.; Kerth, N. (ed.) (1996): Pattern Languages of Program Design 2. Addison-Wesley, Reading, MA, 1996.
- Martin, R.; Riehle, D.; Buschmann, F. (ed.) (1998): Pattern Languages of Program Design 3. Addison-Wesley, Reading, MA, 1998.

Literatur

Entwurfsmuster

- Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.
- Buschmann, F.; Meunier, R.; Rohnert, H.; Sommerlad, P.; Stal, M.: Pattern-Oriented Software Architecture. Wiley, 1996.
- Pree, W.: Design Patterns for Object-Oriented Software Development. Addison-Wesley, 1995.
- Riehle, D.: Entwurfsmuster für Softwarewerkzeuge: Gestaltung und Entwurf von Anwendungen mit grafischer Benutzungsoberfläche. Addison-Wesley, 1997.

Entwurfsheuristiken

- Riel, A.: Object-Oriented Design Heuristics. Addison-Wesley, Reading, MA, 1996.

Literatur

Idiome

- Coplien, J.: Advanced C++: Programming Styles and Idioms. Addison-Wesley, 1992.
- Beck, K.: Smalltalk Best Practice Patterns. Prentice Hall, 1996.

Frameworks

- Weinand, A.; Gamma, E.; Marty, R.: Design and Implementation of ET++, a Seamless Object-Oriented Application Framework. Structured Programming 10(2), 1989, S. 63-87.
- Weinand A., Gamma E.: ET++ – a Portable, Homogenous Class Library and Application Framework. In: Bischofberger, W.; Frei, H. (Hrsg.): Computer Science Research at UBILAB, Strategy and Projects; Proceedings of the UBILAB '94 Conference, Zurich, Universitätsverlag Konstanz, Konstanz, 1994, S. 66-92.