

Vorlesung "Softwaretechnik"

Wiederverwendung

Lutz Prechelt

Freie Universität Berlin, Institut für Informatik

- Arten der Wiederverwendung
 - Fokus, Ziel/Vorteil
- Gegenstände der WV:
 - Anforderungen (→ Muster)
 - Schablonen, Checklisten
 - Entwurfsideen (→ Muster)
 - Komponenten
 - Methoden, Prozesse (→ Muster)
 - Werkzeuge
- Risiken und Hindernisse
- Beispiele für Muster
 - Prinzipien
 - Analysemuster
 - Benutzbarkeitsmuster
 - Prozessmuster
 - Anti-Muster

Teil 2

Wo sind wir?: Taxonomie "Die Welt der Softwaretechnik"

Welt der Problemstellungen:

- Produkt (Komplexitätsprob.)
 - Anforderungen (Problemraum)
 - Entwurf (Lösungsraum)
- Prozess (psycho-soziale P.)
 - Kognitive Beschränkungen
 - Mängel der Urteilskraft
 - Kommunikation, Koordination
 - Gruppendynamik
 - Verborgene Ziele
 - Fehler

Welt der Lösungsansätze:

- Technische Ansätze ("hart")
 - Abstraktion
 - **Wiederverwendung**
 - Automatisierung
- Methodische Ansätze ("weich")
 - Anforderungsermittlung
 - Entwurf
 - Qualitätssicherung
 - Projektmanagement

- Einsicht: Etwas bekanntes wiederzuverwenden kann Qualität und Produktivität stark erhöhen und Risiko senken
- Prinzipien:
 - **Normales Vorgehen**: Vermeide radikales Vorgehen
 - **Universalität**: Fast alles lässt sich im Prinzip wiederverwenden
 - z.B. Anforderungen, Anforderungsmuster, Architekturen, Teilentwürfe, Entwurfsmuster, Testfälle, Dokumentschablonen, Vorgehensbeschreibungen, Checklisten, Prozessmuster
 - **Abwägung**: Wäge sorgfältig den Gewinn an Produktivität und (hoffentlich) Qualität ab gegen den Verlust an Flexibilität und Kontrolle

Mit welcher Orientierung kann man wiederverwenden?

- Produktorientiert
 - Anforderungen
 - Architekturen
 - Teilentwürfe, Entwurfsideen
 - Konkrete Komponenten
 - Testfälle
 - (siehe auch viele Einträge der nachfolgenden Folie)
- Prozessorientiert
 - Beschreibungen von Prozessen und Rollen
 - Methoden
 - Werkzeuge und Automatisierung
 - Sonstige Infrastruktur
 - z.B. Dokumentvorlagen
 - Maße
 - Erfahrungen
 - Verhalten von Kunden
 - Verhalten von Entwicklern
 - Firmenkultur
 - Projektdynamik
 - Erfolg von Prozessen

Was kann man wiederverwenden?

- Problem
 - Funktionale Anforderung
 - bewährte domänenspezifische Anforderung
 - Problemrahmen
 - Analysemuster
 - Nichtfunktionale Anforderung
 - z.B. Kriterien für Erlernbarkeit, Verfügbarkeit, Antwortzeiten etc.
 - Dokumentschablone
 - z.B. für Projektplan, Use Case, API-Dokumentation, Mängelbericht, u.v.a.m.
 - Checkliste
 - z.B. für Anforderungsermittlg., Analyse, Entwurf, Kodierung, Durchsicht, Test, Dokumentation
- Lösung
 - Konkrete Lösung
 - Dienst
 - Binärcode (Bibliothek, Komponente, Anwendung)
 - Quellcode
 - Rahmenwerk
 - Konkrete Lösungsidee
 - Entwurfsmuster
 - Benutzbarkeitsmuster
 - Prozessmuster
 - Allgemeine Lösungsidee
 - Methode
 - Prinzip
- Mischformen
 - Werkzeug
 - Produktfamilie

Mit welchem Ziel kann man wiederverwenden?

- **Aufwandsverminderung jetzt**
 - indem man etwas nicht erst erfinden/entwickeln muss
- **Qualitätsverbesserung**
 - indem man etwas Ausgereiftes benutzt
- **Risikoverminderung**
 - indem man die Eigenschaften schon vorher kennt
- **Standardisierung**
 - indem man das Gleiche vielfach verwendet
 - führt zu den obigen Wirkungen
- **Aufwandsverminderung später**
 - indem man nur 0 oder 1 Lösung anstatt N Lösungen pflegen muss
- **Fokussierung**
 - indem man sich mehr auf seine Hauptkompetenz konzentriert
 - und Nebensachen Anderen überlässt
- **Beschleunigung des Markteintritts**
 - Zusatzwirkung der Aufwandsverminderung
- **Flexibilisierung**
 - indem man ggf. mit geringeren Kosten die Lösung wechseln kann

"Produktisiko":

- Qualität ungewiss
 - Eventuell ist ein Produkt nicht hochwertig genug
 - in einer relevanten Hinsicht
 - Dadurch ist die Risiko- und Aufwandsverminderung ungewiss
- Eignung ungewiss
 - Man versteht evtl. nicht von vornherein, ob das Produkt die Anforderungen wirklich erfüllen kann
- Flexibilität eingeschränkt
 - Man kann Qualitätsmängel oder Funktionslücken evtl. nicht selbst korrigieren

"Lieferantenrisiko":

- Zwangsstandardisierung
 - Bei fremd zugelieferten Produkten geht die künftige Entwicklung evtl. in eine Richtung, die mir ungelegen ist
- Entwicklungsstillstand
 - Evtl. stellt der Hersteller die Fortentwicklung komplett ein

Beide gelten vor allem für die WV ausführbarer Komponenten

- *Etwas* weniger bei OpenSource-Komponenten
- Wiederverwendung von *Ideen* aller Art hat viel geringeres Risiko
 - → **Muster!**

Abwägung:

- **A:** Komponente K von Hersteller H zukaufen
 - Vorher nach Funktionalität auswählen
 - und Qualität überprüfen
 - Vorteile von A:
 - Implementierung gespart
 - Qualitätssicherung weitgehend gespart
 - Nachteile: Produktrestrisiko, Lieferantenrisiko
-
- **B:** Nur Anforderungen und Schnittstellen von K abgucken
 - aber Implementierung selber bauen
 - Vorteile von B:
 - Funktionalität passt sicher(?) zu meinen Anforderungen
 - Kann nötige Qualität erzwingen
 - Habe die Fortentwicklung selbst in der Hand
 - Nachteile: Höherer Aufwand (jetzt und später), Projektrisiko

Die Quantifizierung ist sehr schwierig!

Wiederverwendung existierender Elemente (Produkt, Prozess):

- "Not invented here"
 - Ingenieure mögen evtl. fremde Dinge nicht, die sie auch selbst bauen könnten
- Ignoranz, intellektuelle Faulheit
 - Man weiß zu wenig darüber, was es alles gibt
- Auswahlaufwand
 - Evtl. gibt es Dutzende unbrauchbarer und nur 1–2 brauchbare Kandidaten
 - Dann ist der Auswahlaufwand hoch ("Nadel im Heuhaufen")
 - Zusammen mit verbleibendem Produkt- und Lieferantenrisiko gibt das manchmal den Ausschlag für Eigenentwicklung

Produktion wiederverwendbarer Elemente (Produkt, Prozess):

- Zusatzaufwand
 - Ein wiederverwendbares Teil hat 2–10x so viel Entwicklungsaufwand wie ein nur für ein Projekt gestaltetes Teil
- Flexibilitätseinschränkung
 - Wenn es mehrere Benutzer eines Teils gibt, kann ich es nicht mehr nach Belieben an meine künftigen Anforderungen anpassen
 - sondern muss Rücksicht nehmen, abstimmen, informieren etc.
- Geeignete Organisationsform nötig
 - heute oft: "Inner Source"

- Wiederverwendung ist das Erfolgsmodell in der SWT
- Die Softwaretechnik ist überwiegend eine Lehre von der Wiederverwendung:
 - Architekturen
 - Komponenten
 - Methoden und Notationen
- Gemessen an produzierter Funktionalität pro Zeiteinheit sind Programmierer heute dramatisch viel produktiver als vor einigen Jahrzehnten
 - Dieser Unterschied ist praktisch komplett auf Wiederverwendung zurückzuführen:
 - Bibliotheken, Komponenten (z.B. RDBMS), Infrastrukturen
 - Betriebssysteme und Werkzeuge
 - Sprachen und Methoden

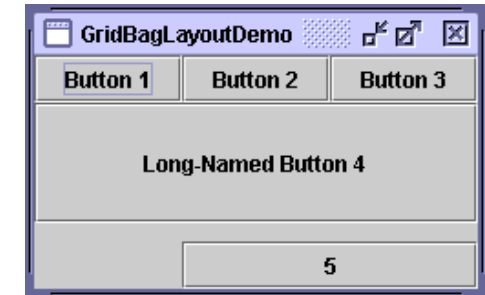
Ist die Produktivität wirklich angestiegen?

- Es wird immer mal wieder behauptet, Programmierer seien seit 40 Jahren nicht produktiver geworden.
- Diese Behauptung ist völliger Unfug
 - Sie beruht meist auf dem Maß "Programmzeilen pro Personenmonat"
 - Aber eine Programmzeile ist heute viel mehr Funktionalität wert
- Tatsächlich kann die gleiche Funktion (auf Anwendungsebene) heute meist viel schneller realisiert werden
 - Ein Datenpunkt dafür stammt von Gerald Weinberg:
 - Quality Software Management 1, 18.3.1, Seite 290
 - Er schrieb 1956 ein Programm zur Simulation von hydraulischen Netzen (Wasserleitungs-Netzen) mit ~500 Stunden Aufwand
 - 1979 schrieb er das gleiche Programm erneut:
 - 2,5 Std. Aufwand
 - Das ist eine Produktivitätsverbesserung von 20.000% oder ca. 25% jährlich!

- Wie in der Vorlesung "Die Welt der Softwaretechnik" besprochen, sollte man bei SW-Entwicklung scharf trennen zwischen
- routinemäßigen Aspekten (normales Vorgehen) und
 - innovativen Aspekten (radikales Vorgehen)
 - und sollte nur dort radikal vorgehen, wo es wirklich nötig ist
-
- Wiederverwendung bedeutet meistens auch Abstützen auf Erfahrungen und fördert somit meistens (nicht immer!) das normale Vorgehen:
 - Aus Sicht des normalen Vorgehens ist an der Wiederverwendung nur die **Risikosenkung** von Interesse
 - Die Kostensenkung und die restlichen Vorteile gibt es quasi gratis dazu
 - Es folgen Beispiele für diese Sichtweise:

- Eine komplette Softwarekomponente benutzen, *die man schon mehrmals zuvor in anderen Projekten für ähnliche Zwecke erfolgreich verwendet hat*

- z.B. eine Klasse, ein Modul, ein Subsystem



- Vorteile / Normalität:

- N1 Wir wissen aus Erfahrung, wie man die Komponente benutzt
 - und brauchen es nicht erst mühsam zu erlernen
- N3 Wir kennen die typischen Probleme dabei (Risikofaktoren)
 - z.B. Defekte, Seltsamkeiten
- N4 Wir verstehen, wofür die Komponente geeignet ist und wo ihre Grenzen liegen (Anwendbarkeitsbereich)
 - Funktionalität, Qualitätsmerkmale, Kapazitätsverhalten, Leistungsgrenzen
- N5 Wir dürfen zuversichtlich sein, ein an dieser Stelle gut funktionierendes System zu erhalten

Risikosenkung

- Eine Softwarekomponente nach den gleichen Überlegungen konstruieren, wie man es schon mehrmals zuvor in anderen Projekten für ähnliche Zwecke erfolgreich getan hat

- z.B. Einsatz eines Architekturmusters oder Entwurfsmusters

- Vorteile / Normalität:

- N1 Wir wissen, wie man die Überlegungen anwendet
- N2 Wir verstehen, was das Wesen des Musters und seiner Nützlichkeit ausmacht (Erfolgsfaktoren)
- N3 Wir kennen die typischen Stolperstellen beim Einsatz (Risikofaktoren)
- N4 Wir verstehen, welche Ziele das Muster erreichen hilft und welche nicht oder wo es gar stört (Anwendbarkeitsbereich)
- N5 Wir dürfen zuversichtlich sein, ein System mit den an dieser Stelle erwarteten Eigenschaften zu erhalten



Risikosenkung

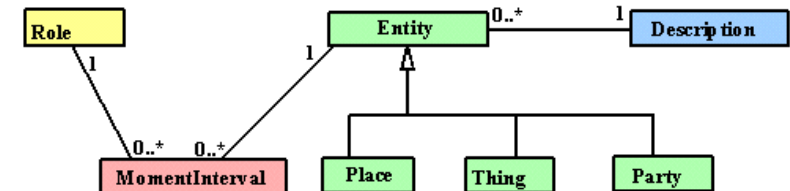
- Von einem System eine Gruppe von Eigenschaften fordern, die man bereits mehrmals zuvor in anderen Projekten für ähnliche Zwecke erfolgreich gemeinsam gefordert und umgesetzt hat

- z.B. Einsatz eines Analysemodells oder Problemrahmens

- Vorteile / Normalität:

- N1 Wir wissen, wie man diese Anforderungen hier formuliert
- N2 Wir verstehen, was an den Anforderungen essentiell ist
 - und was anwendungsspezifisch wechselt (an Formulierung od. Inhalt)
- N4 Wir verstehen, welche Eigenschaften diese Anforderungen bewirken und welche nicht (Anwendbarkeitsbereich)
 - und können die Verträglichkeit mit anderen Anforderungen prüfen
- N5 Wir dürfen zuversichtlich sein, mit diesen Anforderungen ein gewisses angestrebtes Ziel auch zu erreichen
 - und keinen Aspekt zu übersehen oder falsch zu formulieren

Business Modelling Stereotypes



- Im Projekt ein Verfahren verwenden, das man bereits mehrmals zuvor in ähnlichen Zusammenhängen für ähnliche Zwecke erfolgreich verwendet hat
 - z.B. Durchsichten, Testautomatisierung, Objektmodellierung
- Vorteile / Normalität:
 - N1 Wir wissen aus Erfahrung, wie das Verfahren gemacht wird
 - und brauchen es nicht erst mühsam zu erlernen
 - N2 Wir verstehen, worauf es dabei ankommt (Erfolgsfaktoren)
 - weil wir aus früheren Fehlern gelernt haben
 - N3 Wir kennen die typischen Probleme dabei (Risikofaktoren)
 - weil wir aus früheren (Beinahe)Fehlschlägen gelernt haben
 - N4 Wir verstehen, wofür das Verfahren geeignet ist oder nicht und was es erreichen kann oder nicht (Anwendbarkeitsbereich)
 - z.B. übertreiben wir es nicht

- Ein sprachliches oder technisches Hilfsmittel einsetzen, das man bereits mehrmals zuvor in anderen Projekten für ähnliche Zwecke erfolgreich eingesetzt hat
 - z.B. eine Notation/Sprache, einen Codegenerator

```
life+{t1 wv.^3 4=+/,^-1 0 1°.e^-1 0 1°.φ<ω}
```

- Vorteile / Normalität:

- N1 Wir kennen uns bei der Verwendung aus
 - und machen deshalb wenig Fehler
- N2 Wir verstehen, worauf es dabei ankommt (Erfolgsfaktoren)
 - z.B. geschickte anstatt ungeschickte Benutzung
- N3 Wir kennen die typischen Probleme dabei (Risikofaktoren)
 - z.B. Schwächen der Notation, Defekte des Werkzeugs
- N4 Wir verstehen, wofür das Werkzeug geeignet ist und wofür schlecht oder gar nicht (Anwendbarkeitsbereich)
- N5 Wir sind zuversichtlich, den gewünschten Effekt zu erzielen

APL

- Vor ca. 1994 sprach man von Wiederverwendung meist nur auf der Ebene von Programmcode
 - Andere Ebenen, wie Anforderungen oder Entwurf galten als sehr schwierig
- Außerdem gab es natürlich Wiederverwendung von Methoden
 - aber die zählte nicht viel
 - Wiederverwendung wurde oft als ein uneingelöstes Versprechen angesehen
- Dann erschien die Idee von Entwurfsmustern:
 - Paar aus Problem- und Lösungsbeschreibung
 - Abstrakter (und deshalb viel flexibler) als eine konkrete Lösung
- Diese Idee hat sich inzwischen in vielen Bereichen bewährt
 - Deshalb nun dazu ein paar weitere Beispiele

Arten von Mustern

- Anforderungen
 - Analysemuster ←
 - Akzeptanzkriterien
- Entwurf
 - Referenzarchitekturen
 - Architekturstile/-muster, Entwurfsmuster
 - Produktfamilien
- Benutzungsschnittstellen
 - Benutzbarkeitsmuster ←
- Management, Vorgehen
 - Prozessmuster ←
 - Best practices
 - Standards
- Allgemein
 - **Prinzipien** ←
 - Notationen ←
- Zu den markierten (←) folgen nun Erläuterungen/Beispiele

(Die Liste ist sehr unvollständig)

- **Prinzip:** Ein Grundsatz, an dem man sein Handeln orientiert
 - Sehr abstrakt; sehr allgemein; allgemein gültig
 - **Bleibt lebenslang richtig und nützlich!**
- **Methode:** Planmäßig angewandte und begründete Vorgehensweise zum Erreichen festgelegter Ziele
 - Abstrakt; meist auf einen Bereich spezialisiert
 - **Bleibt einige Jahre (ca. 2–20) lang relevant**
- **Verfahren:** Präzise und recht konkrete Vorgehensvorschrift
 - Konkret oder leicht konkretisierbar; für sehr spezifischen Bereich
- **Werkzeug:** Computerprogramm, das ein Verfahren automatisiert oder eine Methode unterstützt
 - **Bleibt oft nur kurze Zeit relevant**
- Folgerung: **Lernen Sie virtuos mit Prinzipien umzugehen!**

- Es gibt eine kleine Zahl von Grundideen, die sich in fast allen Ansätzen der Softwaretechnik wiederfinden:
 - Abstraktion
 - Strukturierung
 - Hierarchisierung
 - Modularisierung
 - Lokalität
 - Konsistenz
 - Angemessenheit
 - Wiederverwendung
- Siehe die nachfolgenden Folien

Prinzip: Abstraktion

- Beschreibe X durch etwas einfacheres, das aber hinsichtlich der relevanten Eigenschaften gleich ist
 - Insbesondere: Beschreibe viele Exemplare durch einen Typ
- ! Abstraktion ist *die* Zentralidee der gesamten Informatik
- Beispiele:
 - Eine Prozedur in einem Programm ist eine Abstraktion einer Anweisungsfolge
 - Eine Temperaturangabe ist eine Abstraktion der Molekularbewegungen in einem Gegenstand
- (Achtung, Obiges ist eine stark verkürzte Darstellungsform):

- Um die vorherige Folie in ein richtiges Muster zu verwandeln, müsste man einiges ergänzen:
 - Anwendungsbereich
 - Klarere Trennung von Problem und Lösung
 - Vorteile
 - Nachteile und Abwägungen
 - Mögliche Variationen
- Ich gehe davon aus, dass es bei den Prinzipien reicht, sie überhaupt bewusst zu machen
 - und spare uns aus Zeitgründen die genauere Diskussion

Prinzip: Strukturierung

- Mache etwas Kompliziertes verständlich, indem Du den Einzelteilen ausdrücklich eine bestimmte Rolle oder Bedeutung im Ganzen zuweist
- ! Strukturierung verwendet immer auch Abstraktion
- Beispiele:
 - Erkläre den Softwareentwicklungsprozess als zusammengesetzt aus Anforderungsbestimmung, Entwurf, Implementierung, Test etc.
 - Erkläre eine Sprache durch eine Grammatik

Prinzip: Hierarchisierung

- Schaffe Übersicht bei einer großen Zahl von Teilen, indem Du
 - Jeweils einige Teile zusammenfasst
 - Und nötigenfalls jeweils Zusammenfassungen wieder als Teile betrachtest
 - In solch einer Weise, dass jede solche Zusammenfassung eine Bedeutung bekommt und zu einem Begriff wird
- Andere Sicht: Ordne eine Menge von Teilen als Blätter in eine Baumstruktur und mache dir die inneren Knoten zunutze
- ! Hierarchisierung ist ein Spezialfall von Strukturierung
- Beispiel:
 - Zusammenfassen von Dateien in Verzeichnisbäumen

Prinzip: Modularisierung

- Mache ein System mit einer großen Zahl von Funktionsteilen konstruierbar, indem Du
 - das gedachte Ganze zerlegst in nichttriviale Teile ("Module")
 - so dass jedes Modul einen klar beschriebenen Zweck erfüllt ("Schnittstelle", "Interface"),
 - das Verstehen und Benutzen eines Moduls von außen einfacher ist als das Verstehen aller seiner inneren Teile (Komplexitätsreduktion)
 - und ein Modul auf möglichst wenig Eigenschaften anderer Module angewiesen ist ("geringe Kopplung", "Trennung von Belangen").
- ! Modularisierung ist ein Spezialfall von Hierarchisierung und von Abstraktion
- Beispiel:
 - Batterie: Interface sind Bauform, Spannung, Kapazität; Innereien sind Bauart (z.B. Alkali, NiMH), Gehäusematerial, etc.

- Versammle alle Informationen, die zum Verstehen eines Teils oder einer Eigenschaft nötig sind, möglichst an einem Ort
- Beispiele:
 - Javadoc (Code und Dokumentation an einem Ort)
 - Java (versus C++: Klassendeklaration und Implementierung in einer Datei)

Prinzip: Konsistenz

- Handhabe gleichartige Dinge möglichst stets auf die gleiche Weise, um Verwirrung und Irrtümer zu vermeiden
- ! Konsistenz erfordert zuvor oft die Bildung einer Abstraktion
 - Gleichartigkeit
- Beispiele:
 - Warnlampen sind immer rot
 - Methodennamen sind immer Imperative
 - Ein Dialog kann immer mit der Taste ESC abgebrochen werden

Prinzip: Angemessenheit

- Bei der Auswahl einer Lösung zu einem Problem ist zu berücksichtigen, inwieweit der erzielte Nutzen den betriebenen Aufwand rechtfertigt
- ! Angemessenheit ist ein Handlungsprinzip aller Ingenieure
- Beispiele:
 - Komplexe technische Apparaturen nur einsetzen, wo nötig
 - Nur um Emails zu schreiben
braucht man keine Hochleistungs-Grafikkarte
 - Nur um eine private Musiksammlung zu verwalten
braucht man kein verteiltes DBMS

Prinzip: Wiederverwendung

- Vermeide die Konstruktion komplexer Teile oder Ideen
 - Suche, ob es ein gleichwertiges Teil schon gibt
 - Oder ein ähnliches, zu dem Du Deine Anforderungen abwandeln kannst
 - Oder ein allgemeineres, das Du passend ausprägen kannst
- ! Wiederverwendung ist ein Spezialfall von Angemessenheit
- Beispiel:
 - Verwende ein existierendes Web-Framework anstatt selbst eines zu bauen
 - Verwende ein etabliertes Prozessmodell anstatt selbst ein ganz neues zu definieren
- Gegenstück: Wiederverwendbarkeit
 - "Vermeide die Konstruktion hoch spezialisierter Teile"

Auch Notationen (z.B. UML, Programmiersprache u.a.) können als Muster aufgefasst werden:

- **Problem:** Softwaretechnik ist auf die Zusammenarbeit mehrerer angewiesen; deshalb müssen Begriffe und Aussagen eindeutig wiederholbar festgehalten werden können
- **Lösungsidee:** Notation: Darstellung relevanter Konzepte durch festgelegte Menge von Symbolen mit definierter Syntax und Semantik
- **Abwägungen:**
 - Notationen sind nur Hilfsmittel zu einem Zweck; jede Notation kann manche Dinge besser ausdrücken als andere; deshalb ist wichtig, dass die jeweilige Notation gut zur Aufgabe passt
 - z.B. domänenspezifische Sprachen (DSLs), in der Praxis ein wichtiges Element von Model-Driven-Architecture (MDA)
 - Andererseits sind Definition und Erlernen von Notationen aufwändig; deshalb sind oft allgemeine Notationen zweckmäßig

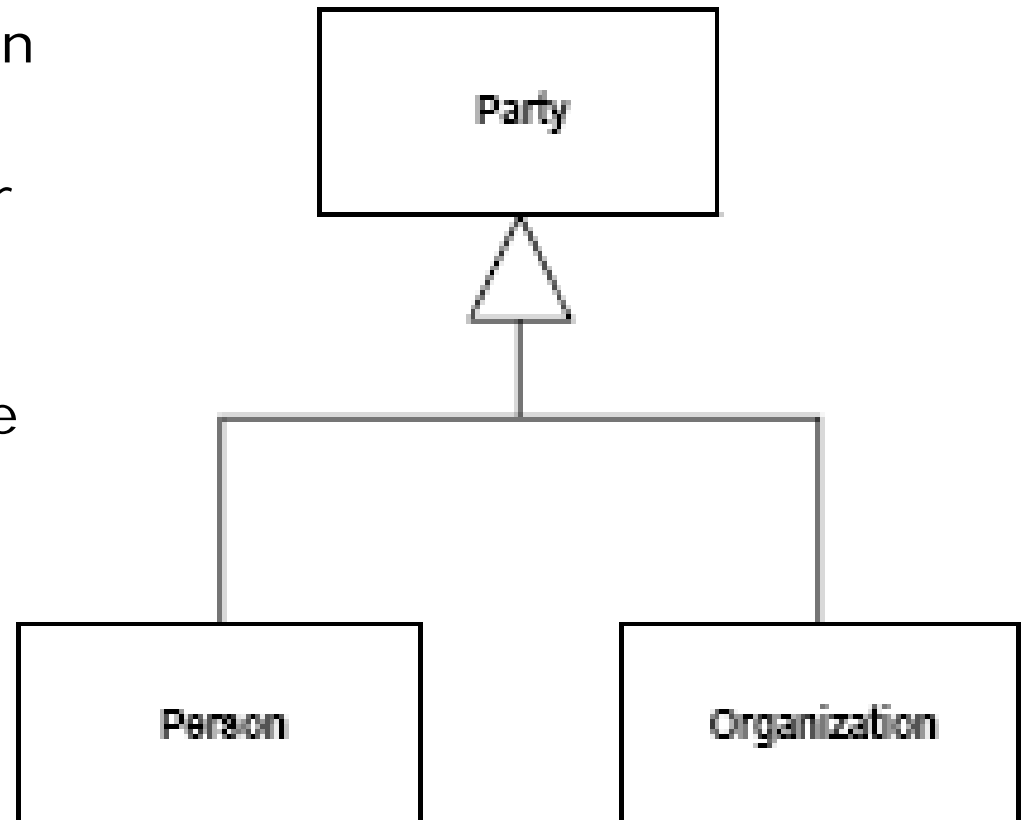
Arten von Mustern

- Anforderungen
 - **Analysemuster** ←
 - Akzeptanzkriterien
- Entwurf
 - Referenzarchitekturen
 - Architekturstile/-muster, Entwurfsmuster
 - Produktfamilien
- Benutzungsschnittstellen
 - Benutzbarkeitsmuster ←
- Management, Vorgehen
 - Prozessmuster ←
 - Best practices
 - Standards
- Allgemein
 - Prinzipien ←
 - Notationen ←
- Zu den markierten (←) folgen nun Erläuterungen/Beispiele

- Im Rahmen der Anforderungsanalyse tauchen bei der Objektbestimmung in vielen Domänen ähnliche Gruppen von Objekten mit ähnlichen Anforderungen auf
- Analysemuster identifizieren solche Gruppen von Klassen und beschreiben die Struktur ihres Zusammenwirkens
 - Der Übergang zu Entwurfsmustern ist fließend
 - Der Schwerpunkt liegt hier aber auf den Domänenobjekten und ihren Beziehungen
 - nicht auf ihren Schnittstellen
 - nicht auf Abläufen und Verhalten
- Wir betrachten hier nur ein einziges **Beispiel**:
 - Eine Gruppe von Mustern ("Mustersprache") zur Abbildung von Organisationshierarchien

Beobachtung 1: Personen versus Organisationen

- An vielen Stellen muss man in SW sowohl Personen als auch Organisationen behandeln
- Meist sind weite Teile der Behandlung gleich
- Also führt man dafür eine Oberklasse ein:



Beobachtung 2: Organisationen sind hierarchisch

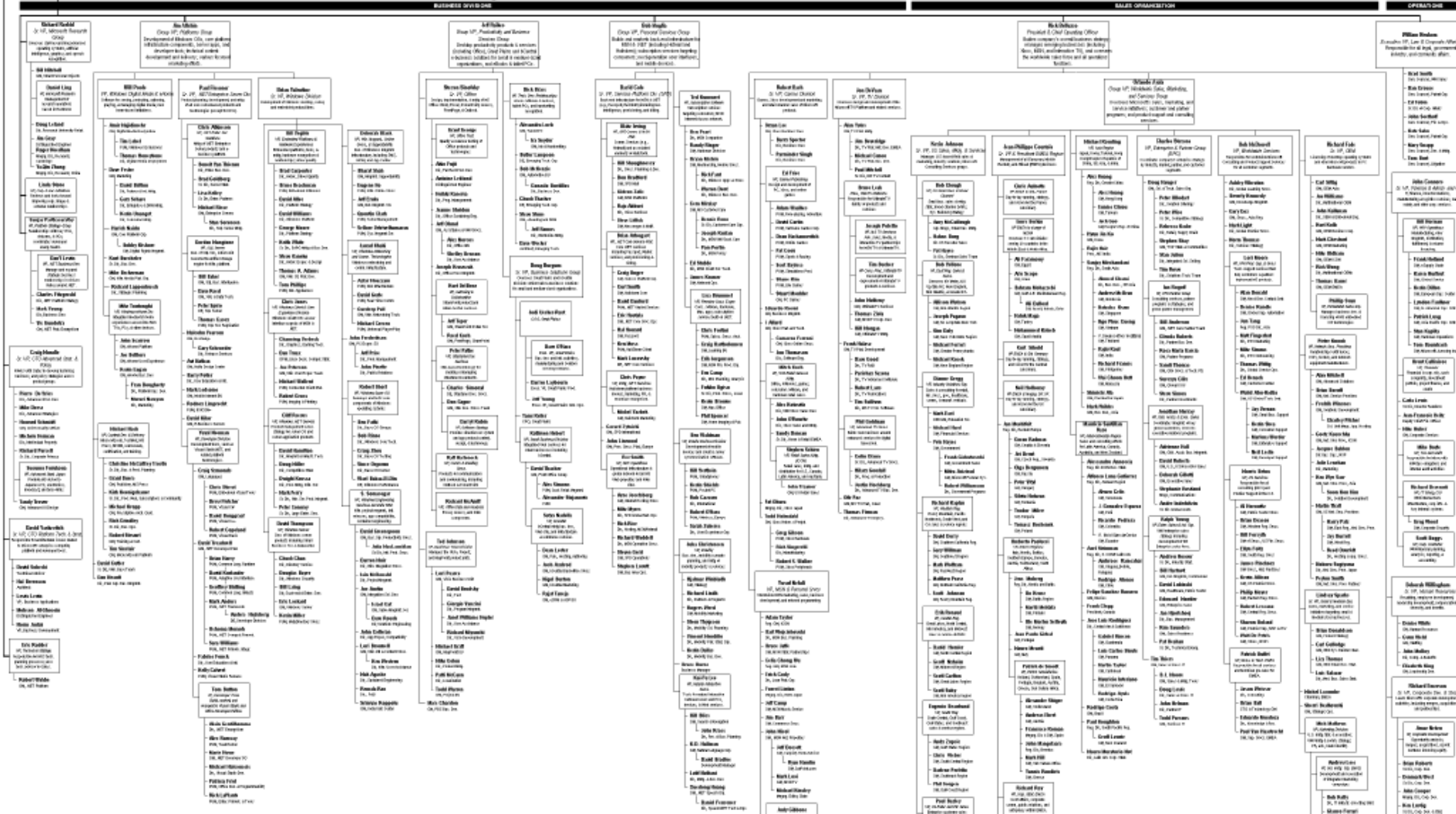
- Die Organisationen haben meist eine Struktur aus Teilorganisationen
 - und die Teilorganisationen bestehen aus Personen



Beispiel einer Organisationshierarchie

The Microsoft OCTOBER 2001 Corporate Organization

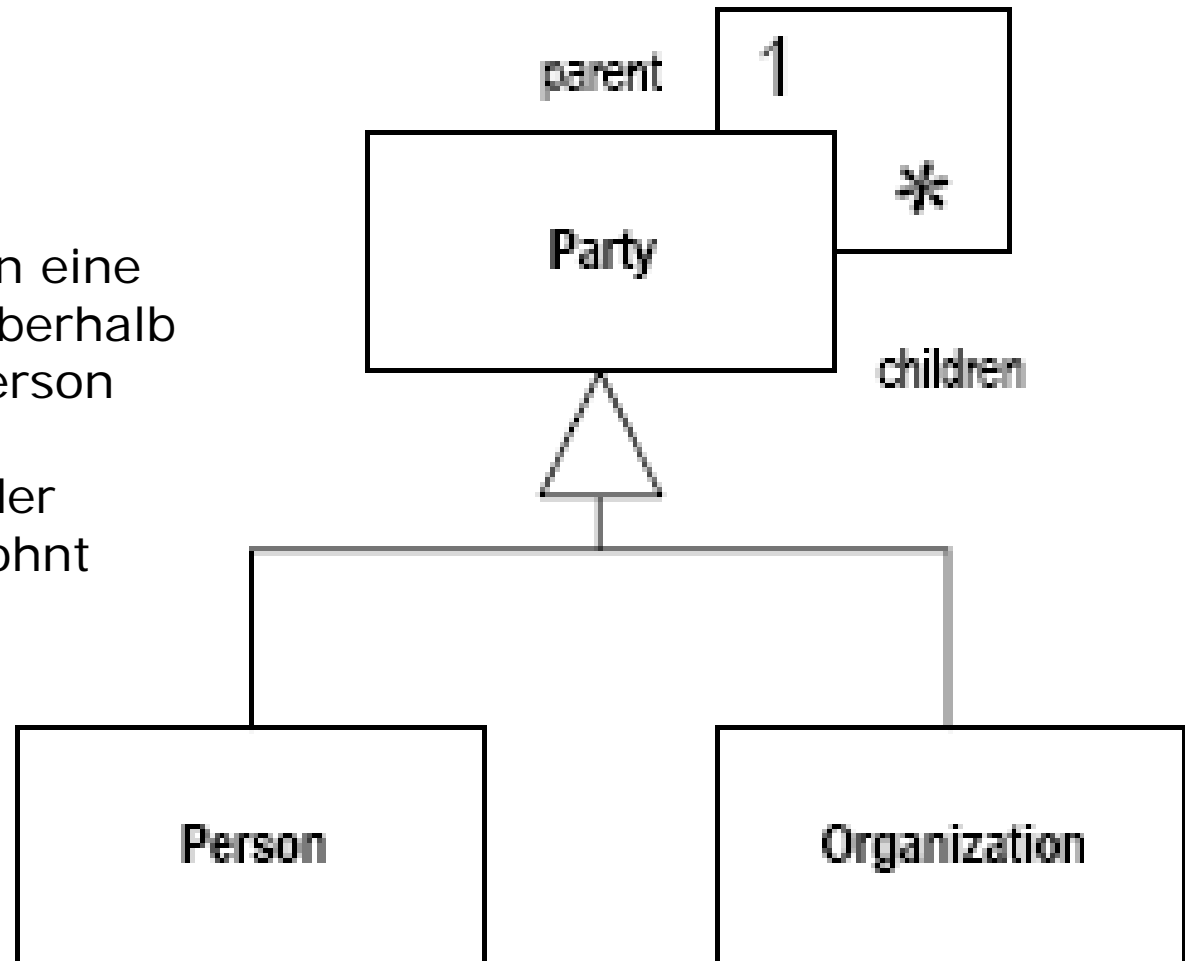
Directions
www.DirectionsOnMicrosoft.com



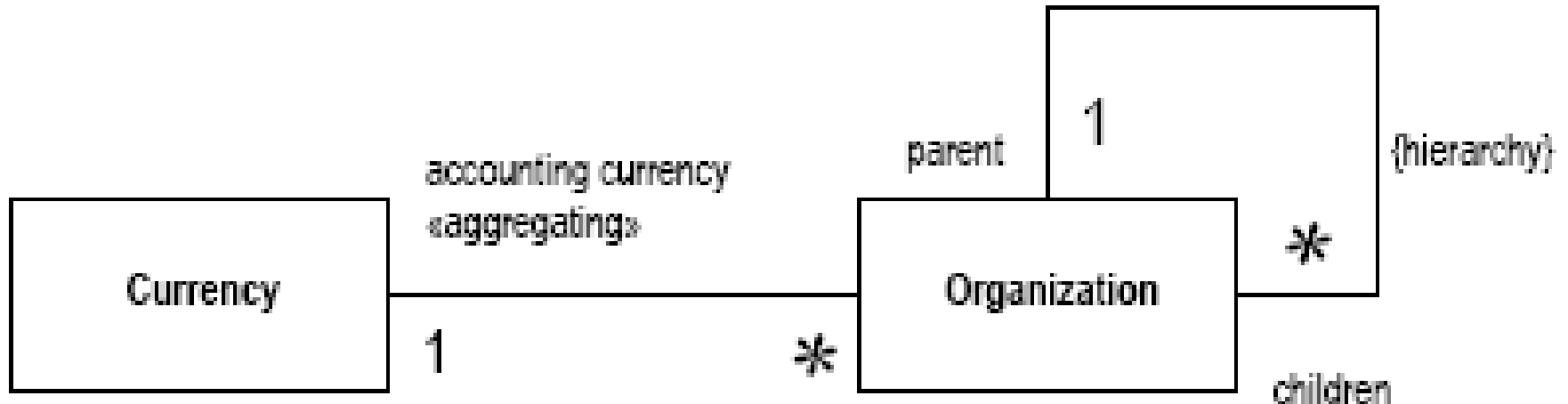
{hierarchy}

(Bei dieser Lösung kann eine Teilorganisation auch oberhalb der Blatt-Ebene eine Person sein.

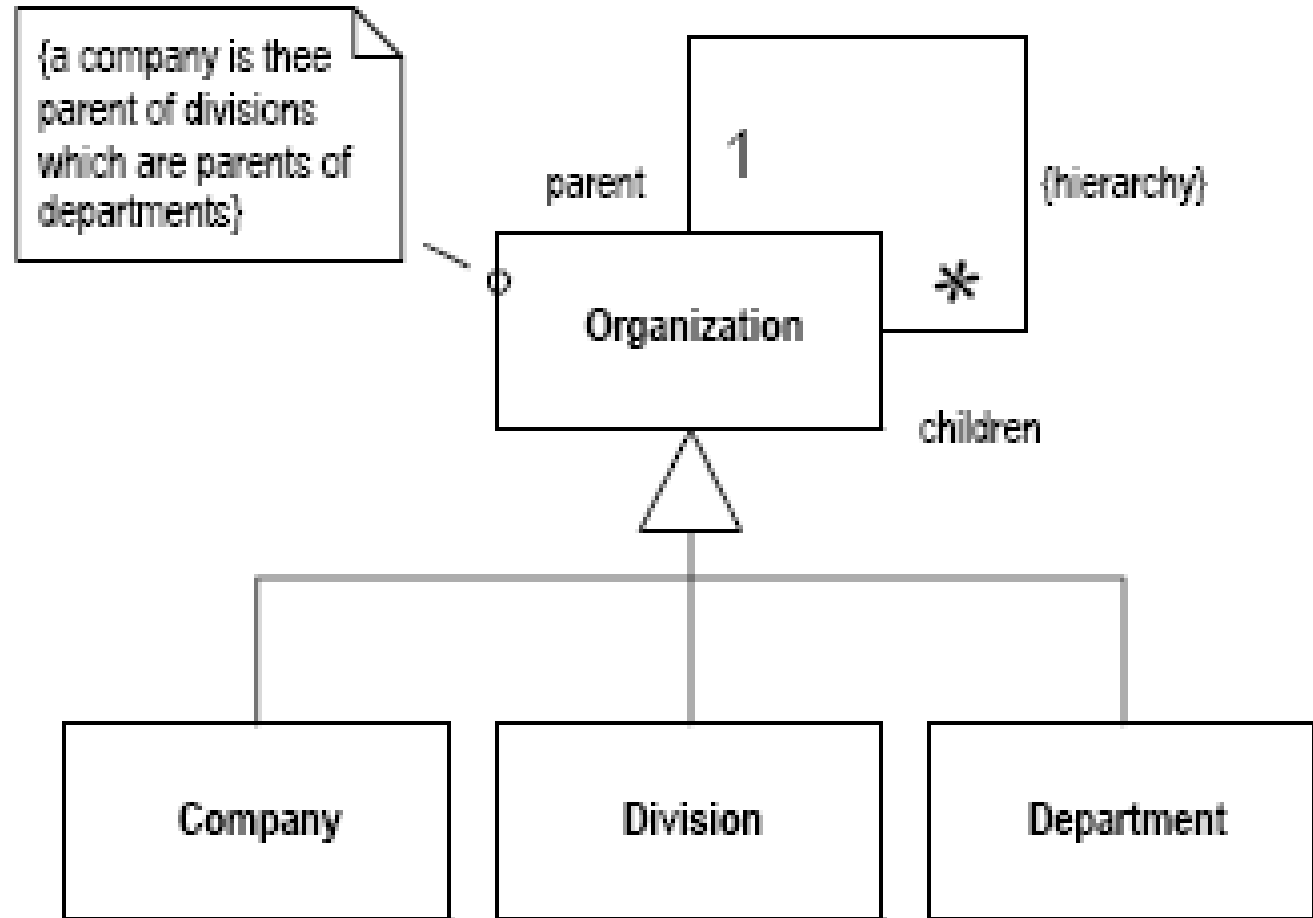
Das kann erwünscht oder unerwünscht sein, es lohnt aber evtl. nicht, das zu verhindern.)



- Man kann Attributwerte entlang der Hierarchie nach unten weitergeben
 - Nicht direkt in UML ausdrückbar, deshalb als Stereotyp notiert
 - Es wird also nicht nur eine Exemplarvariable vererbt, sondern derer aktuelle Wert wird dynamisch von oben nach unten durchgereicht

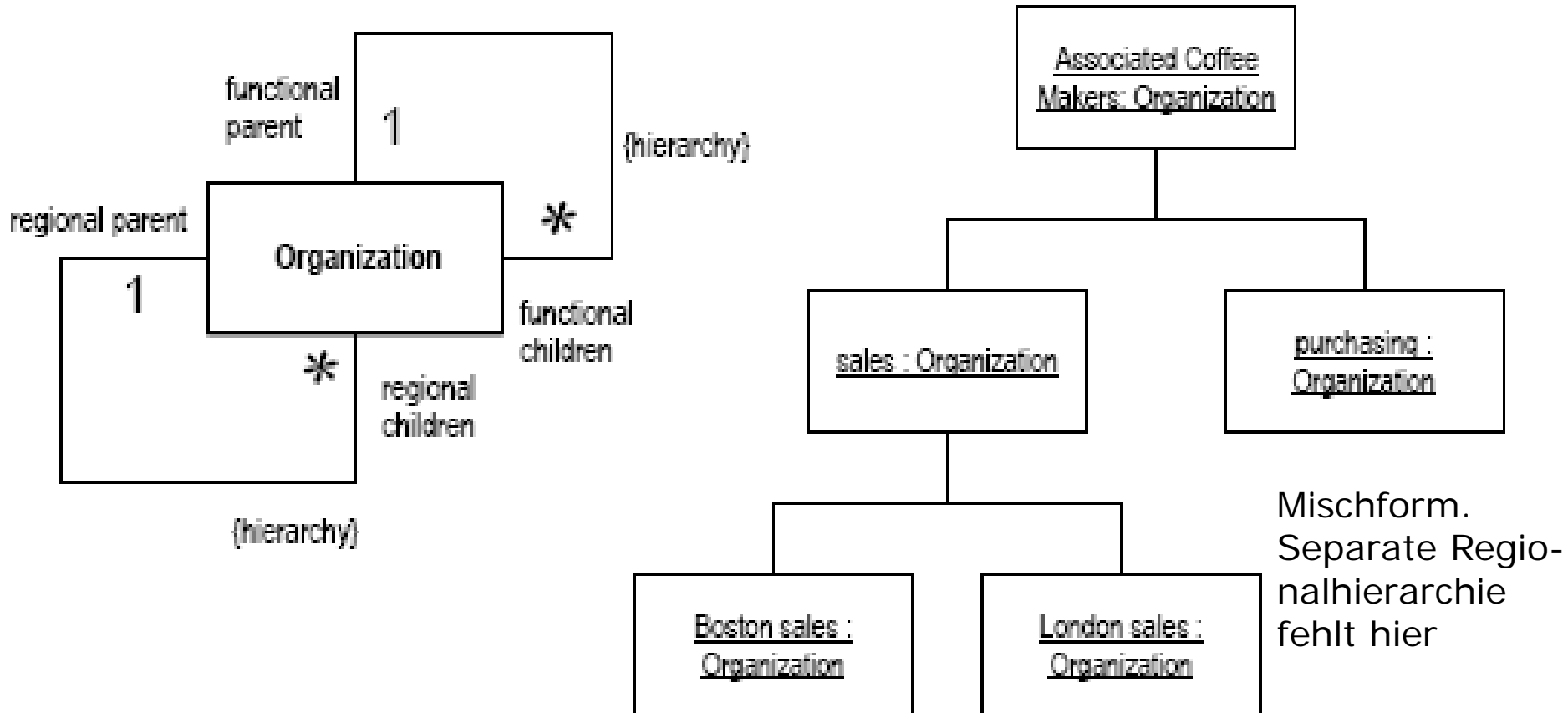


- Eventuell ist es sinnvoll, die einzelnen Stufen (oder manche davon) dennoch durch eigene Klassen darzustellen



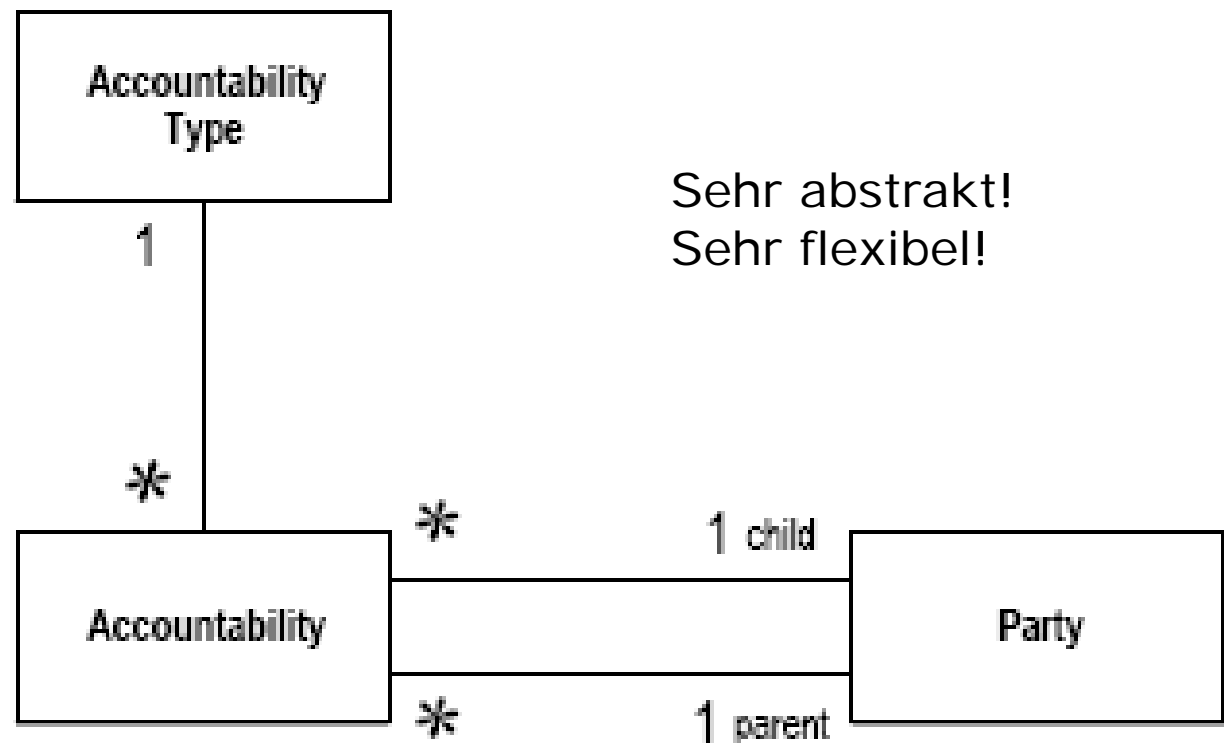
Mehrfache Hierarchien

- Eventuell will man Teilorganisationen nach mehr als einem Kriterium in Hierarchien einordnen
 - z.B. Funktionshierarchie und Regionalhierarchie



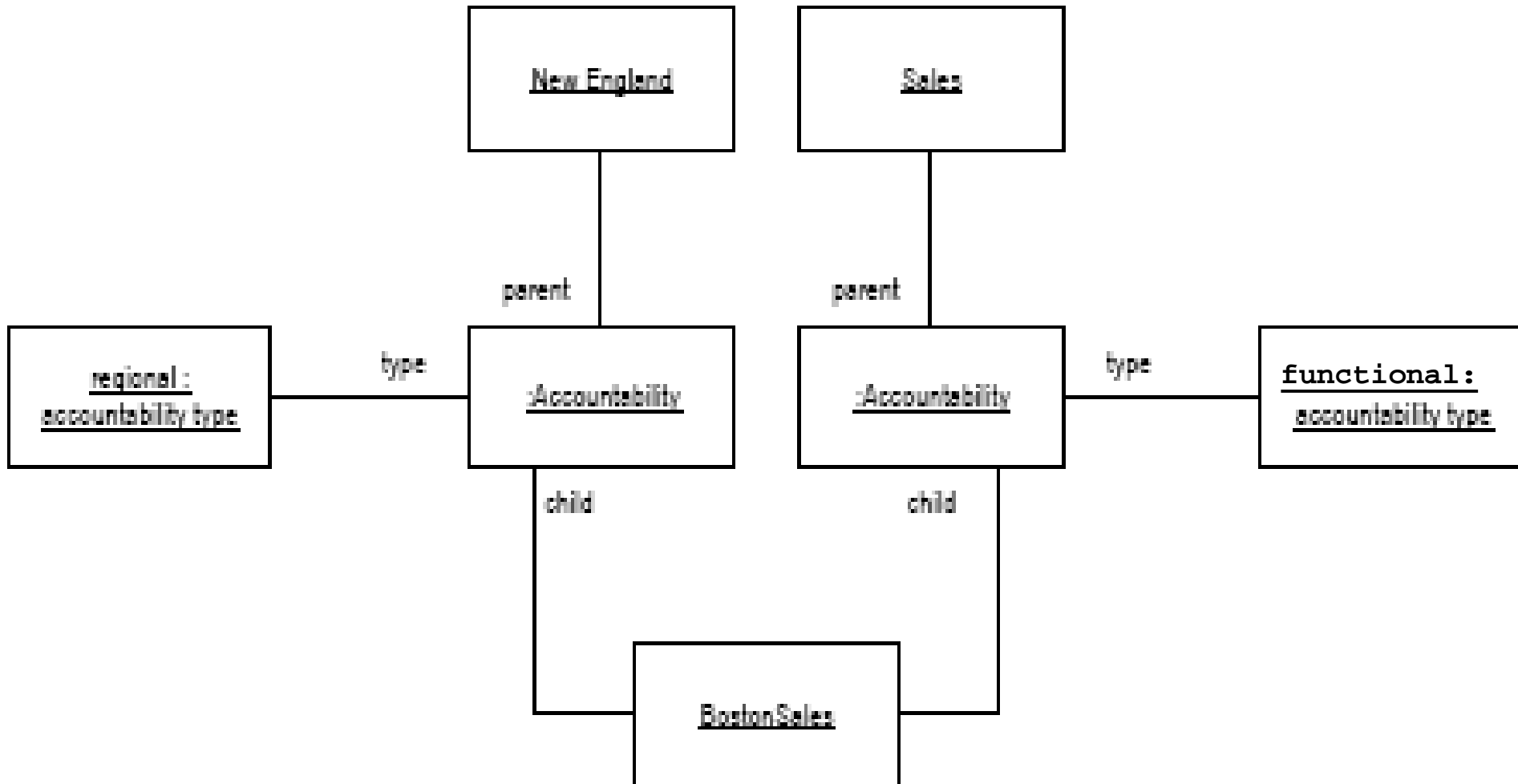
Verallgemeinerte Hierarchien: Verantwortlichkeit

- Ein Verantwortlichkeitsobjekt (Accountability) verbindet eine Oberorganisation (parent) mit einer Unterorganisation (child)
 - Die Art der Verantwortlichkeit wird beschrieben durch ein Beziehungsobjekt (AccountabilityType-Objekt, z.B. "Funktionsgliederung", "Regionalgliederung")



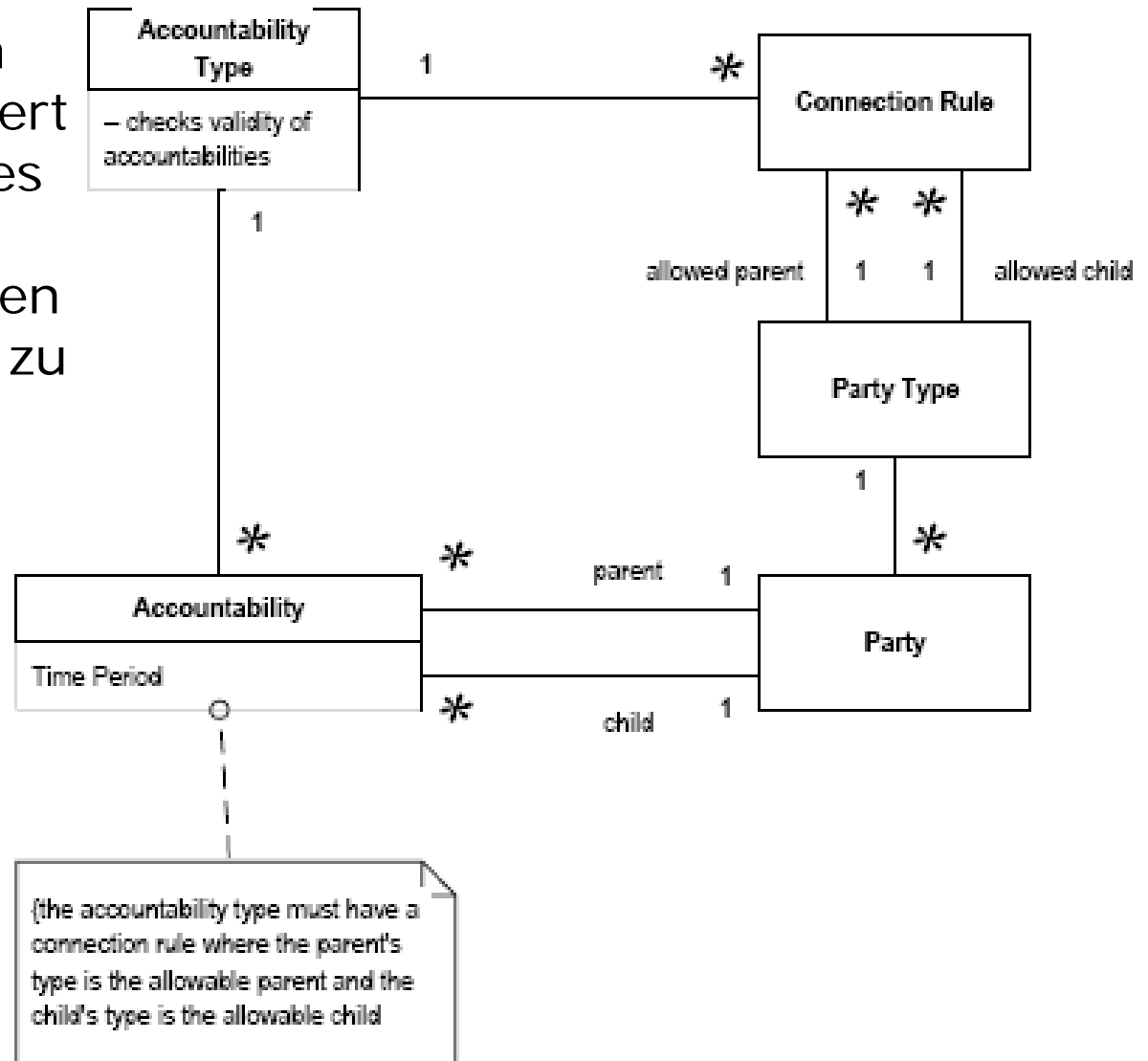
Sehr abstrakt!
Sehr flexibel!

Verantwortlichkeit: Beispiel



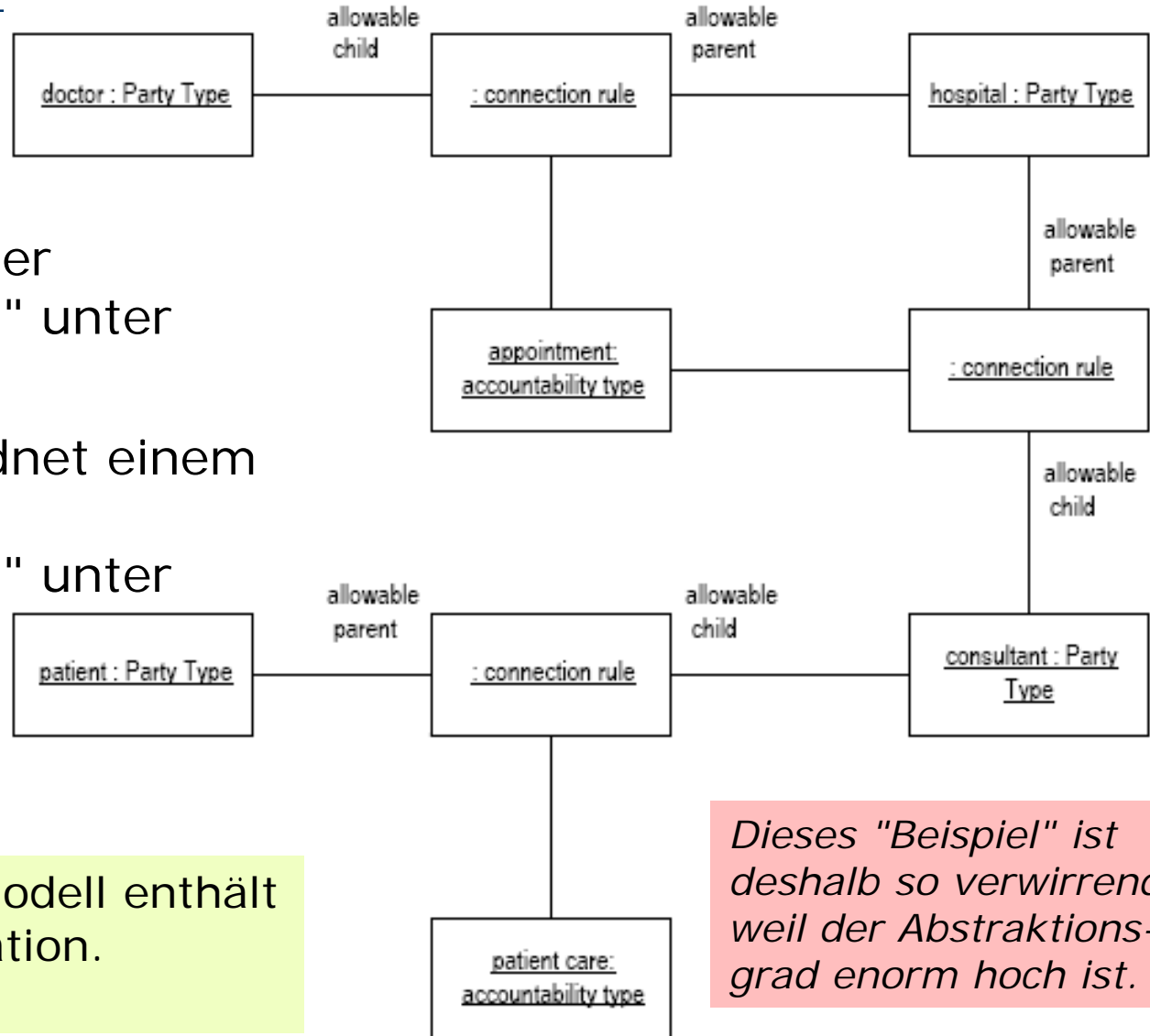
Verantwortlichkeiten mit Regeln

- Sollen Beziehungen zur Laufzeit verändert werden, empfiehlt es sich, die *Regeln* für erlaubte Beziehungen auch mit im Modell zu repräsentieren
 - ConnectionRule
 - PartyType



Verantwortlichkeiten mit Regeln: Beispiel

- "appointment" ordnet einem "hospital" einen "doctor" oder einen "consultant" unter
- "patient care" ordnet einem "patient" einen "consultant" unter
 - (das ist aber keine Organisationshierarchie)

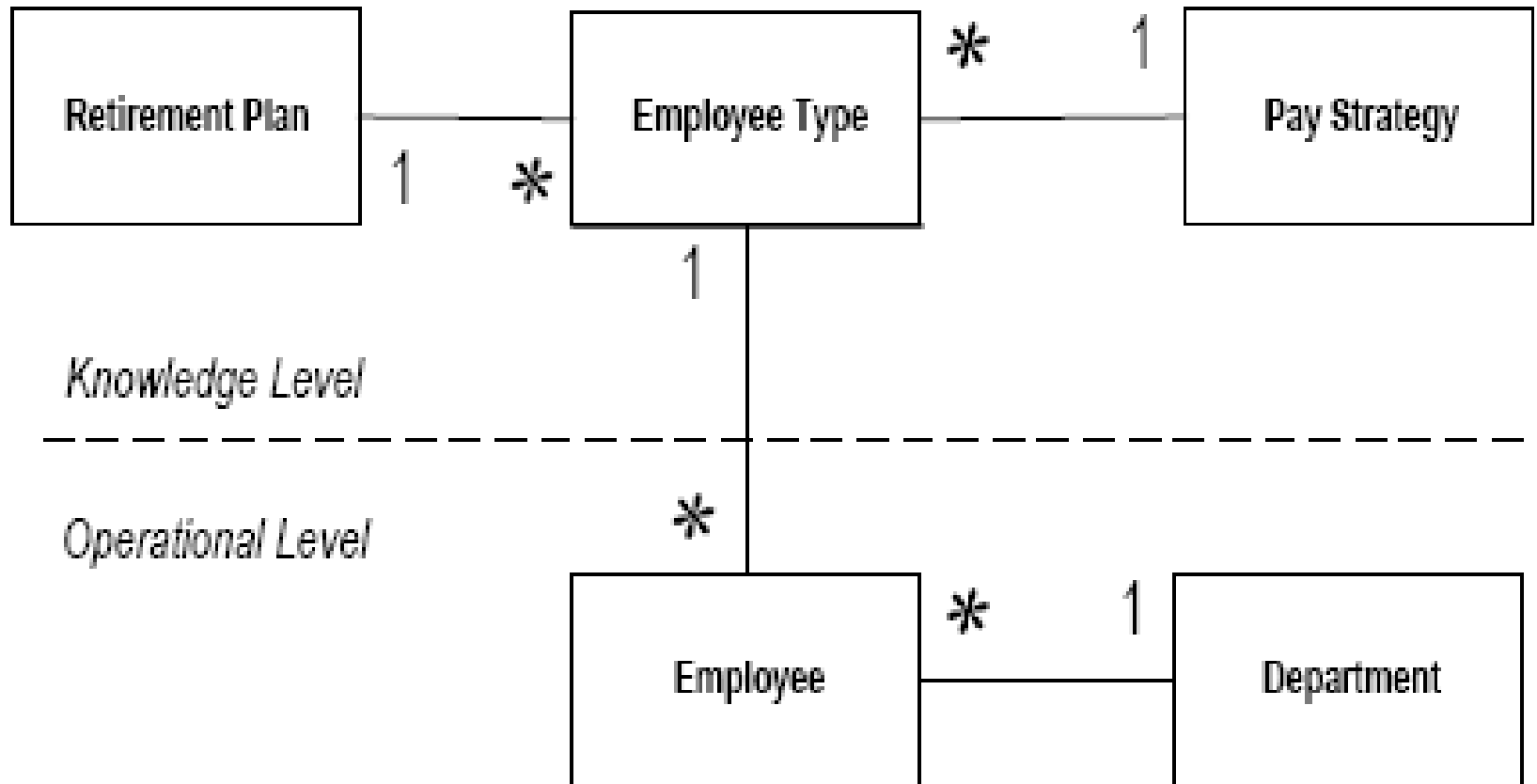


Achtung: Das ganze Modell enthält keine einzige Organisation. Alles nur Metamodell!

Dieses "Beispiel" ist deshalb so verwirrend, weil der Abstraktionsgrad enorm hoch ist.

Verallgemeinerung: Metadaten

- Die Grundidee hinter ConnectionRule lässt sich oft anwenden: Trenne zwischen Inhaltsdaten (operational data) und Beschreibungsdaten (meta data, hier: "knowledge level")



Teil 2:

Arten von Mustern

- Anforderungen
 - Analysemuster ←
 - Akzeptanzkriterien ←
- Entwurf
 - Referenzarchitekturen
 - Architekturstile/-muster, Entwurfsmuster
 - Produktfamilien
- Benutzungsschnittstellen
 - **Benutzbarkeitsmuster** ←
- Management, Vorgehen
 - Prozessmuster ←
 - Best practices
 - Standards
- Allgemein
 - Prinzipien ←
 - Notationen ←
- Zu den markierten (←) folgen nun Erläuterungen/Beispiele

- Jedes Benutzbarkeitsmuster ist eine Regel darüber, wie sich interaktive Software verhalten sollte, um gut benutzbar zu sein
- Die allgemeineren dieser Regeln sind inzwischen recht bekannt, deshalb hier nur in Kurzform
 - d.h. Angabe der Lösung ohne
 - Beschreibung von Kontext und Problem
 - Aufgliederung in einzelne Teilanforderungen ("Verantwortlichkeiten")
 - Diskussion von Varianten oder möglichen Nachteilen
- Es folgt ein Katalog solcher Benutzbarkeitsmuster
 - Genannt "usability scenarios" (Bass und John, 2001)
 - Achtung: Diese Muster sind unterschiedlich nützlich, unterschiedlich häufig anwendbar, stellenweise etwas einseitig und die Liste ist ganz sicher nicht vollständig -- aber ein Anfang.

Benutzbarkeit ist schwierig

Mantra der
Benutzbarkeitsspezialisten:

- "Benutzer sind anders als wir"
 - Sie brauchen andere Funktionen und Eigenschaften von SW als wir



CUNO FAND DIE SPRACHFUNKTION SEINES PCS NUR AM ANFANG WITZIG.

- Aggregating Data
 - Ermögliche, dieselbe Operation auf mehreren Datenelementen zugleich auszuführen
- Aggregating Commands
 - Ermögliche, mehrere verschiedene Operationen zusammen auszuführen (Makros)
- Canceling Commands
 - Ermögliche, längere Operationen unterwegs abubrechen
- Using Applications Concurrently
 - Ermögliche die gleichzeitige (evtl. verschachtelte) Nutzung mehrerer Anwendungen ohne Konflikte
 - z.B. Spracherkennung in Textverarbeitung ohne Konkurrenz um Cursor

- Checking for Correctness
 - Prüfe Benutzereingaben und zeige Warnungen
 - Biete evtl. vollautomatische Korrektur an (z.B. "hte" → "the")
- Maintaining Device Independence
 - Arbeite einheitlich und konfliktfrei mit allen in Frage kommenden Eingabe-, Ausgabe- und Speichergeräten zusammen
- Evaluating the System
 - Baue Mechanismen zur Unterstützung von Benutzbarkeitstests gleich mit ins System ein
- Recovering from Failure
 - Minimiere den möglichen Datenverlust bei HW- oder SW-Ausfällen

- Retrieving Forgotten Passwords
 - Sehe einfache, aber hinreichend sichere Mechanismen vor, wie Benutzer nach Vergessen des Passworts wieder Zugang erlangen können
- Providing Good Help
 - Biete Hilfe kontextabhängig und detailliert genug an, um ein Problem auch wirklich zu lösen
 - Hilfe sollte dafür sorgen, dass die Benutzer die Grundkonzepte des Systems verstehen
- Reusing Information
 - Erlaube, Daten leicht von und zu anderen Anwendungen zu übertragen (import, export, cut/paste, etc.)
- Supporting International Use
 - Unterstütze die Umstellung auf andere Sprachen (Zeichensätze, Fonts, Tastaturen etc.)
 - Unterstütze Umstellung auf andere Kulturen (Farben, Symbole)

- Leveraging Human Knowledge
 - Lehne Bedienung an bekannte Software an
 - insbesondere an ggf. frühere Versionen desselben Produkts
- Navigating Within a Single View
 - Erlaube schnelle und einfache Navigation auch in großen Datenmengen und ohne die Ansicht verlassen zu müssen
- Observing System State
 - Zeige alle benutzerrelevante Information über den Systemzustand in verständlicher und übersichtlicher Form an
- Working at the User's Pace
 - Passe die Geschwindigkeit automatischer Aktionen (z.B. Rollen, Timeouts) der Arbeitsweise der Benutzer an (+Einstellbarkeit)

- Predicting Task Duration
 - Zeige bei längeren Arbeitsphasen des Rechners deren voraussichtliche Dauer an
- Supporting Comprehensive Searching
 - Erlaube Suche flexibel über alle Daten mittels aller benutzerrelevanter Kriterien
- Supporting Undo
 - Erlaube, Aktionen rückgängig zu machen; auch mehrere hintereinander
- Working in an Unfamiliar Context
 - Biete einen Anfängermodus an, in dem viele Erklärungen angezeigt werden

- Verifying Resources
 - Prüfe vor einer Operation, ob diese vermutlich erfolgreich ausgeführt werden kann, insbesondere alle Ressourcen verfügbar sind (typischer Fall: Plattenplatz)
- Supporting Visualization
 - Biete mehrere strukturell verschiedene Ansichten der Daten an
- Operating Consistently Across Views
 - Stelle in verschiedenen Ansichten der gleichen Daten möglichst immer die selben Operationen zur Verfügung
- Making Views Accessible
 - Umgekehrt sollten während aller Operationen und Modi möglichst alle Ansichten leicht zugänglich bleiben

Puh. Viel Arbeit!

Eine Gruppierung der Anforderungen (als Auswahlhilfe):

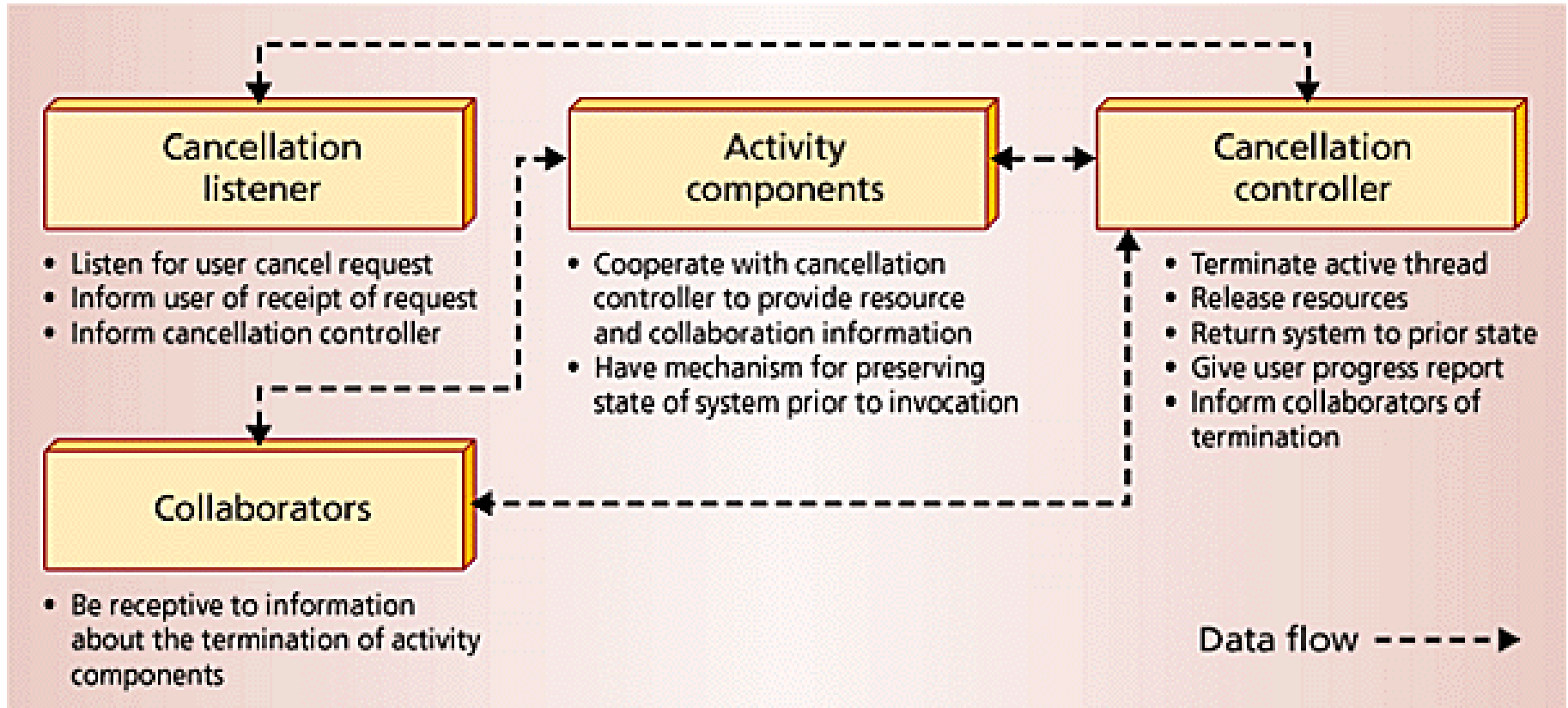
- Increases individual user effectiveness
 - *Expedites routine performance*
 - Accelerates error-free portion of routine performance
 - Reduces the impact of routine user errors (slips)
 - *Improves non-routine performance*
 - Facilitates learning
 - Supports problem-solving
 - *Reduces the impact of user errors caused by lack of knowledge*
 - Prevents mistakes
 - Accommodates mistakes
- Reduces the impact of system errors
 - *Prevents system errors*
 - *Tolerates system errors*
- Increases user confidence and comfort

- Usability-supporting architectural patterns (USAP)
 - beschreiben den Zusammenhang zwischen den obigen Benutzbarkeitsanforderungen und den Mechanismen zu Ihrer Implementierung

Zunächst werden Basismechanismen identifiziert und beschrieben:

- Separation
 - Encapsulating function
 - Sep. data from commands
 - Sep. data from the view
 - Sep. authoring from execution


- Replication
 - Data
 - Commands
- Indirection
 - Data
 - Function
- Recording
- Preemptive scheduling
- Models
 - Task
 - User
 - System
- Jedes USAP erläutert für eine Anforderung, wie die Mechanismen einzusetzen sind.
 - Beispiel:



- Muster für Einzelplatz-Desktop-Anwendungen
 - Im Original natürlich mit genauerer verbaler Beschreibung

Arten von Mustern

- Anforderungen
 - Analysemuster ←
 - Akzeptanzkriterien ←
- Entwurf
 - Referenzarchitekturen
 - Architekturstile/-muster, Entwurfsmuster
 - Produktfamilien
- Benutzungsschnittstellen
 - Benutzbarkeitsmuster ←
- Management, Vorgehen
 - **Prozessmuster** ←
 - Best practices
 - Standards
- Allgemein
 - Prinzipien ←
 - Notationen ←
- Zu den markierten (←) folgen nun Erläuterungen/Beispiele

- Prozessmuster lassen sich auf unterschiedlichen Ebenen formulieren:
 - Projektmanagement 
 - Wer macht was wann und wie wird geplant?
 - "Technische" Aufgaben
 - Vorgehensweisen für z.B. Durchsichten, Entwurfsdiskussionen, Systemfreigaben
 - Rahmenbedingungen
 - Aufbau der Organisation, Verantwortlichkeiten, Form der Zusammenarbeit
- Wir besprechen im Folgenden einige Projektmanagement-Muster
 - Muster über Planung und Termineinhaltung
 - Muster über Umgang mit Störungen

- Jemand erzählt:
 - "Wir übertrugen ein erfolgreiches Produkt auf eine völlig neue Technologie. Der Erfolg unserer Prototypen gab uns das Vertrauen, ein Lieferdatum festzulegen.
 - Leider ging unser Lieferzeitpuffer verloren. Daraufhin rief der Projektleiter alle zusammen, gab eine lange Terminverzögerung bekannt und erhielt von allen wichtigen Projektmitgliedern gemeinsame Zustimmung zum neuen Plan.
 - Wir hielten diesen neuen Plan ein und lieferten sehr gute Qualität: Teile des Produktes wurden noch viele Jahre später in anderen Projekten wiederverwendet."
- Die unterstrichenen Phrasen deuten auf die Anwendung von Prozessmustern
 - Wir besprechen diese (und einige mehr) auf den nächsten Folien
 - Beachte: Muster beschreiben stets etwas **Bewährtes**, nicht etwas *Neues*

Muster: Baue Prototypen (*build prototypes*)

- Wenn
 - Anforderungen genauer verstanden oder mit Kunden validiert werden müssen
 - Bedienbarkeit überprüft werden soll
 - oder die Vor- und Nachteile wichtiger Entwurfsentscheidungen unklar sind
- dann
 - baue einen Prototyp: Ein möglichst kleines System, das nur den Zweck hat, diese Fragen zu klären und daraus zu lernen
 - Wirf den Prototyp anschließend weg (den Code, nicht unbedingt den Entwurf)
- Vorteile:
 - Senkt das Risiko der anschließenden Entwicklung

Muster: Liefertermin festlegen (*size the schedule*)

- Wenn
 - die Anforderungen stabil und verstanden sind und
 - der Projektumfang geschätzt werden kann
- dann
 - lege einen verbindlichen Liefertermin gegenüber dem Kunden fest
 - lege einen etwas früheren verbindlichen Abschlusstermin in Abstimmung mit dem Projektteam fest
 - (die Differenz erzeugt einen Liefertermin-Puffer)
 - **belohne das Team, wenn es diesen Termin einhält**
- Vorteile:
 - Erzeugt klare Orientierung und gute Motivation
 - Es ist meistens sinnvoller, nötigenfalls die Funktionalität zu reduzieren als den Liefertermin hinauszuschieben

Muster: Beobachte den Liefertermin-Puffer (*completion headroom*)

- Wenn
 - ein fester Liefertermin eingehalten werden muss
 - und ein realistischer Zeitplan dafür vorliegt
- dann
 - aktualisiere ständig (z.B. wöchentlich) die Restzeit-Schätzungen
 - beobachte die Größe des Zeitpuffers zwischen Schätzung und verbindlichem Termin
 - und ergreife Gegenmaßnahmen, wenn der Puffer verschwindet und dauerhaft verschwunden bleibt
- Vorteile:
 - Kontinuierlich hohe Glaubwürdigkeit der Terminplanung
 - Rechtzeitiges Warnsignal, sobald ernste Terminschwierigkeiten auftauchen



Muster: Keine kleinen Terminverzögerungen (*take no small slips*)

- Wenn
 - der Liefertermin-Puffer negativ wird
 - und sich ein Trend von weiteren Verzögerungen einstellt
- dann
 - mache eine neue Liefertermin-Schätzung
 - und sehe genug zusätzliche Zeit vor, dass später keine weiteren Verzögerungen des Termins nötig werden
- Vorteile:
 - Vermeidet Verluste, die durch zu großzügige anfängliche Terminsetzungen entstehen können
 - Vermeidet Ausbrennen von Entwicklern durch dauerhaft zu hohen Termindruck; sorgt für Glaubwürdigkeit der Terminplanung
 - Vermeidet Verärgerung von Kunden durch mehrfache Verzögerungen

Muster: Gemeinsame Zustimmung zum neuen Plan (*recommitment meeting*)

- Wenn
 - der bisherige Zeitplan eindeutig nicht eingehalten werden kann
 - und Hilfsmaßnahmen wie Wochenendarbeit oder Zusatzpersonal nicht als Lösung ausreichen
- dann
 - führe ein Treffen mit Management und den Projekt-Schlüsselpersonen durch
 - in dem verschiedene Lösungsmöglichkeiten diskutiert werden (Funktionalität verringern, Zeit verlängern),
 - ein neuer Inhalts- und Zeitplan erarbeitet wird
 - und sich abschließend alle auf diesen neuen Plan verpflichten
- Vorteile:
 - Erzeugt angemessene und realistische neue Pläne
 - Erzeugt die Bereitschaft aller, einen neuen Plan wirklich ernst zu nehmen

Ablenkungen sind Wünsche, die von außen neu an das Projekt herantreten und den Arbeitsfluss stören/unterbrechen

- **Opfere eine Person** (*sacrifice one person*):
 - Für eine kleinere Ablenkung, stelle eine Person ab, die sich allein um dieses Problem (und evtl. weitere) kümmert.
Alle anderen arbeiten ungestört weiter
- **Ein Team pro Aufgabe** (*team per task*):
 - Für jede große Ablenkung ("Krise"), stelle mehrere Personen ab, die sich gemeinsam um das Problem kümmern.
Alle übrigen arbeiten ungestört weiter
- **Irgendjemand macht Fortschritt** (*someone always makes progress*):
 - Wenn ständige Ablenkungen den Fortschritt bremsen, Sorge unter allen Umständen dafür, dass zumindest irgendjemand noch auf das ursprüngliche Ziel zuarbeitet

- **Kindertagesstätte** (*day care*):
 - Wenn die erfahrenen Entwickler zu stark durch die Einweisung von Neulingen aufgehalten werden, stelle einen davon ab, sich um alle Neulinge zu kümmern und lasse die übrigen in Ruhe
 - Die Neulinge müssen nur wenig produktive Arbeit abliefern, die aber in hoher Qualität
- **Söldner-Autor** (*mercenary analyst*)
 - Wenn die erfahrenen Entwickler zu stark durch das Schreiben von (Entwurfs-)Dokumentation aufgehalten werden, heuere einen erfahrenen Technischen Autor mit Domänenwissen an, um alle Dokumentation zu verfassen
- **Löse Blockaden** (*interrupts unjam blocking*)
 - Jemand, der zwingend für eine Aufgabe benötigt wird, deren Verzögerung das ganze Projekt zum Stillstand brächte, wird sofort unterbrochen und dort hin gerufen
 - und kann dann keinesfalls unterbrochen werden



- Wenn
 - Anforderungen und Grobentwurf verstanden sind,
 - eine große Menge Code zu schreiben ist und
 - die eigene Konzentrationsfähigkeit begrenzt ist
- dann
 - finde und verwende einen Rhythmus des Entwickelns in sinnvoll abgeschlossenen Episoden:
 - Wähle für die Episode aus, was darin fertiggestellt werden wird,
 - stelle ausreichend Zeit und Konzentration dafür bereit,
 - treffe alle nötigen Entwurfsentscheidungen und setze sie um.
 - Breche eine Episode nicht vor Fertigstellung ab.
- Vorteile:
 - Erzeugt hohe Qualität
 - Erlaubt effizient einen verlässlichen, planbaren Fortschritt

- Diese Beschreibungen sind nur Skizzen
 - Im Original sind viel mehr Details, Überlegungen, Auswirkungen, Verbindungen zu anderen Mustern etc. beschrieben
 - Die Muster sind nicht so naiv, wie sie hier zum Teil vielleicht geklungen haben
- Wie alle Muster, so stammen auch diese direkt aus der Praxis
 - Sie sind echte, erfolgreiche Lösungen aus realen Software-Organisationen
- Sie wirken vielleicht banal, sind es aber nicht
 - Sie sind relevante und erprobte Antworten auf häufige und schwierige Entscheidungsprobleme

- Anforderungen
 - Analysemuster ←
 - Akzeptanzkriterien ←
 - Entwurf
 - Referenzarchitekturen
 - Architekturstile/-muster, Entwurfsmuster
 - Produktfamilien
 - Benutzungsschnittstellen
 - Benutzbarkeitsmuster ←
 - Management, Vorgehen
 - Prozessmuster ←
 - Best practices
 - Standards
 - Allgemein
 - Prinzipien ←
 - Notationen ←
- Jetzt folgen noch:
- Mustersprachen (pattern languages) ←
 - Anti-Muster ←

Anmerkung zu Mustern (fast aller Art):

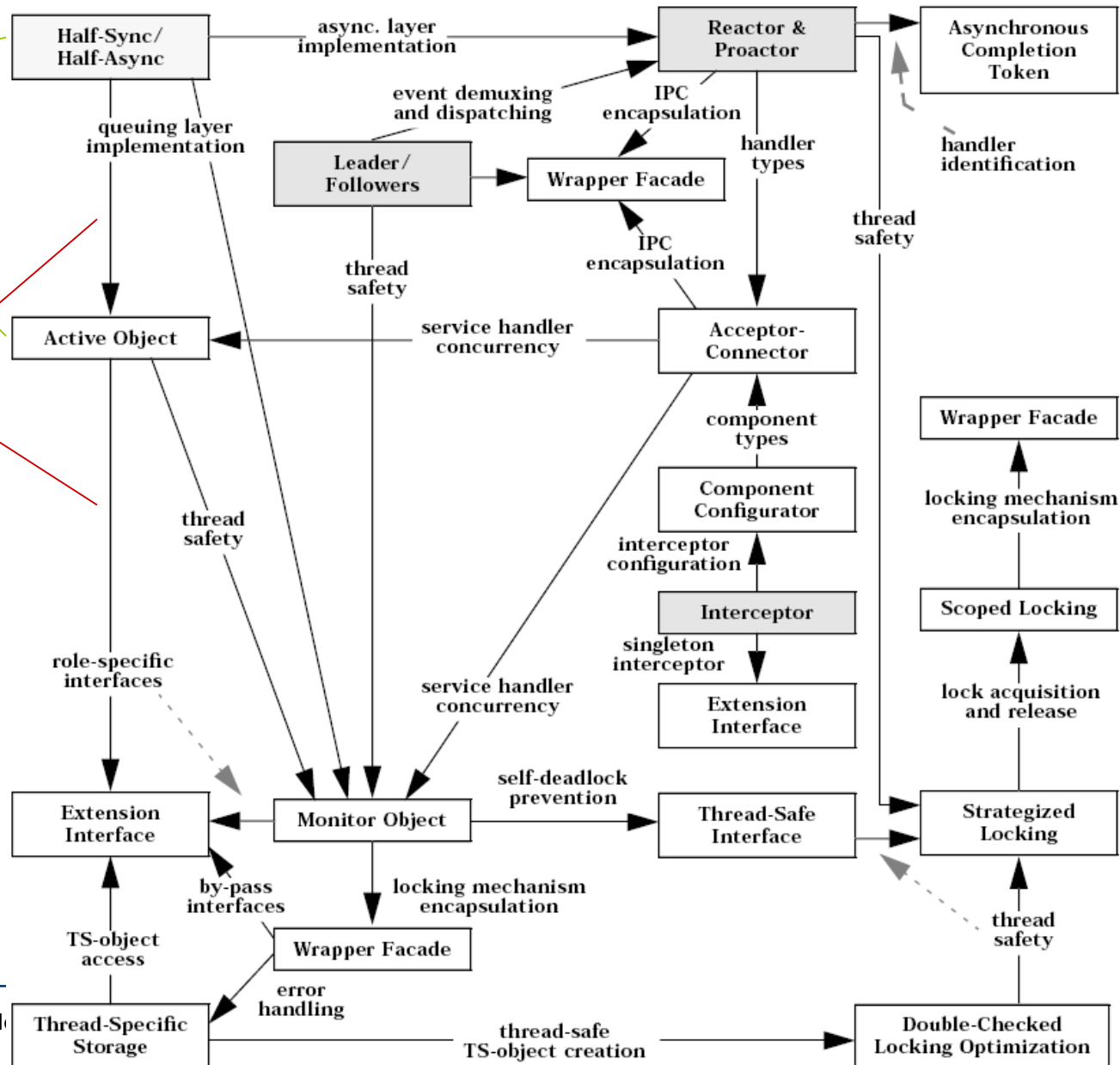
- Wo Muster anwendbar sind, treten oft immer wieder mehrere ähnliche Probleme zusammen auf
 - aber nicht unbedingt immer alle zusammen
- Dementsprechend kann man auch häufig mehrere Muster gemeinsam einsetzen
- Um das zu erleichtern, werden Muster oft nicht einzeln präsentiert, sondern zu einer Gruppe zusammengeschlossen
 - genannt **Mustersprache**
 - Die Beschreibungen der Muster beziehen sich dann aufeinander und beschreiben, wie die Muster zusammenwirken (können)
- Versuchen Sie ggf. stets die Mustersprache als ein Ganzes zu verstehen
 - nicht nur die einzelnen Muster

Beispiel: Eine Mustersprache für verteilte Anwendungen

Muster

unterstützt-von

aus:
D. Schmidt et al.:
"Pattern-orientierte Softwarearchitektur: Muster für nebenläufige und vernetzte Objekte"



Nachbemerkung: Anti-Muster

- Antimuster sind Lösungen die besonders ungünstige Eigenschaften haben,
 - aber ebenso wie Muster in der Praxis wiederholt auftreten
- Sie zu studieren, kann ebenfalls lohnend sein
 - vor allem als Argumentationshilfe gegen drohende inkompetente Lösungen oder für deren Reparatur

Es folgen einige Beispiele (in zufälliger Reihenfolge):

- Prozess: Tod durch Planen (*death by planning*)
Prozess: Analyse-Paralyse (*analysis paralysis*)
Prozess: Kernkompetenz Bildentwurf (*viewgraph engineering*)
 - Perfektion anstreben in der Anforderungsermittlung, in einer reinen Entwurfsphase oder in der Projektplanung und dadurch enorm viel Zeit verschenken
- Überall: Das Rad wiedererfinden (*reinvent the wheel*)
 - Eine ausgereifte, bekannte Lösung nicht benutzen, sondern etwas Schlechteres selbst entwickeln
 - Vor allem bei Architektur, Entwurf, Implementierung; aber auch Anforderungen und Prozess
 - bei uns hieß das: Unnötiges radikales Vorgehen
- Überall: Goldener Hammer (*golden hammer*)
 - Ein bekanntes und bewährtes Konzept, das übermäßig intensiv eingesetzt wird



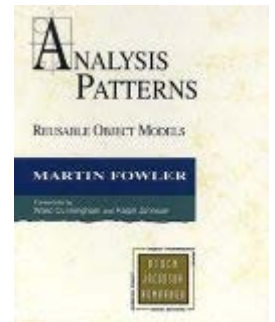
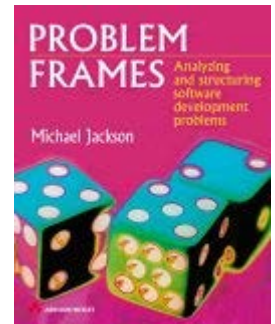
Beispiele für Anti-Muster (2)

- Wiederverwendung: Sackgasse (*dead end*)
 - Man hat eine erhebliche Veränderung an einer Komponente eines Drittherstellers vorgenommen und ist nun von deren Fortentwicklung abgekoppelt
- Wiederverwendung: Minenfeld (*bleeding edge*)
 - Man benutzt zu viel unausgereifte, supermoderne Technologie und die Software ist deshalb chronisch instabil
 - ein Fall von unnötigem radikalem Vorgehen
- Prozess: Dumm halten
 - Die Entwickler werden ausdrücklich gegen jeden Kontakt mit Benutzern abgeschirmt und entwickeln deshalb nie ein brauchbares Verständnis für die Anforderungen und Prioritäten

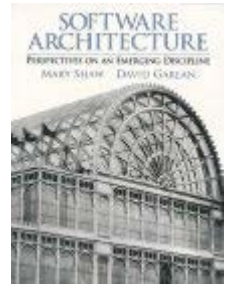
- Prozess: Pseudo-Experten-Herrschaft (*blowhard jamboree*)
 - Entscheidungen (vor allem Technologieauswahl) werden zu sehr auf Basis irgendwelcher Presseberichte getroffen anstatt auf Basis lokal vorhandenen Sachverstands
 - Entwickler müssen unentwegt dem Management Auskunft geben
- Projektmanagement: Todesmarsch (*death march*)
 - Zeitplan und Budget sind von vornherein so knapp, dass ein Erfolg praktisch unmöglich ist
 - Damit eng verwandt ist das Muster *Blendwerk (vaporware)*: Etwas noch nicht existentes als existent verkaufen
- Kommunikation: Email-Glaube
 - Verwendung von Email als Kommunikationsmedium, wo mündliche Verständnisklärung (inhaltlich) oder sensibles Kommunizieren (zwischenmenschlich) nötig wäre

- Architektur: Entwurf per Komitee (*design by committee*)
 - "Zu viele Köche verderben den Brei" und führen zu einer Architektur, die in keiner Hinsicht wirklich gut ist
- Architektur: Conways Gesetz (*Conway's law*)
 - Die Architektur eines großen Systems bildet die bestehende Organisationsstruktur der am Bau beteiligten Gruppen ab
 - <http://www.melconway.com/research/committees.html>
 - (Kann, klug eingesetzt, auch ein positives Muster sein!)
- Entwurf: Krake (*the blob*)
 - Es gibt eine Klasse, die an fast allem irgendwie beteiligt ist, und viel zu viele Kopplungen und Aufgaben hat
 - Überkomplexe Schnittstellen sind auch bekannt als *Eierlegende Wollmilchsau* (unübersetzbar) oder etwas schwächer als *Schweizer Messer* (*swiss army knife*)

- Michael Jackson: *"Problem Frames: Analyzing and Structuring Software Development Problems"*, Addison Wesley 2001
 - Identifiziert einige grundlegende Problemklassen
- Martin Fowler: *"Analysis Patterns: Reusable Object Models"*, Addison Wesley Longman 1997
 - wie vorhin beispielhaft gesehen
 - Achtung: Das Buch benutzt eine Vor-UML-Notation, bei der Rollennamen immer genau auf der anderen Seite der Assoziation stehen als bei UML üblich
 - Aktualisierungen und Ergänzungen siehe <http://www.martinfowler.com/articles.html#ap>
 - Die beschriebenen Beispiele entstammen <http://www.martinfowler.com/apsupp/accountability.pdf>



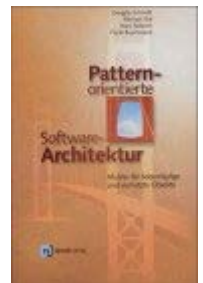
- Mary Shaw, David Garlan: "Software Architecture: Perspectives on an Emerging Discipline", Prentice Hall 1996
 - Erstes Buch zum Thema Architekturstile und –muster
 - Nur wenige Muster, aber konzeptuell sehr gute Einführung
- Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal: "*Pattern-orientierte Softwarearchitektur: Ein Pattern-System*", Addison-Wesley 1998
 - "Pattern-oriented software architecture: a system of patterns"
 - Architektur-, Entwurfs- und Kodiermuster



Shaw



- Martin Fowler: *"Patterns für Enterprise Application-Architekturen"*, mitp 2003
 - *"Patterns for Enterprise Application Architectures"*, Addison Wesley 2000
 - schöne breite Sammlung für Informationssysteme
- Erich Gamma, Richard Helms, Ralph Johnson, John Vlissides: *"Entwurfsmuster"*, 1995
 - der Klassiker: das "Gang-of-Four"-Buch (GoF)
 - Allgemeine Muster und gute Einführung in OO-Entwurf
- Douglas Schmidt, Michael Stal, Hans Rohnert, Frank Buschmann: *"Pattern-orientierte Softwarearchitektur: Muster für nebenläufige und vernetzte Objekte"*, dpunkt 2002
 - siehe oben die Mustersprache



- <http://www.cmis.brighton.ac.uk/research/patterns/home.html>
 - Grundlegende Regeln für die Interaktionsgestaltung
- Len Bass, Bonnie E. John, Jesse Kates:
"Achieving Usability Through Software Architecture",
CMU/SEI-2001-TR-005
 - Quelle für vorhin beschriebene Muster
 - <http://www.sei.cmu.edu/publications/documents/01.reports/01tr005.html>



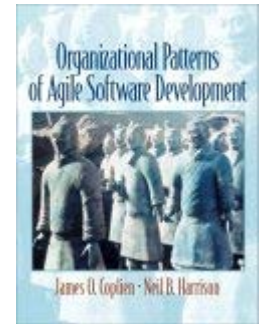
Bass



John



- James Coplien, Neil Harrison: "*Organizational Patterns of Agile Software Development*",



Pearson Prentice Hall 2005

- wie oben beispielhaft gesehen
- Sehr gutes Buch für Projektmgmt. und Prozessmgmt.
 - (und gar nicht nur speziell für agile Prozesse)

- Inhalt:

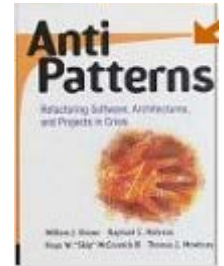
- Project Management Pattern Language 26 Muster
- Piecemeal Growth Pattern Language 32 Muster
- Organizational Style Pattern Language 23 Muster
- People and Code Pattern Language 23 Muster
- Organizational Principles 18 Erwägungen
- Antropological Foundations
- Case Studies 2 Fallstudien
- Summary Patlets Kurzbeschreibungen



Coplien

Quellen: Anti-Muster

- William Brown, Raphael Malveau, Hays McCormick: "Anti-Patterns: Refactoring Software, Architectures and Projects in Crisis", John Wiley 1998
 - Die meisten der oben zitierten Antimuster sind in diesem Buch genauer beschrieben
- <http://en.wikipedia.org/wiki/Antipattern>
 - enthält auch eine Liste von Mustern
- <http://c2.com/cgi/wiki?AntiPattern>
 - <http://c2.com/cgi/wiki?AntiPatternsCatalog>



Danke!