



AntiPatterns

Gliederung:

1. Einführung

- 1.1 Was ist ein AntiPattern?
- 1.2 Bedeutung von AntiPatterns
- 1.3 Unterschied zum Entwurfsmuster
- 1.4 Entstehung von AntiPatterns
- 1.5 Die Perspektiven des AntiPatterns

2. Das Referenzmodell

- 2.1 Hauptursachen
- 2.2 Urkräfte
- 2.3 Software- Design- Level- Modell (SDLM)

3. Vorlagen für Entwurfsmuster und AntiPatterns

3.1 Warum Vorlagen?

3.2 Vorlagen für AntiPatterns

3.3 Vollständige AntiPattern- Vorlage

4. AntiPatterns

4.1 The Blob

4.2 Poltergeist

4.3 Goldener Hammer

1.1 Was ist ein AntiPattern?

Realität der Softwarebranche:

- 5 von 6 Softwareprojekte scheitern (84%), bzw. erfüllen nicht die gesetzten Erwartungen
- in Quelltext geht man von mehreren Fehlern pro Zeile aus!
- Software Entwicklung befindet sich noch in der Steinzeit!
- Es gibt noch keine gewährleistenden Mittel, Fehler in der Software-Entwicklung zu vermeiden

AntiPattern (zu deutsch: Antimuster) bezeichnet in der Softwareentwicklung einen häufig anzutreffenden **schlechten Lösungsansatz für ein bestimmtes Problem. Es bildet damit das Gegenstück zu den Mustern (Entwurfsmuster, Analysemuster, Architekturmuster...)**

AntiPatterns- Was ist ein AntiPattern Fortsetzung

Anti Pattern

- beschreiben Fehler die sehr häufig unterlaufen
- bieten Lösungen an
- verdeutlichen negative Muster

2 wichtige Zwecke:

- Sie helfen beim Erkennen von Problemen
- Sie helfen bei der Implementierung von Lösungen

Entstehung:

- mangelnde Kenntnisse eines Projektleiters oder Entwicklers
- mangelnde Erfahrung zur Problemlösung
- Anwendung eines an sich perfekten Musters im falschen Kontext

1.2 Die Bedeutung von AntiPatterns

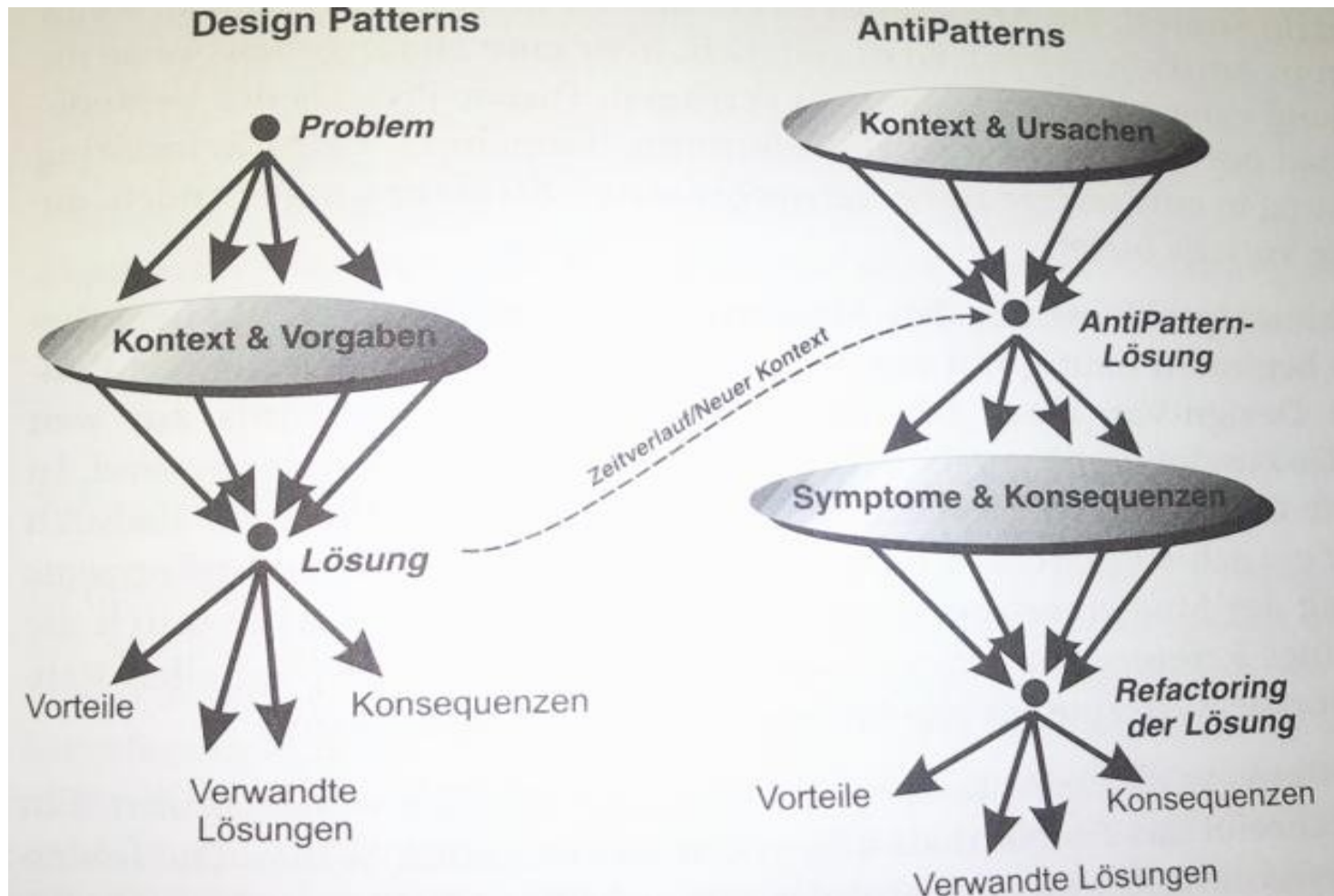
- fast alle entwickelten Systeme sind Stovepipe-Systeme, die nicht verändert werden können
- Anpassungsfähigkeit wichtige Qualität von Software
- Über die Hälfte der Software-Entwicklungskosten wird für notwendige Änderungen oder Systemerweiterungen ausgegeben
- Zirka 30% der Entwicklungskosten werden durch Veränderungen während der Systementwicklung verursacht

Stovepipe-System: *In sich abgeschlossenes System, alle Subsysteme sind voneinander in hohem Maß abhängig und nahezu untrennbar miteinander verkoppelt; Kaum erweiterbar, sehr komplex*

1.3 Unterschied zum Entwurfsmuster

- Ein Entwurfsmuster wird zum AntiPattern wenn es mehr Probleme verursacht als es löst
- AntiPattern(=AP) ist ein Muster im falschen Kontext
- Entwurfsmuster beginnen mit einer wiederholt angewendeten Lösung
- AP beginnen mit einem immer wieder auftretenden Problem;
Zweite Lösung = Refactoring

Refactoring: *Art der Codemodifikation zur Verbesserung der SW-Struktur für spätere Erweiterungen und die langfristige Programmpflege - meist ist es das Ziel, Code umzuwandeln ohne dessen Korrektheit zu beeinträchtigen*



Entwurfsmuster: Kontext, Problem und Vorgaben eines Entwurfsmusters werden formuliert um zu einer eindeutigen Lösung zu führen

AntiPa.: beginnen mit einer zwingenden, problematischen Lösung und bieten anschließend eine alternative Lösung mit einem Refactoring des Problems -> Lösung muss nicht eindeutig sein

AntiPatterns sind effektiver weil:

- Probleme verdeutlicht werden, indem Symptome und Konsequenzen direkt aufgezeigt werden
- ein Motiv für Veränderung und die Notwendigkeit des Refactoring vermitteln
- für das Verständnis allgemeiner Probleme notwendig sind, mit denen die meisten Software- Entwickler konfrontiert werden.

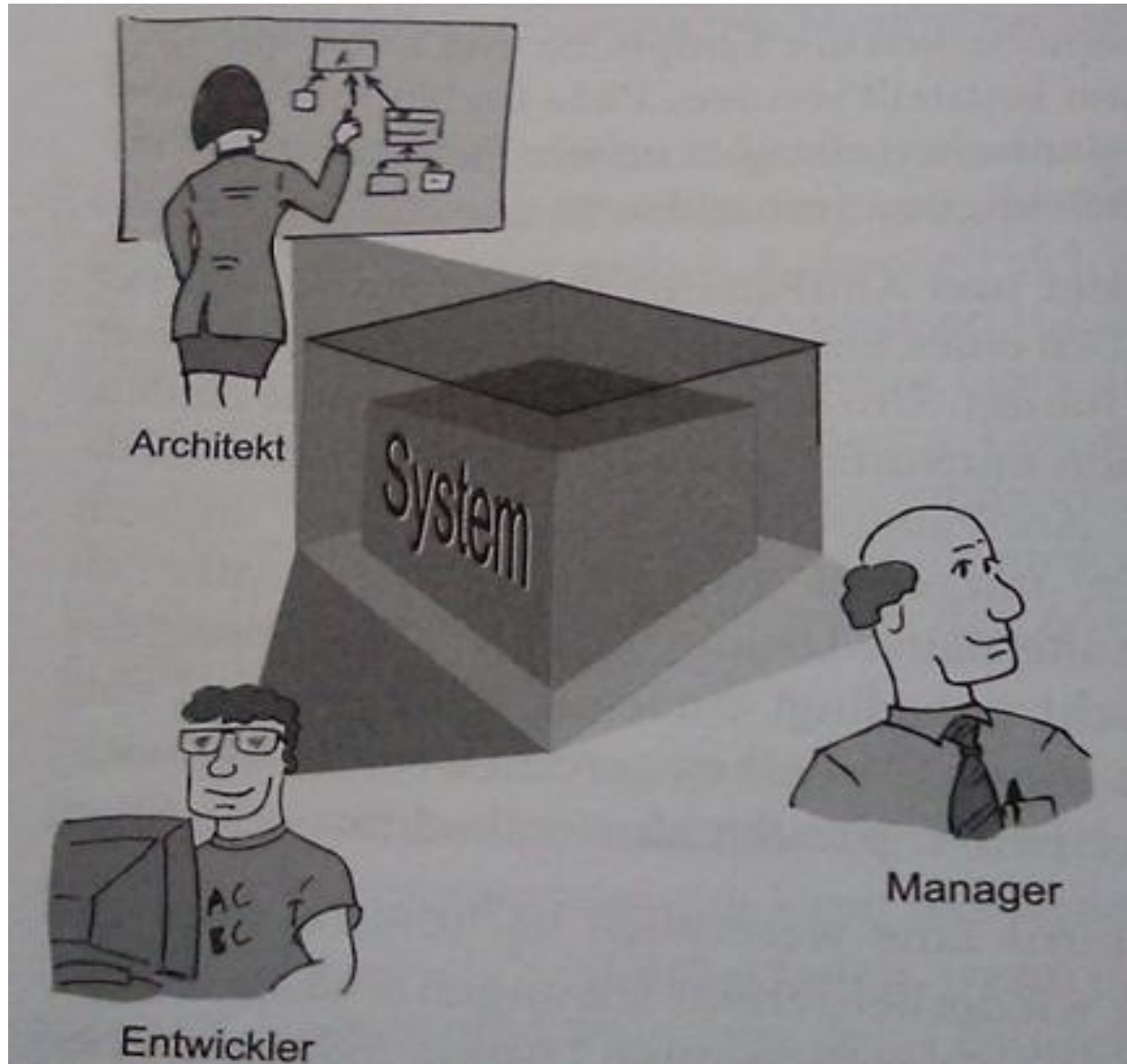
1.4 Die Entstehung von AntiPatterns

- Idee von Architekten Christopher Alexander (70iger Jahre)
- Sprache für die Stadtplanung und den Bau von Gebäuden innerhalb der Städte
- Sprache formulierte klar die Vision, wie Architektur modelliert werden sollte, warum bestimmte Städte und Gebäude eine bessere Umwelt schaffen als andere
- 1987 mehrere führenden SW-Entwickler greifen Arbeit von Alexander auf und wenden sie an
- 1994 hat Entwurfsmuster die Hauptströmung der OOSE erobert.
- Hillside Group historische Konferenz zum Thema Software- Entwurfsmuster und dem Titel "Pattern Languages of Program Design(PLoP)" veranstaltet.

- Immer mehr Literatur, Veröffentlichungen zum Thema Entwurfsmuster erschien
- 1996 Präsentation von Michael Akroyd bei der Konferenz Object World West unter dem Titel "**AntiPattern**"

1.5 Die Perspektiven der AP

- 3 Perspektiven
- **Entwicklern:** technische Probleme und Lösungen für Programmierer
- **SW-Architekten:** benennen und lösen allgemeine Probleme der Systemstruktur
- **Projektleiters:** allgemeine Probleme bei Software-Prozessen und der Software- Entwicklung --> betrifft Personen aller SWE Bereiche



2.1 Hauptursachen

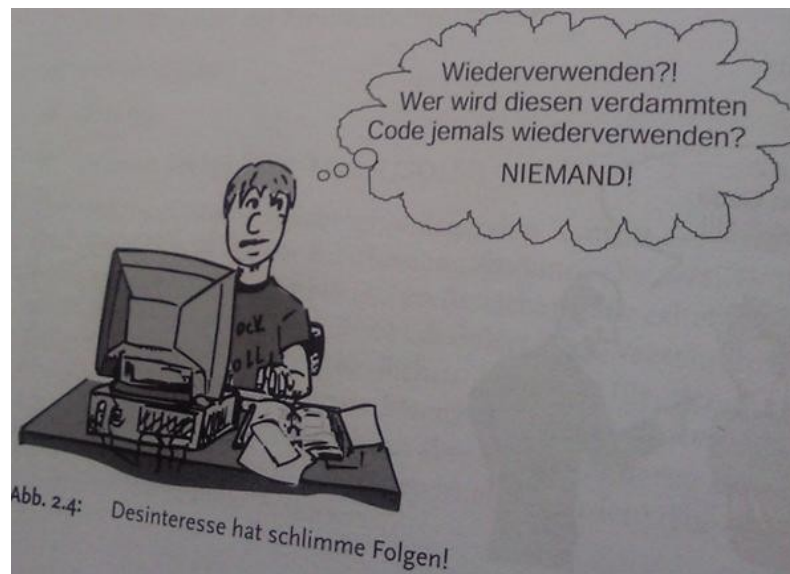
Schlüsselkonzepte des AntiPattern:

Hauptursachen, Urkräfte, Software-Design-Level-Modell

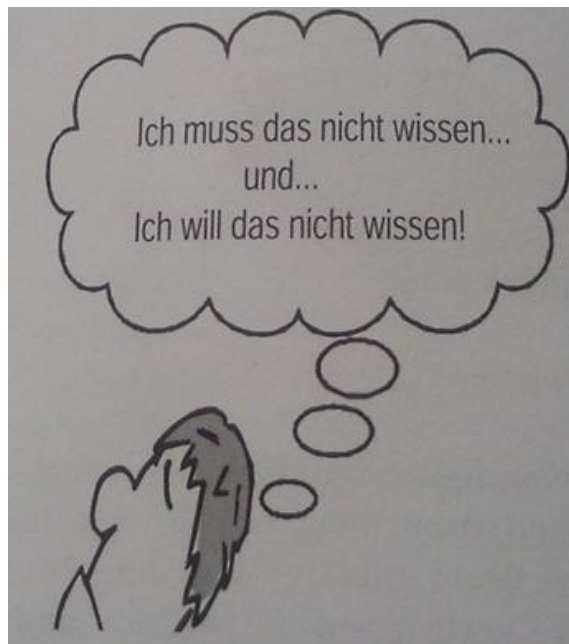
Hauptursachen:

- **Hast:** Kompromissen hinsichtlich der Qualität der SW
- knappe **Zeitplanung**, unrealistische Zeitvorgaben
- **Budgeteinsparungen**
- Verzicht auf ausführliche **Tests**
- **mangelnde Erfahrungen** in der Entwicklung
- **Desinteresse:** Unwilligkeit Lösungen zu suchen
- **Entstirnigkeit**
- **Faulheit**
- **Geiz:** Übertriebene Komplexität
- **Ignoranz:** intellektuelle Faulheit, Implementierungsabhängigkeit
- **Stolz:** unnötige neue Elemente

Implementierungsabhängigkeit =



Beispiele für Desinteresse, Faulheit und Stolz



2.2 Urkräfte:

- **Funktionalitätsmanagement:** Den Anforderungen entsprechen, Funktionalität
- **Leistungsmanagement:** erforderliche Operationsgeschwindigkeit erreichen
- **Komplexitätsmanagement:** Einfachheit, einfache Schnittstellen, einheitliche Architektur, bessere Objektmodelle
- **Management der Änderungen:** Steuerung der weitere Entwicklung der SW
- **Management der IT-Ressourcen:** Überwachung und Einsatz von Mitarbeitern und Ausrüstung, Hardware - Software, Inventar, Schulungen, Wartung, Upgrade, Sicherheit und technischer Support
- **Management des Technologietransfers:** Überwachung technologischer Änderungen, Transfer von Software und Technologien

Kräfte/Bereiche	Global	Unternehmen	System	Anwendung
Funktionalität	unwichtig	marginal	wichtig	kritisch
Leistung	wichtig	wichtig	kritisch	kritisch
Komplexität	wichtig	kritisch	wichtig	marginal
Änderungen	unwichtig	kritisch	kritisch	wichtig
IT-Ressourcen	unwichtig	kritisch	wichtig	marginal
Technologietransfer	kritisch	wichtig	wichtig	marginal

Tabelle zeigt Auswirkung der Kräfte auf unterschiedliche Bereiche

Unwichtig = Die Auswirkungen müssen nicht beachtet werden

Marginal = Konsequenzen können häufig ignoriert werden, da sie nur einen minimalen Teil der SW betreffen

Wichtig = Auswirkungen dürfen nicht vernachlässigt werden, da sie einen bedeutenden Teil der SW betreffen

Kritisch = Auswirkungen sind fundamental und betreffen gesamte SW

2.3 SDLM

- Modell der Software- Designebenen
- Software- Architektur wichtig
- Trennung der Aufgabenbereiche
- Problem in lösbar Teilprobleme zerlegen

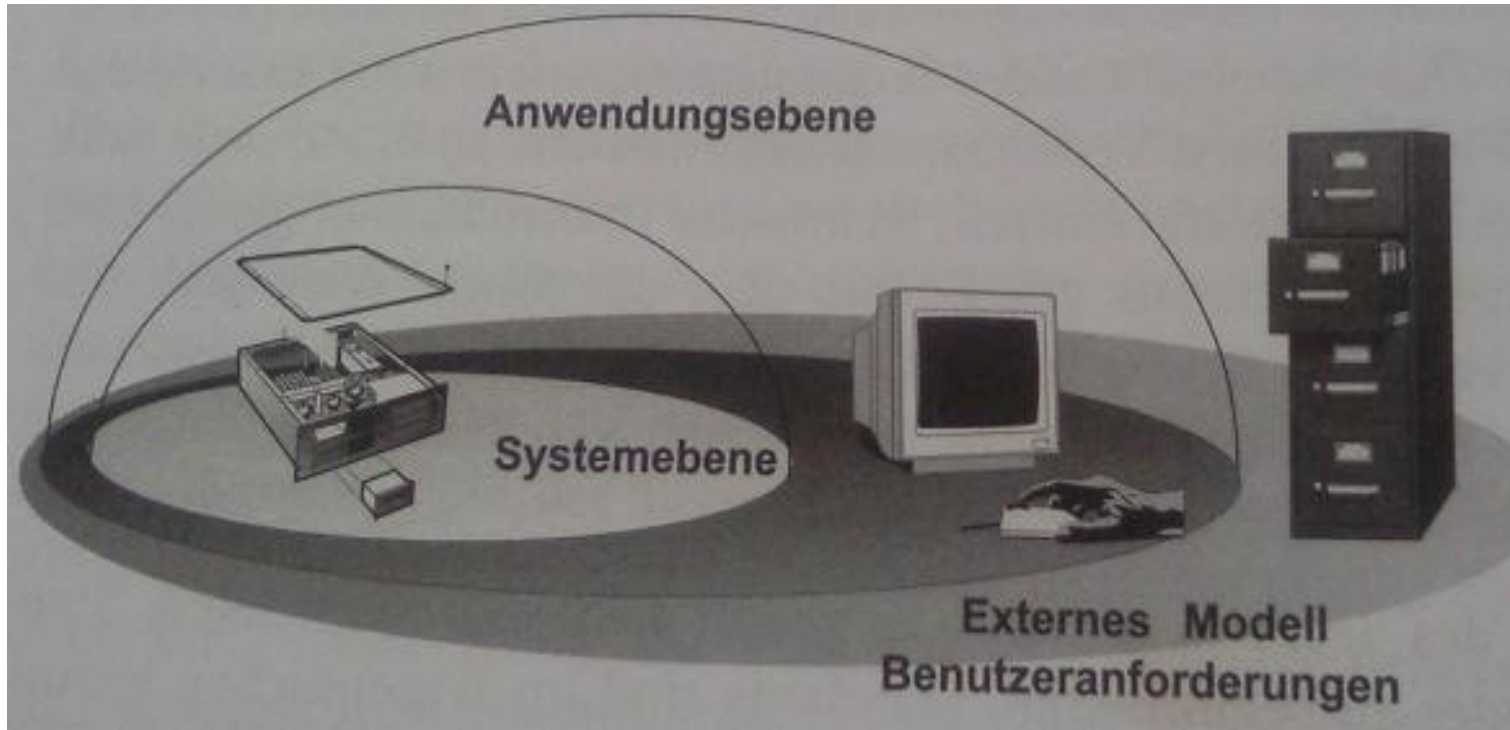
2 Ebenen in SW- System:

* Externes Modell (Anwendungsebene):

- widmet sich Anforderungen des Anwenders, Benutzerschnittstellen, GUI
- Benutzer kommuniziert direkt mit der Anwendung

* Internes Modell (Systemebene):

- Verbindungen zwischen den Anwendungen
- Keine unmittelbare Schnittstelle zum Benutzer
- Systemebene umfasst die Architektur des SW-Systems, Kommunikation, Koordination Zusammenarbeit zwischen Anwendungen
- Zugriff auf Datenspeicher, Verwaltung von Prozessen

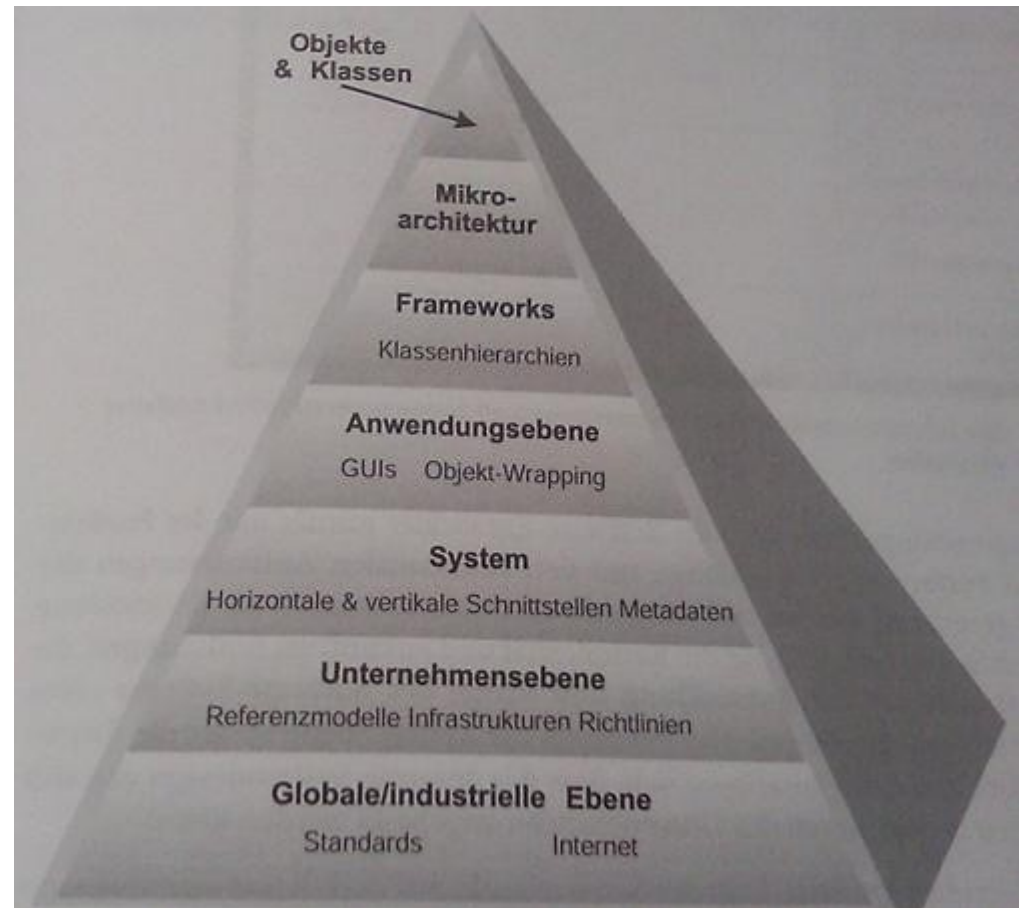


Externes und internes Modell eines Software- Systems



Abb. 2.11: Das Modell der Software-Designebenen

Das Modell der SW- Designebenen



Globale Ebene:

- Grenzen sind kaum festzulegen
- breiteste architektonische Schicht, mehrere Unternehmen angesiedelt
- umfasst global anwendbare Designfragen für alle Systeme
- Kommunikation, Informationsaustausch von Unternehmen
- Betrifft Sprachen, Standards und Richtlinien von mehreren Unternehmen
- Ziel ist die Verwirklichung gemeinsamer Ziele zwischen Unternehmen
- Bsp. Internet = Globales System, SW- Standards z.B. ISO, ANSII, TCP/IP

- **Unternehmensebene:**

- Koordination und Kommunikation innerhalb eines Unternehmens
- umfangreichste architektonische Schicht innerhalb eines Unternehmens
- Unternehmen hat Kontrolle über seine Ressourcen und Strategien

- **Systemebene:**

- Kommunikation, Koordination zwischen Anwendungen und Gruppen von Anwendungen
- drei Schlüsselemente dieser Ebene:
 - horizontalen Schnittstellen: für Wiederverwendung im Unternehmen
 - vertikalen Schnittstellen: fachspezifische Schnittstellen für bestimmte Anwendungen
 - Metadaten: Daten, Dienste, Infos im System

- **Anwendungsebene:**

- Einrichtung von Anwendungen für unterschiedliche Benutzeranforderungen
- kann aus ein oder mehreren Frameworks bestehen
- Funktionalität der Anwendungen im Mittelpunkt

- **Framework- Ebene:**

- Wiederverwendung von Code und Designs
- Frameworks versuchen große Teile des Designs und der Software zu übernehmen innerhalb eines bestimmten Fachgebiets

- **Microarchitektur:**

- Entwurfsmuster, welche mehrere Objekte oder Objektklassen kombinieren,
- Aufgabe: Wiederverwendung der Einkapselung von Komponenten

- **Objektebene:** feinkörnigste Ebene, Software und Dokumentation, wiederverwendbare Objekte und Klassen

- **Makrokomponentenebenen:** Einrichtung und Entwicklung eines Anwendungs-Frameworks

- **Mikrokomponentenebene:** Entwicklung von SW-Komponenten zur Lösung immer wieder auftretender SW- Probleme.

Framework = „Rahmenstruktur, Fachwerk“, Programmiergerüst

3.1 Warum Vorlagen

- Für eine Problemstellung das passende Entwurfsmuster finden
- Ohne eine Vorlage ist ein Entwurfsmuster nur unstrukturierter Vorschlag nach dem Motto: "Dies ist ein Entwurfsmuster, weil ich es sage"
- wir brauchen eine Begründung, eine Struktur für ein Entwurfsmuster. Ohne Struktur ist es schwer ein Entwurfsmuster ausfindig zu machen.

Vorlagen für Entwurfsmuster

- Es gibt viele formale Vorlagen, um ein Entwurfsmuster zu spezifizieren, z.B. degeneriertes Entwurfsmuster, Entwurfsmuster nach Alexander, GoF-Entwurfsmuster, ...
- AntiPattern- Vorlagen funktionieren genau nach demselben Prinzip, man versucht mit Vorlagen Muster zu spezifizieren

3.2 Vorlagen für AntiPatterns

Pseudo - AntiPattern-Vorlage:

Art degenerierte Vorlage - Name und Problem

Bsp.: **Name:** Wie heißt das AntiPattern?

Problem: Welches sind die nachteiligen Eigenschaften?

Mini Anti Patterns:

Name + Erste Lösung AntiPattern- Problem + Zweite Lösung Refactoring

Bsp.: **Name:** Wie soll dieses AntiPattern für die Praxis bezeichnet werden?

AntiPattern- Problem: Welches ist die immer wiederkehrende Lösung mit den negativen Konsequenzen?

Refactoring: Wie lässt sich das Problem des AntiPatterns vermeiden, minimieren oder die Lösung umgestalten?

3.3 Vollständige AP Vorlage

- **AntiPattern Name:** eindeutiger Name um es identifizieren zu können
- **Auch bekannt als:** zusätzliche Bezeichnung für AP
- **Häufigste Ebene:** gibt an wo AP im SDLM angesiedelt ist
- **Name der Lösung nach dem Refactoring**
- **Lösungstyp des Refactoring:** Typ der Aktion, der sich aus Lösung des AP ergibt
 - +Software: verlangen das Erstellen neuer SW
 - +Technologie: Anschaffung neuer Technologie oder Produkte
 - +Prozess: Lösung setzt Prozess voraus
 - +Rolle: Zuweisung von Verantwortlichkeiten an Person oder Gruppe

AntiPatterns- Vollständige AntiPattern- Vorlage

- **Hauptursachen:**

allgemeinen Gründe für dieses AntiPattern

(z.B. Hast, Desinteresse, Engstirnigkeit, Faulheit, Geiz, Ignoranz, Stolz oder Verantwortlichkeit)

- **Nicht ausgeglichene Kräfte (Urkräfte):**

ignorierten, missbrauchten oder überstrapazierten Urkräfte dieses AP

(Funktionalitätsmanagement, Leistungsmanagement, Komplexitätsmanagement, Änderungen, usw.)

AntiPatterns- Vollständige AntiPattern- Vorlage

- **Anekdotisches:** optionaler Abschnitt für unterhaltsames Material zum AP
- **Hintergrund:** optional

- **Allgemeine Form dieses AP:** enthält oft ein Diagramm, das die allgemeinen Symptome und Konsequenzen verdeutlicht

- **Symptome und Konsequenzen:** wie oben ohne Diagramm
- **Typische Ursachen**
- **Bekannte Ausnahmen**

AntiPatterns- Vollständige AntiPattern- Vorlage

- **Refactoring:** die Probleme aus dem Abschnitt "Allgemeine Form" werden aufgehoben, Kräfte werden aufgehoben
- **Varianten:** optional, alternative Lösungen beschrieben, strukturiert
- **Beispiel:** Lösung für das Problem veranschaulichen, Problemdiagramm, usw.
- **Verwandte Lösungen:** Literaturverweise, Querverweise, nahe-verwandte AP werden aufgeführt, Erklärung der Unterschiede
- **Anwendbarkeit auf andere Gesichtspunkte und Ebenen:** Darstellung aus anderem Blickwinkel- aus dem des Managers, Architekten oder Entwicklers

4.1 The Blob

Name = The Blob

Auch bekannt als = Winnebago, The God Class

Häufigste Ebene = Anwendung

Name des Refactoring = Refactoring der Verantwortlichkeiten

Typ des Refactoring = Software

Hauptursachen = Faulheit, Hast

Nicht ausbalancierte Kräfte: Funktionalitätsmanagement, Performance, Komplexität

Anekdotisches = „Diese Klasse ist das Herzstück unserer Architektur“

Hintergrund = Schwarz-Weiß-Film "Blob- Schrecken ohne Namen?"



Allgemeine Form =

- prozedurales Design
- Eine Klasse mit Hauptanteil an Verantwortlichkeit
- andere Klassen enthalten nur Daten oder einfache Prozesse
- Lösung: einheitlichere Verteilung der Verantwortlichkeiten
- durch ein Klassendiagramm gekennzeichnet
- komplexe CONTROLLER_CLASS von einfachen Datenklassen umgeben

Entstehung =

- falsche Zuweisung der Anforderungen, Verantwortung
- Ergebnis schrittweisen Entwicklung (Programmierer fängt mit einer Klasse an, verteilt Verantwortlichkeiten später nicht mehr)

Symptome und Konsequenzen =

- Eine einzige Klasse mit vielen Attributen, Operationen oder beidem
- Sammlung bezugsloser Attribute und Operationen in einer Klasse eingekapselt
- Controller Klasse
- bei Systemveränderung werden eingekapselte Objekte beeinträchtigt
- zu komplex für Wiederverwendung
- nimmt sehr viel Platz in Anspruch

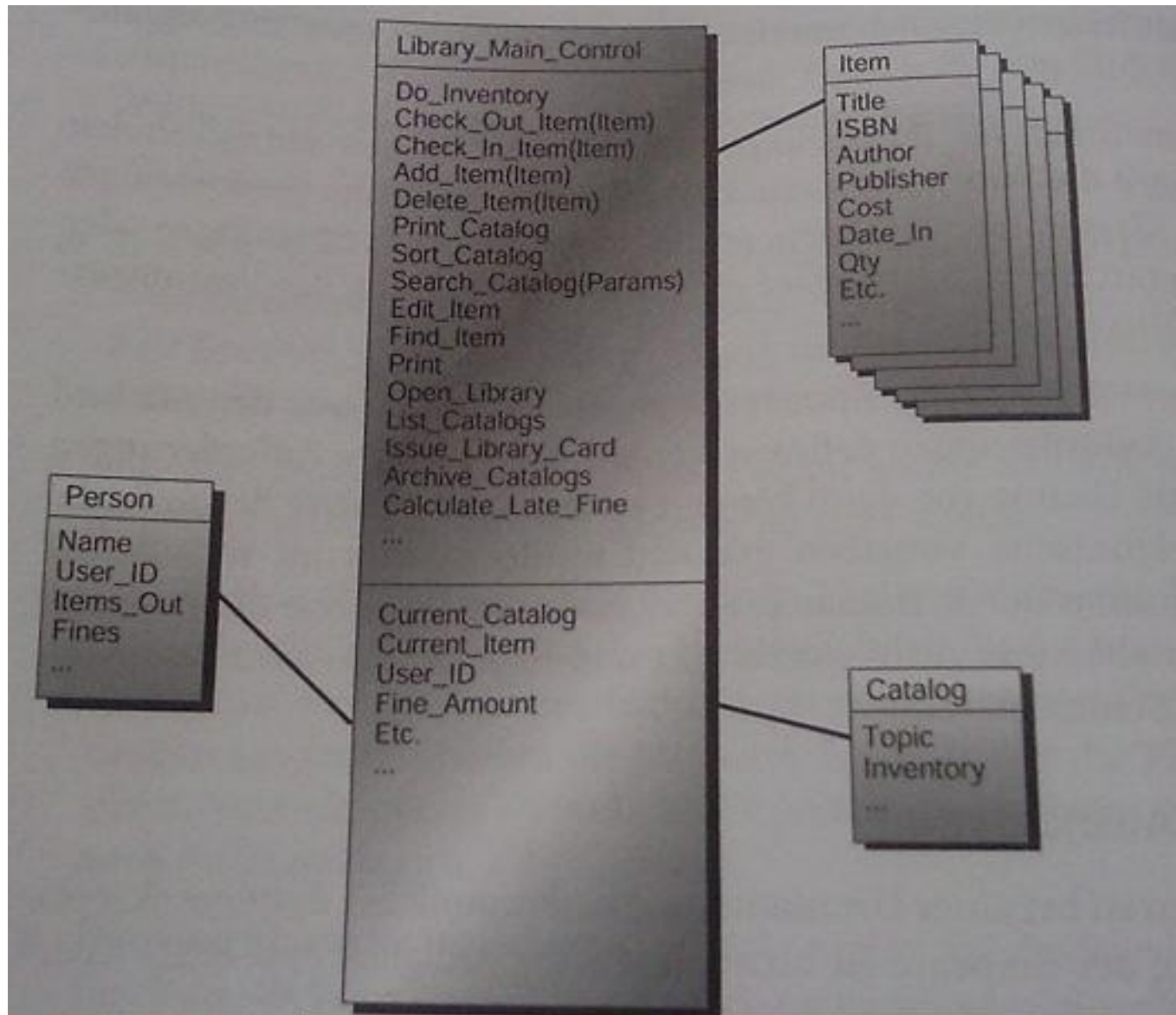
AntiPatterns- The Blob Fortsetzung

Ursachen =

- keine Objektorientierte- Architektur
- überhaupt keine Architektur: Ad-hoc Entwicklung der Programme
- mangelnde Umsetzung der Architektur: Gute Struktur schlecht umgesetzt
- zu geringe Bereitschaft für Erweiterungen, keine neuen Klassen

Refactoring = Lösung ist das Verschieben von Operationen aus dem Blob, Komplexität reduzieren

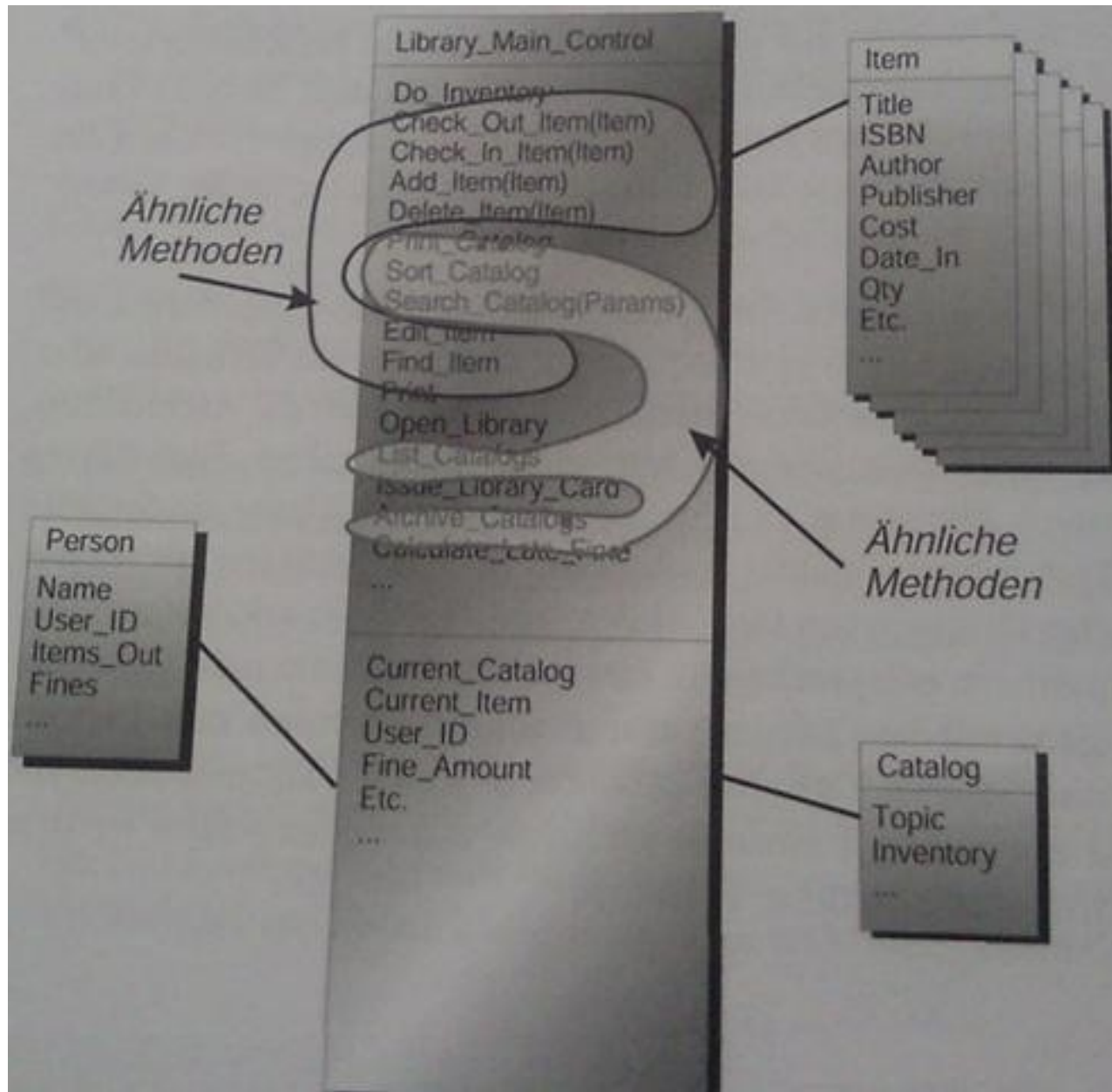
Ad-hoc: spontane improvisierte Handlung, „für diesen Augenblick gemacht“



Der Library- Blob

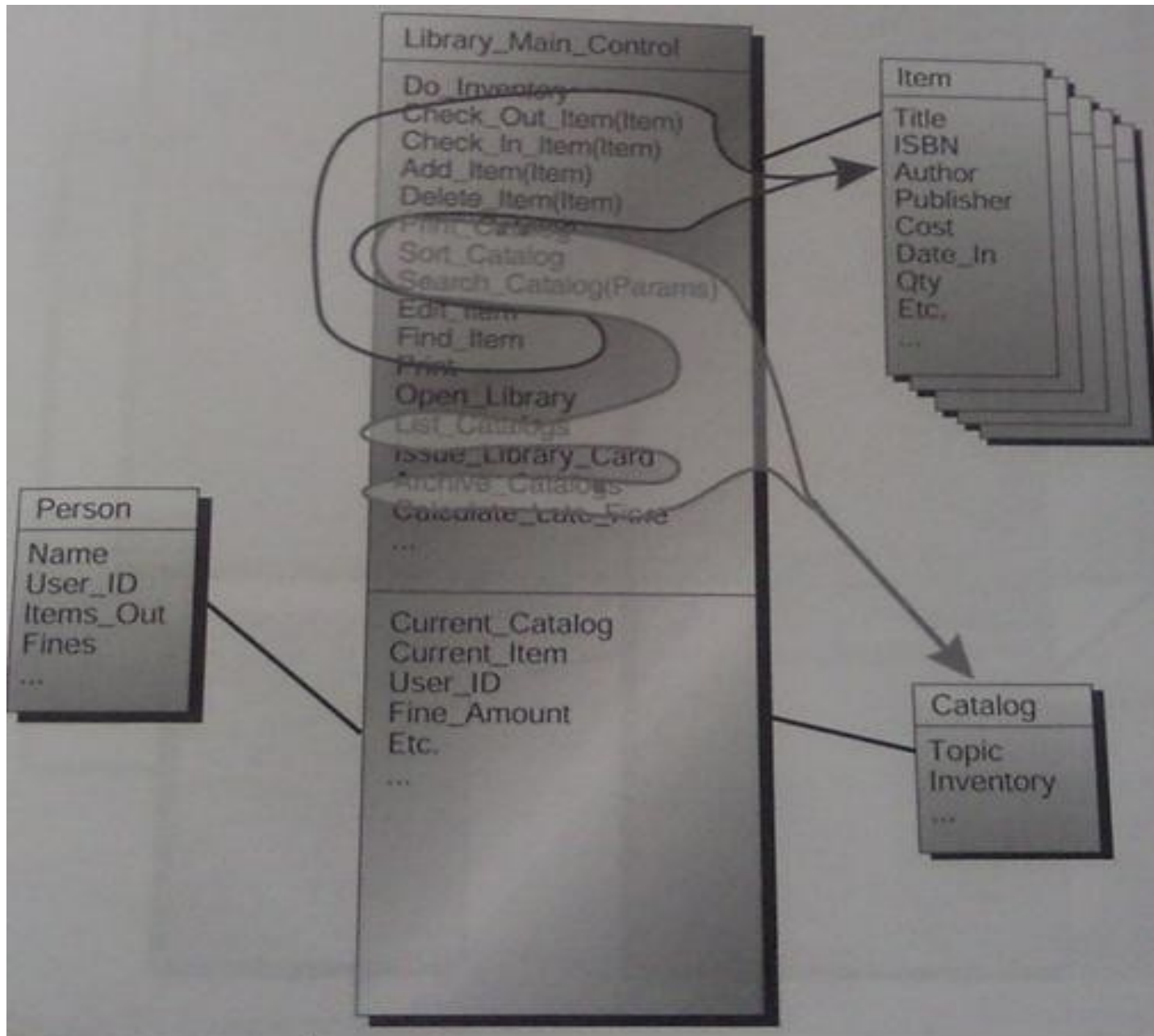
Neugruppierung des Blobs nach Vertragspartnern

1. Kategorisieren ähnlicher Attribute und Operationen nach bestimmten Muster(Vertragspartner),
Neugruppierung nach Vertragspartnern



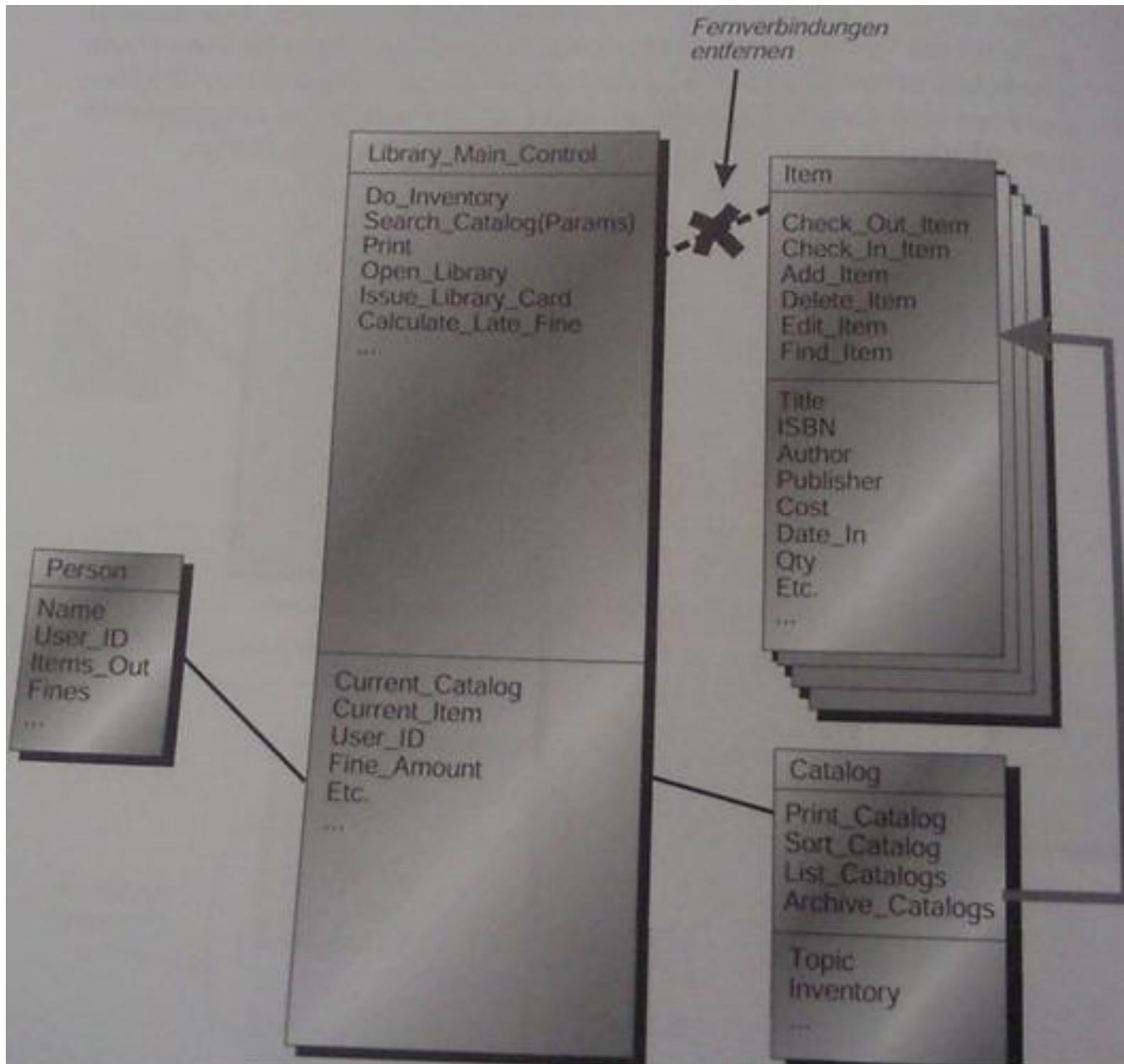
Verschieben der funktionalen Gruppen

2. Es wird nach "Eigenheimen" für die Vertragspartner gesucht und umgesiedelt -> Methoden, Attribute CATALOG kommen in Klasse Catalog -> Resultat: Library Klasse vereinfacht und ITEM CATALOG werden zu mehr als nur einfache eingekapselte Datentabellen; Verbessertes OO - Design



Fernverbindungen entfernen

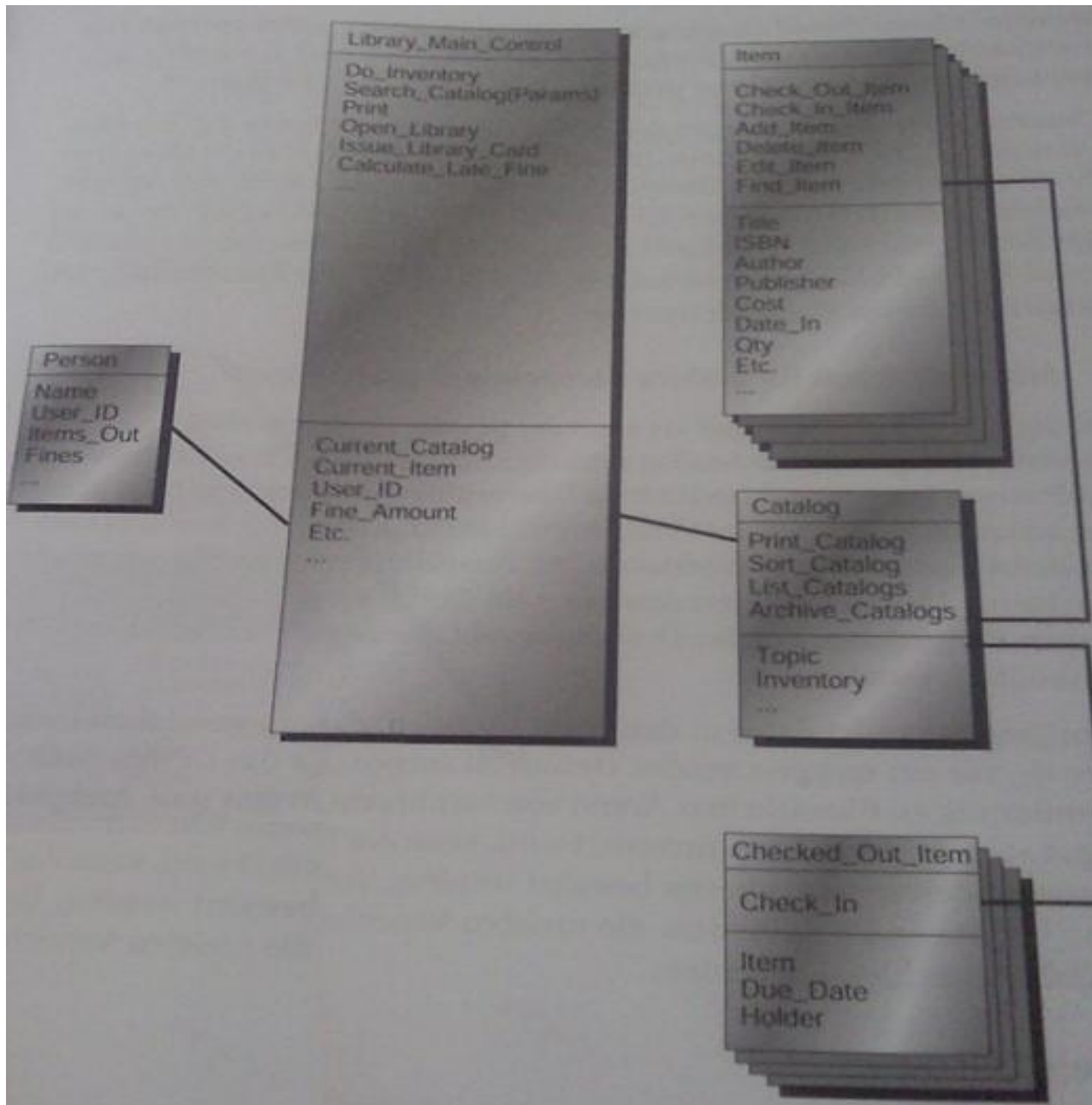
3. Redundante, Indirekte Bezüge entfernen, "Fernverbindungen" löschen, CHECK_OUT_ITEM aus Library Class entfernen!



Transiente Verbindungen entfernen

4. Neue Verbindungen erstellen Catalog-> Item - allgemeine Basisklasse

5. Transiente Verbindungen entfernen - transiente Prozesse können in eine separate transiente Klasse mit lokalen Attributen verschoben werden



Varianten:

- weitere 80%ige Lösung: Klasse von einer Controllerklasse auf eine Koordinator Klasse herabstufen
- Blob Klasse regelt Systemfunktionalität, Datenklassen werden mit eigenen Operationen erweitert

transient: „flüchtige“ Daten, „vorübergehend“



4.2 Poltergeister

Name = Poltergeister

Auch bekannt als= Gypsy, Proliferaton of Classes

Name des Refactoring= Ghostbusting

Typ des Refactoring = Prozess

Hauptursachen= Ignoranz

Nicht ausbalancierte Kräfte = Funktionalitätsmanagement, Komplexität

Anekdotisches = „Ich bin mir nicht ganz sicher was diese Klasse macht, aber es ist sicher wichtig!“

Hintergrund = Michael Akroyd stellt AP 1996 auf der Objekt World West vor

Häufigste Ebene = Anwendung

AntiPatterns- Poltergeister

Allgemeine Form =

- Poltergeister sind Klassen mit sehr beschränkten Aufgaben, Verantwortlichkeiten und effektiven Lebenszyklen
- Sie starten oft Prozesse für andere Objekte.
- Zum Refactoring gehört eine Neuzuweisung der Verantwortlichkeiten an langlebigere Objekte, die die Poltergeister ausschalten
- stören das Software Design wegen unnötiger Abstraktionen
- komplex und schwer zu verstehen und zu pflegen
- tauchen plötzlich auf um etwas in Klassen auszuführen

Symptome und Konsequenzen =

- sie sind überflüssig, beanspruchen unnötige Ressourcen
- Sie sind uneffektiv, weil sie zahlreiche redundante Navigationspfade verwenden
- stehen Oo-Design im Wege, bringen es durcheinander
- flüchtige Zuordnungen, Beziehungen
- Zustandslose Klassen
- temporäre Objekte und Klassen mit kurzer Lebensdauer
- Klassen mit nur einer Operation, die nur Vorhanden sind um andere Klassen über temporäre Verbindungen aufzurufen
- Klassen mit Bezeichnungen die control enthalten oder start_process_alpha

AntiPatterns- Poltergeister Fortsetzung

Ursachen =

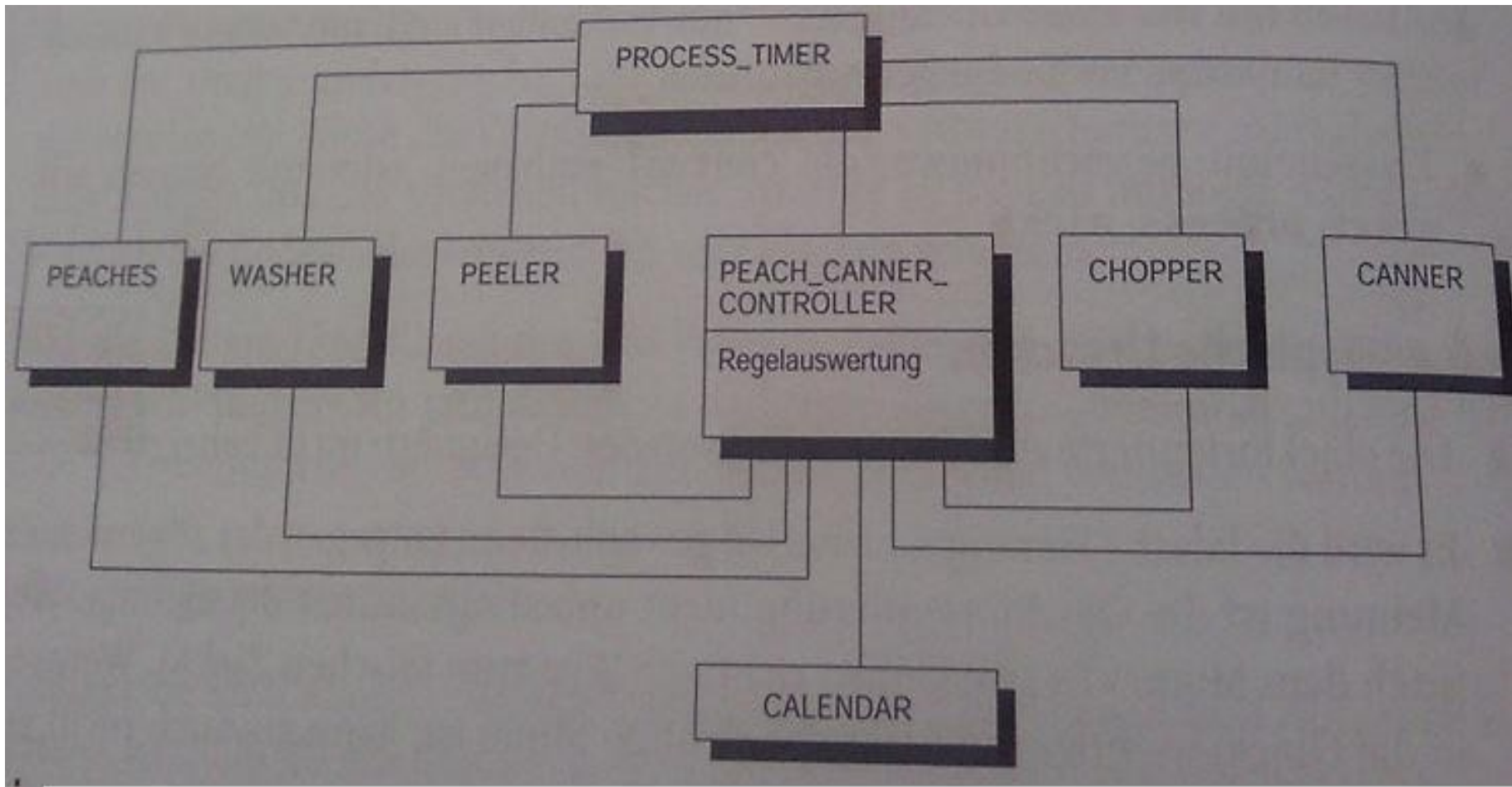
- unerfahrene SW-Architekten, die Architektonisches- Konzept nicht verstanden haben
- fehlerhafte architektonische Vorgaben: nicht immer ist Oo zu bevorzugen!

Refactoring =

- vollständiges Entfernen aus der Klassenhierarchie
- nach Ersetzen muss Funktionalität ersetzt werden
- Korrektur an der Architektur
- Einkapselte steuernde Aktionen in Poltergeistern müssen in die Klassen verschoben werden, die sie aufgerufen haben

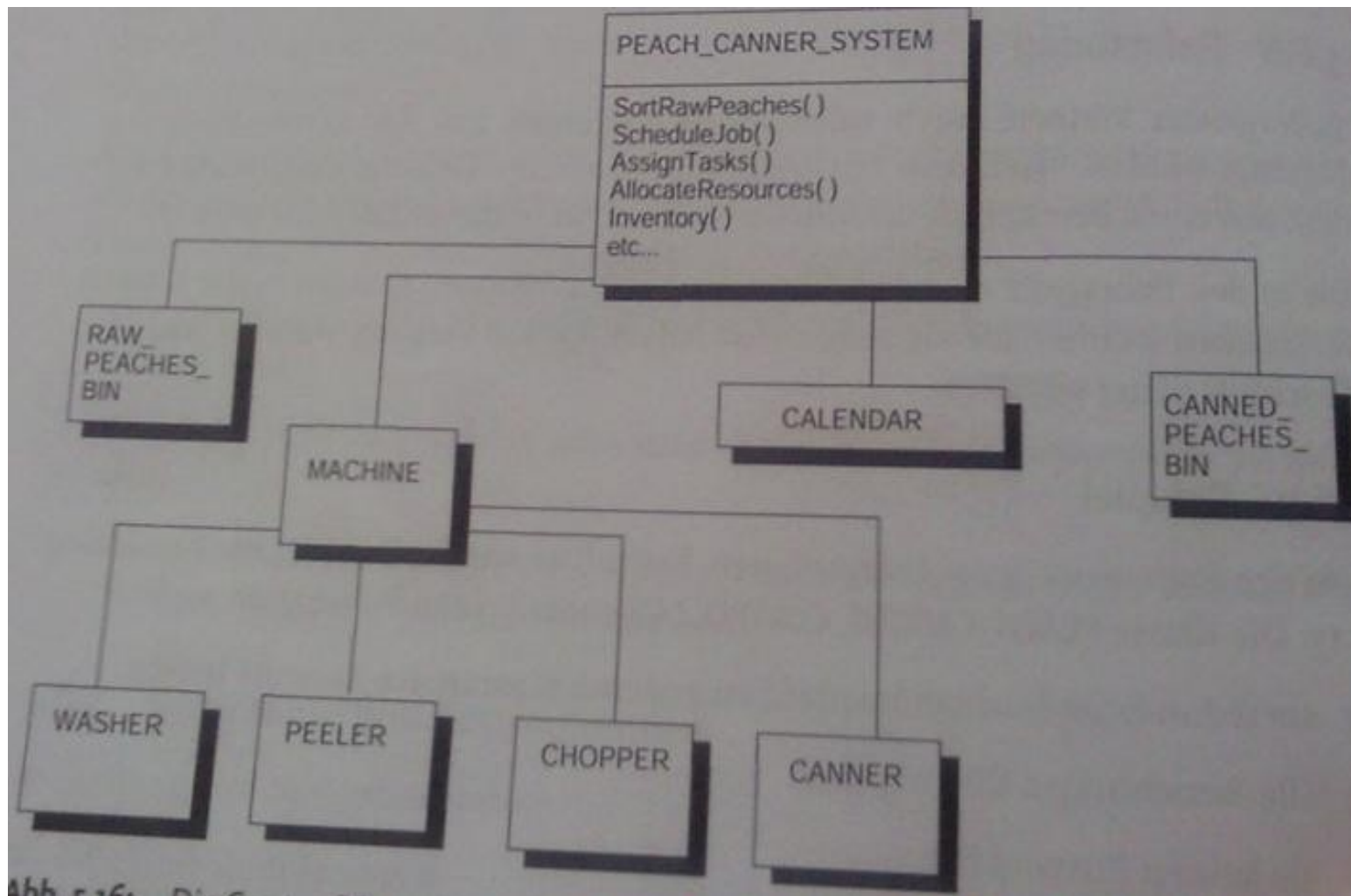
Bekannte Ausnahmen:

sind für dieses AntiPattern nicht bekannt!



Beispiel Poltergeister PEACH_CANNER_CONTROLLER

- redundante Navigationspfade zu anderen Klassendiagramm
- Beziehungen sind flüchtig
- kein Zustand der Klassen
- temporäre Klasse mit kurzer Lebensdauer, ruft nur kurz andere Klassen auf
- keine Interaktion
- Prozesse sind ungeordnet



Die Controller- Klasse nach dem Refactoring

- > Poltergeist Klasse entfernen
- > übrigen Klassen sind nicht mehr daran gebunden
- > Möglichkeit zur Interaktion
- > neue Hierarchie-Möglichkeiten

4.3 Goldener Hammer

Name = Goldener Hammer

Auch bekannt als = Old Yeller, Head-in-the-Sand

Name des Refactoring = Erweitern Sie Ihren Horizont

Typ des Refactoring = Prozess

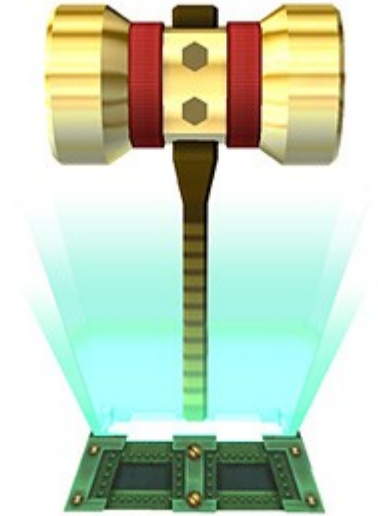
Hauptursachen = Ignoranz, Stolz, Engstirnigkeit

Nicht ausbalancierte Kräfte = Management des Technologietransfers

Anekdotisches = „Ich besitze einen Hammer und alles Übrige sind Nägel“

Hintergrund = Weit verbreitetes AP, Hersteller preisen ihr SW-Produkt an

Häufigste Ebene = Anwendung



Allgemeine Form =

- bekannte Technologie oder bekanntes Konzept, das wie besessen auf viele SW-Probleme angewendet wird
- Lösung: Schulung, Training Lektüre der Fachliteratur für alternative Lösungen
- Team von Entwicklern “Goldener Hammer” kreiert, Lösung für alle Probleme im Unternehmen
- falsche Versprechen: Risiko und Kosten werden reduziert
- es werden minimale Versuche unternommen andere Lösungen zu suchen
- Keine Bereitschaft zu Alternativen

Symptome und Konsequenzen =

- für unterschiedliche Produkte identische Werkzeuge und Produkte
- Performance zu anderen Lösungen wesentlich schlechter
- Bei Erörterung der Systemanforderungen mit Analysten und Endanwendern tritt das Werkzeug in den Vordergrund
 - lenkt von Bereichen ab, bei denen Lösung unbefriedigend ist
- Entwickler besitzen Mangel an Erfahrung mit alternativen Herangehensweisen
- Eingeführte Produkte bestimmen das Design und die Systemarchitektur
- neue Entwicklungen hängen sehr stark vom Produkt und der Technologie eines bestimmten Herstellers ab

Ursachen =

- Entwickler bevorzugt obsessiv ein SW- Konzept
- falsche Anwendung eines bevorzugten Werkzeugs oder Konzepts
- mit einer bestimmten Vorgehensweise wurden viele Erfolge erzielt

AntiPatterns- Goldener Hammer Fortsetzung

- hohe Investitionen in Schulung für bestimmtes Produkt, bestimmte Technologie
- Entwicklergruppe isoliert sich von der industriellen Entwicklung & anderen Firmen

Refactoring =

- philosophischer Aspekt:

- Unternehmen braucht Engagement für die Nutzung neuer Technologien
- Weiterbildung der Entwickler
- Einsicht des Managements bzgl. Weiterbildung, Investitionen in die anfängliche Entwicklung -> solides Fundament,
- Team zusammenstellen mit breitem Erfahrungsspektrum
- Mitarbeiter honorieren und unterstützen

- Entwicklungsprozess:

- neue Entwicklungsstrategien welchen Wechsel der Technologien erlauben
- SW-Komponenten müssen einzeln ausgetauscht werden können
- zwischen Systemen Brücke schlagen
- standardisierte Schnittstellen- Spezifikationen benutzen, CORBA; TCP/IP

AntiPatterns- Goldener Hammer Fortsetzung

Bekannte Ausnahmen =

- wenn das Produkt Teil einer Herstellersuite ist, die die meisten SW-Bedürfnisse abdeckt
- Produkt als langfristige strategische Lösung gedacht, internes System wie z.B. DB mit keinen großartigen Neuerungen

Varianten =

- Entwickler verwendet „Goldenen Hammer“ in allen Phasen der SW- Analyse, des Designs und der Implementierung
- es ist schwer Entwickler zum Umdenken zu bewegen

Beispiel =

- Versicherungsunternehmen das sich beim Wechsel zum Client/Serversystem dazu entschied Access DB von Microsoft als Grundlage für dauerhafte Lösung zu nutzen
- gesamtes Frontend des Call- Center- Systems wurde um frühere Version des Produkts herumkonstruiert
- Entwicklung des Systems kaum mehr möglich
- System hatte keine sechs Monate bestand

Vielen Dank für die Aufmerksamkeit!