Software-Restrukturierung

Andreas Zeller

Lehrstuhl Softwaretechnik
Universität des Saarlandes, Saarbrücken

Restrukturierung

Restrukturierung ist nach dem *Reverse Engineering* (Programmverstehen) der zweite wesentliche Teil des *Software-Reengineerings.*

Wir betrachten zwei Techniken

Refactoring – Restrukturierung von OO-Systemen

Wrapping – Verpacken eines Altsystems in OO-Schnittstelle







Refactoring

Refactoring („Refaktorisieren“) = *Aufspalten* von Software in weitgehend unabhängige *Faktoren*

⇒ Umstrukturieren von Software gemäß Zerlegungsregeln zur Modularisierung

Es gibt keine allgemeine Methode des Refactorings – aber einen *Katalog* von Methoden, ähnlich wie bei *Entwurfsmustern*







Ein Refactoring-Muster

Problem: *Fallunterscheidungen* innerhalb einer Klasse können fast immer durch Einführen von *Unterklassen* ersetzt werden.

Das ermöglicht weitere Lokalisierung – jede Klasse enthält genau die für sie nötigen Berechnungsverfahren.

Ein Refactoring-Muster (2)

Das Muster „Replace Conditional Logic with Polymorphism“ hat die allgemeine Form:

> Eine Fallunterscheidung bestimmt verschiedenes Verhalten, abhängig vom Typ des Objekts.
>
> *Bewege jeden Ast der Fallunterscheidung in eine überladene Methode einer Unterklasse. Mache die ursprüngliche Methode abstrakt.*







Ein Refactoring-Muster (3)

Original-Code:

```
double getSpeed() {
    switch (_type) {
        case EUROPEAN:
            return getBaseSpeed();
        case AFRICAN:
            return getBaseSpeed() -
                        getLoadFactor() * numberOfCoconuts();
        case NORWEGIAN_BLUE:
            return (_isNailed) ? 0 : getBaseSpeed(_voltage);
    }
}
```

. . . und weitere ähnliche Fallunterscheidungen
an zahlreichen Stellen im Code

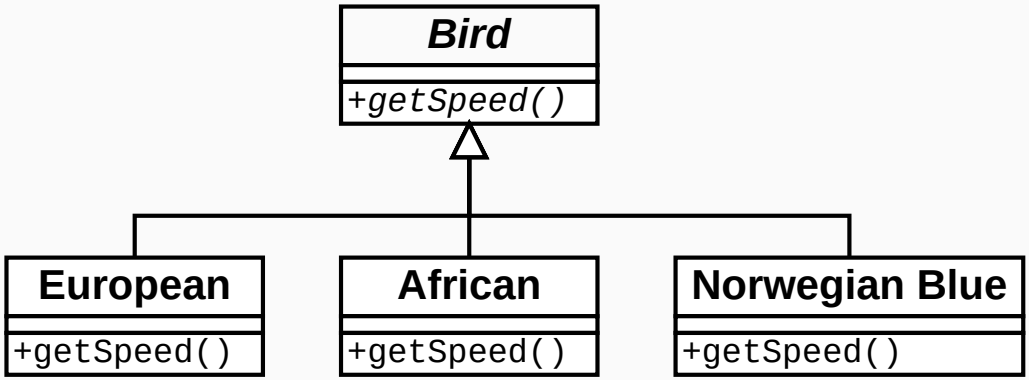






Ein Refactoring-Muster (4)

Das Muster führt eine neue Klassenhierarchie ein:



Die Fallunterscheidung ist nun in die Unterklassen ausgelagert!

Wann wird restrukturiert?

Typische Code-Probleme können auffallen

- Beim Gegenlesen
- Durch Anwendung von *Metriken*

Wir betrachten einige typische Probleme und die passenden Refactoring-Muster.







Wann wird restrukturiert? (2)

Mehrfach auftretender Code.







Wann wird restrukturiert? (2)

Mehrfach auftretender Code.

⇒ Neu zusammenfassen *(Extract Method, Extract Class)*







Wann wird restrukturiert? (2)

Mehrfach auftretender Code.

⇒ Neu zusammenfassen *(Extract Method, Extract Class)*

Lange Methoden.







Wann wird restrukturiert? (2)

Mehrfach auftretender Code.

⇒ Neu zusammenfassen *(Extract Method, Extract Class)*

Lange Methoden.

⇒ Neue Methoden bilden *(Extract Method)*







Wann wird restrukturiert? (2)

Mehrfach auftretender Code.

⇒ Neu zusammenfassen *(Extract Method, Extract Class)*

Lange Methoden.

⇒ Neue Methoden bilden *(Extract Method)*

Große Klassen.







Wann wird restrukturiert? (2)

Mehrfach auftretender Code.

⇒ Neu zusammenfassen *(Extract Method, Extract Class)*

Lange Methoden.

⇒ Neue Methoden bilden *(Extract Method)*

Große Klassen.

⇒ In Klassen aufspalten *(Extract Class)*







Wann wird restrukturiert? (2)

Mehrfach auftretender Code.

⇒ Neu zusammenfassen *(Extract Method, Extract Class)*

Lange Methoden.

⇒ Neue Methoden bilden *(Extract Method)*

Große Klassen.

⇒ In Klassen aufspalten *(Extract Class)*

Lange Parameterlisten.







Wann wird restrukturiert? (2)

Mehrfach auftretender Code.

⇒ Neu zusammenfassen *(Extract Method, Extract Class)*

Lange Methoden.

⇒ Neue Methoden bilden *(Extract Method)*

Große Klassen.

⇒ In Klassen aufspalten *(Extract Class)*

Lange Parameterlisten.

⇒ Parameter-Objekt *(Introduce Parameter Object)*







Wann wird restrukturiert? (2)

Mehrfach auftretender Code.

⇒ Neu zusammenfassen *(Extract Method, Extract Class)*

Lange Methoden.

⇒ Neue Methoden bilden *(Extract Method)*

Große Klassen.

⇒ In Klassen aufspalten *(Extract Class)*

Lange Parameterlisten.

⇒ Parameter-Objekt *(Introduce Parameter Object)*

Fremdgänger. Eine Methode benutzt mehr Dienste einer fremden Klasse als Dienste der eigenen

Wann wird restrukturiert? (2)

Mehrfach auftretender Code.

⇒ Neu zusammenfassen *(Extract Method, Extract Class)*

Lange Methoden.

⇒ Neue Methoden bilden *(Extract Method)*

Große Klassen.

⇒ In Klassen aufspalten *(Extract Class)*

Lange Parameterlisten.

⇒ Parameter-Objekt *(Introduce Parameter Object)*

Fremdgänger. Eine Methode benutzt mehr Dienste einer fremden Klasse als Dienste der eigenen

⇒ Methode bewegen *(Move Method)*

Three strikes and you refactor!







Wann wird restrukturiert? (3)

Datenklumpen. Dieselben Daten treten regelmäßig als Parameter auf







Wann wird restrukturiert? (3)

Datenklumpen. Dieselben Daten treten regelmäßig als Parameter auf

⇒ (Parameter-)Klasse einführen *(Extract Class, Introduce Parameter Object)*







Wann wird restrukturiert? (3)

Datenklumpen. Dieselben Daten treten regelmäßig als Parameter auf

⇒ (Parameter-)Klasse einführen *(Extract Class, Introduce Parameter Object)*

Der Reiz des Primitiven. Häufiger Gebrauch von nicht-OO-Datentypen wie Records







Wann wird restrukturiert? (3)

Datenklumpen. Dieselben Daten treten regelmäßig als Parameter auf

⇒ (Parameter-)Klasse einführen *(Extract Class, Introduce Parameter Object)*

Der Reiz des Primitiven. Häufiger Gebrauch von nicht-OO-Datentypen wie Records

⇒ Durch Klassen ersetzen *(Replace Data Value with Object, Replace Type Code with Class)*

Wann wird restrukturiert? (3)

Datenklumpen. Dieselben Daten treten regelmäßig als Parameter auf

⇒ (Parameter-)Klasse einführen *(Extract Class, Introduce Parameter Object)*

Der Reiz des Primitiven. Häufiger Gebrauch von nicht-OO-Datentypen wie Records

⇒ Durch Klassen ersetzen *(Replace Data Value with Object, Replace Type Code with Class)*

Datenklassen. Eine Klasse enthält nur Daten







Wann wird restrukturiert? (3)

Datenklumpen. Dieselben Daten treten regelmäßig als Parameter auf

⇒ (Parameter-)Klasse einführen *(Extract Class, Introduce Parameter Object)*

Der Reiz des Primitiven. Häufiger Gebrauch von nicht-OO-Datentypen wie Records

⇒ Durch Klassen ersetzen *(Replace Data Value with Object, Replace Type Code with Class)*

Datenklassen. Eine Klasse enthält nur Daten

⇒ Dienste der Klasse zuordnen *(Move Method)*

Wann wird restrukturiert? (3)

Datenklumpen. Dieselben Daten treten regelmäßig als Parameter auf

⇒ (Parameter-)Klasse einführen *(Extract Class, Introduce Parameter Object)*

Der Reiz des Primitiven. Häufiger Gebrauch von nicht-OO-Datentypen wie Records

⇒ Durch Klassen ersetzen *(Replace Data Value with Object, Replace Type Code with Class)*

Datenklassen. Eine Klasse enthält nur Daten

⇒ Dienste der Klasse zuordnen *(Move Method)*

Switch-Anweisungen. Meist Zeichen mangelnder Objektorientierung

Wann wird restrukturiert? (3)

Datenklumpen. Dieselben Daten treten regelmäßig als Parameter auf

⇒ (Parameter-)Klasse einführen *(Extract Class, Introduce Parameter Object)*

Der Reiz des Primitiven. Häufiger Gebrauch von nicht-OO-Datentypen wie Records

⇒ Durch Klassen ersetzen *(Replace Data Value with Object, Replace Type Code with Class)*

Datenklassen. Eine Klasse enthält nur Daten

⇒ Dienste der Klasse zuordnen *(Move Method)*

Switch-Anweisungen. Meist Zeichen mangelnder Objektorientierung

⇒ Dynamische Bindung ausnutzen *(Replace Conditional Logic with Polymorphism)*

Wann wird restrukturiert? (4)

Temporäres Attribut. In einem Attribut werden temporäre Zwischenergebnisse gespeichert







Wann wird restrukturiert? (4)

Temporäres Attribut. In einem Attribut werden temporäre Zwischenergebnisse gespeichert

⇒ Zwischenergebnisse in Klasse auslagern *(Extract Class)*







Wann wird restrukturiert? (4)

Temporäres Attribut. In einem Attribut werden temporäre Zwischenergebnisse gespeichert

⇒ Zwischenergebnisse in Klasse auslagern *(Extract Class)*

Nachrichtenketten. Beispiel:

```
john.getGroup().getDepartment().getManager()
```







Wann wird restrukturiert? (4)

Temporäres Attribut. In einem Attribut werden temporäre Zwischenergebnisse gespeichert

⇒ Zwischenergebnisse in Klasse auslagern *(Extract Class)*

Nachrichtenketten. Beispiel:

```
john.getGroup().getDepartment().getManager()
```

⇒ Abkürzungsmethoden wie `john.getManager()` einführen *(Hide Delegate)*







Wann wird restrukturiert? (4)

Temporäres Attribut. In einem Attribut werden temporäre Zwischenergebnisse gespeichert

⇒ Zwischenergebnisse in Klasse auslagern *(Extract Class)*

Nachrichtenketten. Beispiel:

```
john.getGroup().getDepartment().getManager()
```

⇒ Abkürzungsmethoden wie `john.getManager()` einführen *(Hide Delegate)*

Vermittler. Zu viele Abkürzungsmethoden (`john.getManager()`, `john.getManagerLastName()`, `john.getDepartmentLocation()...`)







Wann wird restrukturiert? (4)

Temporäres Attribut. In einem Attribut werden temporäre Zwischenergebnisse gespeichert

⇒ Zwischenergebnisse in Klasse auslagern *(Extract Class)*

Nachrichtenketten. Beispiel:

`john.getGroup().getDepartment().getManager()`

⇒ Abkürzungsmethoden wie `john.getManager()` einführen *(Hide Delegate)*

Vermittler. Zu viele Abkürzungsmethoden (`john.getManager()`, `john.getManagerLastName()`, `john.getDepartmentLocation()...`)

⇒ Vermittler entfernen *(Remove Middle Man)*







Wann wird restrukturiert? (4)

Temporäres Attribut. In einem Attribut werden temporäre Zwischenergebnisse gespeichert

⇒ Zwischenergebnisse in Klasse auslagern *(Extract Class)*

Nachrichtenketten. Beispiel:

```
john.getGroup().getDepartment().getManager()
```

⇒ Abkürzungsmethoden wie `john.getManager()` einführen *(Hide Delegate)*

Vermittler. Zu viele Abkürzungsmethoden (`john.getManager()`, `john.getManagerLastName()`, `john.getDepartmentLocation()...`)

⇒ Vermittler entfernen *(Remove Middle Man)*

Kommentare. Oft Hinweise auf schwer verständlichen Code

Wann wird restrukturiert? (4)

Temporäres Attribut. In einem Attribut werden temporäre Zwischenergebnisse gespeichert

⇒ Zwischenergebnisse in Klasse auslagern *(Extract Class)*

Nachrichtenketten. Beispiel:

```
john.getGroup().getDepartment().getManager()
```

⇒ Abkürzungsmethoden wie `john.getManager()` einführen *(Hide Delegate)*

Vermittler. Zu viele Abkürzungsmethoden (`john.getManager()`, `john.getManagerLastName()`, `john.getDepartmentLocation()...`)

⇒ Vermittler entfernen *(Remove Middle Man)*

Kommentare. Oft Hinweise auf schwer verständlichen Code

⇒ Methoden explizit machen *(Extract Method, Introduce Assertion)*







Gefahren der Restrukturierung

Änderungen am laufenden System sind stets *gefährlich*, da bestehende Funktionalität gefährdet sein könnte

Never change a running system!







Voraussetzungen für Restrukturierung

Automatisierte Tests nach jeder Änderung







Voraussetzungen für Restrukturierung

Automatisierte Tests nach jeder Änderung

Entwurfswerkzeuge, die einfache Änderungen am Entwurf ermöglichen







Voraussetzungen für Restrukturierung

Automatisierte Tests nach jeder Änderung

Entwurfswerkzeuge, die einfache Änderungen am Entwurf ermöglichen

Dokumentationswerkzeuge, mit denen die Dokumentation stets auf dem neuesten Stand gehalten werden kann







Voraussetzungen für Restrukturierung

Automatisierte Tests nach jeder Änderung

Entwurfswerkzeuge, die einfache Änderungen am Entwurf ermöglichen

Dokumentationswerkzeuge, mit denen die Dokumentation stets auf dem neuesten Stand gehalten werden kann

Versionsverwaltung zum Erhalt früherer Versionen







Voraussetzungen für Restrukturierung

Automatisierte Tests nach jeder Änderung

Entwurfswerkzeuge, die einfache Änderungen am Entwurf ermöglichen

Dokumentationswerkzeuge, mit denen die Dokumentation stets auf dem neuesten Stand gehalten werden kann

Versionsverwaltung zum Erhalt früherer Versionen

Gute Kommunikation innerhalb des Teams, damit Mitglieder über Änderungen informiert werden







Voraussetzungen für Restrukturierung

Automatisierte Tests nach jeder Änderung

Entwurfswerkzeuge, die einfache Änderungen am Entwurf ermöglichen

Dokumentationswerkzeuge, mit denen die Dokumentation stets auf dem neuesten Stand gehalten werden kann

Versionsverwaltung zum Erhalt früherer Versionen

Gute Kommunikation innerhalb des Teams, damit Mitglieder über Änderungen informiert werden

Systematisches Vorgehen – etwa indem existierende und bekannte Refaktorisierungen eingesetzt werden, statt unsystematisch „alles einmal zu überarbeiten“.

Voraussetzungen für Restrukturierung

Automatisierte Tests nach jeder Änderung

Entwurfswerkzeuge, die einfache Änderungen am Entwurf ermöglichen

Dokumentationswerkzeuge, mit denen die Dokumentation stets auf dem neuesten Stand gehalten werden kann

Versionsverwaltung zum Erhalt früherer Versionen

Gute Kommunikation innerhalb des Teams, damit Mitglieder über Änderungen informiert werden

Systematisches Vorgehen – etwa indem existierende und bekannte Refaktorisierungen eingesetzt werden, statt unsystematisch „alles einmal zu überarbeiten“.

Vorgehen in kleinen Schritten mit Tests nach jeder Überarbeitung.

Restrukturierungs-Werkzeuge

Zur Sicherheit tragen auch spezielle *Refaktorisierungs-Werkzeuge*, die auf Knopfdruck bestimmte Refaktorisierungen durchführen

Ziel: Semantik des Programms erhalten!

Leider derzeit noch in den Kinderschuhen

Software-Sanierung

Während Refactoring ein Verfahren für das Überarbeiten *im Kleinen* ist, spricht man bei großen Systemen von *Sanierung.*

Eine Sanierung ist häufig Alternative zu einer Neuentwicklung oder dem Einsatz einer Standardsoftware.

Zur Kosten/Nutzen-Analyse muß man wissen,

- was die derzeitige Wartung kostet
- was die Sanierung kosten wird
- wieviel durch eine Sanierung eingespart werden kann
- wann sich die Sanierungskosten amortisieren







Wann wird ein Altsystem saniert?

1. Altsystem ist häufig wegen Fehlern außer Betrieb
2. Code ist älter als 7 Jahre
3. Struktur zu komplex
4. Computersystem nicht mehr aktuell
5. Programme werden emuliert
6. Module sind zu groß geworden
7. Ressourcenbedarf ist zu groß
8. Fest eingebaute Konstanten müssen umgestoßen werden
9. Ausbildung der Wartungsmitarbeiter zu teuer
10. Technische Dokumentation unbrauchbar
11. Anforderungsdefinition mangelhaft

Wann wird ein Altsystem saniert? (2)

Sanierung ist nur dann attraktiv,

- wenn das Altsystem noch wenigstens drei Jahre zu leben hat und
- der Sanierungsaufwand nicht mehr als 50% des Aufwands einer Neuentwicklung beträgt.

Sanierung ist ein *Transformationsprozeß:* Das Altsystem wird Schritt für Schritt in eine bessere Form gebracht







Wrapping

Wrapping = Verpacken eines Altsystems in objektorientierte Schnittstelle

Grundidee: Altsoftware (oder Teile davon) werden durch eine Schicht nach außen so verpackt, daß objektorientierte Benutzung moglich ist

Anwender merkt nicht, daß sich hinter der „Verpackung“ ein Altsystem befindet

Gut geeignet für *Migration:* Erst objektorientiert verpacken, dann durch Neuimplementierungen ersetzen

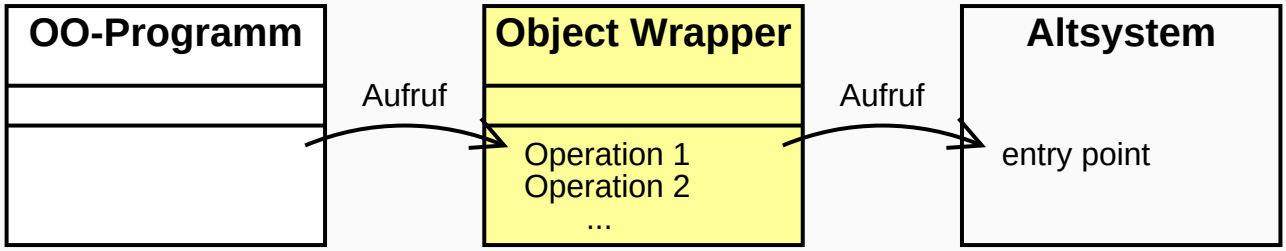






Objektorientierter Verpacker

Objekt verkapselt Altsoftware und transformiert funktionale Schnittstelle in objektorientierte Schnittstelle



- Code des Altsystems bleibt unverändert
- Neue Teile können objektorientiert geschrieben werden
- Altsysteme finden ihren Weg in neue Anwendungen



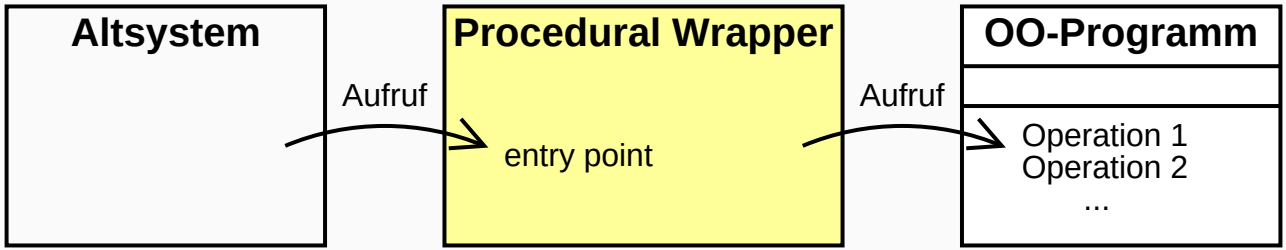




Prozeduraler Verpacker

Ziel: Einsatz eines Objekts in Altsystem

Altsystem ruft *prozeduralen Verpacker* auf, der die Aufrufe zum Objekt weiterleitet



- Code des Altsystems bleibt nahezu unverändert
- Neue Teile können objektorientiert geschrieben werden
- OO-Programme können von beliebig vielen Altsystemen benutzt werden

Alle Verpackungsmöglichkeiten

Prozeßebene Kompletter Prozeß wird mit OO-Schnittstelle versehen; Erzeugen der Eingabe und Abfangen der Ausgabe

Transaktionsebene OO-Schnittstelle simuliert Ein/Ausgaben am Bildschirm; Ausfüllen von Masken und Verarbeiten der Ausgabedaten

Programmebene Starten eines Programms (wie Prozeß)

Modulebene Aufruf eines Unterprogramms mit benötigten Eingabeparametern

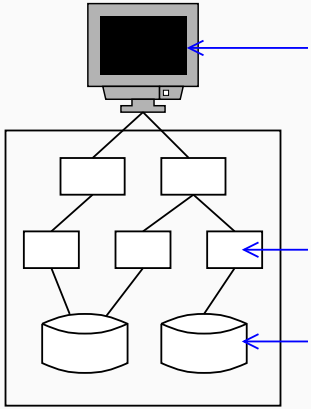
Prozedurebene Aufruf einer Prozedur des Altsystems; benötigt vorheriges Einrichten der globalen Daten

Dateiebene Zugriff auf persistente Daten (= Dateien, Datenbanken) des Altsystems

Migration

Gegeben sei ein Altsystem, das objektorientiert verpackt werden soll

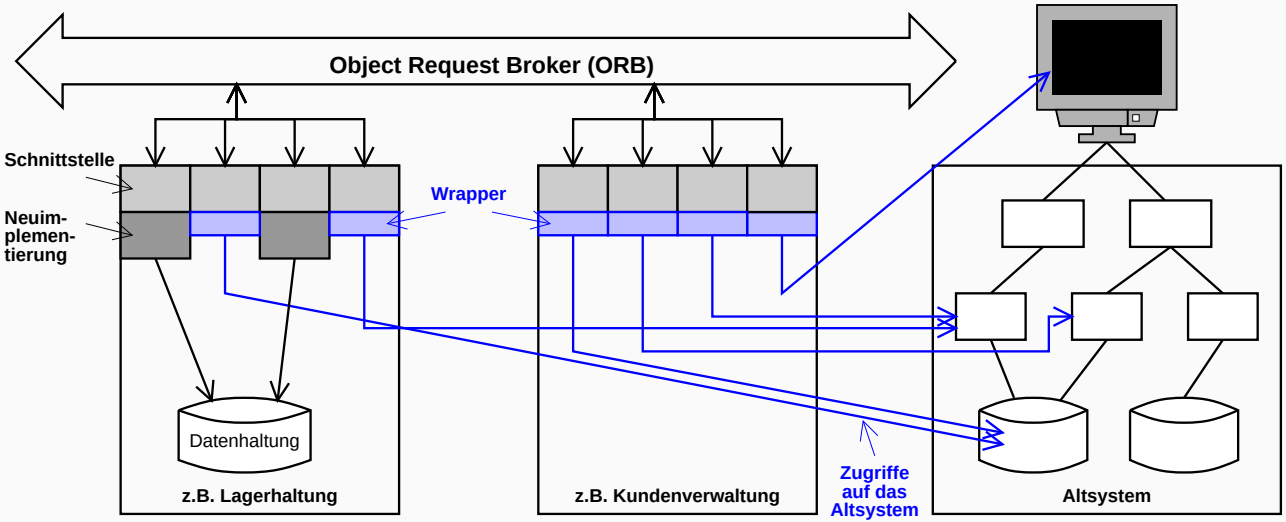
Auf das Altsystem kann an drei Stellen zugegeriffen werden:



1. Simulation von Benutzer-Ein/Ausgaben
   Problem: Effizienz
2. Aufruf einzelner Funktionen
   Problem: Altsystem ist top-down aufgebaut
3. Zugriff auf Datenbank
   Problem: Implementierungsaufwand

Migration (2)

Komponentenbasierte Neuentwicklung



- Komponenten können Stück für Stück neu entwickelt werden
- Gleichzeitig wird der Zugriff auf das Altsystem abgeschaltet

Praktische Probleme

Beim Binden von OO-Komponenten und funktionaler Komponenten können leicht Probleme auftreten:

- Sprachanbindung (z.B. COBOL/C++)
- Versorgen globaler Datenstrukturen im Altsystem
- Verwaltung mehrfacher Objekt-Instanzen
- Verwaltung unterschiedlicher Laufzeitumgebungen (Dateien, Freispeicher, Garbage collection, Exceptions, Lebensdauer von Objekten)

Verpacken auf Prozeß- oder Programmebene ist am einfachsten!







Zusammenfassung

- *Software Reengineering* = Reverse Engineering + Restrukturierung
- *Refactoring* ist das Umstrukturieren von Software *im Kleinen* gemäß Modularisierungskriterien
- *Sanieren* ist das *Überarbeiten im Großen* von Altsystemen
- Refactoring und Sanieren sind stets mit *Risiken* verbunden („Läuft mein Programm noch?“)
- *Wrapping* hilft, Altsysteme in OO-Umgebungen einzusetzen und schrittweise nach OO zu migrieren