Software-Architekturen

Sommersemester 2002

Prof. Dr. Wolfgang Pree
Universität Salzburg
www.SoftwareResearch.net/SWA

OOAD
Richtlinien & Tips

© 2002, W. Pree, Software-Architekturen, SS2002; Teil I

Metriken (I)

- Klassen:

  Anzahl der Methoden: > 5
  < 30

- Methoden:

  durschnittliche Methodenlänge:
  10 – 20 LOC

Metriken (II)

- System:

  Anzahl der zentralen abstrakten Klassen:
  < 10

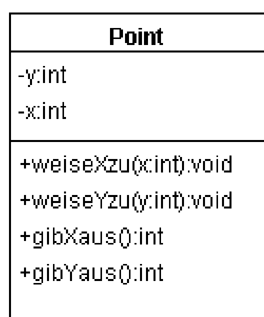
- Vererbungstiefe: < 15

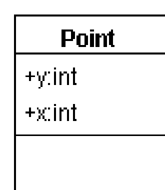
UML Tools

Trotz der Empfehlung der Tool-Hersteller, die Entwicklung möglichst auf UML abzustützen (OOAD; Code-Generierung), hat es sich in der Praxis bewährt, UML-Diagramme lediglich für eine überblicksmäßige Visualisierung der wichtigen Aspekte eines Systems zu benutzen.

Information Hiding

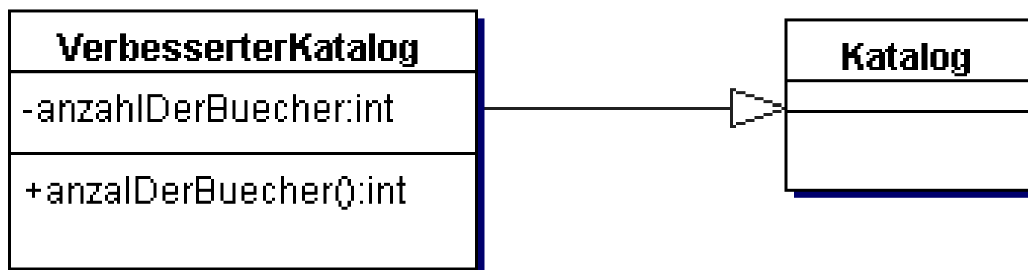
Mittels Zugriffsrechten werden Implementierungsdetails vor dem Benutzer der Klasse versteckt und gesichert, daß Objekte immer in konsistenten Zuständen sind.





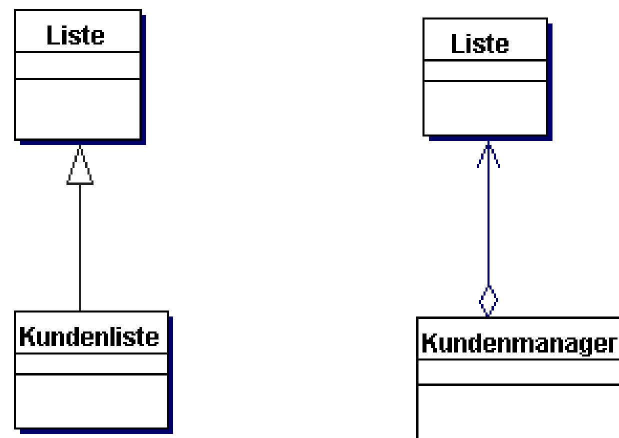
Vererbung—nicht immer!

Vererbung sollte nicht verwendet werden, um Klassen zu korrigieren oder Implementierungen zu verbessern.



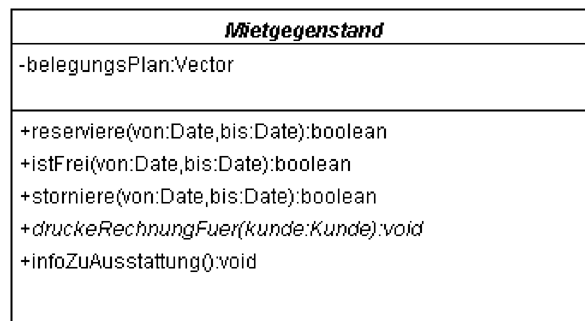
Is-A versus Has-A

Vererbung (Is-A) sollte keinesfalls verwendet werden, um „Has-A-Beziehungen“ (Associations oder Aggregations) zu modellieren.



Abstrakte Klassen (I)

30% - 40% der Methoden, die in eine abstrakte Klasse herausfaktorisiert werden sollten, sind „offensichtlich“.



Abstrakte Klassen (II)

Wird zuwenig „herausfaktorisiert“, so wird es schwierig, Halbfertigfabrikate zu implementieren.

- Wird zuviel „herausfaktorisiert“, so werden die Unterklassen zu überladen bzw. mit unnötigem Verhalten belastet.

Interfaces versus abstrakte Klassen

SOFTWARE RESEARCH LAB

Gründe für Interfaces:

- Die kritische Masse an Methoden für eine Klasse wird nicht erreicht.
- Es werden keine Instanzvariablen oder keine konkreten Methoden gebraucht.
- Unabhängigkeit von der Klassenhierarchie

SOFTWARE RESEARCH LAB

(Re-)Design-Strategien

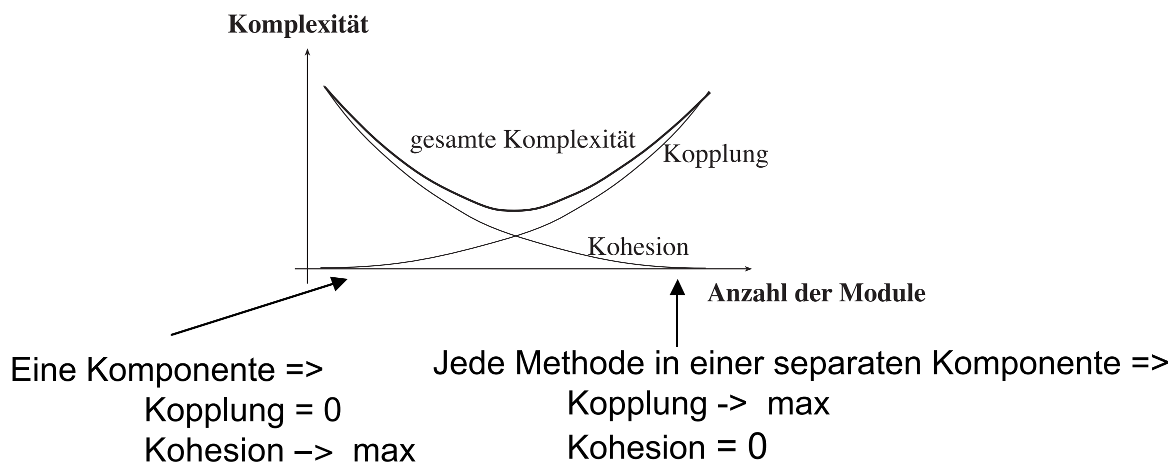
Allgemein

- Architektur-Zerlegung in Pakete/Module/Klassen/Komponenten/ Objekte prüfen
- Hots Spots (Flexibilität) von Frameworks prüfen

Modularisierung/Strukturierung in der Theorie (I)

Gleichgewicht zwischen

- **Maximierung der Kohesion innerhalb einer Komponente (Modul/Klasse)**
- **Minimierung der Kopplung zwischen Komponenten**



Modularisierung/Strukturierung in der Theorie (II)

SOFTWARE RESEARCH LAB

- Ausgeglichene Verteilung von „Responsibilities“ zwischen Komponenten
- Minimale Schnittstelle einer Komponente (erhöht die Kopplung innerhalb der K.)
  - enge Verbindung zw. Methoden und Instanzvariablen
  - keine redundanten Methoden
  - kleine Anzahl von Parametern
  - ausdrucksstarkes und konsistentes Namensschema
  - keine globalen Daten/Komponenten/Objekte

Anwendung der Theorie auf OO Systeme

SOFTWARE RESEARCH LAB

- grobkörniges Design von Klassenhierarchien
- Kopplung/Interaktion von Komponenten
- Kohesion innerhalb einer Komponente
- Evolution von Klassenhierarchien
- Hot Spots (Variation Points) von Frameworks

Grobkörniges Design von Klassenhierachien (I)

SOFTWARE RESEARCH LAB

Klassen können grob in **Familien, Teams und Subsysteme** unterteilt werden.

Klassenfamilien

- basieren auf abstrakten Klassen/Schnittstellen
  - Container Klassen (abstrakte Klasse Container)
  - GUI Komponenten (abstrakte Klasse Component)
- bottom-up oder top-down Entwicklung von Familien
- Wurzeln von Klassenfamilien sollten „leicht“ sein, insbesondere Datenrepräsentationen (Instanzvariable) vermeiden

Grobkörniges Design von Klassenhierachien (II)

SOFTWARE RESEARCH LAB

Klassenteams

- bestehen aus mehreren Klassenfamilien
  - ET++: Text team besteht aus Text, TextView undTextFormatter Familien
  - Container und Iterator Familien bilden ein Team
- werden als ganzes wiederverwendet
- sind abstrakt gekoppelt => Vermeidung von enger Kopplung

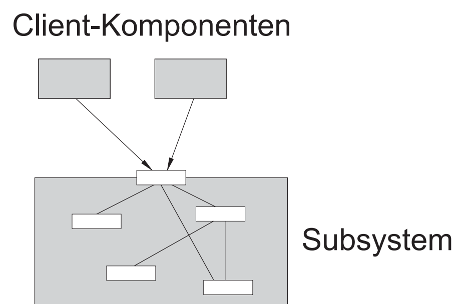
Grobkörniges Design von Klassenhierachien (III)

SOFTWARE RESEARCH LAB

Subsysteme

Explizite Kapselung eines Klassenteams
(–> Facade Pattern)

- reduziert die Kopplung zwischen Klassen eines Teams
- Definition einer zusätzlichen Abstraktion, um Wiederverwendung für Clients zu vereinfachen

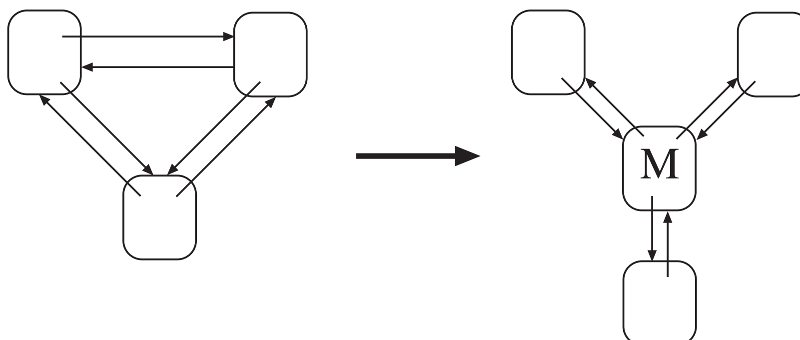


© 2002, W. Pree, Software-Architekturen, SS2002; Teil I

Reduktion der Kopplung durch Mediatoren

SOFTWARE RESEARCH LAB

Effekt des Mediator Patterns:
Komponenten können eher unabhängig voneinander wiederverwendet werden.



© 2002, W. Pree, Software-Architekturen, SS2002; Teil I

Erreichen von Kohesion innerhalb einer Komponente

SOFTWARE RESEARCH LAB

Um die Kohesion qualitativ zu prüfen, sind folgende Fragen hilfreich

- Kann eine Komponente
  - verstanden
  - getestet

  werden, ohne zu wissen, wie sie in ein System eingebracht wird?

  siehe Fallstudie: Ereignissimulation
- Hat eine Komponente Seiteneffekte auf andere Komponenten?

Evolution von Klassenhierarchien

SOFTWARE RESEARCH LAB

Vertikale Reorganisationen

- Verschieben von gemeinsamen Eigenschaften/Verhalten hinauf in der Klassenhierarchie (-> abstrakte Klassen)
- Aufsplitten einer zu komplexen Klasse in eine Klassenfamilie
- Vermeiden des Überschreibens von zu vielen spezifischen Methoden, indem abstrakte Klassen eingeführt werden

Horizontale Reorganisationen

- Aufsplitten einer zu komplexen Klasse in ein Klassenteam/Subsystem
- Verschieben von Verhalten in „Strategie“-Klassen (Stategy/Bridge-Patterns)

Analyse von Variation Points (= Hot Spots)

SOFTWARE RESEARCH LAB

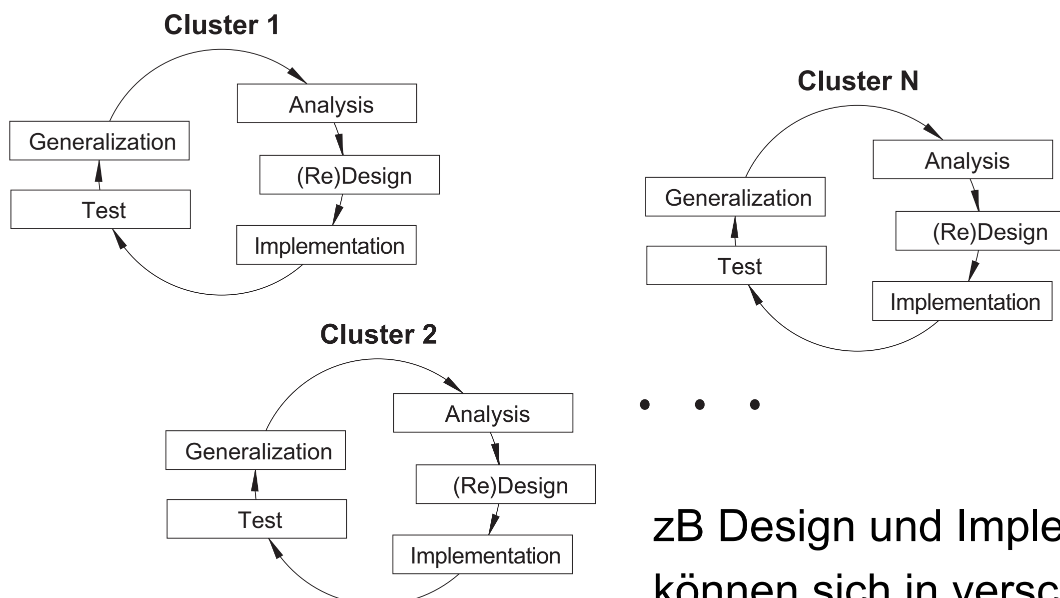
Was macht die Qualität eines Frameworks aus?

Flexibilität um der Flexibilität willen—indem möglichst viele Patterns angewendet werden—führt zu unnötig komplexen Frameworks

- Flexibilität muß in der „richtigen Dosis“ hinein
- Identifikation von Hot Spots stellt eine explizite Aktivität im Framework-Entwicklungsprozeß dar

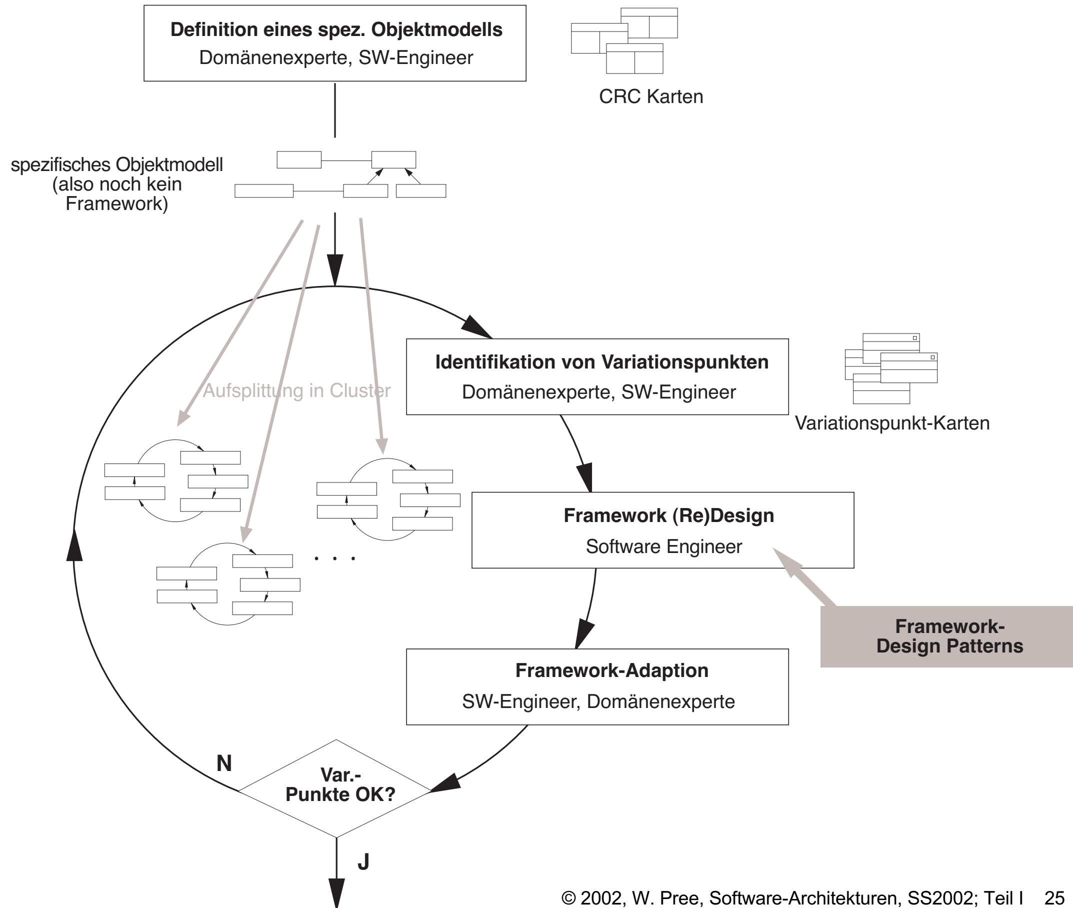
OO: Cluster-Modell (Bertrand Meyer)

SOFTWARE RESEARCH LAB



zB Design und Implementierung können sich in verschiedenen Clustern überlappen

erweitertes Cluster-Modell bei Framework-Entwicklung



Literaturhinweise

Literaturhinweise (I)

Gamma E., Helm R., Johnson R. and Vlissides J. (1995). Design Patterns—Elements of Reusable OO Software. Reading, MA: Addison-Wesley (auch als CD verfügbar)

Pree W. (1995). Design Patterns for Object-Oriented Software Development. Reading, Massachusetts: Addison-Wesley/ACM Press

Fontoura M., Pree W., Rumpe B. (2001) The UML-F Profile for Framework Architectures, Addison Wesley

Szyperski C. (1998) Component Software—Beyond Object-Oriented Programming, Addison-Wesley.

Fayad M., Schmidt D., Johnson R. (1999) Building Application Frameworks: Object-Oriented Foundations of Framework Design, Wiley

Fayad M., Schmidt D., Johnson R. (1999) Implementing Application Frameworks: Object-Oriented Frameworks at Work, Wiley

Fayad M., Schmidt D., Johnson R. (1999) Domain-Specific Application Frameworks: Manufacturing, Networking, Distributed Systems, and Software Development, Wiley

Shannon et al.: Java 2 Platform, Enterprise Edition—Platform and Component Specifications, Addison-Wesley, 2000

Horstmann C., Cornell G.: Core Java—Advanced Features, Vol. II, Prentice Hall, 1998

Hamilton G. et al.: JDBC Database Access with Java, Addison-Wesley, 2000

Monson-Haefel R.: Enterprise JavaBeans, O'Reilly, 2000

Englander R.: Developing Java Beans, O'Reilly, 1999

Vanhelsuwe L.: Mastering JavaBeans (full text: http://www.lv.clara.co.uk/masbeans.html)

Literaturhinweise (II)

Booch G., Jacobson I, Rumbaugh J. (1999)

- Objektorientierte Analyse und Design. Mit praktischen Anwendungsbeispielen
- Das UML-Benutzerhandbuch (Unified Modeling Language User Guide)
- Unified Modeling Language Reference Manual,
- The Objectory Software Development Process,
- Addison-Wesley

Österreich B. (2000) Erfolgreich mit Objektorientierung, Oldenburg Verlag

Balzert H. (2000) Objektorientierung in 7 Tagen, m. CD-ROM. Vom UML-Modell zur fertigen Web-Anwendung, Spektrum Akadem. Verlag

Gabriel R.P. (1996). Patterns of Software—Tales from the Software Community. New York: Oxford University Press

Wirth N, Gutknecht J. (1993) Project Oberon, Addison-Wesley.