# Implementierung im Domain und Application Engineering

**Hochschule Mannheim**
**Seminar im Master-Studiengang Informatik**

Technische Aspekte von Software-Produktlinien

Arne Arnold und Christian Mohr
14. Juni 2005

# Agenda

- **Variabilität in der Produktlinienentwicklung**
  - Referenzmodell
  - Darstellung von Variabilität in den einzelnen Phasen
  - Orthogonales Variabilitätsmodell
- Technologiedimensionen
- Vertikale Variabilitätsmechanismen
- Horizontale Variabilitätsmechanismen
- Auswahl und Bewertung von geeigneten Mechanismen
- Fazit

## Einleitung – Software-Produktlinien (SPL)

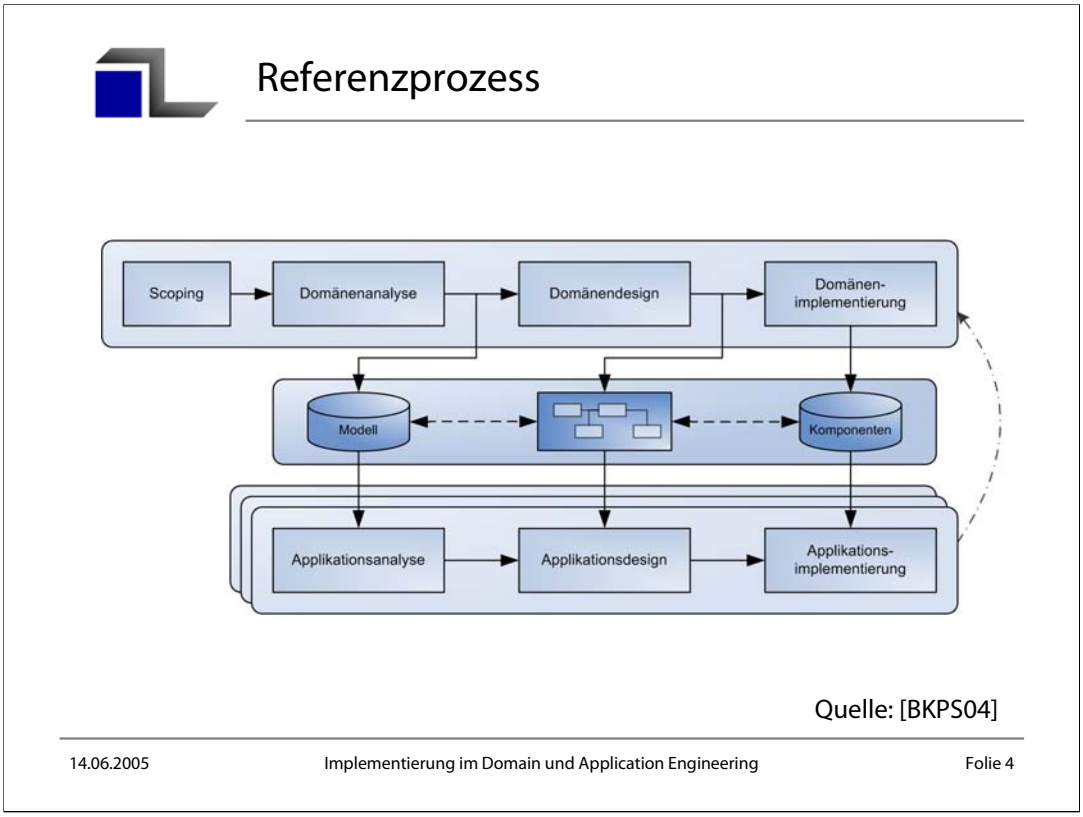
- Gründe für den Einsatz von SPL:
  - Aufwandsreduktion
  - Reduktion von time-to-market
  - Niedrigere Kosten für den Kunden
  - Höhere Qualität (intensiv geprüfte Produktbasis)
- Eigenschaften:
  - Organisierte Wiederverwendung und organisierte Variabilität durch Variationspunkte
  - Entwicklung einer Produktlinienplattform für Wiederverwendung

14.06.2005 Implementierung im Domain und Application Engineering Folie 3

Für die Entwicklung einer Software-Produktlinie (SPL) sprechen neben den oben genannten Gründen eine komponentenbasierte Entwicklung, die eine hohe Wiederverwendung ermöglicht, und eine größere Produktpalette. Dabei orientiert man sich am Bedarf einer bestimmten (Anwendungs-)Domäne.

Da die Entwicklung einer Produktlinie sehr aufwendig ist, muss abgewogen werden, wie viele Produkte potentiell aus der Produktlinie abgeleitet werden sollen. Die Einführung einer Produktlinie lohnt sich ab der 2. bis 3. Produktableitung.

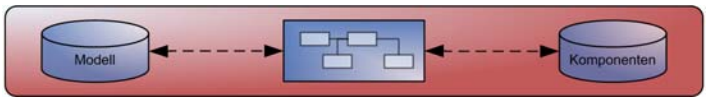
Hauptkennzeichen einer Produktlinienentwicklung ist eine gezielte Wiederverwendung und die systematische Organisation von Variabilität in Variationspunkten und Varianten. Diese drücken die Unterschiede zwischen den ableitbaren Produkten aus.



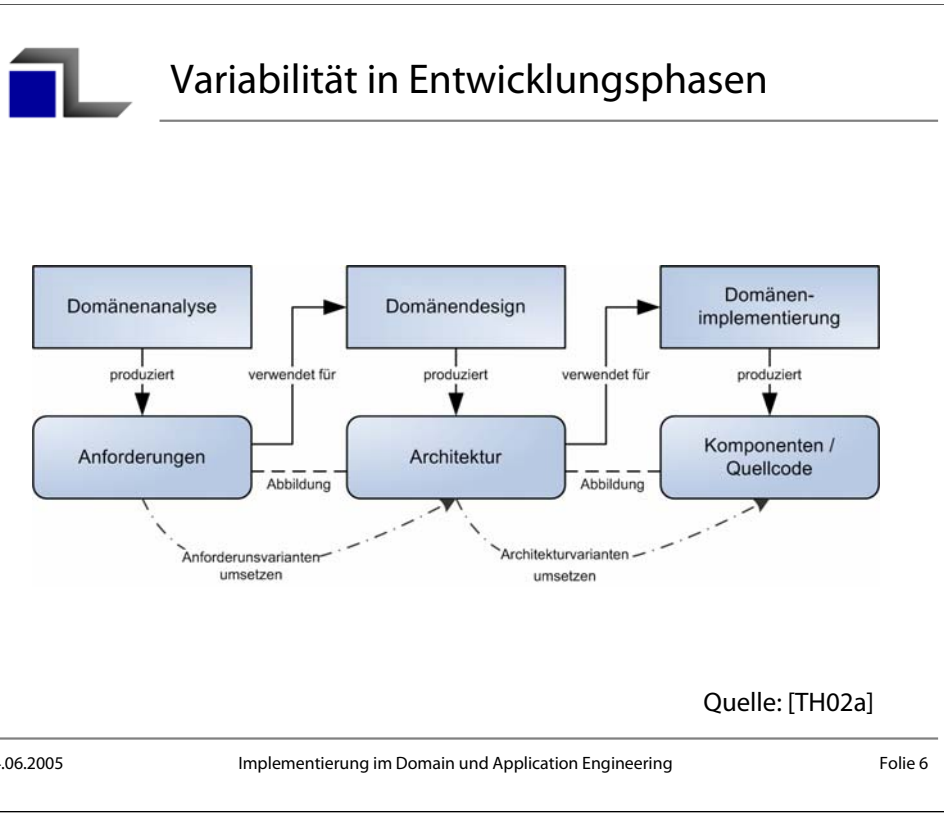

Das Referenzmodell für die Entwicklung von Software-Produktlinien ist aus den vorangegangenen Seminarvorträgen bekannt. Scoping, sowie Analyse und Design im Domain- und Application Engineering wurden bereits behandelt. Im Folgenden wird die Implementierung im Domain- und Application Engineering näher betrachtet.

Um darzustellen, wie sich Variabilität in der Implementierungsphase darstellen lässt, muss beachtet werden, das Variabilität in allen Phasen der Produktlinienentwicklung eine wichtige Rolle spielt.

# Referenzprozess




Variabilität einer Produktlinie wirkt sich in mehreren Abstraktionsebenen aus! [Bec04]

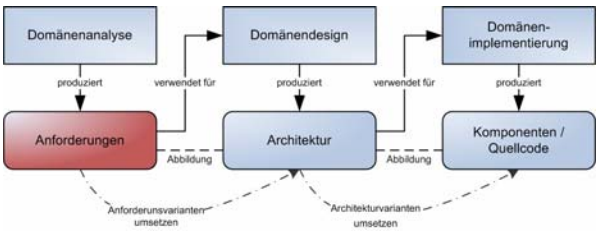



Variationspunkte und zugehörige Varianten ziehen sich durch alle Schritte der Produktlinienentwicklung! Die Ergebnisse einer Phase werden in der nachfolgenden berücksichtigt und weiter verfeinert bzw. genauer spezifiziert.

Die im Design entwickelte Architektur setzt beispielsweise die in den Anforderungen identifizierten Variationspunkte um. Während der Implementierung müssen die Variationspunkte in der Architektur in Komponenten und Quellcode umgesetzt werden.

Dabei stellen sich Variabilitätspunkte in den einzelnen Abstraktionsebenen jeweils unterschiedlich dar, so dass unterschiedliche Darstellungsformen verwendet werden.

## Variabilität in den Anforderungen




- Ausprägungen
  - Features
  - Datenumfang
  - Systemzugang
  - Systemschnittstelle
  - Abläufe
  - Datenformat
  - Benutzerschnittstelle
  - Qualität

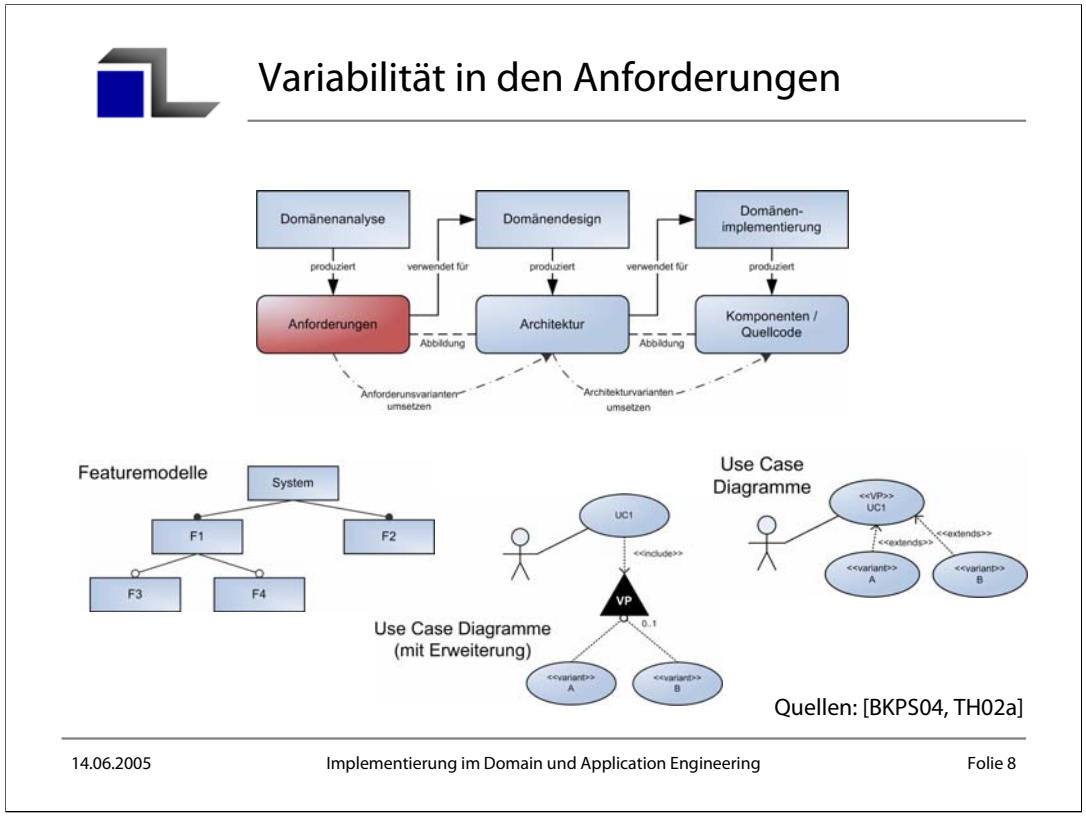
Quellen: [BKPS04]



Variabilitäten, die in der Anforderungsanalyse für ein bestimmtes System identifiziert wurden, können verschiedene Ausprägungen haben.

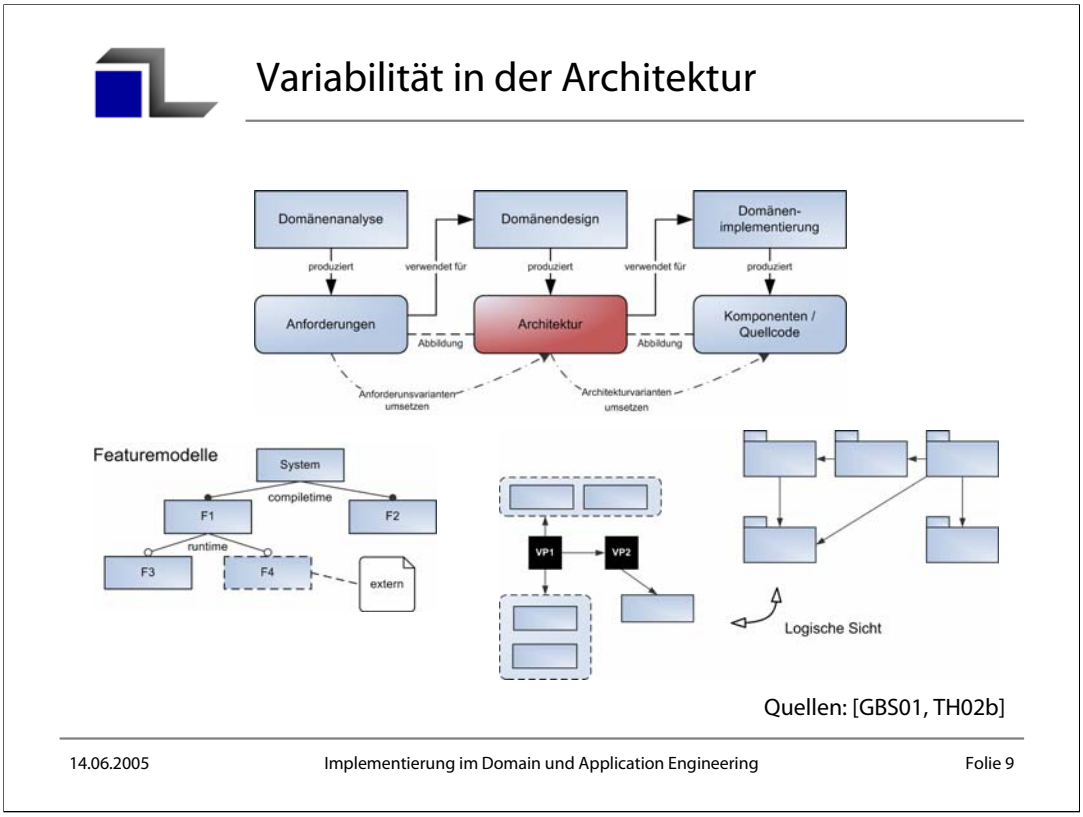
Neben Variabilität in Features ist beispielsweise auch eine Variabilität im Datenumfang oder in Qualität möglich. Dies könnte beispielsweise die Verfügbarkeit von Produkten betreffen (Wartungsfenster).

Die identifizierten Variabilitäten und deren Ausprägung wirkt sich auf die gesamte nachfolgende Entwicklung aus.




Die Variabilität in den Anforderungen kann durch unterschiedliche Modelle beschrieben und dokumentiert werden. Dazu gehören:

- Featuremodelle (vgl. [TH02a], S. 67)
- Use Case Diagramme
- Use Case Diagramme mit Erweiterungen zur expliziten Darstellung von Variationspunkten (vgl. [BKPS04], S. 71 f.)



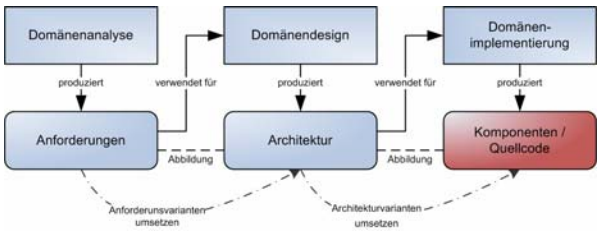

Auch in der Beschreibung der Architektur werden Variationspunkte und Varianten explizit festgehalten.

> *„An explicit design of variants in the architecture and the definition of variation points that support finding those variants in the architecture enables developers who are not necessarily the creators of the architecture to fully employ designed variations.“* (Vgl. [BB01], S. 132)

Auch in der Architektur können Variationen durch verschiedene Modelle beschrieben und dokumentiert werden. Beispielsweise können die in den Anforderungen spezifizierten Featuremodelle verfeinert werden (z.B. um die Binding Time). Außerdem ist eine Darstellung von Variationspunkten und Varianten in den architektonischen Sichten möglich.

Die Folie zeigt beispielhaft die Beziehungen von Komponenten und Variationspunkten in der logischen Sicht. Thiel und Hein stellen ein Modell vor, mit dem sich Variationen auch in anderen Sichten darstellen lassen (vgl. [TH02b]).

## Variabilität in der Implementierung

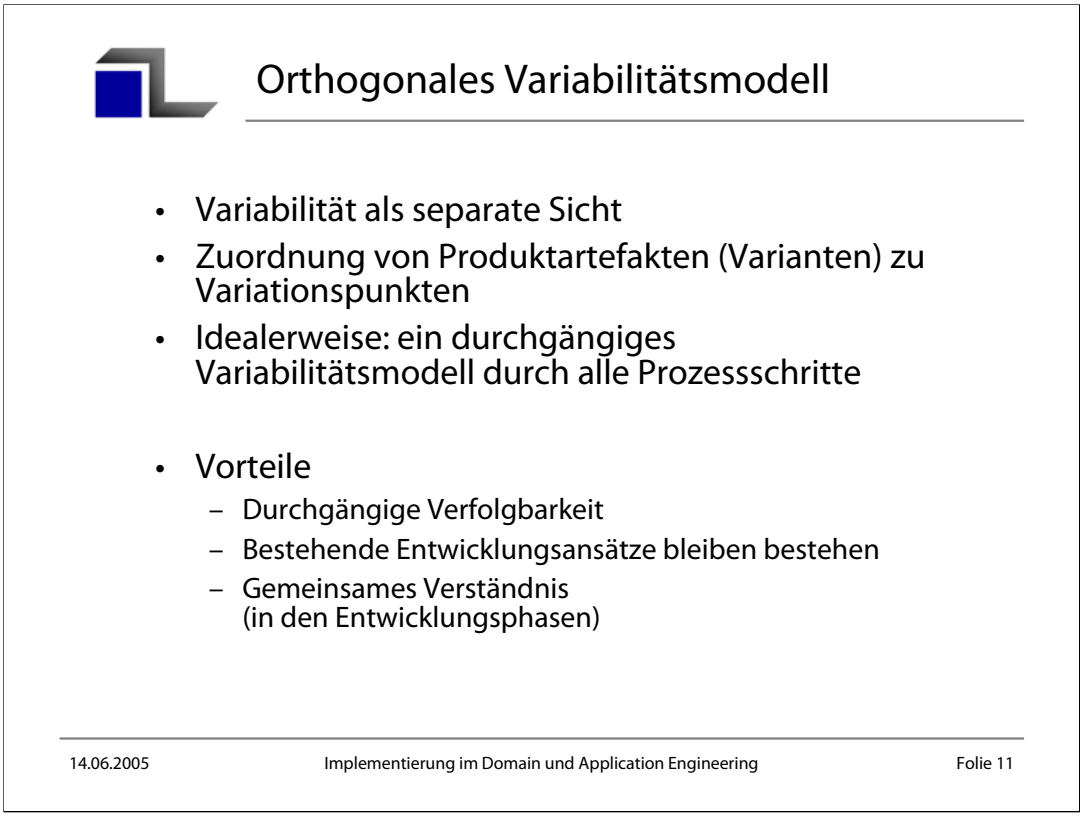



- Die Variabilität in der Implementierung wird durch die Variabilität in den Anforderungen und der Architektur beeinflusst.
- Architektur ist wichtig, Implementierung ist essentiell
  →Es werden adäquate Implementierungstechniken benötigt.

14.06.2005 Implementierung im Domain und Application Engineering Folie 10

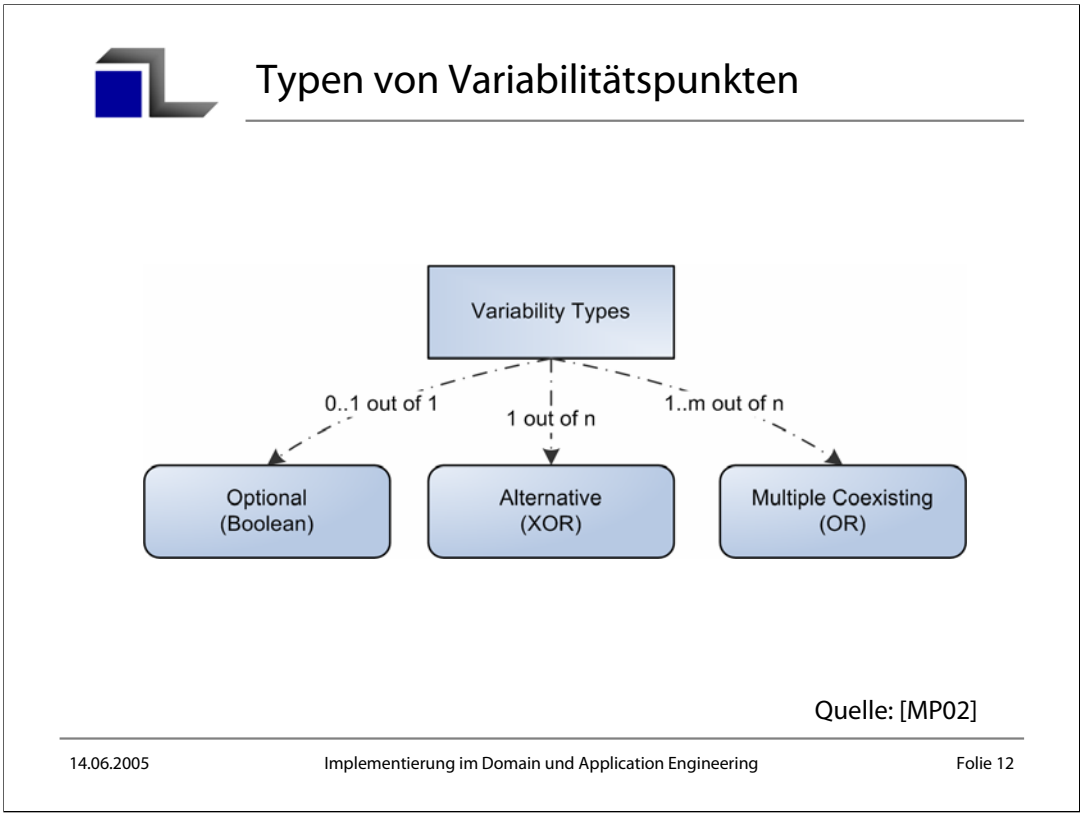
Die Umsetzung von Variationspunkten auf Implementierungsebene steht im Zentrum des Vortrags. Um die in der Architektur festgelegten Ziele zu erreichen, müssen dazu Mechanismen zu Implementierung von Variabilität gefunden werden.

Mit diesen Variabilitätsmechanismen müssen sich unterschiedliche Typen von Variabilitätspunkten realisieren lassen.




Da Variabilität die Entwicklung durchgängig beeinflusst und in den einzelnen Phasen der Entwicklung explizit beschrieben wird, macht es Sinn, sie als separate Sicht zu betrachten. Dies kann durch ein *orthogonales Variabilitätsmodell* (vgl. [BKPS04], S. 22) erreicht werden. Das Wissen über Variabilität wird dabei zentral dokumentiert und Elementen in den Modellen der einzelnen Prozessschritte zugeordnet.

Durch dieses Vorgehen lässt sich eine durchgängige Verfolgbarkeit von Variationspunkten (durch die Modelle der Entwicklungsphasen) erreichen. Man erspart sich dadurch die einzelnen Modelle um Darstellungsmöglichkeiten für Varianten und Variationspunkte zu erweitern. Den Projektbeteiligten wird durch die zentrale Dokumentation ein gemeinsames Verständnis und eine gemeinsame Terminologie bereitgestellt.

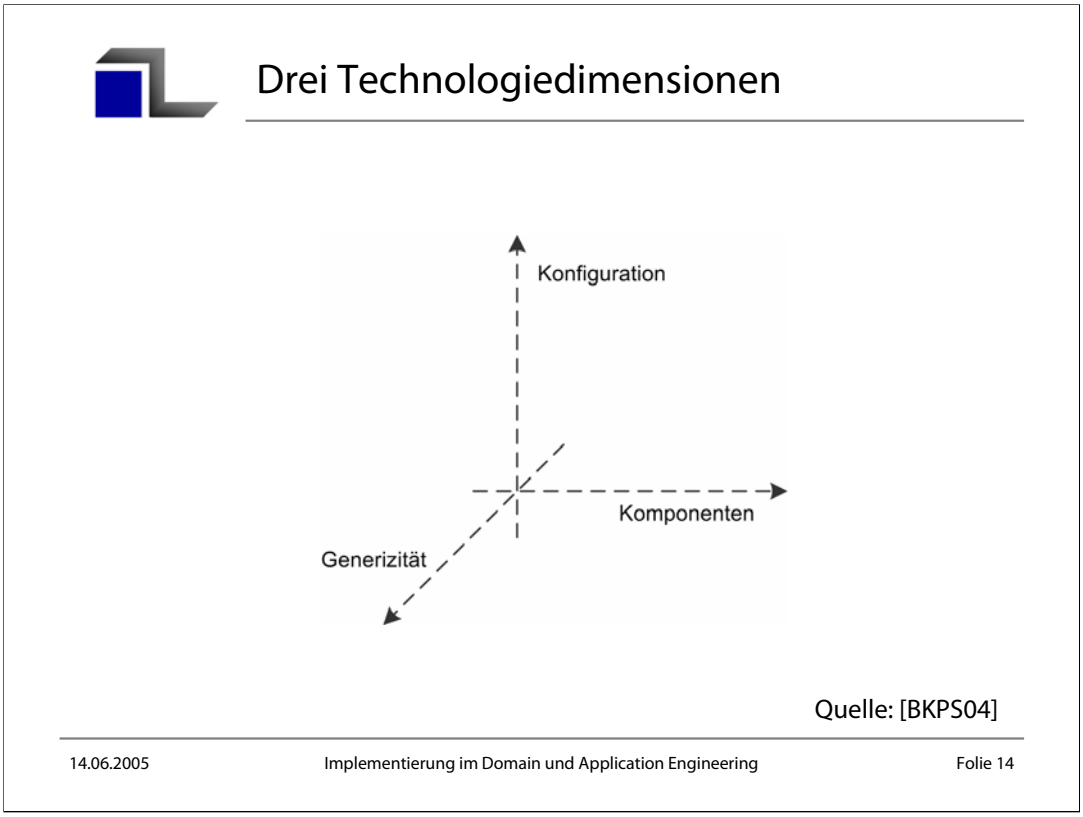



Variabilitätspunkte lassen sich in drei Typen unterteilen:

- **Boolean:** Eine optionale Auswahlmöglichkeit
- **XOR:** Mehr als eine Auswahlmöglichkeit (sich gegenseitig ausschließend)
- **OR:** Mehr als eine Auswahlmöglichkeit (sich nicht gegenseitig ausschließend)

# Agenda

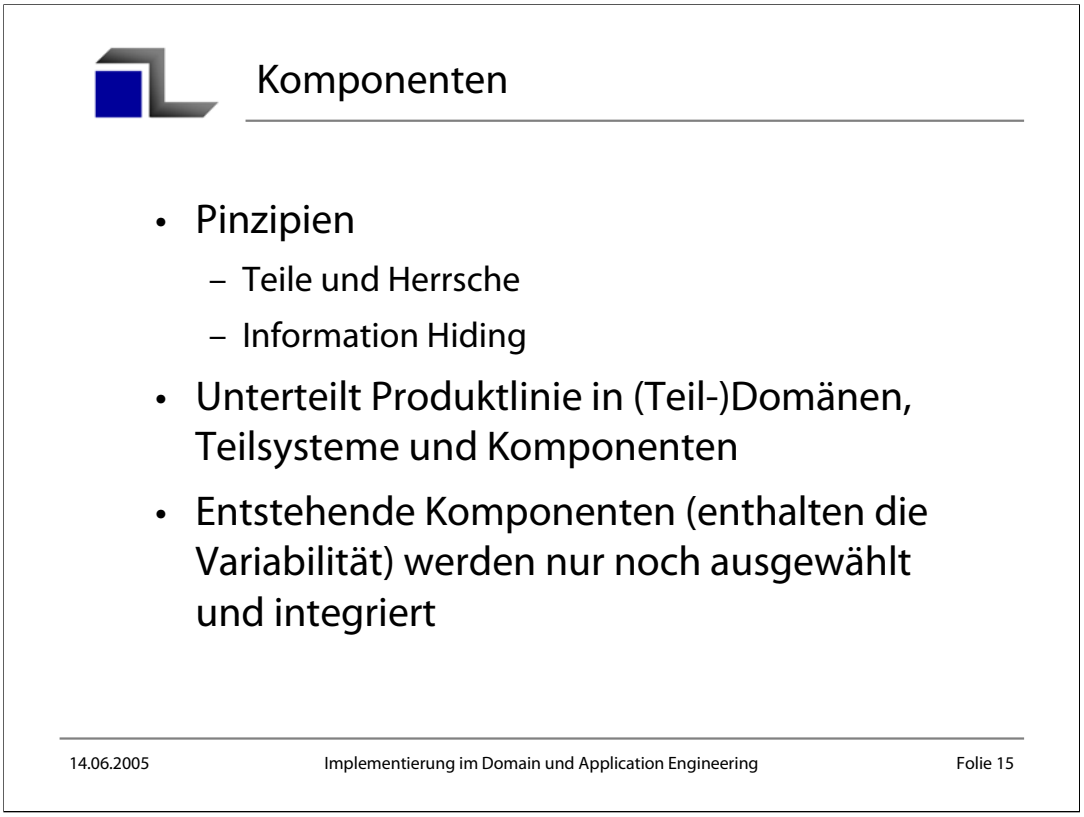
- Variabilität in der Produktlinienentwicklung
- **Technologiedimensionen**
  - Komponenten
  - Konfigurationsmanagement
  - Generizität
- Vertikale Variabilitätsmechanismen
- Horizontale Variabilitätsmechanismen
- Auswahl und Bewertung von geeigneten Mechanismen
- Fazit




Im Rahmen des PoLITe Projektes (**Pro**duct **L**ine **I**mplementation **Te**chnologies) wurden drei Technologiedimensionen von Produktlinien definiert (vgl. [BKPS04], S. 110).

- Komponenten
- Konfiguration
- Generizität

Die drei Dimensionen werden auf den nachfolgenden Folien genauer beschrieben.

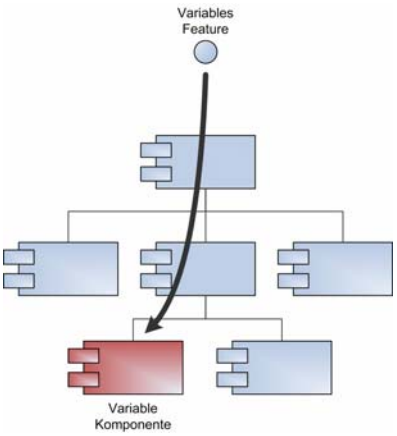



„Die Effiziente Entwicklung von Systemvarianten wird in erster Linie durch umfassende Wiederverwendung erreicht“ (vgl. [BKPS04], S. 111). Dabei spielen die beiden bekannten Prinzipien „Teile und Herrsche“ und das „Information Hiding“ eine große Rolle. Die Architektur einer Produktlinie zerteilt das System in Subsysteme und diese weiter in Komponenten. Deren Details und Belange werden gekapselt.

So können die entstandenen Komponenten einfach selektiert und integriert werden, wenn Produktvarianten aus der Produktlinienplattform abgeleitet werden.

## Variabilität in Komponenten

- Variablitität wird entlang der Dekompositionshierarchie weitergereicht
- Realisierung von Variabilitätspunkten durch alternative Komponenten möglich
- Oft aber Adaption statt einfachem Einfügen der Komponente nötig
- Einsatz von Dynamic Class Loading oder einem Komponenten-Manager auf Komponenten-Ebene löst dieses Problem.




Quelle: [BKPS04]



Bei der Unterteilung des Systems in Komponenten wird die Variabilität entlang der Dekompositionshierarchie weiter gereicht. Die Abbildung auf der Folie verdeutlicht dies.

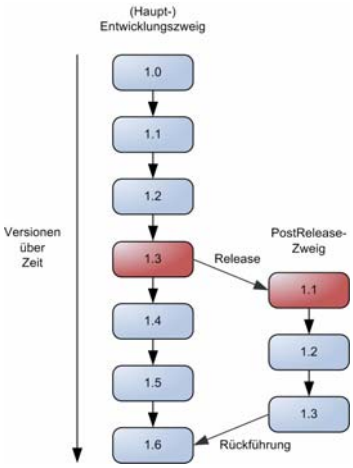
Es sind aber auch verschiedene Realisierungen der gleichen Komponente für unterschiedliche Anforderungen möglich. So lassen sich Variabilitätspunkte mit Varianten realisieren. Durch Selektion einer Komponente wird eine Variante ausgewählt.

Oft ist aber die einfache Wiederverwendung einer bestehenden Komponente im entsprechenden Kontext nicht möglich. Statt dessen ist die Adaption der Komponente nötig. Dieser Nachteil entsteht oft durch eine nicht ausreichend Komponenten-orientierte Architektur. Sie bedeutet zusätzlicher Aufwand und macht die Wiederverwendung schwierig.

Durch Ansätze wie Dynamic Class Loading oder die Verwendung eines Komponenten-Managers können solche Probleme vermieden werden.

# Konfigurationsmanagement

- **Konfiguration**
  - Vektor, der auf eine Menge von optionalen bzw. alternativen Komponenten verweist
  - Ein Produkt entspricht dabei einer Konfiguration
- **Konfigurationsmanagementsystem**
  - Erfassung und Verwaltung von zeitlichen Änderungen der Implementierung von Komponenten
  - Ermöglicht Verzweigungen in der Versionshistorie (zur Problem-behebung gedacht)




Quelle: [BKPS04]



Eine Konfiguration kann als Vektor verstanden werden, der auf verschiedene optionale oder alternative Komponenten verweist (BOOLEAN oder XOR). Sie entspricht einem Produkt, das aus den selektierten Komponenten aufgebaut ist und beschreibt damit eine Ableitung aus der Produktlinienplattform.

Ein Konfigurationsmanagementsystem erfasst und verwaltet Versionen von Komponentenimplementierungen. Dazu speichert es jedes Artefakt in einem Repository und dokumentiert die Revisionshistorie im zeitlichen Verlauf.

Konfigurationsmanagementsystem ermöglichen darüber hinaus die Verwaltung von Konfigurationen aus den Komponenten.

## Variabilität durch Zweige (Branches)

- Verzweigungen zur Realisierung von alternativen Implementierungen einzelner Komponenten:
  - Zweige laufen nicht mehr aus
  - Inkompatible Implementierungen entstehen (Rückführung in Hauptstrang nicht mehr möglich)
  - Anzahl der Zweige wächst ständig an
  - Stetig wachsende Zahl von zu pflegenden Implementierungen

➔ Wiederverwendung mit möglichst geringem Wissen über Details der SPL-Plattform ist nicht mehr möglich!



Prinzipiell besteht die Möglichkeit Verzweigungen in der Versionsverwaltung zur Realisierung von alternativen Komponentenimplementierungen zu verwenden. Doch dieses Vorgehen für zu Problemen.

Die Zweige laufen nicht mehr aus, weil inkompatible Implementierungen entstanden sind (ein Zusammenführen der Alternativen war nicht das Ziel). Dadurch wächst die Anzahl der offenen Zweige ständig an, was zu einem hohen Verwaltungsaufwand führt.

Wiederverwendung ist durch dieses Vorgehen nur noch mit großem Wissen über die Interna der Produktlinie möglich. Daher sollte dieses Vorgehen vermieden werden!

## Generizität

- Implementierung der Variabilität in den Komponenten mit geeigneten Mechanismen
- Umsetzung von Variabilitätsarten, die nicht mit Konfigurationsmanagement umsetzbar sind (beispielsweise Variation in der Qualität)
- Vertikale Codefragmente:
  Zerteilung der Variabilität entlang der Struktur



Unter Verwendung von Konfigurationsmanagement im Zusammenhang mit einer Komponenten-basierten Architektur lassen sich BOOLEAN und XOR Variabilitätspunkte realisieren, die mit einer Aufteilung der Varianten auf Komponenten vereinbar sind.

Um in den einzelnen Subsystemen und Komponenten Variabilitätspunkte zu realisieren, die sich nicht durch Konfigurationsmanagement abbilden lassen, werden generische Ansätze benötigt.

Vertikale Codefragmente zerteilen die Variabilität dabei entlang der Struktur. Sie lassen sich mit verschiedenen Variabilitätsmechanismen erreichen.

# Agenda

- Variabilität in der Produktlinienentwicklung
- Technologiedimensionen
- **Vertikale Variabilitätsmechanismen**
  - Bedingte Übersetzung
  - Bedingte Ausführung
  - Subtyp-Polymorphismus
  - Parametrisierter Polymorphismus
  - Frames
  - Dynamic Class Loading
- Horizontale Variabilitätsmechanismen
- Auswahl und Bewertung von geeigneten Mechanismen
- Fazit

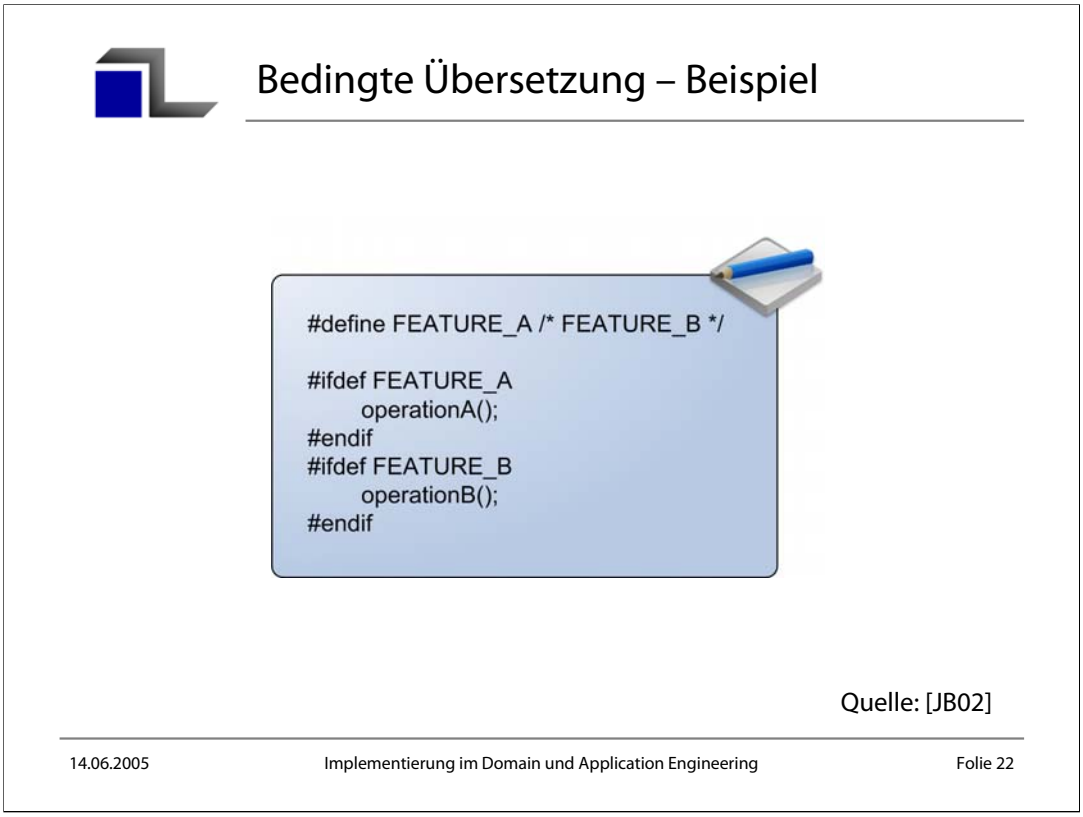
## Bedingte Übersetzung

- Verwendung von Präprozessordirektiven
- z.B. Makros im C/C++ Präprozessor
  - Definition von optionalen oder alternativen Codestücken
  - **`#define`**
  - **`#ifdef/#ifndef`**

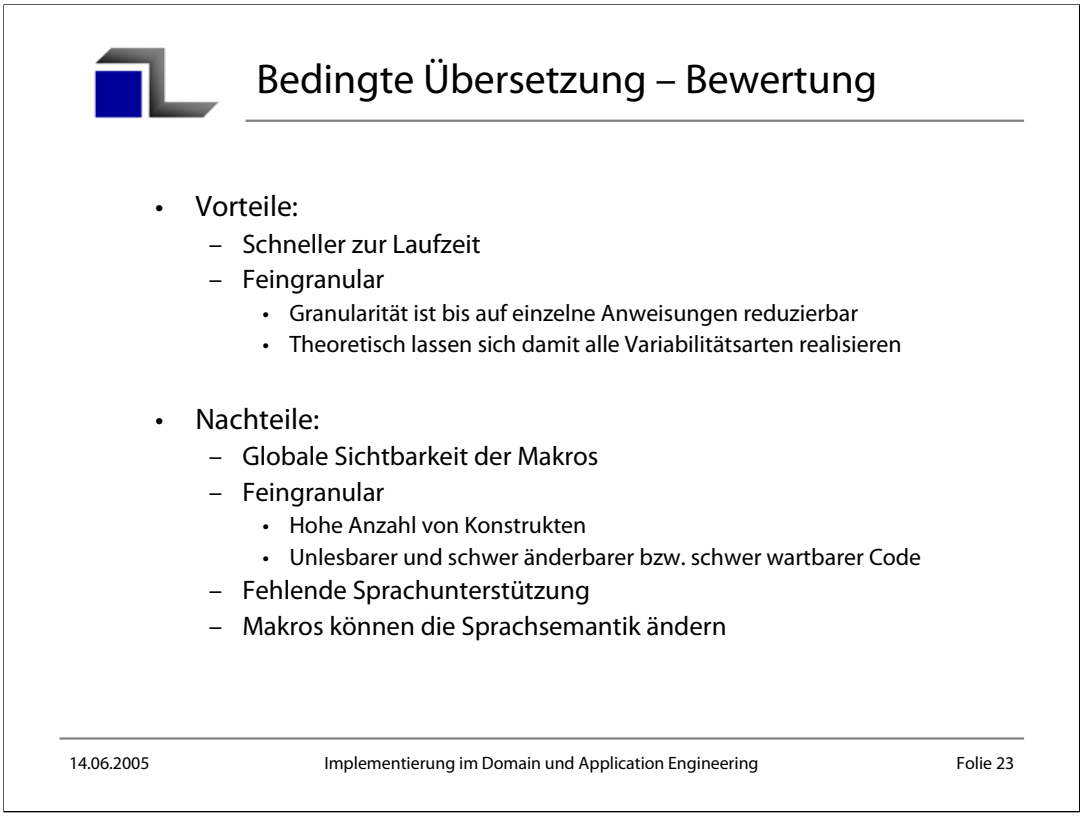


Die bedingte Übersetzung ist eines der einfachsten, statischen Modelle zur Implementierung von Variabilitätspunkten. Sie wird mittels eines Präprozessors ermöglicht.

Vgl. auch [Spi02], [JB02].




Im obigen Beispiel wird in Abhängigkeit zur symbolischen Konstanten *FEATURE_A* bzw. *FEATURE_B* der jeweilige Codeblock durch den Präprozessor selektiert und im Anschluss entsprechende die jeweilige Methode *operationA*() bzw. *operationB*() kompiliert.



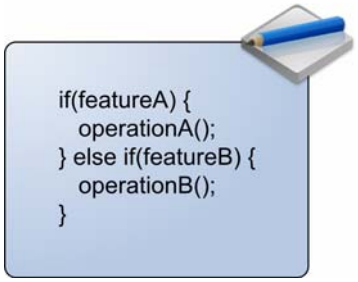

Diese Art der Variabilität wird bereits zur Übersetzungszeit aufgelöst, weshalb kein Overhead zur Laufzeit entsteht und das Binary schlank und schnell bleibt.

Besondere Vorsicht bedarf es aufgrund der fehlenden Sprachunterstützung von Makros, sie können z.B. die Sprachsemantik ändern indem sie  das Schlüsselworte private in public umdeklarieren. Tippfehler in Variablennamen führen schnell zu permanentem Auslassen ganzer Codesegmente. Fehlende Sprachunterstützung erschwert jedoch das Auffinden dieser Bugs.

Die Verwendung von *bedingter Übersetzung* mittels #ifdef, #ifndef,… ist in Verbindung mit Software-Produktlinien eher als Problematisch anzusehen, da bei Änderung bzw. beim Hinzufügen neuer Varianten alle betroffenen Klassen bzw. Module lokalisiert und angepasst werden müssen. Die Variabilitätspunkte sind so über den gesamten Code zerstreut, was Wartbarkeit und Codequalität mindert.

## Bedingte Ausführung

- Abarbeitung abhängig von Parameterwerten
- Ausführung von unterschiedlichen Anweisungen






„In Abhängigkeit von Parameterwerten werden bei der Abarbeitung eines Programms unterschiedliche Anweisungen ausgeführt“ [Bec04]

Bedingte Ausführung ist somit das Pendant zur bedingten Übersetzung mit dem Unterschied, dass die Variabilität erst zur Laufzeit aufgelöst wird.

Da Entscheidungen Bestandteil jeder heutigen Programmiersprache sein sollten, kann auch das Prinzip der bedingten Ausführung für die Implementierung von Variabilität in all diesen Sprachen eingesetzt werden

## Bedingte Ausführung – Bewertung

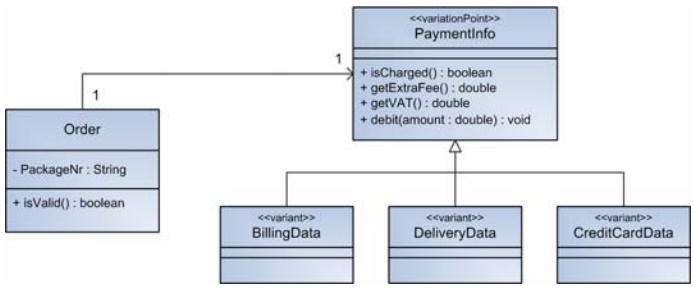
- Vorteile:
  - Flexibilität zur Laufzeit
    (z.B. durch Konfigurationsdateien)
- Nachteile:
  - Overhead
  - Performance
  - Unsicher
    (wenn Features freigeschaltet werden können)



Während die späte Auflösung der Variabilität als Vorteil zu bewerten ist, ist der damit verbundene Overhead an Code (alle Variabilitäten müssen im Binary enthalten sein), die negativen Performanceauswirkungen und die Gefahr das Features (ungewollt) freigeschaltet werden können eher Nachteilig zu bewerten.

## Subtyp-Polymorphismus

- Wird von OO-Sprachen implementiert
  - Zugriff erfolgt dynamisch über Oberklasse
  - Konkretes Verhalten wird in Unterklassen definiert




Quelle: [Cla01]



Subtyp- oder auch Untertyp-Polymorphismus meint die Möglichkeit, Variabilität über Vererbung auszudrücken. In C++ geschieht dies durch die Ableitung von Basisklassen, welche einfach leere bzw. Standardmethoden und Attribute implementieren. In Java kann dies mit Hilfe von Interfaces elegant realisiert werden.

In Abhängigkeit vom abgeleiteten Objekt wird eine bestimmte Funktionalität bereitgestellt (implementiert) oder nicht. Der Zugriff erfolgt über die, alle Varianten beschreibende, Basisklasse bzw. das Interface.

Im obigen Beispiel wird eine Basisklasse mit Defaultverhalten implementiert, d.h. sie gibt Verhalten vor, indem sie leere Methoden implementiert.

Dies macht Sinn, da eine von einer abstrakte Klassen abgeleitete Klasse zwingend alle vorgegebenen Methoden implementieren müsste, auch wenn dies zur Folge hat, dass die für die konkrete Ausprägung „leer“ sind. Dies hat zur Folge, dass wenn einer Basisklasse nachträglich Methoden hinzugefügt werden, auch alle von Ihr abgeleite Klassen angepasst werden müssen.

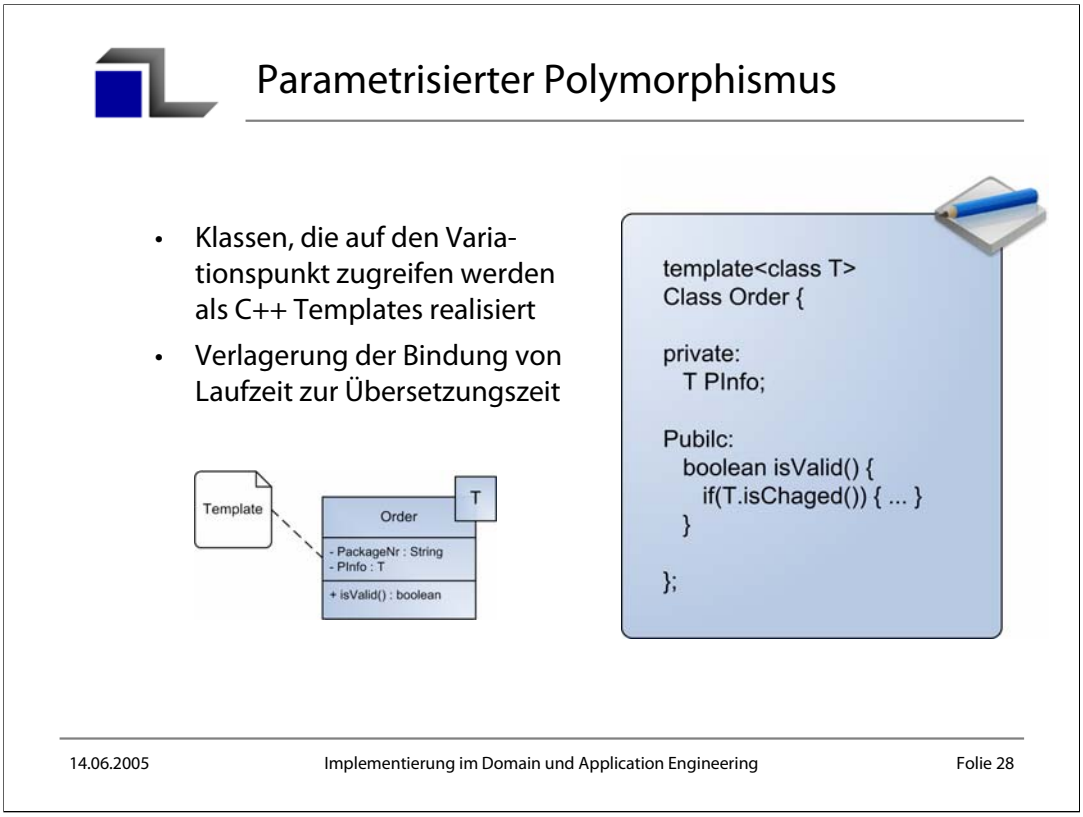
## Subtyp-Polymorphismus – Bewertung

- Vorteile:
  - Abbildung von XOR-Variabilitätstypen
    - Jede Subklasse eine XOR-Variante
    - Kapselung von Features
- Nachteile:
  - Negativer Einfluss auf die Performance durch die Laufzeitbindung
  - OR-Variabilitätstypen können nicht klar ausgedrückt werden



Da jede Subklasse in der Regel eine eigene Variante repräsentiert, kann nur eine Variabilität auf einmal hinzugefügt werden. Mit anderen Worten bedeutet das, dass OR-Variabilitätstypen mittels Subtype-Polymorphismus nicht klar ausgedrückt werden können. Vgl. [MP02a]

Umgekehrt kann Laufzeitbindung explizit erwünscht bzw. vom Design vorgegeben sein, sodass man gezwungen ist ggf. verschachtelte Konstrukte zu bilden um eine solchen Typ von Variabilität abzubilden.




Da Templates ein sprachspezifisches Konstrukt sind, existiert für sie keine einheitliche UML Notation. Ein verbreiteter Ansatz Templates trotzdem darzustellen zu können, ist die Darstellung als normale Klasse mit Hervorhebung des Typparameters wie oben dargestellt.

In Anlehnung an das vorangegangene Beispiel wird hier die Klasse *Order* mit einem Typparameter versehen und entsprechend des übergebenen Subtyps die Methode *isCharged*() z.B. von der Klasse *CreditCardData* aufgerufen.

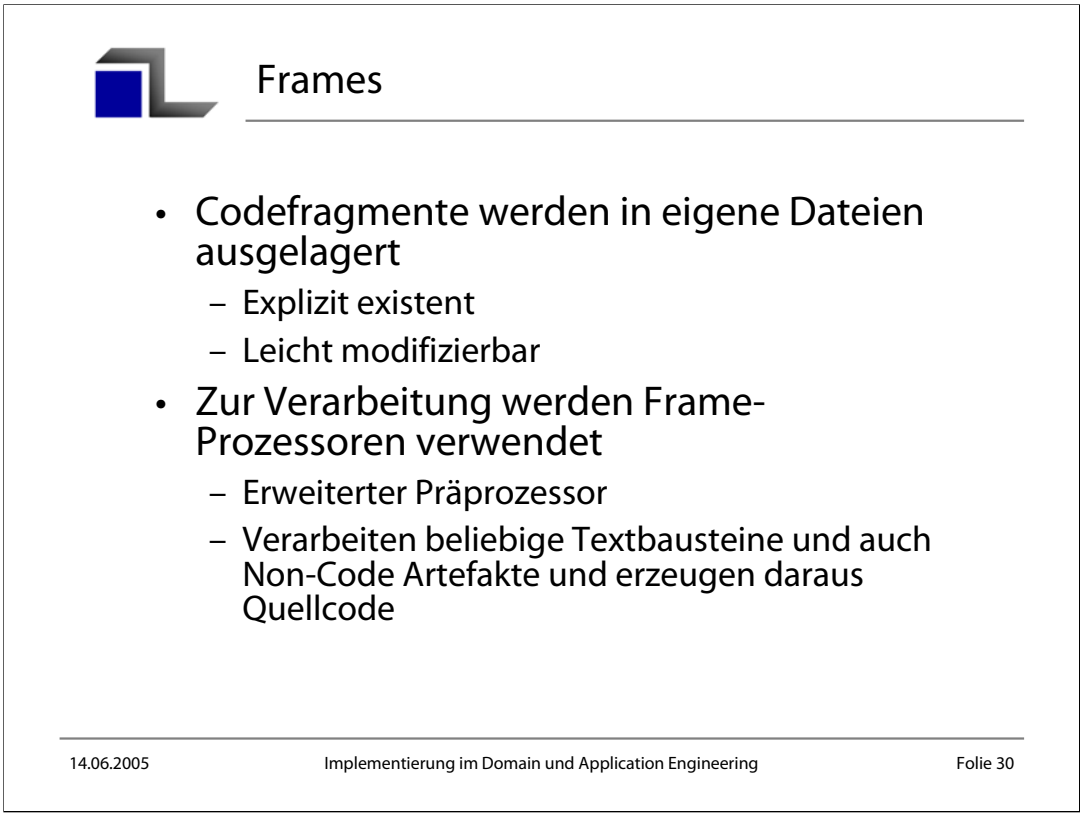
## Parametrisierter Polymorphismus – Bewertung

- Vorteile:
  - Abbildung von XOR-Variabilitätstypen
  - Bindung zur Übersetzungszeit
  - Sprachunterstützung
    - Typprüfung zur Übersetzungszeit
- Nachteile:
  - OR-Variabilitätstypen können nicht klar ausgedrückt werden



Neben Subtyp- und parametrisiertem Polymorphismus existieren noch weitere Polymorphismus-Typen, die an dieser Stelle jedoch nicht weiter behandelt werden (Vgl. hierzu [MP02a], S. 7 und [MP02b], S.21).

Abschließend sei darauf verwiesen, dass mit Java Tiger (5.0) nun auch generische Templates in Java verwendet werden können.




Aufbau eines Frames (ANGIE, XFramer):

Deklaration von Frame X

Deklaration von Frame-Variable a mit Vorgabewert

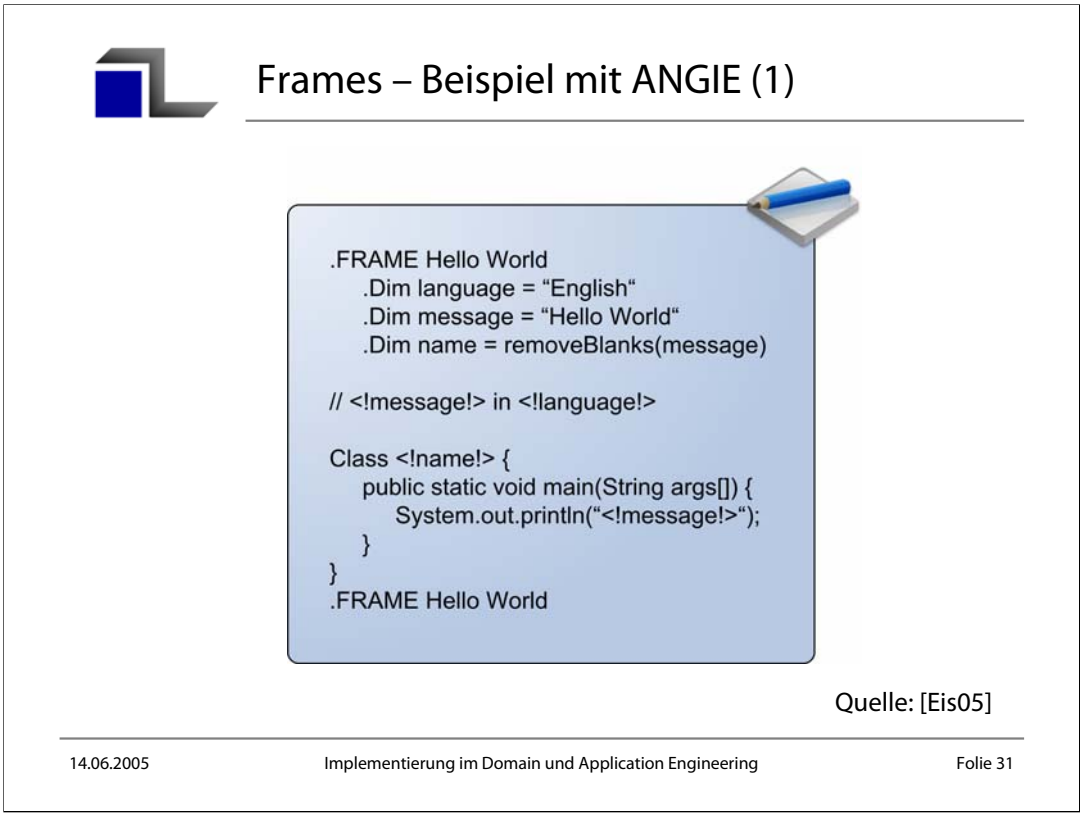
Deklaration von Frame-Variable b mit Vorgabewert

Frame-Text (auch *Host-Code* genannt) mit Referenzen auf Frame-Variablen wie a oder b. Eine Referenz (*Slot*) auf eine Frame-Variable, z.B. a, wird später durch den aktuellen Inhalt der Frame-Variable a ersetzt

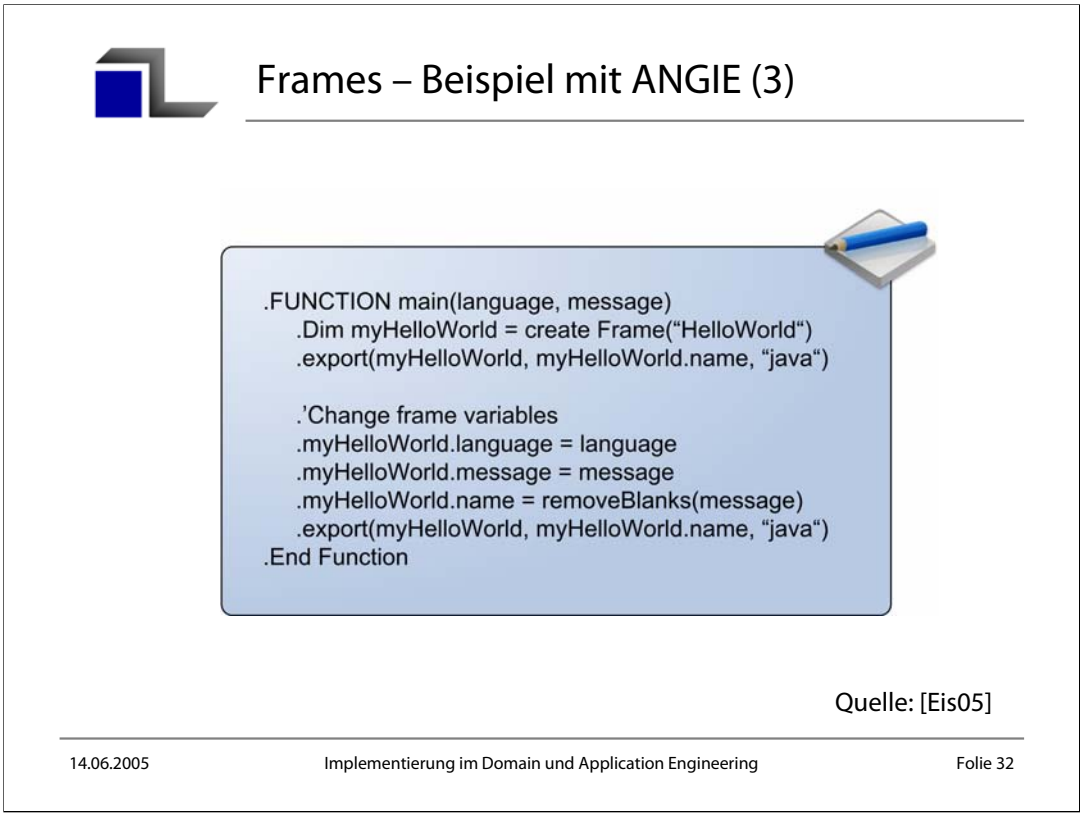
Ende des Frames

Frame-Instanz-Variablen können andere Frames bzw. Frame-Instanzen referenzieren. Diese Referenzen bilden einen gerichteten zyklenfreien Graphen, dessen Scheitelpunkt „Spezifikationsframe“ oder „Root-Frame“ genannt wird. Während des Exports werden alle Referenzen auf Frames bzw. Frame-Instanzen aufgelöst. Mehrfach referenzierte Frames werden zu entsprechend vielen individuelle Kopien.

Quelle: [Eis05] und [MP02a]




Im obigem Beispiel wird ein Frame „*Hello World*“ beschrieben, welcher die Frame-Variablen *language*, *message* und *name* und ihre default-Werte definiert. *name* referneziert dabei auf die Frame-Funktion *removeBlanks*(). Diese hat lediglich veranschaulichenden Charakter und soll hier nicht weitergehend beschrieben werden. Darunter findet sich der eigentliche Frame-Text, welcher später in eine konkrete Klasse und deren *main*-Methode abbildet wird.




Die Funktion *main()* hat die Aufgabe unseren Beispiel-Frame zu instanziieren und als Quellcode zu exportieren. Ähnlich wie bei dem vorangegangenem Frame beginnt ihre Definition mit *.Function* und schließt mit *.End Function*.

Den eigentliche Frame erzeugt dann der Aufruf der Funktion *createFrame*(). Sie ist Bestandteil des Frame-Prozessors und wird daher als intrinsische Funktion bezeichnet. Ihr erster Parameter ist der Name des Frames, alle ggf. folgenden Parameter übergibt die Funktion dem Frame bei seiner Instanziierung.

Die intrinsische Funktion *export()* sorgt dafür, dass der Inhalt der Frame-Instanz in der Datei *HelloWorld.java* abgelegt wird, wobei der Klassenname als zweiter und das Suffix separat als dritten Parameter übergeben wird.

Im konkretem Beispiel erzeugen wir demnach zunächst eine Frame-Instanz mit Namen *myHelloWorld* und exportieren dann die Klasse *HelloWorld.java* unter Verwendung der default-Werte. Anschließend ändern wir zur Veranschaulichung die Attribute der Frame-Instanz über den Instanznamen und erzeugen durch erneuten Aufruf der Methode *export*() eine weitere Klasse (der Name häng vom Parameter *message* ab).

## Frames – Bewertung

- Vorteile:
  - Hohe Lokalität (ein Frame kann mehr Variabilität aufnehmen als etwa ein Klassentemplate)
  - Schnelle Verarbeitung der Frames
  - Debugging möglich (für manche Frame-Prozessoren)
  - Auch Generierung von None-Code-Artefakten
- Nachteile:
  - Keine Information über Host-Code (etwa Typen)



Die Vorzüge von Frames lassen sich z.B. im Vergleich mit Makros und Templates sehr gut verdeutlichen. Während Makrodefinitionen auf eine Zeile beschränkt sind und man mit Fortsetzungszeichen arbeiten muss, kann man mit Frames auf elegante Wiese lesbare und verständlichste Konstrukte (Frames) bilden. Zudem lässt sich der erzeugte Code problemlos mit dem Debugger untersuchen. [ES02]

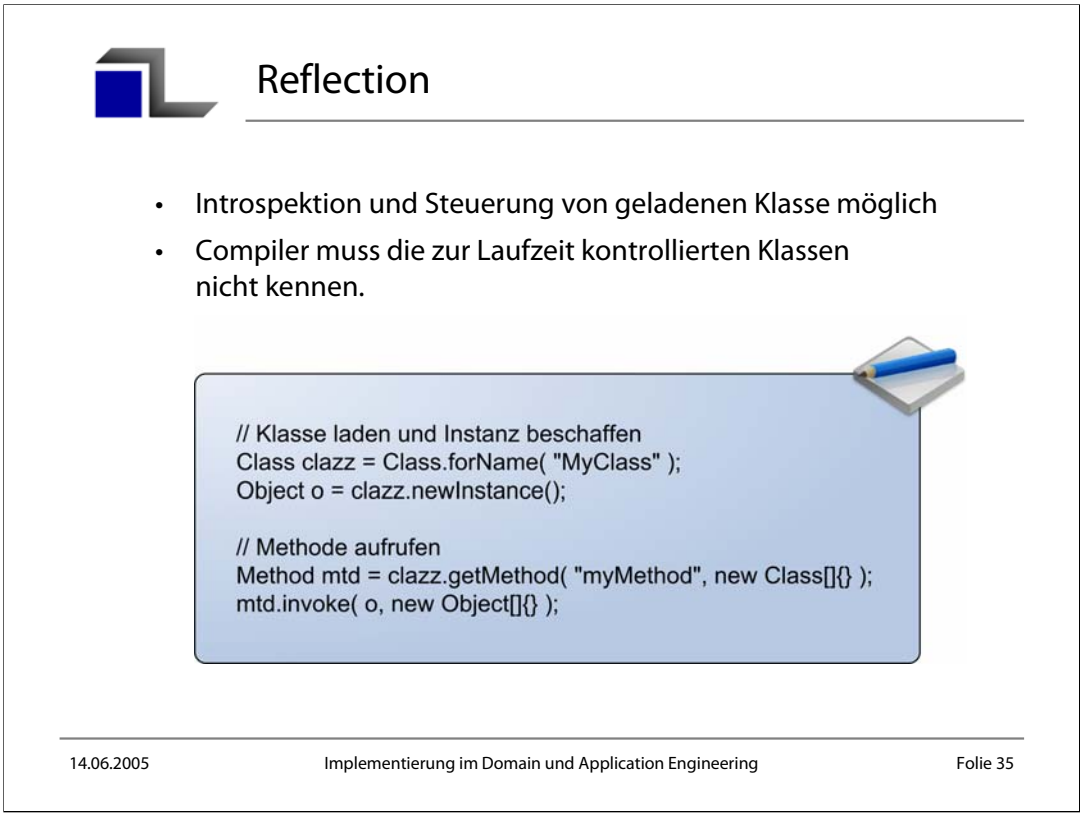
## Dynamic Class Loading

- Dynamisches Laden von Klassen zur Laufzeit
  - Normalfall in Java, Klassen werden „nach Bedarf“ geladen
  - Berücksichtigung des Laufzeitkontextes beim Laden möglich (z.B. für Portabilität)
  - Definition der Varianten ist zeitlich unabhängig von der des zugehörigen Variationspunktes
- Dynamic Class Loading kann mit Reflection kombiniert werden



Beim Dynamic Class Loading werden Klassen erst dann geladen, wenn sie von der Laufzeitumgebung tatsächlich benötigt werden. Dieses Vorgehen ist bei Java der Standard. Eine Klasse kann jedoch auch explizit von einer laufenden Anwendung selbst über einen entsprechenden ClassLoader geladen werden.

Für Produktlinien eröffnet Dynamic Class Loading die Möglichkeit zur Laufzeit unterschiedliche Klassen zu laden. Dabei kann zunächst der Laufzeitkontext bestimmt werden. So können je nach Kontext beispielsweise unterschiedliche Klassen für den Datenbankzugriff geladen werden




Das Reflection-Konzept ermöglicht es einem System geladene Klassen zu untersuchen und zu steuern oder auf ihren eigenen Zustand zugreifen und sich so zu modifizieren (vgl. [Bec04]). Die Struktur dieser Klassen muss dabei nicht bekannt sein.

Eine Kombination aus Dynamic Class Loading und Reflection ermöglich das dynamische Laden, untersuchen von Klassen. Die Erzeugung von Instanzen und deren Steuerung ist ebenfalls möglich.

Beim Einsatz in Produktlinien lassen sich auf diese Art sehr flexible Plug-in Mechanismen entwerfen, sie das Hinzufügen von Funktionalität auch nach der Auslieferung erlaubt.

Das Beispiel zeigt das Zusammenspiel von Dynamic Class Loading und Reflection in Java. Die Klasse *MyClass* wird geladen und eine Instanz erzeugt. Danach wird per Reflection die Methode *myMethod()* ermittelt und aufgerufen.

Dabei wird der Methode *invoke*() als erster Parameter das Objekt übergeben, auf dem die Methode ausgeführt werden soll. Der zweite Parameter ist ein Object-Array, der die Argumenten der aufzurufenden Methode enthält. In unserem Beispiel wird ein leeres Array übergeben, da die Methode keine Parameter besitzt.

## Dynamic Class Loading und Reflection

- Vorteile:
  - Ermöglicht äußerst flexible Plug-in Mechanismen
  - Hinzufügen von (neuen) Varianten zur Laufzeit möglich
  - Variabilität in Klassen gekapselt
  - Auf Basis der durch Reflection gewonnen Informationen kann Variabilität zur Laufzeit gesteuert werden.
    (z.B. Variabilität im Datenumfang)
- Nachteile:
  - Negative Auswirkungen auf Performance
    (bei Dynamic Class Loading sprachabhängig)
  - Reflection führt zu schlechterer Verständlichkeit und erschwert das Debugging

## Generizität

- *Problem*: Bestimmte Variabilitäten lassen sich nicht lokal mit wenigen Codefragmenten darstellen
  - Ziehen sich durch das ganze System
  - Schwer zu erfassen
  - Schwer zu ändern
- *Lösung*: Variabilitäten werden in horizontalen Codefragmenten zusammengefasst
  - Zusammenziehen der horizontal verteilt codierten Codefragmente
  - Lokale Isolierung von eigentlich verteilter Varabilität



In den vielen Fällen betrifft Variabilität mehrere Stellen im Code zugleich. Die Gesamtheit eines Variabilitätspunktes ist of nur schwer zu erfassen, da keine explizite Struktur vorliegt. Beispiele hierfür sind Variation in Qualität: z.B. Sicherheit, Performance und Wartbarkeit.

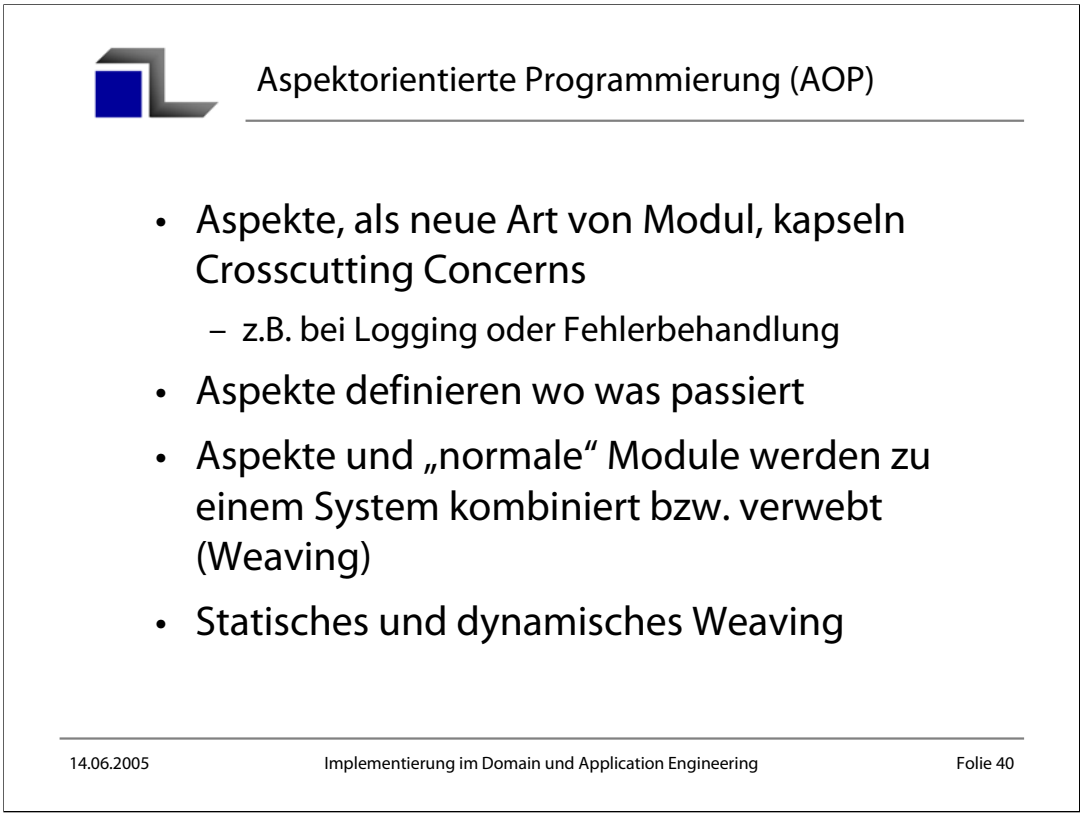
Für Änderungen müssen alle Codefragmente gefunden und die Konsistenz der Codeänderungen muss sicher gestellt werden.

# Agenda

- Variabilität in der Produktlinienentwicklung
- Technologiedimensionen
- Vertikale Variabilitätsmechanismen
- **Horizontale Variabilitätsmechanismen**
  - Refinements
  - Aspektorientierte Programmierung
  - Kollaborations-basierte Mechanismen
- Auswahl und Bewertung von geeigneten Mechanismen
- Fazit

## Refinements

- Verschiedene Post-OO-Ansätze
- Variabilität wird in einem eigenen Modell abgebildet (losgelöst von Klassenhierarchien)
- Fragmente werden als *Komponenten* gekapselt
- Refinements repräsentieren jeweils zusätzliche Funktionalität, die mehrere Implementierungseinheiten betrifft.
- Zwei Unterkategorien:
  - Kollaborations-basierte Mechanismen
  - Aspektorientierte Programmierung (AOP)




Ziel der Aspektorientierung ist die Verbesserung der Wartbarkeit und Widerverwendbarkeit. Mit aspektorientierten Techniken wird versucht, modulübergreifende Merkmale – so genannte *crosscutting concerns* - zu modularisieren. (Vgl. [Sic04])

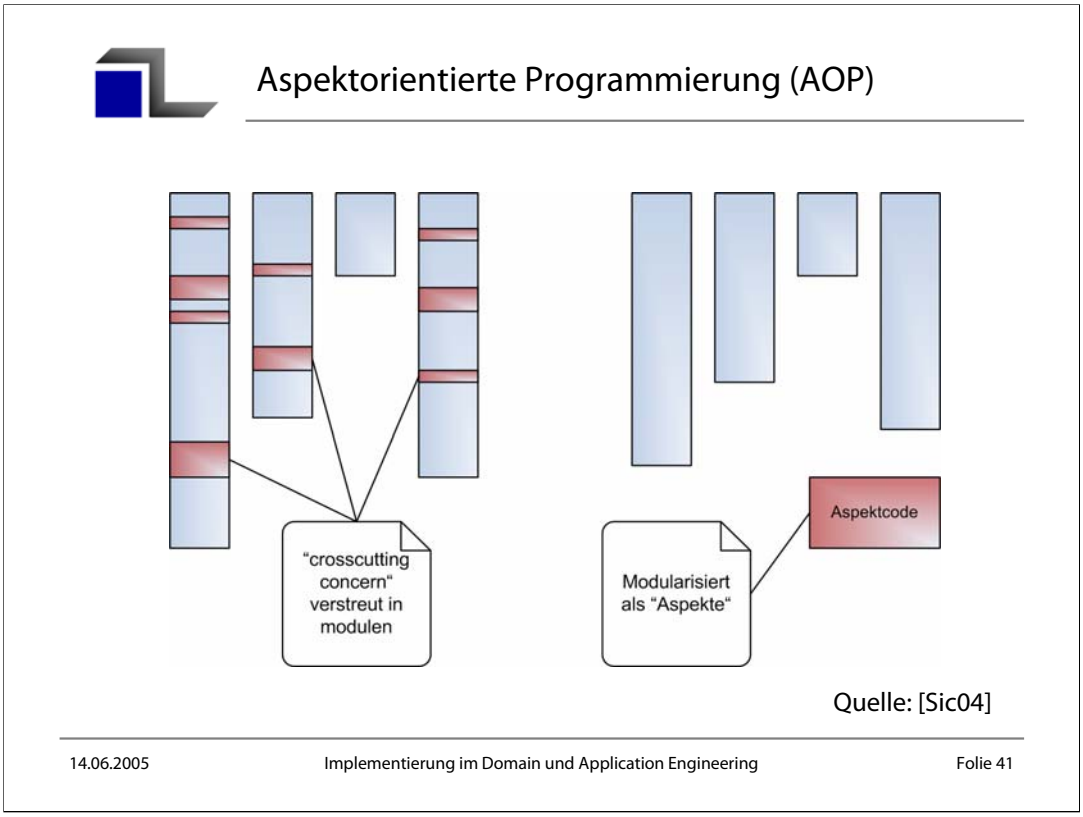
Mit *crosscutting concerns* (auch *querschneidenden Modellen)* sind unabhängige Modelle gemeint, die unterschiedliche, sich überschneidende Sichten auf ein gemeinsames System repräsentieren und in einer gegebenen Modulstruktur (oder simultan mit anderen Aspekten im Programmtext) nicht lokalisiert werden können. (Vgl. [Ost03])

Neben dem statischen Codeweaving (zur Übersetzungszeit) existieren in der Java-Welt auch noch drei weitere Ansätze welche ein dynamisches Codeweaving ermöglichen:

• Load-time weaving

• Just in Time (JIT) compiler weaving

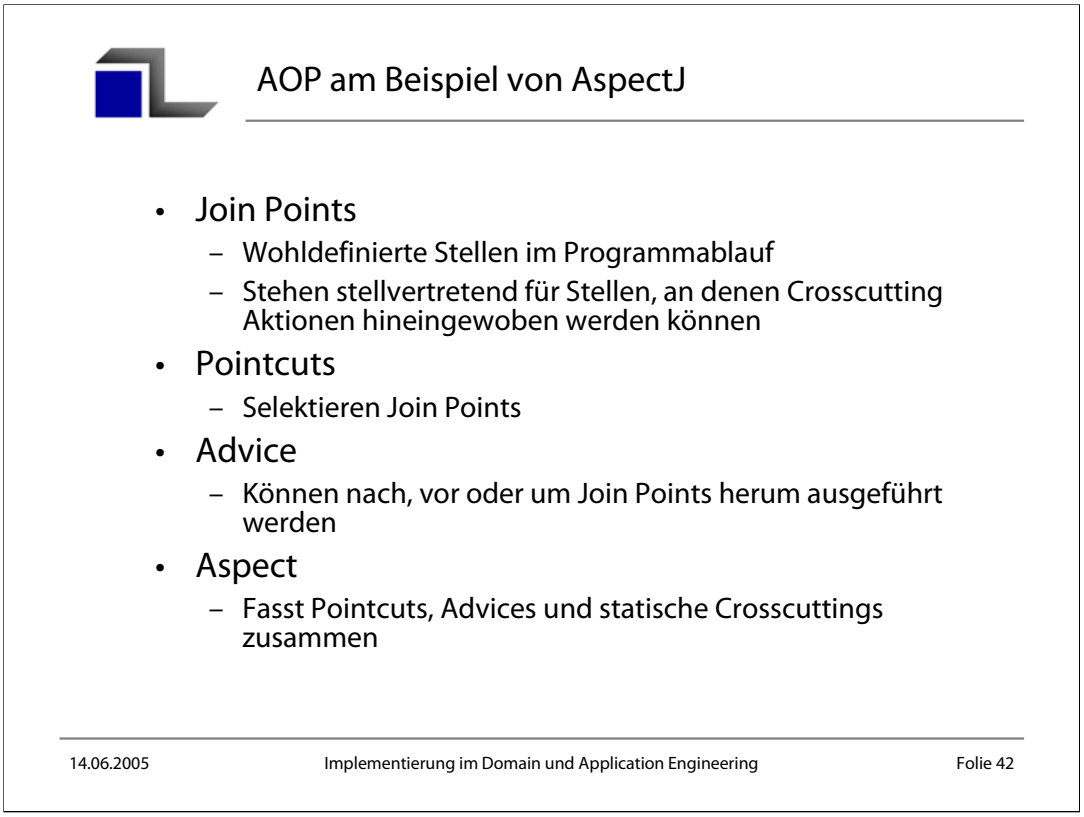
• Dynamic proxies.



## Aspektorientierte Programmierung (AOP)



Quelle: [Sic04]




AspectJ ist eine AOP Umsetzung für Java, die von Eclipse mittels der entsprechenden Plug-ins unterstützt wird.

AspectJ definiert die folgenden Konstrukte:

**Join Points**

Join Points (JP) werden unterschieden in: Constructor JP (Call, Execution), Method JP (Call, Execution), Field Access JP (nur Klassenvariablen), Exception Handling JP, Class Initialization JP, Object Initialization JP, Object Pre-Initialization JP, Advice Execution JP.

Join Points an Schleifen und anderen Kontrollstrukturen werden nicht unterstützt.

**Pointcuts**

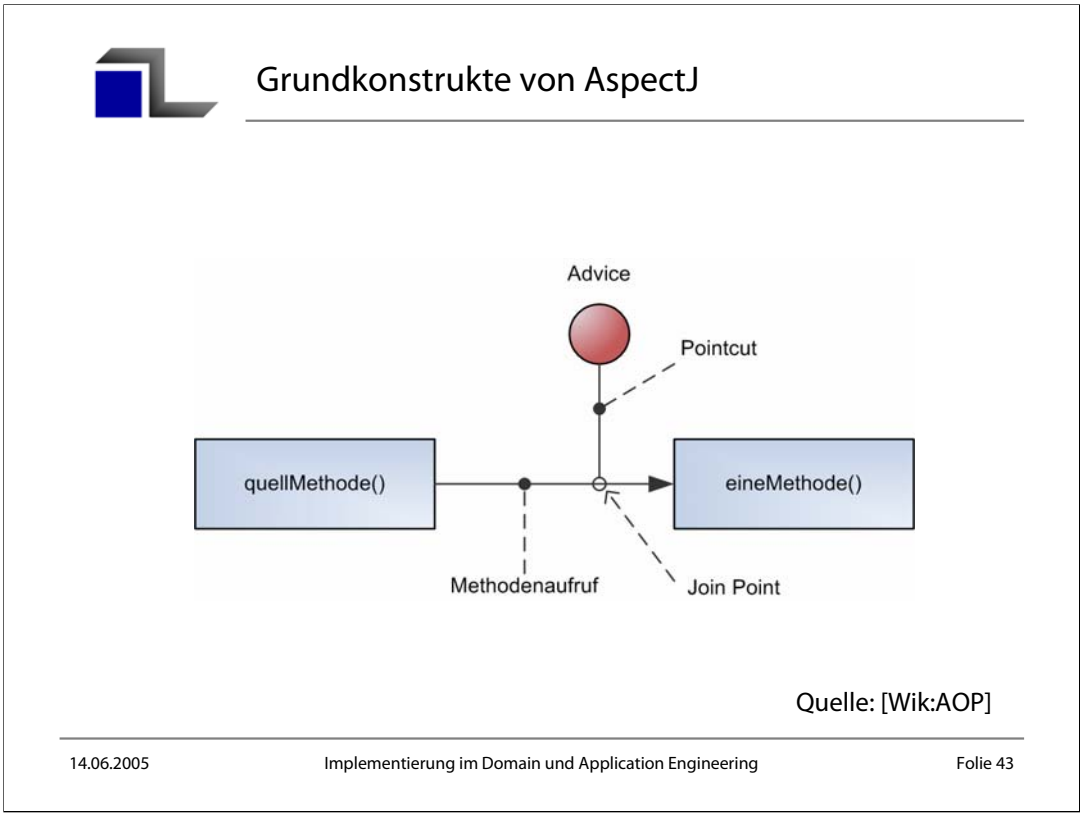
Pointcuts (PC) werden unterschieden in PC mit direkter Referenzierung (auch als Kinded Pointcuts bezeichnet) und PC mit indirekter Referenzierung (eingeleitet durch *cflow* oder *cflowbelow*) .

Pointcuts unterstützen folgende Wildcarts:

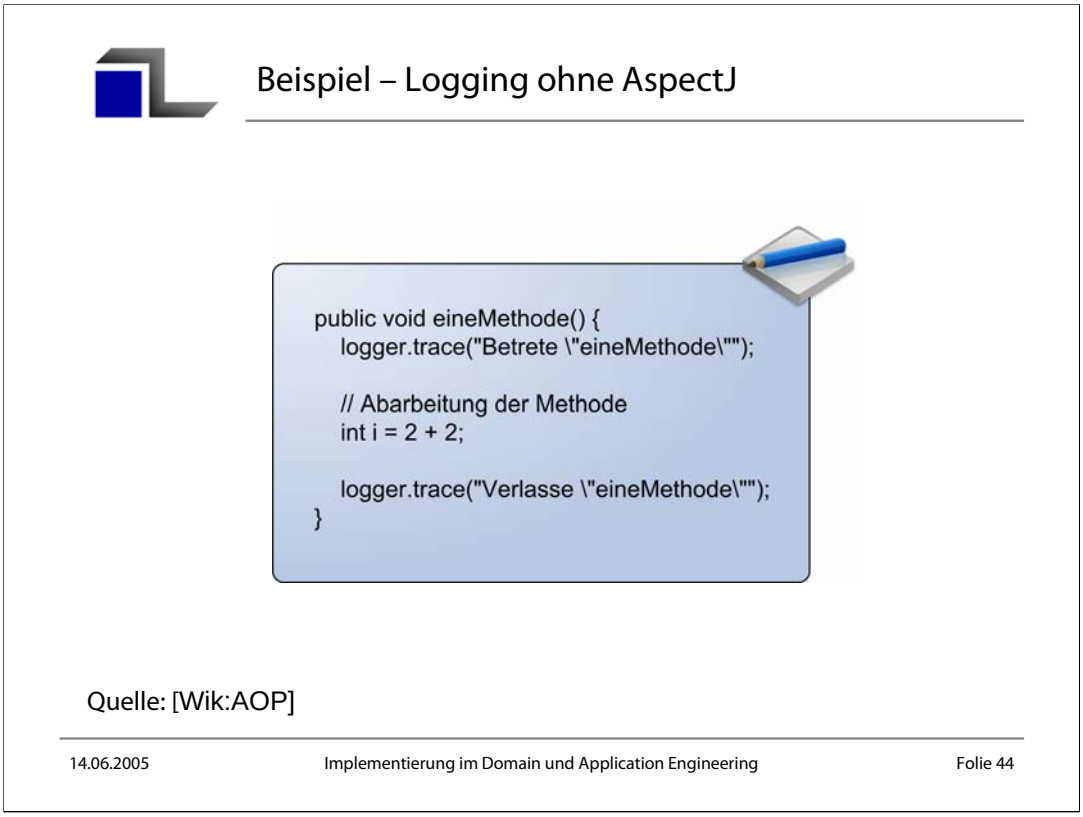
- ***** beliebige Anzahl von Zeichen außer Perioden durch **.** und **,**
- **..** beliebige Anzahl von Zeichen einschließlich Perioden mit **.** und **,**
- **+** jede erbende oder implementierende Klasse eines gegebenen Typs

**Advice**

Ein Advice definiert, was und wann (*before*, *after*, *around)* an einem Join Point genau geschehen soll. Mit *proceed kann die Abarbeitung der ursprünglichen Methode fortgesetzt werden*

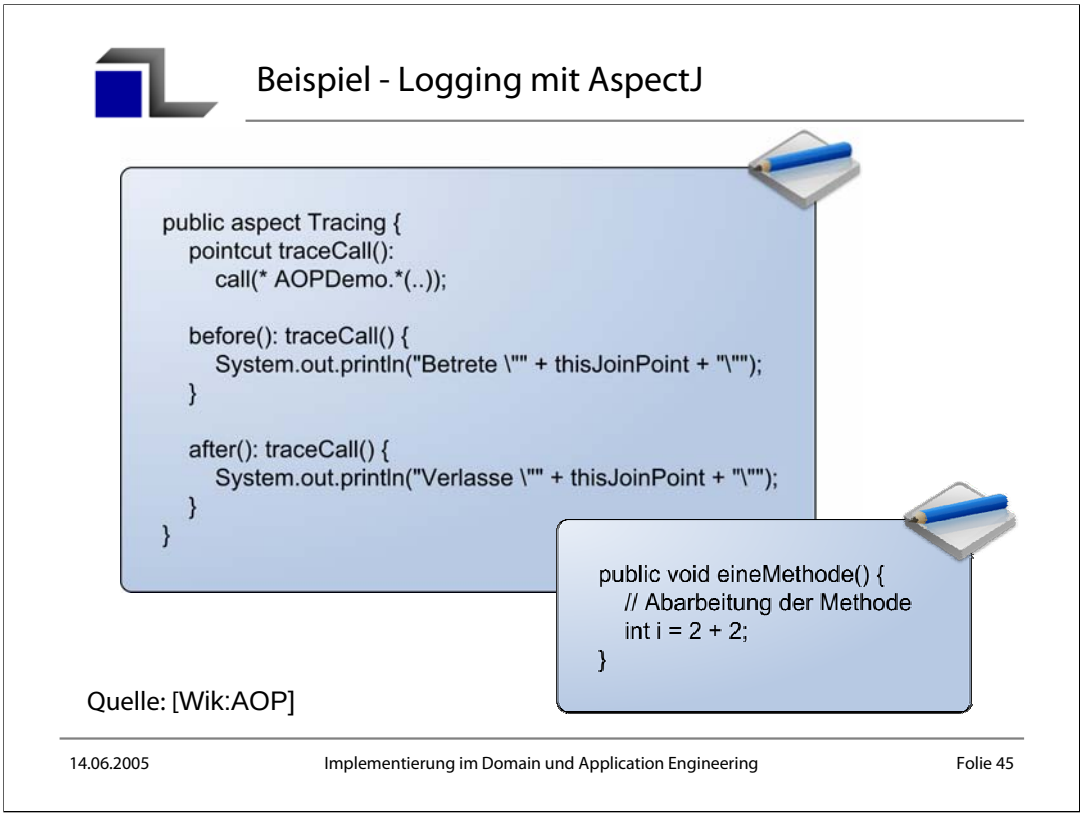



Ein Pointcut ist die Spezifikation der Codestelle (Join Point) an der eine Aktion (Advice) ausgeführt bzw. verwoben werden soll. Zusammengefasst wird dies ein Aspekt genannt. Ein Aspekt alleine erledigt noch nichts, er muss erst noch mit dem Quell-Programm, das die Join Points enthält, verwoben werden. Die geschieht durch Aufruf des erweiterten AspectJ-Compilers.




Mit obigem Beispiel soll gezeigt werden, wie mit herkömmlichen Mitteln ein Logging implementiert werden würde. Betrachtet man lediglich diese Methode mag ein Einsatz von AspectJ nicht gerechtfertigt sein.

Stellt man sich jedoch die Methode innerhalb eines Transaktionsprozesses vor, wo das Logging noch an vielen anderen Stellen implementiert werden muss und zusätzlich auch noch Mechanismen zur Sicherung und Validitätsprüfung hinzukommen, wird schnell ersichtlich, dass dieser Ansatz zu hochgradig redundantem, schwer lesbarem und unnötig aufgeblähtem Code führt. Zudem muss der Logger jedem Objekt bekannt gemacht werden, und kann nicht ohne weiteres an einer zentralen Stelle ausgetauscht werden.

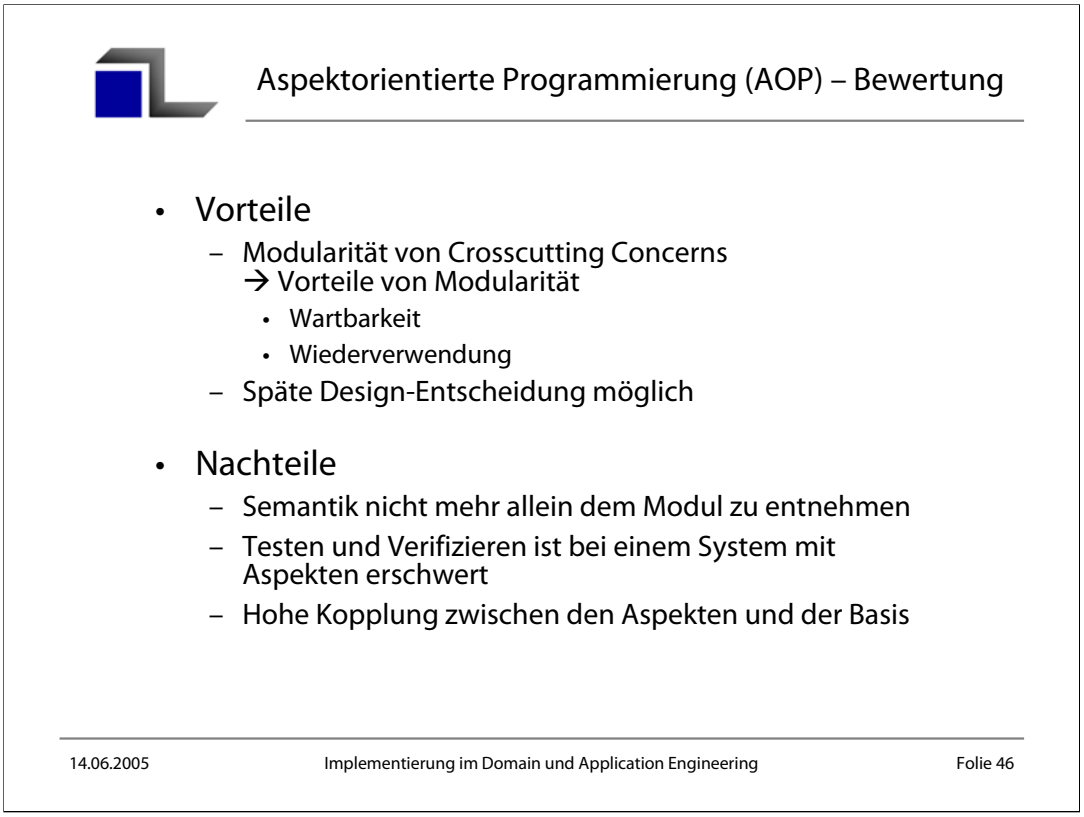



Das vorangegangene Beispiel hat gezeigt, dass eine verhältnismäßig einfache Klasse durch orthogonale Anforderungen sehr schnell und immer weiter aufgebläht wird. Mit Aspekten lassen sich die existierenden, funktionalen Abhängigkeiten separat von den orthogonalen, funktionalen Anforderungen beschreiben.

Für unser Beispiel heißt das, dass die ursprüngliche Methode unverändert bleiben kann und lediglich ihre funktionale Anforderung (hier die Addition von 2+2) implementiert.

Separat davon wird in einem Aspekt ein Pointcut *traceCall*() definiert, welcher auf alle Methoden der Classe *AOPDemo* abzielt, also jede Methode der Klasse als Join Point beschreibt. Die beiden Advices *before*():*traceCall*() und *after*():*traceCall*() fügen dann das Logging vor und nach jedem Aufruf der Methode ein.

Damit wir später wissen, woher das Logging stammt, fügen wir noch den Namen des aktuell bearbeitet Join Points dem Logging hinzu, welcher uns durch die Variable *thisJoinPoint* zur Verfügung steht.




Das hier vorgestellte Beispiel ist im Vergleich zu den Möglichkeiten, welche sich durch den Einsatz AspectJ eröffnen, vergleichsweise simpel. Um einen Anreiz zu geben, sich evtl. noch mehr mit AspectJ zu beschäftigen, sei abschließend noch ein sehr interessantes Beispiel zur Verwendung der *proceed*()-Methode gegeben.

Das folgende Codebeispiel macht sich zu Nutze, dass ein Advice durchaus mehrmals die *proceed*()-Methode aufgerufen kann um so eine fehlertolerante Codeimplementierung zu ermöglichen:

```
Object around():call(public * SomeClassWithBuggyMethods.*(..)){
        Object toreturn=null;
        try{
                toreturn=proceed();
        }catch(Exception e){
                repair(e);
                toreturn=proceed();
        }
        return toreturn;
}
```

*(Vgl. auch* [Sch03])

## Kollaborations-basierte Mechanismen

- Zusammenarbeit von Objekten steht im Mittelpunkt (Kollaborationen)
- Kollaboration
  - Ist eine Sicht auf einen OO-Entwurf
  - Realisiert eine funktionale Einheit (ein Feature oder einen Dienst)
  - Besitzt ein Protokoll für die Zusammenarbeit von Objekten
- Objektfragmente nehmen Rollen in einer Kollaborationen ein



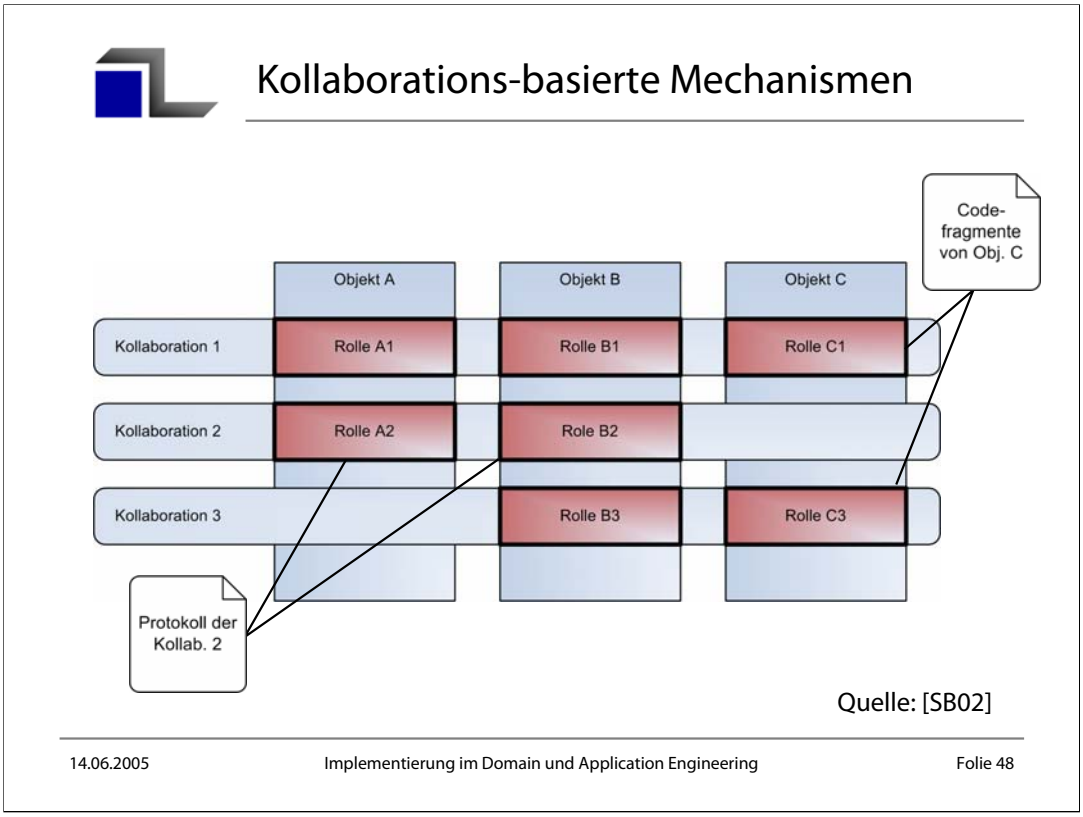
Mitglieder einer Software-Produktlinie sollen aus einer gemeinsamen Basis abgeleitet werden. Eine Umsetzung kann durch schrittweise Verfeinerung erfolgen.

Eine Kollaboration von Objekten implementiert genau ein Merkmal. Merkmale bilden als Schichten von Kollaborationen die gewünschte Software. Beginnend mit einer minimalen Basis wird durch Hinzufügen von Schichten die Funktionalität inkrementell erweitert.

Da für eine Kollaboration zwischen Objekten oft nur ein Teil einer Objektschnittstelle genutzt wird, besteht das Protokoll üblicherweise nur aus einer Untermenge der von den Objekten angebotenen Schnittstellen. Der Teil eines Objektes (das Objektfragment), der innerhalb einer Kollaboration für die Umsetzung des Protokolls verantwortlich ist, wird Rolle eines Objektes genannt.

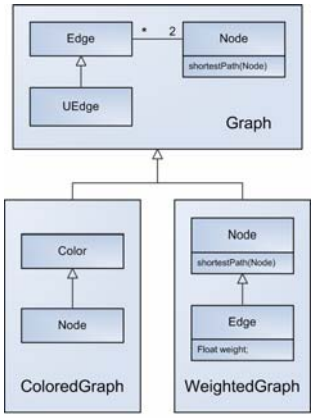
Durch ein einfaches Hinzufügen, Austauschen und Weglassen einzelner Schichten wird Erweiterbarkeit, Wartbarkeit und Anpassbarkeit ermöglicht.

Ein Beispiel der Implementierung zur Übersetzungszeit sind die so genanten „Mixin-Layers“ nach VanHilst und Notkin auf Basis von GenVoca Grammatiken.

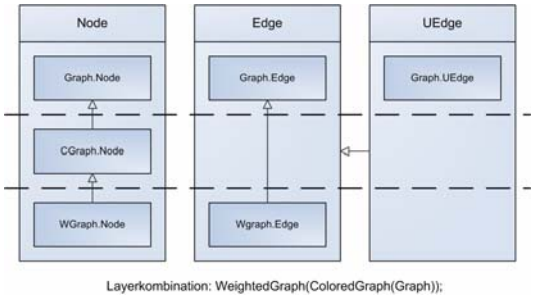



Obige Abbildung zeigt drei Kollaborationen. Objekte sind vertikal (Objekt A,B,C) und Kollaborationen horizontal (Kollaboration 1,2,3) angeordnet. Codefragmente der Objekte nehmen verschiedene Rollen in den Kollaborationen ein.

## Kollaborations-basierte Mechanismen – Beispiel




Kollaborations-Vererbung




Layerkombination: WeightedGraph(ColoredGraph(Graph));

Quellecode siehe Overhead!

Quelle: [Ost03]



Das Beispiel soll den Grundsätzlichen Ansatz verdeutlichen und zeigt eine Implementierung von Kollaboration unter Verwendung von „Mixin-Layers“ nach VanHilst und Notkin mit Hilfe von C++ Templates und inneren Klassen.

Ein Mixin ist eine Klasse, mit der man eine Erweiterung einer anderen Klasse implementiert, ohne dabei zu wissen, wie die zu erweiternde Basis aussieht. Die erweiterte Basisklasse ist ein Parameter bei der Mixin Definition.

Dieser Mechanismus kann zur Implementierung von Software-Produktlinien genutzt werden, wenn er mit dem Konzept der inneren Klassen (engl. inner classes) verknüpft wird. Mixins kapseln dann Fragmente verschiedener Klassen statisch, wodurch ein hoher Grad an Modularisierung ermöglicht wird.

Da von einer benutzten Schicht lediglich die Schnittstelle, aber nicht der Name bekannt sein muss, können Schichtimplementierungen entsprechend der Regeln der GenVoca Grammtik ausgetauscht werden.

(Vgl. [Spi02],[LA05])

Im Konkretem Beispiel erweitern die Mixins *ColoredGraph* und *WeightedGraph* die Basisklasse Graph um die Eigenschaften *color* bzw. *weight*. Die konkrete Ausprägung (Kollaboration) kann im folgenden durch eine *typdef*-Anweisung beschrieben werden, wobei gemäß der Klassenhierarchie (oder besser einer zuvor definierten GenVoca-Gramatik) die einzelnen Mixins gegeneinander ausgetauscht und/oder weggelassen werden können.

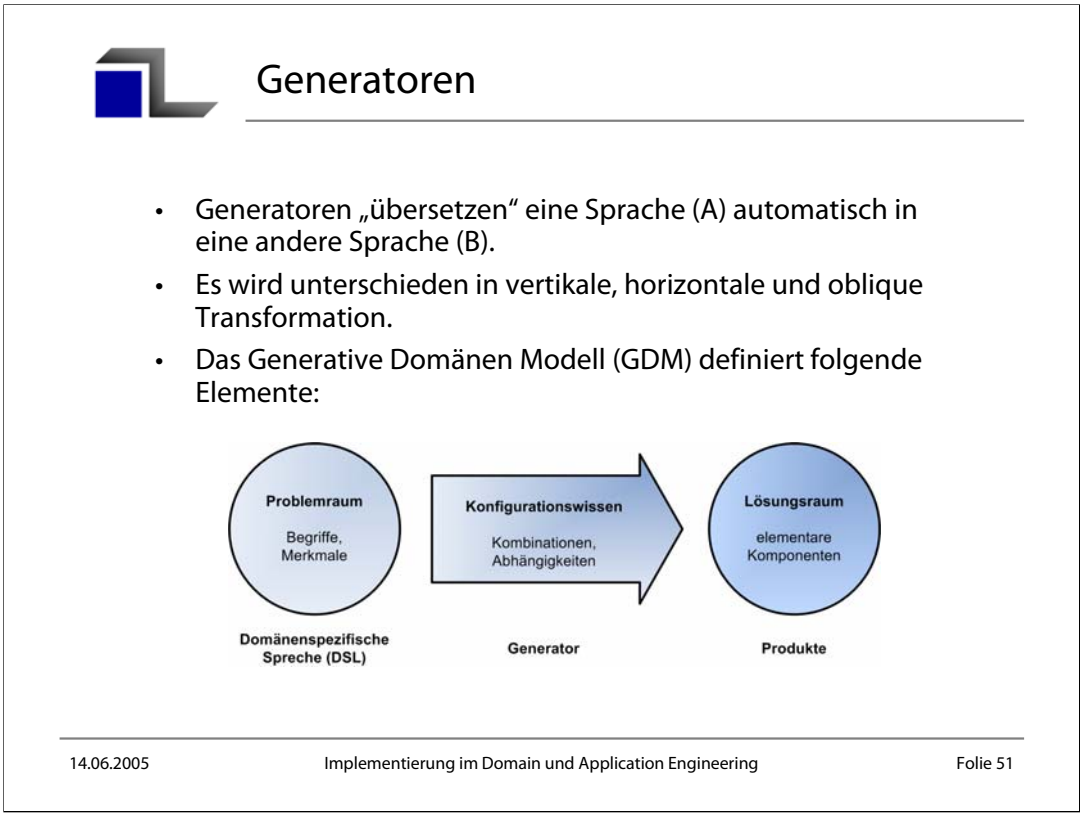
## Kollaborations-basierte Mechanismen – Bewertung

- Vorteile
  - Kollaborationen lassen sich einfach erweitern und wieder verwenden
  - Erweiterungen betreffen nicht den existierenden Code
  - Funktionale Änderungen betreffen nur die jeweilige Kollaboration
  - OR-Variationspunkte können einfach realisiert werden.
- Nachteile
  - Sehr komplex
  - Theorie ist schwer verständlich
  - Erfordert beim Entwurf der Kollaborationen grünliche Planung



Abgrenzung zu AOP:

- Aspekte fassen an zentraler Stelle zusammen (extrahieren)
- Kollaborationen definieren eine Sicht auf tatsächlich verstreute Codefragmente.




**Typen von Generatoren:**

*Vertikale Transformation:*

Eine Abstraktionsebene wird auf eine detailiertere, weniger abstrakte Ebene darunter abgebildet (*forward refinment*). Die Struktur einer Anwendung, d.h. die der Komponenten wird nicht verändert.

*Horizontale Transformation:*

Änderungen bewegen sich auf der selben Abstraktionsebene. Es wird die modulare Struktur verändert bzw. optimiert, z.B. Komponenten zusammengefasst oder entfernt

*Oblique Transformation :*

Kombination aus beiden vorangegangenen Typen.

Compiler die Optimierungen, wie zum Beispiel das oben genannte *Inlining* durchführen würden in diese Klasse eingeordnet werden, da diese Techniken die modulare Struktur verändern. Beim Inlining wird der Methodenrumpf an die Stelle kopiert, an der die Methode aufgerufen wird um den Overhead eines Methodenaufrufs umgehen zu können.

**Das Generative Domänen Modell (GDM):**

Der *Problemraum* besteht aus Anwendungs-orientierten Konzepten und Features und spezifiziert ein konkretes Produkt anhand einer DSL.

Das *Konfigurationswissen* definiert welche Komponenten wie miteinander kombiniert werden dürfen und welche nicht.

Der *Lösungsraum* enthält alle möglichen Kombinationen, d.h. alle möglichen Produkte der SPL, also auch nicht zulässige Kombinationen.

## Generatoren – Bewertung

- Vorteile:
  - Einfache Beschreibung von Produkten einer SPL durch die DSL
  - Produktableitungen können schnell durchgeführt werden → time-to-market stark verkürzt.
  - Qualität des (generierten) Codes ist hoch.
- Nachteile:
  - Definition einer DSL und Erstellen eines Generators ist mit einem hohem Aufwand verbunden.
  - Nachträgliche Anpassung der Variabilitätspunkte erfordert Anpassung der DSL und des Generators
  - Fehler können zentral (im Generator) behoben werden, sind aber schwer zu finden → Wartbarkeit erschwert.

# Agenda

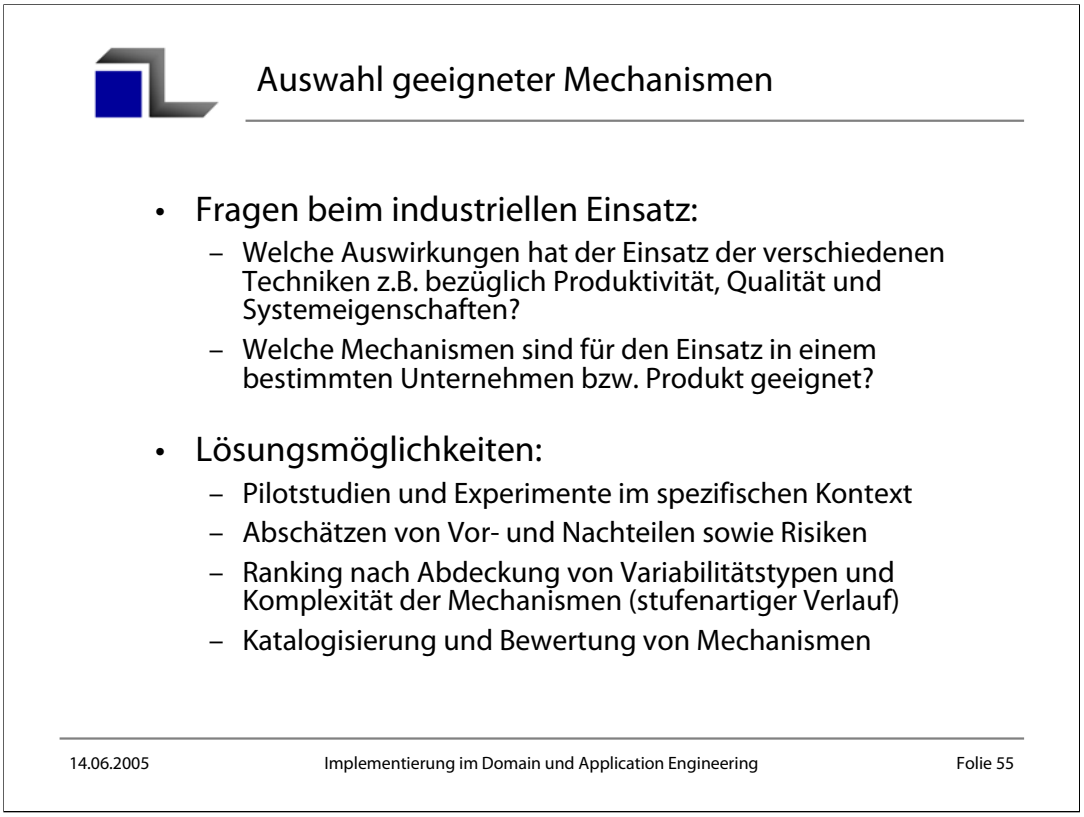
- Variabilität in der Produktlinienentwicklung
- Technologiedimensionen
- Vertikale Variabilitätsmechanismen
- Horizontale Variabilitätsmechanismen
- **Auswahl und Bewertung von geeigneten Mechanismen**
  - Übersicht: vorgestellte Mechanismen
  - Systematische Katalogisierung
- Fazit

## Vergleich der vorgestellten Mechanismen

| Variabilitätsmechanismen | Binding Time | Sprachspezifisch | Sprachunterstützung | VP-Typ | Beispiel |
|---|---|---|---|---|---|
| Konfigurationsmanagement | compilation | nein | (ja) | BOOL, XOR | Plattformanhängigkeit |
| Bedingte Übersetzung | compilation | nein | nein | BOOL, XOR | Ebedded Software |
| Bedingte Ausführung | runtime | nein | ja | BOOL, XOR | |
| Subtyp-Polymorphismus | runtime | OOP | ja | XOR | alternative Algorithmen |
| Parametrisierter Polymorphismus | compilation | C++, ... | ja | XOR, (OR) | |
| Frames | compilation | nein | nein | BOOL, XOR, OR | |
| Dynamic Class Loading | startup/runtime | Java, ... | ja | XOR | Plug-in Mechanism. |
| Aspektorientierte Programmierung | compilation/runtime | Java, C++,... (Aspect Weaver) | ja | BOOL, XOR, OR | Logging, Sicherheit |
| Kollaborationsbasierte Mechanismen | compilation | C++,... (Nested Classes) | ja | BOOL, XOR, OR | Frameworks |



Die dargestellte Matrix fasst die Eigenschaften der vorgestellten Variabilitätsmechanismen zusammen. Für jeden Mechanismus werden dabei auch die unterstützten Typen von Variabilitätspunkten (VP-Typ) angegeben.




Will man Software-Produktlinien nun in der Industrie einsetzen, so ergeben sich neben Kenntnis der technischen Details noch weitere Fragestellungen (vgl. [BKPS04], S. 117).

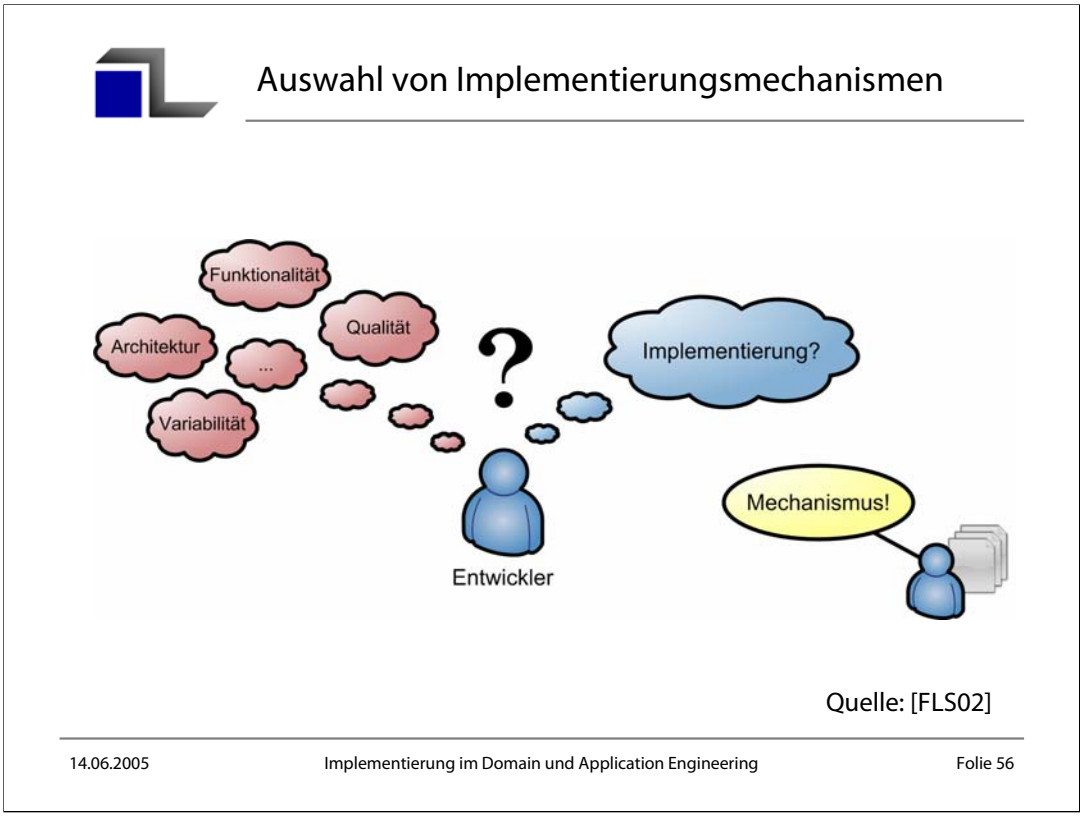
Im Zentrum steht die Frage, wie sich der Einsatz der beschriebenen Variabilitätsmechanismen auf Produktivität, Qualität und die Systemeigenschaften auswirkt. Die Herausforderung besteht darin, für den spezifischen Kontext in einem Unternehmen bzw. einem Projekt die geeigneten Mechanismen zu ermitteln.

Eine Organisation hat verschieden Möglichkeiten die Auswirkungen verfügbarer Variabilitätsmechanismen für ihre spezielle Situation zu bestimmen. Mit Pilotstudien und Experimenten können Vor- und Nachteile bestimmt und Risiken identifiziert werden.

Die einzelnen Mechanismen lassen sich nach verschiedenen Kriterien beschreiben und gegenüberstellen. Die auf der vorangegangenen Folie vorgestellte Matrix ist ein einfaches Beispiel.

Eine weitere Vergleichsmöglichkeit ist ein Ranking in zwei Dimensionen, wie es Muthig und Patzke beschreiben (vgl. [MP02a]). Dabei wird für jeden Mechanismus die Abdeckung der Variabilitätstypen und die Komplexität beurteilt. Stellt man die Ergebnisse in einem Diagramm dar, so ergibt sich ein stufenartiger Verlauf. Die Komplexität steigt also mit der Abdeckung an. Mechanismen mit einer hohen Abdeckung besitzen auch eine höhere Komplexität.

Fritsch, Lehn und Strohm beschreiben die Katalogisierung und Bewertung von Mechanismen anhand von Szenarien als weiter Option (vgl. [FLS02]).




Sehr hilfreich könnte für einen Entwickler, der mit der Implementierung in einer Produktlinie betraut ist, ein Katalog von bewerteten Variabilitätsmechanismen sein. Mit dessen Hilfe ließen sich geeignete Variabilitätsmechanismen finden und auswählen (ähnlich der expliziten Beschreibung und Auflistung von Entwurfsmustern).

Katalogisierung von Implementierungsmechanismen

- Katalog enthält
  - Beschreibungen der Mechanismen
  - Evaluierung der Mechanismen bezüglich relevanter Eigenschaften
  - Kiviat-Diagramme als Index

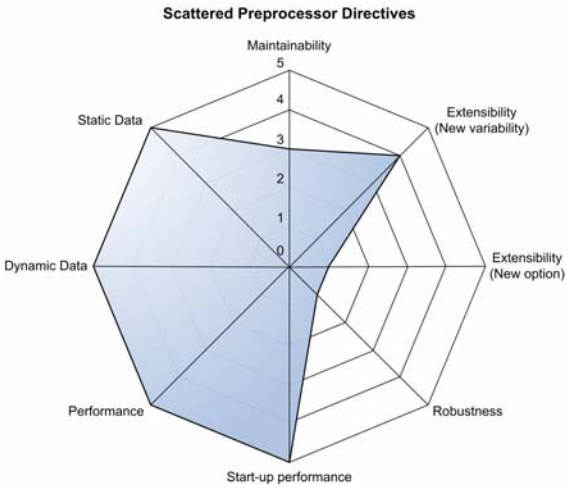
Quelle: [FLS02]



Ein Katalog enthält neben einer gründlichen Beschreibung der enthaltenen Mechanismen jeweils auch eine Evaluierung. Die Evaluierung bezieht sich auf potentiell beeinflusste Systemeigenschaften. Diese werden zuvor für alle Mechanismen festgelegte.

Als Index werden im Katalog Kiviat-Diagramme festgelegt. Ein Kiviat-Diagramm fasst die Evaluierungsergebnisse für einen Mechanismus zusammen.

## Kiviat-Diagramm – Beispiel



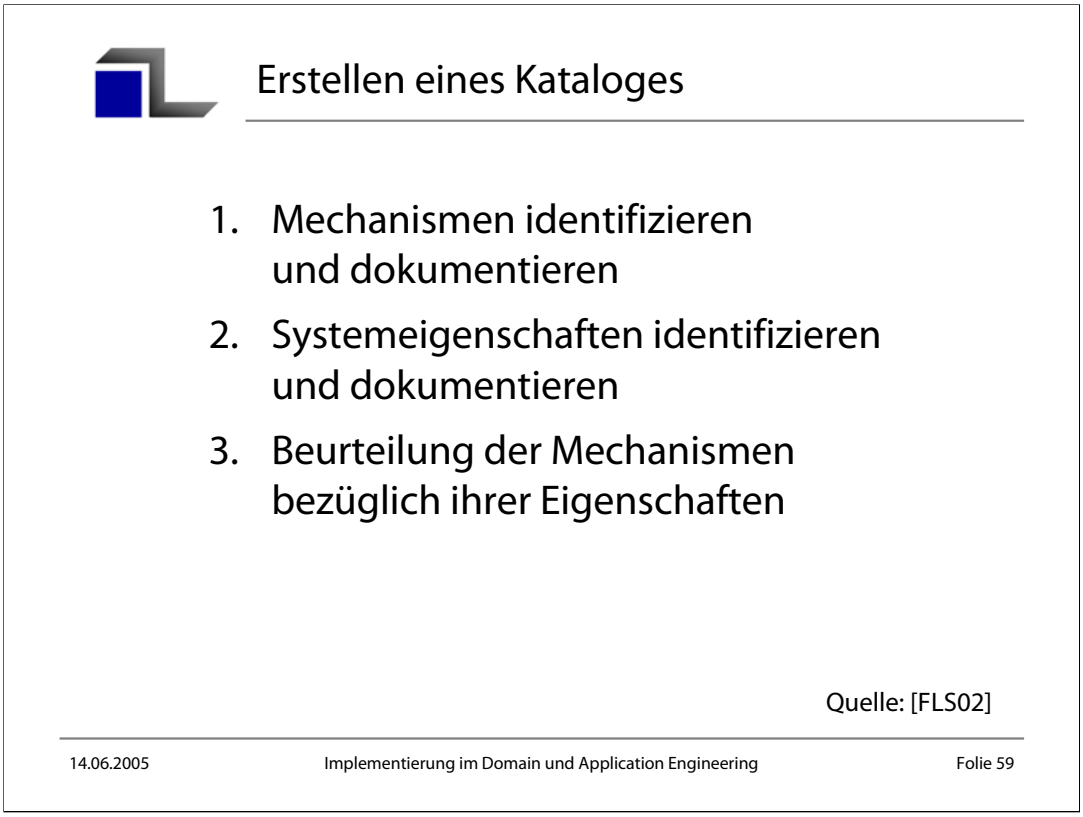

Quelle: [FLS02]



Kiviat-Diagramme dienen als Index im Katalog. Sie erlauben es einem Entwickler die Auswirkungen von verschiedene Implementierungsmechanismen einfach zu vergleichen oder auf der Grundlage bestimmter Anforderungen systematisch nach passenden Mechanismen zu suchen.

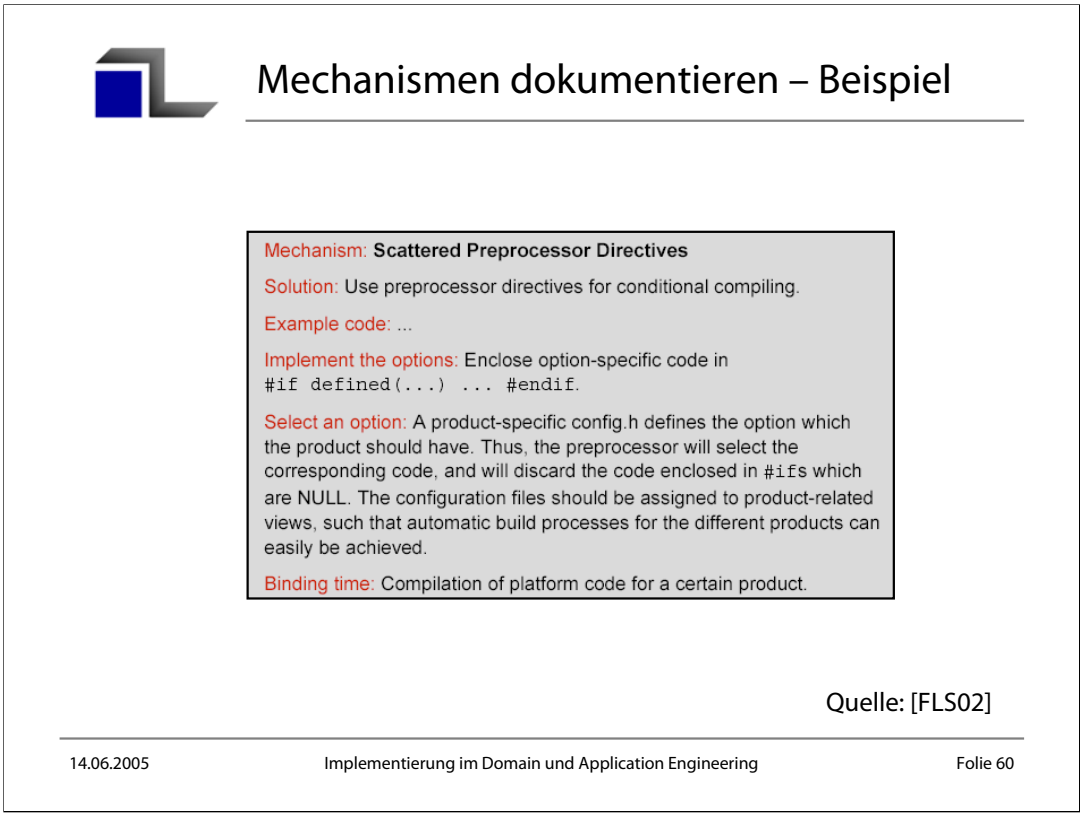
Das Beispiel auf der Folie zeigt das Diagramm für „Verstreute Präprozessoranweisungen“. Während sich für die Performance keine negativen Auswirkungen ergeben und sich neue Variabilitätspunkte (Variability) ohne große Probleme hinzufügen lassen ist zu erkennen, dass das Hinzufügen neuer Varianten (Optionen) zu einem Variationspunkt sehr schwierig ist.

Die folgenden Folien beschäftigen sich mit der Frage, wie man zu einem solchen Diagramm gelangt.




Ein Katalog wird in drei Schritten erstellt:

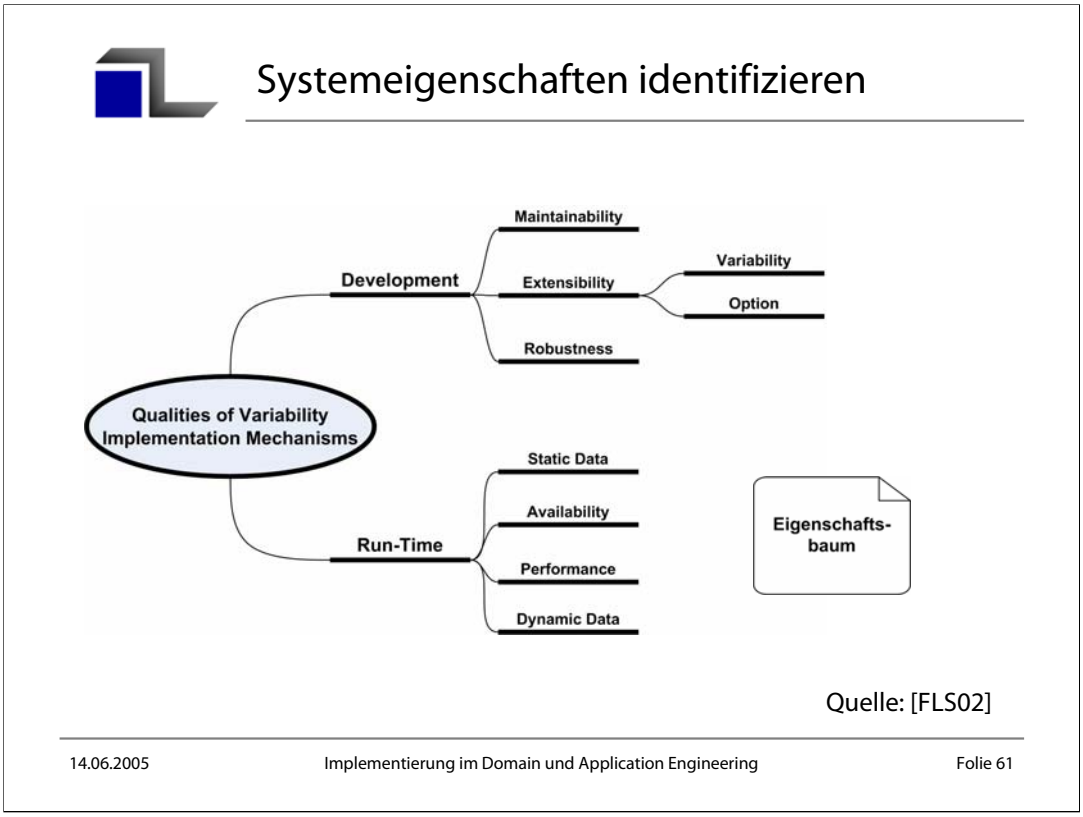
1. Zuerst müssen in Frage kommende Variabilitätsmechanismen identifiziert werden. Diejenigen, die der Katalog enthalten soll, müssen sorgfältig dokumentiert werden.
2. Systemeigenschaften, die potentiell von der Auswahl einer Mechanismus betroffen sind müssen ermittelt und Dokumentiert werden. Auf den identifizierten Eigenschaften aufbauend werden Szenarien für die Bewertung der Mechanismen erstellt.
3. Die Mechanismen werden bewertet. Dazu werden ihre Eigenschaften und die Auswirkungen auf die in Schritt 2 identifizierten Systemeigenschaften betrachtet. Als Grundlage dienen dabei die entwickelten Szenarien.




Die Folie zeigt beispielhaft die Beschreibung des Mechanismus „Verstreute Präprozessoranweisungen“.

Für jeden Mechanismus sollte die Dokumentation folgendes beinhalten (vgl. [FLS02]):

- Name des Mechanismus
- Lösung
- Design-Diagramm
- Beispielcode
- Wie Varianten implementiert werden
- Wie eine Variante ausgewählt wird
- Binding Time

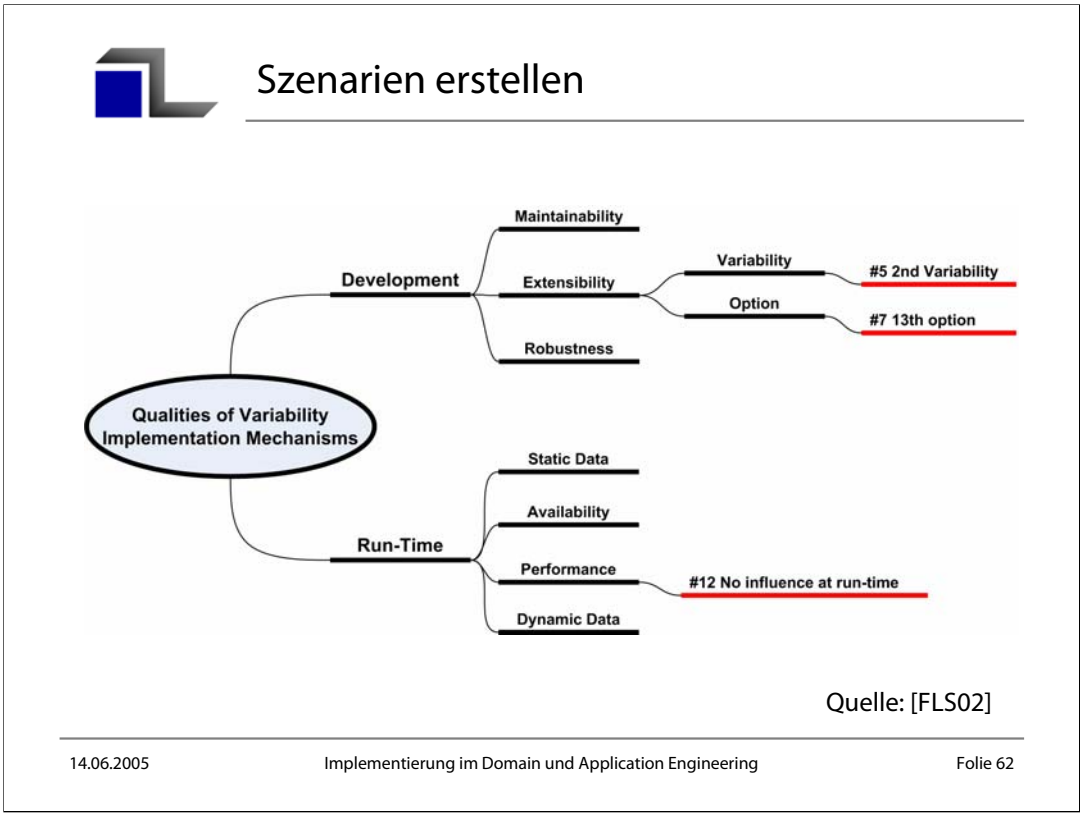



Gesucht sind Systemeigenschaften, die von den eingesetzten Variabilitätsmechanismen beeinflusst werden. Um diese zu ermitteln werden identifizierte Systemeigenschaften hierarchisch untergliedert. Die Folie zeigt ein mögliches Beispiel für eine Unterteilung in Form eines Baumes.

Die Eigenschaften werden dabei in zwei Kategorien eingeteilt:

- *Entwicklungszeiteigenschaften* (Development time qualities) treten bei der Domänen und Applikations-Implementierung, bei der Wartung oder bei der Erweiterung der Produktlinie um Variabilitätspunkte oder Varianten auf.
- *Laufzeiteigenschaften* (Run time qualities) werden bei der Verwendung einer Produktlinie wichtig.

Unterhalb dieser Hauptkategorien wurden verschiedene Eigenschaften identifiziert.



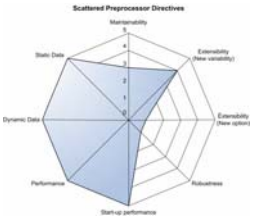

Für die identifizierten Systemeigenschaften werden nun Szenarien definiert. Ein Beispiel ist das Szenario #5 mit dem Namen „2nd Variability“ (vgl. [FLS02]):

> *„Ein Modul besitzt einen Variabilitätspunkt. Ein zweiter Variabilitätspunkt muss implementiert werden. Der zweite Variabilitätspunkt ist unabhängig vom bereits bestehenden befindet aber im gleichen Modul. Die Einführung des zweiten Variabilitätspunktes ist genauso einfach wie die des ersten.“*

Natürlich kann pro betroffener Systemeigenschaft mehr als ein Szenario existieren.

## Mechanismen beurteilen

- Relative Bewertung
- Jeweils Besten und schlechtesten Mechanismus finden
- Einfache Skala




| Mechanism | | | Quality: Maintainability | Extensibility (Var.) | Extensibility (Opt.) | Robustness | Availability | Performance | Dynamic Data | Static Data |
|---|---|---|---|---|---|---|---|---|---|---|
| | [B] | Branches on Source Files | + | + | −− | + | ++ | ++ | ++ | ++ |
| | [P1] | Scattered Preprocessor Directives | o | + | −− | −− | ++ | ++ | ++ | ++ |
| | [P2] | Structured Preprocessor Directives | ++ | o | o | o | ++ | + | + | + |
| | [C] | C++ Conditional Compilation | + | o | o | o | ++ | + | + | + |
| | [O] | Polymorphism with Subclasses | ++ | − | + | + | o | o | o | o |
| | [SS] | Strategy with Subclasses | ++ | − | + | + | o | o | o | o |
| | [ST] | Strategy with Template Parameters | ++ | − | + | + | + | o | o | o |
| | [Se] | Selection on Configuration Data | + | o | o | o | − | o | − | −− |
| | [RI] | Resource & Interpreter | ++ | −− | ++ | ++ | −− | − | | |

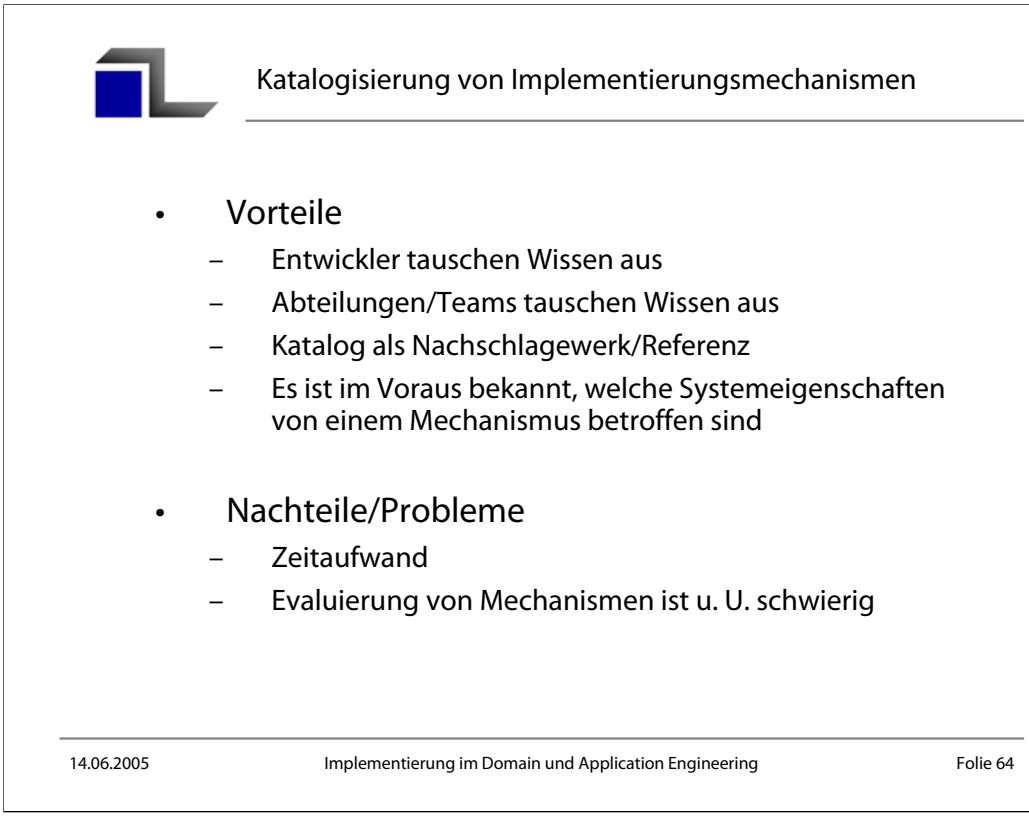
Quelle: [FLS02]

14.06.2005 Implementierung im Domain und Application Engineering Folie 63

Die im vorangegangen Schritt erstellten Szenarien werden jetzt verwendet, um die identifizierten Mechanismen zu bewerten. Dafür wird eine einfache Skala definiert.

Für jedes Szenario wird jeder Mechanismus betrachtet. Die Auswirkungen werden dokumentiert. Fasst man die Ergebnisse der einzelnen Szenarien für die Systemeigenschaften zusammen, so kann man eine Matrix erstellen, wie sie auf der Folie dargestellt ist.

Die Spalten stellen die Systemeigenschaften, die Zeilen die Mechanismen dar. Für die Bewertung wurde eine sehr einfache Skala (--,-,o,+,++) verwendet. Die Ergebnisse der Bewertung lassen sich dann zeilenweise in Kiviat-Diagramme für die einzelnen Mechanismen überführen. Das Diagramm besitzt dabei so viele Ecken, wie Systemeigenschaften identifiziert wurden.




Durch die Verwendung eines Katalogs für Variabilitätsmechanismen wird das Wissen über technische Aspekte der Mechanismen und deren Auswirkungen auf das System explizit beschrieben. Die am Projekt beteiligten Entwickler oder auch Teams haben so eine gemeinsame, systematisierte Entwicklungsgrundlage und eine gemeinsame Terminologie. Somit werden Diskussionen und der Austausch von Wissen erleichtert. Ein Entwickler kann den Katalog konkret als Nachschlagewerk verwenden.

Die Katalogisierung der Mechanismen ist jedoch stark vom Kontext einer Organisation abhängig und muss zeitaufwändig jeweils neu durchgeführt werden. Eine einheitliche Bewertung, wie sie beispielsweise für Entwurfsmuster existiert, ist schwierig.

Außerdem können viele der Systemeigenschaften nicht eindeutig gemessen werden, so dass keine absolute Skala zur Bewertung existiert. Daher müssen die Mechanismen relativ bewertet werden. Dabei ist jeweils der beste und der schlechteste Mechanismus zu ermitteln und die Skala entsprechend auszurichten.

# Agenda

- Variabilität in der Produktlinienentwicklung
- Technologiedimensionen
- Vertikale Variabilitätsmechanismen
- Horizontale Variabilitätsmechanismen
- Auswahl und Bewertung von geeigneten Mechanismen
- **Fazit**

## Zusammenfassung

- Variabilität muss über alle Entwicklungsphasen hinweg betrachtet und dokumentiert werden
- Für die Implementierung von Variabilität stehen viele unterschiedliche Mechanismen zur Verfügung
- Die Auswahl der Implementierungsmechanismen muss sich nach der Art der Variabilität und anderen Anforderungen richten
- Eine systematische Bewertung der Mechanismen, hinsichtlich spezifischer Rahmenbedingungen, ist sinnvoll.

## Persönliches Fazit

- Die Frage welcher Mechanismus wofür eingesetzt werden soll lässt sich nicht allgemein beantworten.
  - Es existieren komplexe und weniger komplexe Mechanismen
  - Höhere Unterstützung von Produktlinien (Variabilitätstypen) benötigt komplexere Mechanismen
  - Einfache Variabilitätsmechanismen sind für kleine, wenig komplexe Projekte durchaus geeignet.
- Eine kontextabhängige Bewertung aller Mechanismen ist bei größeren Projekten unabdingbar.

- AOP ist ein interessanter und vielseitig anwendbarer Ansatz
  - Vergleichsweise leicht verständlich
  - Für den breiten Einsatz geeignet

# Literatur (1)


[BB01] BACHMANN, Felix ; BASS, Len: Managing Variability in Software Architecture. In: *Proceedings of the ACM SIGSOFT Symposium on Software Reusability (SSR'01)*, 2001, S. 126–132

[Bec04] BECKER, Martin: *Anpassungsunterstützung in Software-Produktfamilien*. Kaiserslautern, Technische Universität Kaiserslautern, Fachbereich Informatik, Dissertation, 2004

[BKPS04] BÖCKLE, Günter (Hrsg.) ; KNAUBER, Peter (Hrsg.) ; POHL, Klaus (Hrsg.) ; SCHMIDT, Klaus (Hrsg.): *Software Produktlinien*. Heidelberg : dpunkt.verlag, 2004

[Cla01] CLAUSS, Matthias: *Untersuchung der Modellierung von Variabilität in UML*. Dresden, Technische Universität Dresden, Fakultät Informatik, Diplomarbeit, Juli 2001

[Eis05] EISENECKER, Ulrich: *Softwareentwicklung mit Frames: Techniken und Einsatzmöglichkeiten*. Leipzig, Univarsität Leipzig, Wirtschaftswissenschaftliche Fakultät, Institut für Wirtschaftsinformatik, Folien zur Vorlesung, Februar 2005. – online verfügbar – URL http://www.uni-leipzig.de/softwareentwicklung/frames.pdf (Zugriff 5. Juni 2005)

[ES02] EISENECKER, Ulrich ; SCHILLING, Rüdiger: Zum Entwickeln entwickelt: Generative Programmierung mit einem Frameprozessor. In: *iX* 10 (2002)

# Literatur (2)


[FLS02] FRITSCH, Claudia ; LEHN, Andreas ; STROHM, Thomas: Evaluating Variability Implementation Mechanisms. In: *Proceedings of the Second International Workshop on Product Line Engineering - The Early Steps: Planning, Modeling, and Managing (PLEES'02)*, 2002, S. 59–64

[GBS01] VAN GURP, Jilles ; BOSCH, Jan ; SVAHNBERG, Mikael: On the Notion of Variability in Software Product Lines. In: Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA'01), 2001

[GHJV95] GAMMA, Erich ; HELM, Richard ; JOHNSON, Ralph ; VLISSIDES, John: *Design Patterns: Objects of Reusable Object-Oriented Software*. 1. Auflage. Reading : Addison Wesley, 1995

[JB02] JARING, Michel ; BOSCH, Jan: Representing Variability in Software Product Lines: A Case Study. In: *Proceedings of the Second Software Product Line Conference*, 2002, S. 15–36

[LA05] LEICH, Thomas ; APEL, Sven: Ein merkmalsorientierter Speichermanager für eingebettete Systeme. In: BRASS, Stefan (Hrsg.) ; GOLDBERG, Christian (Hrsg.): *Tagungsband zum 17. GI-Workshop über Grundlagen von Datenbanken (17th GI-Workshop on the Foundations of Databases)*. Wörlitz, Germany : Institute of Computer Science, Martin-Luther-University Halle-Wittenberg, Mai 2005, S. 73–77

# Literatur (3)


[Les04] VAN LESSEN, Tammo: *Generatives Programmieren*. Stuttgart, Universität Stuttgart, Fakultät für Elektrotechnik und Informatik, Seminar Produktlinien, Seminararbeit, Januar 2004

[MP02a] MUTHIG, Dirk ; PATZKE, Thomas: Generic Implementation of Product Line Components. In: *Proceedings of the Net.ObjectDays (NODE'02)*. Erfurt, Germany, Oktober 2002, S. 316–333

[MP02b] MUTHIG, Dirk ; PATZKE, Thomas: Product Line Implementation Technologies: Programming Language View / Fraunhofer Institute for Experimental Software Engineering (IESE). 2002 (IESE-Report No. 057.02/E). – Forschungsbericht

[Ost03] OSTERMANN, Klaus: Modules for Hierarchical and Crosscutting Models. Darmstadt, Technischen Universität Darmstadt, Fachbereich Informatik, Dissertation, April 2003

[Sch03] SCHMIDMEIER, Arno: Aspekte von Java. In: Java Magazin 12 (2003)

[SB02] SMARAGDAKIS, Yannis ; BATORY, Don S.: Mixin layers: an object-oriented implementation technique for refinements and collaboration-based designs. In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 11 (2002), April, Nr. 2, S. 215–255

# Literatur (4)

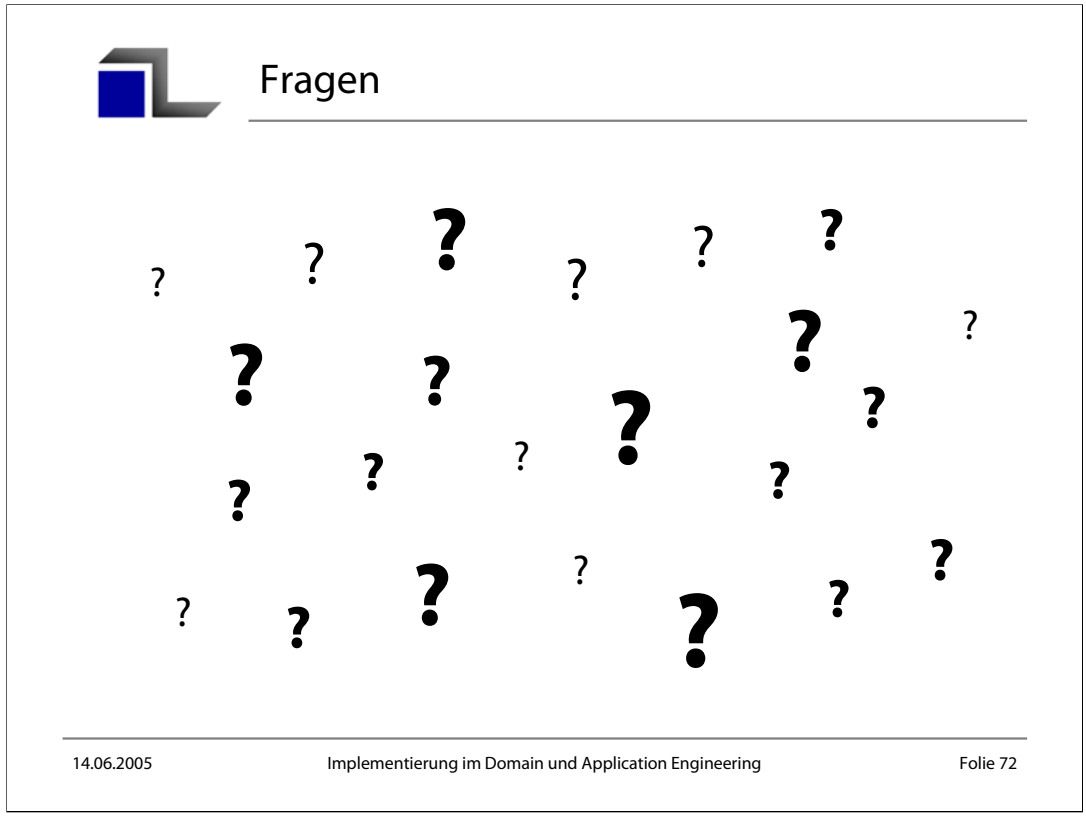

[Sic04] SICHTING, Helge: Middleware-Architektur für mobile Informationssysteme. Magdeburg, Otto-von-Guerike-Univarsität Magdeburg, Fakultät für Informatik, Institut Technische und Betriebliche Informationssysteme, Diplomarbeit, Januar 2004

[Spi02] SPINCZYK, Olaf: Aspektorientierung und Programmfamilien im Betriebssystembau. Magdeburg, Otto-von-Guericke-Universität Magdeburg, Fakultät für Informatik, Dissertation, Dezember 2002

[TH02a] THIEL, Steffen ; HEIN, Andreas: Modeling and Using Product Line Variability in Automotive Systems. In: *IEEE Software* 19 (2002), Juli/August, Nr. 4, S. 66–72

[TH02b] THIEL, Steffen ; HEIN, Andreas: Systematic Integration of Variability into Product Line Architecture Design. In: *Proceedings of the Second Software Product Line Conference*, 2002, S. 130–153

[Wik:AOP] o.V.: Aspektorientierte Programmierung. In: *Wikipedia. Die Freie Enzyklopädie*. Stand: 9. Juni 2005, 19:21 Uhr – URL http://de.wikipedia.org/wiki/Aspektorientierte_Programmierung (Zugriff 8. Juni 2005)


Fragen
14.06.2005
Implementierung im Domain und Application Engineering
Folie 72