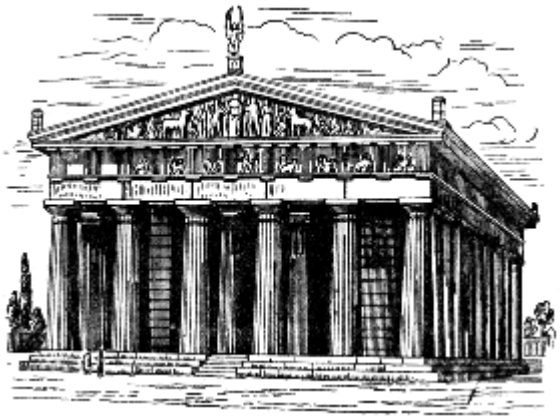
Softwarearchitektur

(Architektur: *αρχή* = Anfang, Ursprung + *tectum* = Haus, Dach)

5. Sichten bei der Softwarearchitektur – Teil II



Vorlesung Wintersemester 2008 / 2009

Technische Universität München
Institut für Informatik
Lehrstuhl von Prof. Dr. Manfred Broy

Dr. Klaus Bergner, Prof. Dr. Manfred Broy, Dr. Marc Sihling

Inhalt

- Rekapitulation
- Entwicklungssicht
  - Überblick und Ziele
  - Modellgetriebene Entwicklung
  - Code-Organisation
  - Build-Management
- Testsicht
- Bereitstellungssicht
- Andere Ansätze
- Zusammenfassung

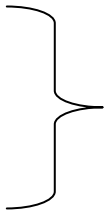
Inhalt

- Rekapitulation
- Entwicklungssicht
  - Überblick und Ziele
  - Modellgetriebene Entwicklung
  - Code-Organisation
  - Build-Management
- Testsicht
- Bereitstellungssicht
- Andere Ansätze
- Zusammenfassung

Arten von Sichten auf die Architektur eines Systems

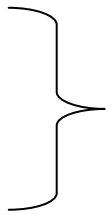
- Wir unterscheiden folgende Sichten:

- Spezifikationssichten
  - Fachliche Sicht
  - Technische Sicht
  - Verteilungssicht



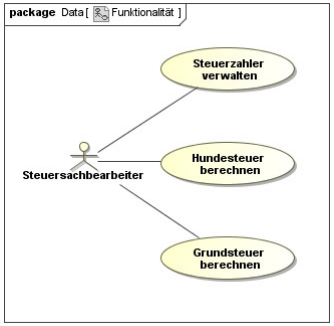
Thema der letzten Doppelstunde

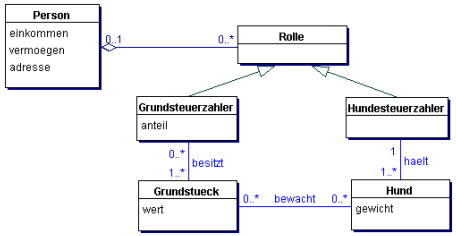
- Erstellungssichten
  - Entwicklungssicht
  - Testsicht
  - Bereitstellungssicht

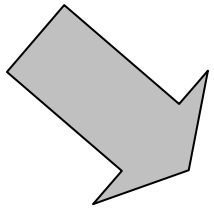


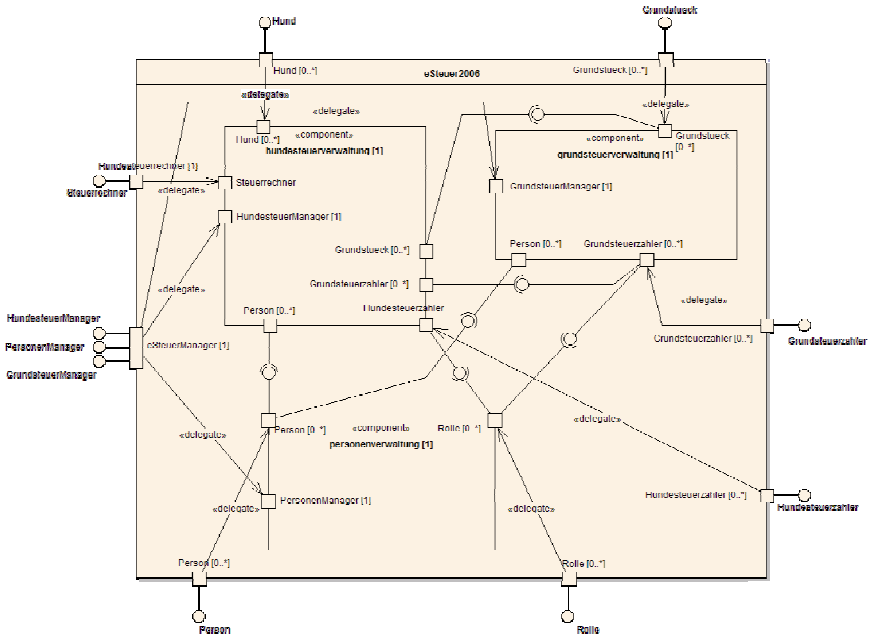
Thema der heutigen Doppelstunde

Spezifikationssichten: Vom CIM zum PIM









Zusammenfassung vom letzten Mal

- Eine Sicht stellt jeweils bestimmte, zusammengehörige Eigenschaften eines Systems dar.
- Es gibt zwei Hauptarten von Sichten: Spezifikationssichten zur Beschreibung des Systems selbst und Erstellungssichten zur Beschreibung seiner Entwicklung.
- Bei den Spezifikationssichten sollte die fachliche Sicht vor der technischen Sicht bearbeitet und ausmodelliert werden.
- Sowohl in der fachlichen als auch in der technischen Sicht ist die Bildung von Komponenten mit klar definierten Schnittstellen und Abhängigkeitsbeziehungen wichtig.
- Zur Entkopplung von fachlicher und technischer Sicht gibt es eine Reihe von Standard-Techniken. Dazu gehören insbesondere die Kapselung der Technik hinter Schnittstellen, Container-Architekturen sowie Generierung.

Inhalt

- Rekapitulation
- Entwicklungssicht
  - Überblick und Ziele
  - Modellgetriebene Entwicklung
  - Code-Organisation
  - Build-Management
- Testsicht
- Bereitstellungssicht
- Andere Ansätze
- Zusammenfassung

Entwicklungs-Sicht - Überblick

- Zweck
  - Darstellung der erstellten Beschreibungen des Systems und der Vorgänge bei Entwicklung und Integration des Systems
- Inhalte
  - erstellte Beschreibungen zum System, insbesondere die unterschiedlichen Modelle und der Quellcode
  - Beziehungen zwischen den Beschreibungen
  - Entwicklungsumgebung und zugehörige Abläufe
- Beschreibung
  - textuelle Beschreibungen, z.B. von Ablagestrukturen, Entwicklungsabläufen und Werkzeugen
  - UML-Aktivitätsdiagramme für Entwicklungsabläufe, UML-Deployment-Diagramme
  - Konfigurationsdateien für Entwicklungs- und Build-Systeme

Ziele in der Entwicklungs-Sicht

- Durchsetzung von Architekturentscheidungen, insbesondere von Abhängigkeitsbeziehungen
- Unterstützung der Parallel-Arbeit einzelner Entwickler und Entwicklungsteams (Edit, Compile, Build, Test)
  - Möglichst einfache und schnelle Entwicklungsprozesse
  - Entkopplung einzelner Entwickler und Teams voneinander
  - Einfache Freigabe von Teilsystemen an andere Teams zur Integration
- Unterstützung beim Bau von Teilsystemen und unterschiedlichen Varianten des Systems
- Verwaltung unterschiedlicher Versionen des Systems und seiner Teilsysteme (Konfigurationsmanagement, KM)
- Optimierung der Entwicklungsabläufe
  - Automatisierung von Entwicklungsschritten
  - Generierung von Ergebnissen

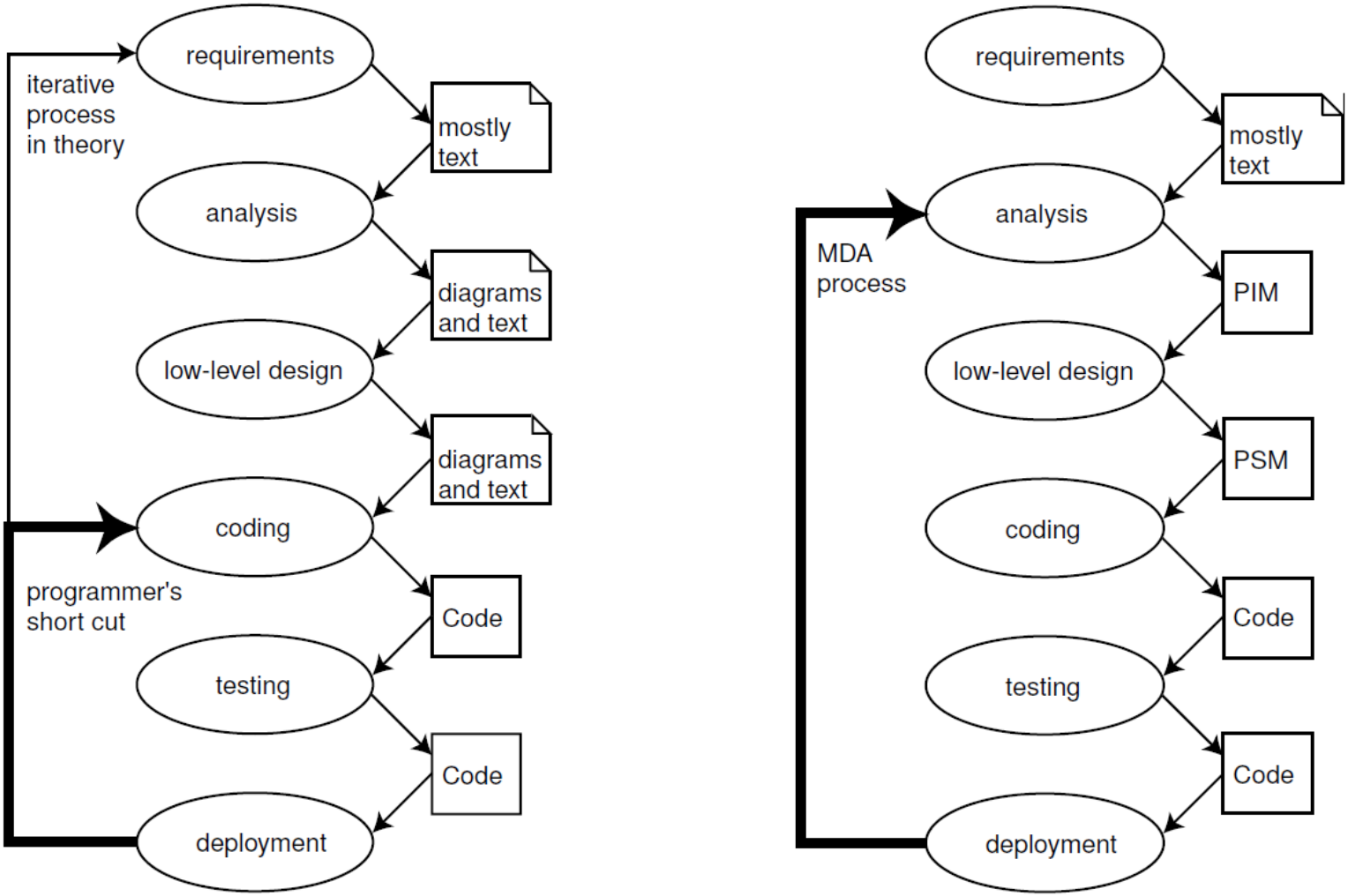
Vorgehen beim Entwurf der Entwicklungs-Sicht

- Aspekte der Entwicklungssicht sind:
  - Generierungs-, Entwicklungs- und Build-Prozesse und Werkzeuge
  - Organisation der Entwicklungsartefakte: Modelle und Programmcodes
- Die Entwicklungssicht ist insbesondere abhängig von der Technischen Sicht, der Testsicht und der Verteilungssicht:
  - Verfügbare Generatoren und Entwicklungswerkzeuge
  - Basissoftware, gewählte Programmiersprachen
  - Abbildung auf die Systemarchitektur
- Meist wird die Entwicklungsumgebung von einem kleinen Team anhand eines Durchstichs des Systems entwickelt und vorbereitet.
  - Auswahl und Integration existierender Werkzeuge
  - Entwicklung spezieller Generatoren und Werkzeuge
- In der Folge erfolgt dann die Realisierung des Gesamtsystems, und die Entwicklungsumgebung wird laufend optimiert.

Inhalt

- Rekapitulation
- Entwicklungssicht
  - Überblick und Ziele
  - Modellgetriebene Entwicklung
  - Code-Organisation
  - Build-Management
- Testsicht
- Bereitstellungssicht
- Andere Ansätze
- Zusammenfassung

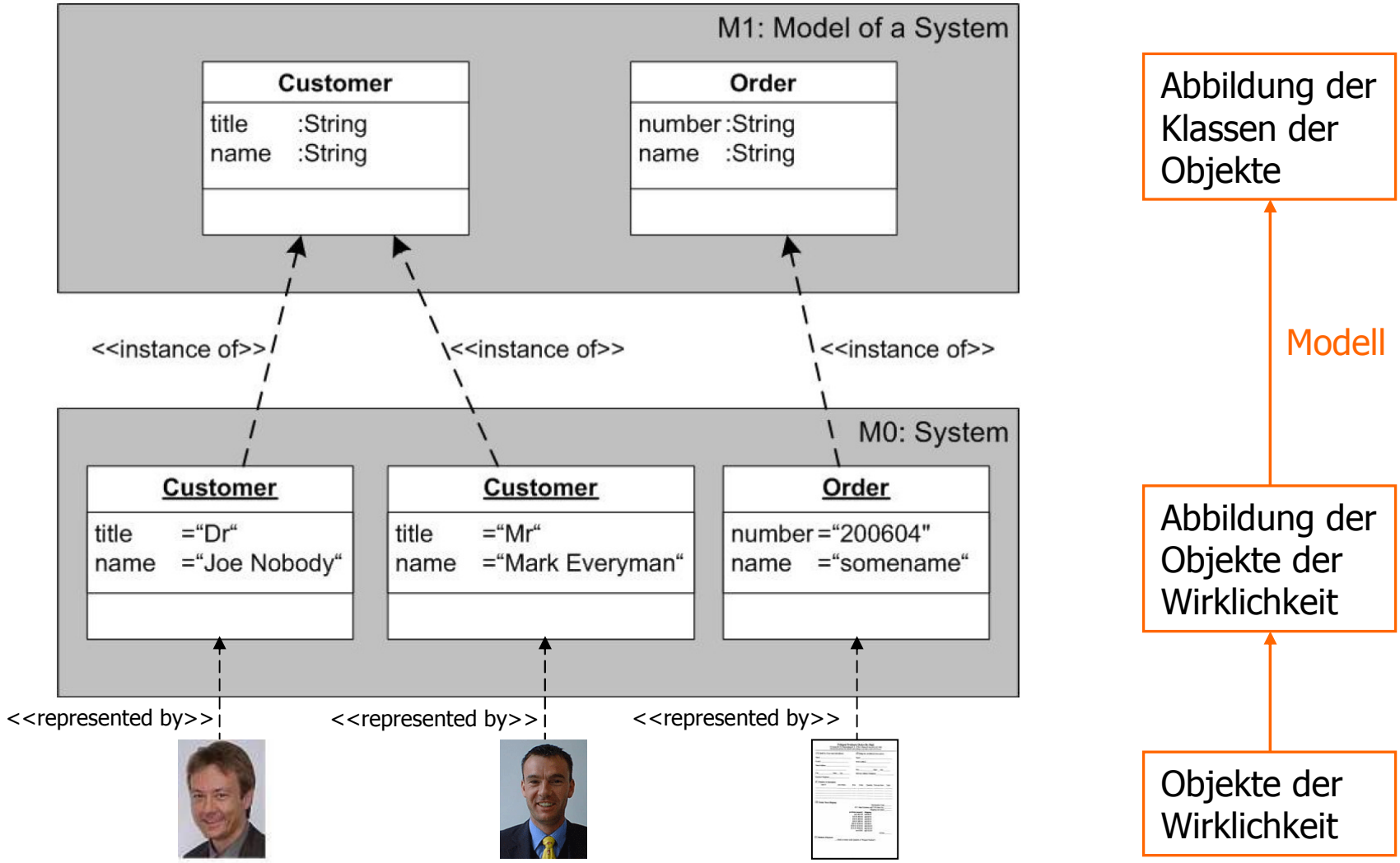
Entwicklungsprozesse: Traditionell und Modellgetrieben



Grundkonzepte der MDA

- Modell
  - Entspricht dem Konzept der **Beschreibung** aus der dritten Vorlesung. Gibt Eigenschaften und Verhalten einer Menge von Instanzen vor.
- Meta-Modell
  - Beschreibung einer Beschreibung. Sinnvoll, um die Erstellung und Verwendung von Modellen zu modellieren und zu spezifizieren.
- Meta Object Facility
  - Standard der OMG für einen vierstufigen Meta-Modell-Ansatz.
- Transformation
  - Operator, der eine Beschreibung in eine andere übersetzt.
  - Arten von Transformationen:
    - Model-to-Model Transformation
    - Model-to-Code Transformation
    - Code-to-Code Transformation

MOF und UML am Beispiel – M0 und M1



MOF und UML am Beispiel – M1 und M2

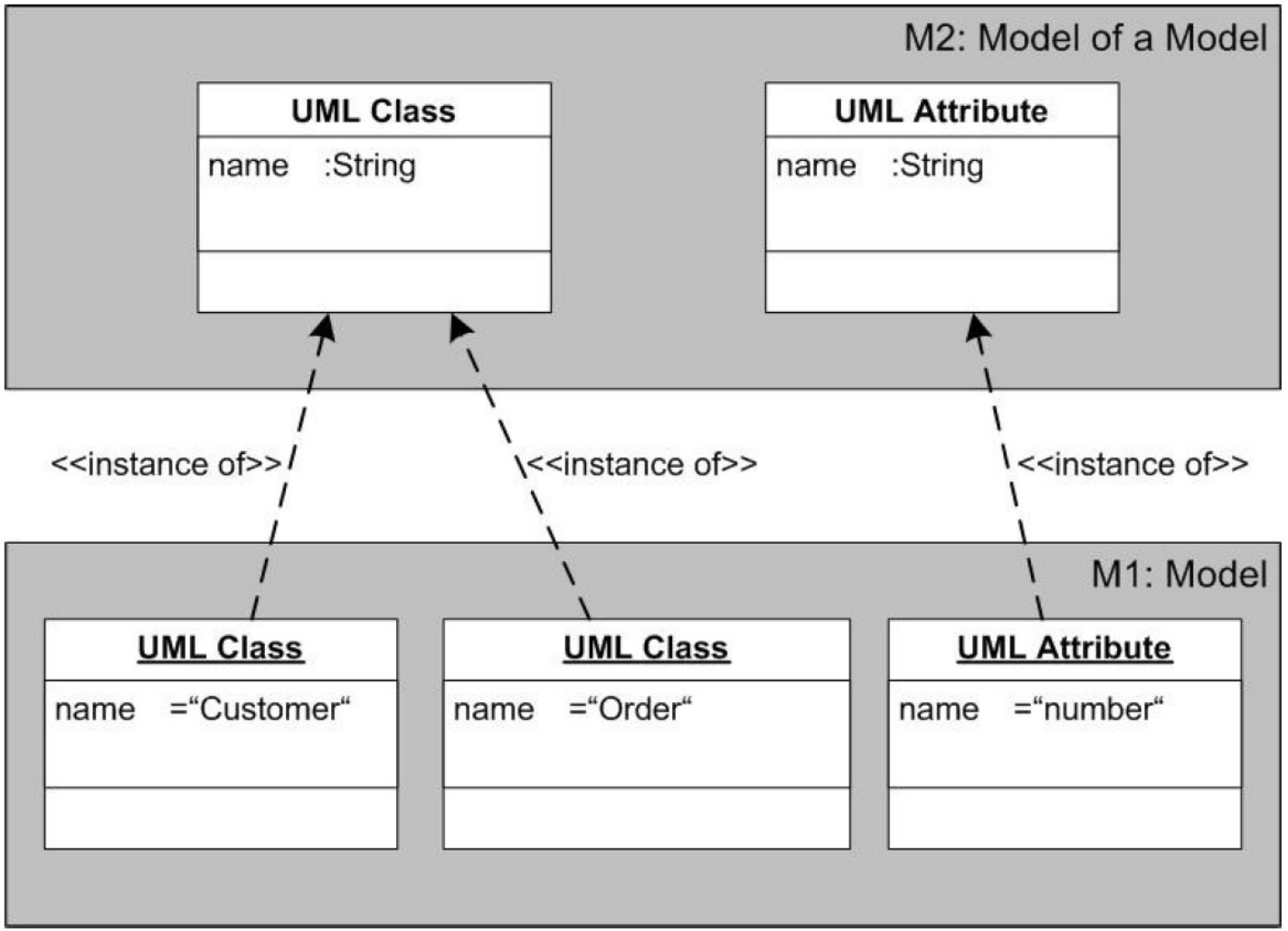


Abbildung der Konzepte in UML-Klassen-diagrammen

Meta-Modell

Abbildung der Klassen der Objekte

MOF und UML am Beispiel – M2 und M3

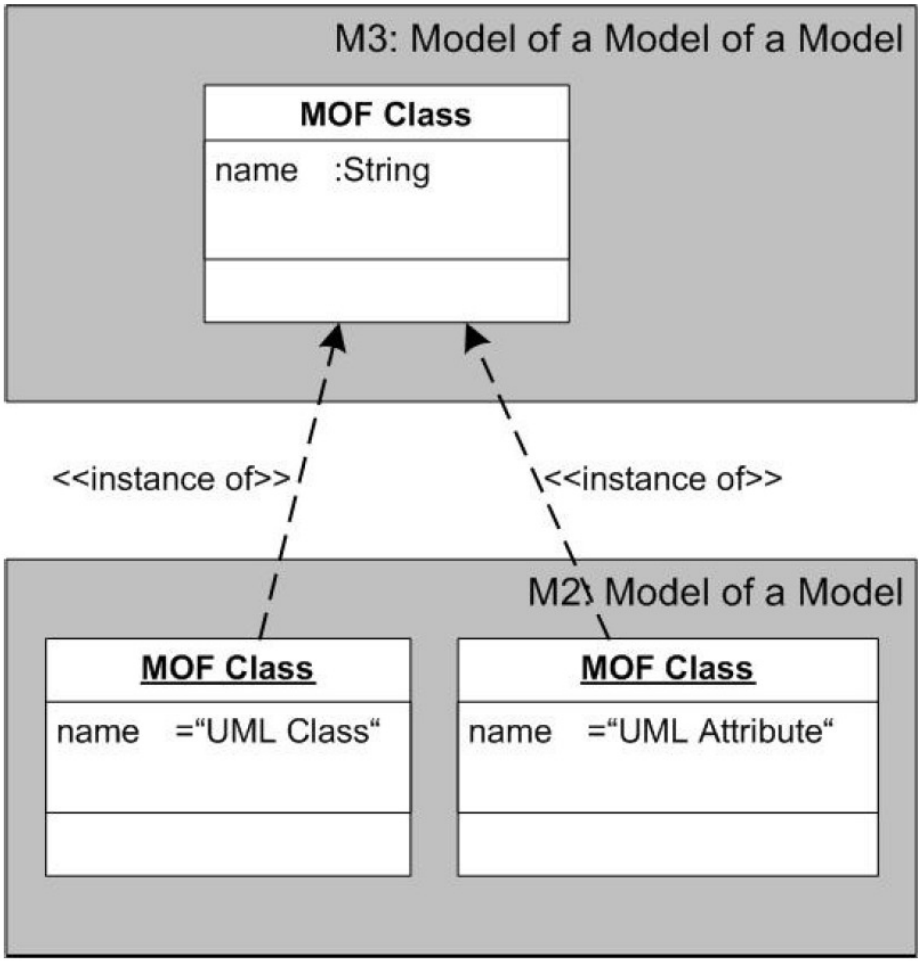
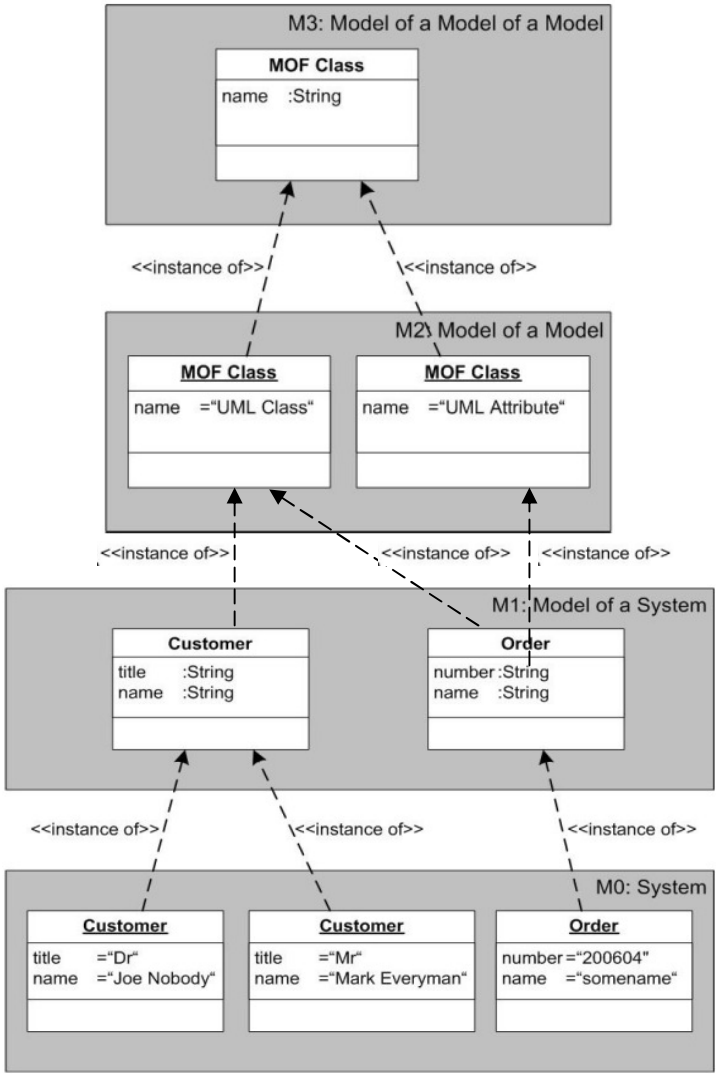


Abbildung von allgemeinen Konzepten zur Modellierung

Meta-Meta-Modell

Abbildung der Konzepte in UML-Klassen-diagrammen

MOF: Vierschichtiges Meta-Modell



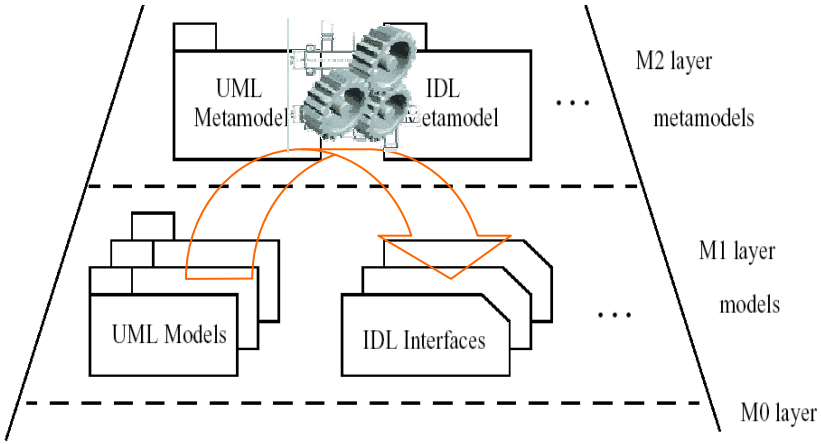
Welt des Meta-Methodikers: Allgemeine Modellierungskonzepte, Werkzeugbaukästen, allgemeine Generatoren (z.B. QVT)

Welt des Methodikers: Modellierungssprachen (Domain-Specific Languages), Entwicklungswerkzeuge (z.B. Generatoren)

Welt des Entwicklers: Klassen und Operationen eines Systems

Welt des Anwenders: Instanzen und Abläufe zur Laufzeit eines Systems

Beispiel: Codegenerator UML2IDL



Charakteristika:

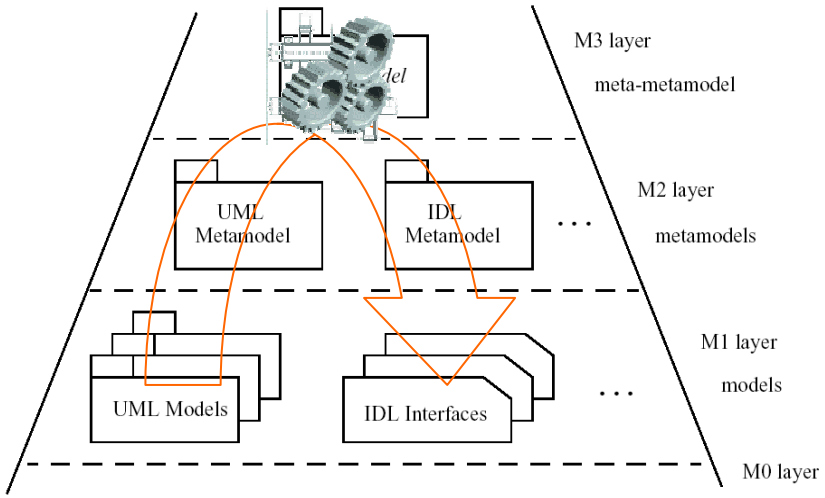
- Handgeschriebener Generator
- Setzt auf API eines CASE-Werkzeugs auf
- Erzeugt Quellcode-Dateien für CORBA IDL

Typischer Ausschnitt des Codes eines derartigen Generators:

```
void generateIDLInterface(UMLClass x) {
  writeln("interface " + x.getClassName + " {\n";
  foreach (a in UMLClass.getAttributes())
      generateIDLAttribute(a);
  …
}
```

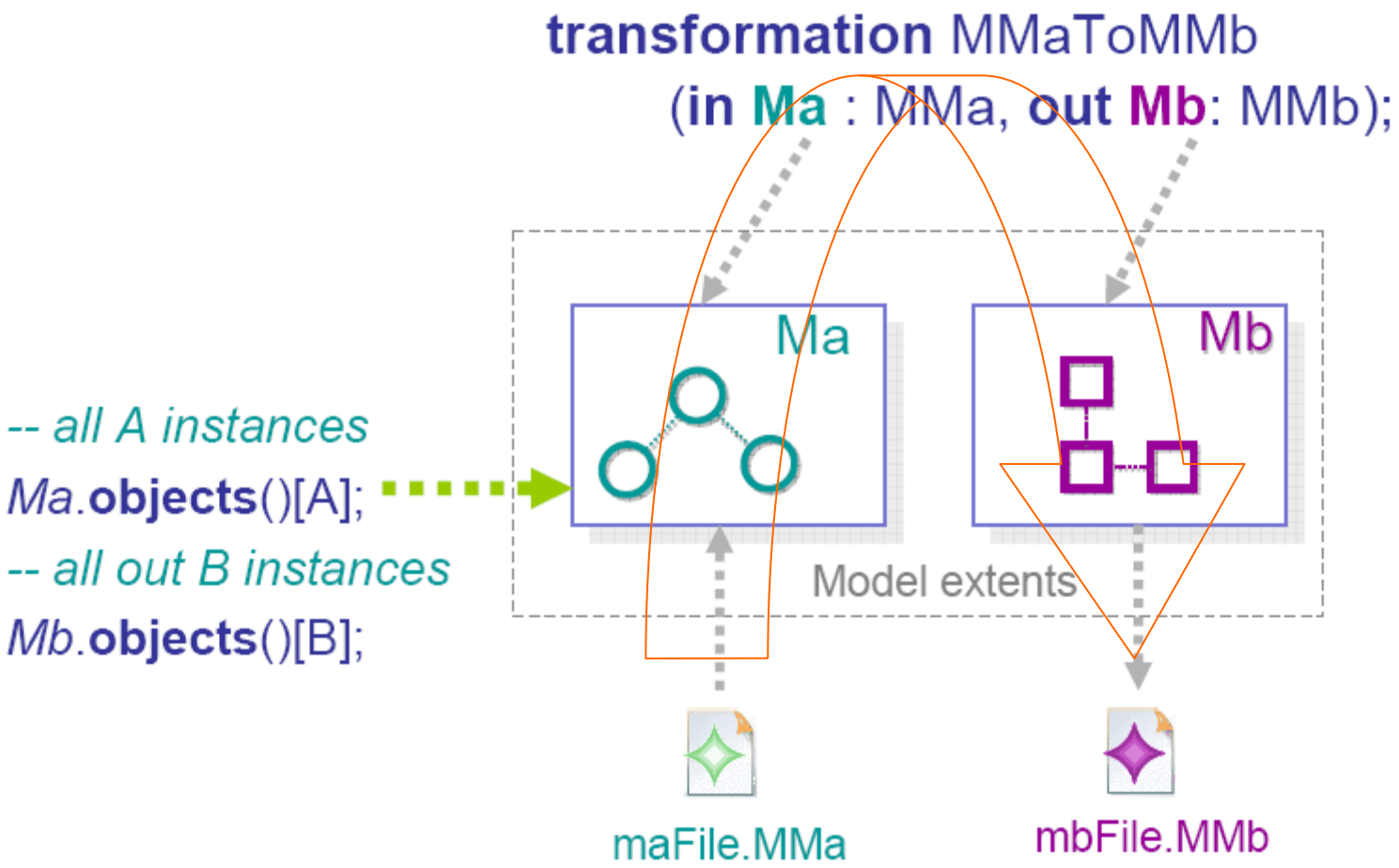
Beispiel: QVT – Queries, Views, Transformations

- Allgemeine Modelltransformationssprache
  - für alle Modelle auf M1,
  - die ein Metamodell auf M2 haben,
  - das durch MOF auf M3 abgebildet ist.



- Charakteristika
  - Spezifiziert Patterns für zwei oder mehr Modell-Ausschnitte, die aufeinander abgebildet werden sollen
  - Kennt unterschiedliche Arten von Transformationen (insbesondere auch bidirektionale Transformationen, inverse Transformationen)
- Status von QVT
  - Standard verabschiedet, operationale und deklarative Teilsprachen
  - Einige erfolgreiche „Konkurrenz-Standards" (z.B. ATL)

QVT als allgemeine Transformationssprache



Beispiel für operationale QVT-Transformation

```
-- Transformiere ein UML-Modell in ein ER-Modell
transformation uml2erm( in uml : UML, out erm : ERM )

-- „Class"-Modellelemente abbilden mittels class2table-Transformation
main() {
      uml.objects()[#Class]->map class2table();
}

 -- Abbildung von Class auf Table
mapping Class::class2table() : Table
when { self.stereotypes->includes('persistent') }
{
    name := self.name;
}
```

Quell-Modell : UML
Ziel-Modell : ERM

Status: MDA und Generatoren

- Vor dem Einsatz Risiken sowie Kosten und Nutzen analysieren!
  - Akzeptanz und Know-how bei Entwicklern
  - Kosten für Erstellung und Integration der Generatoren vs. gesparte Aufwände durch Automatisierung
  - Komplexität und Roundtrip-Zeiten der Entwicklungsumgebung
  - Abhängigkeit von Tools und Werkzeugherstellern
- Methodik und Entwicklungsprozesse sorgfältig definieren
  - Feste Einbindung in Build-Prozess ist erforderlich.
  - Wiederholte Generierung muss möglich sein!
  - Trennung von generiertem und manuell erstelltem Code nötig.
- Werkzeuge und Generatoren müssen flexibel sein.
  - Anpassung an generiertem Code darf nicht länger dauern als die gleiche Änderung an manuellem Code.
  - Anpassungen von Generatoren müssen projektspezifisch möglich sein.
  - Definition domänenspezifischer Sprachen (DSL) bietet viel Potenzial.

Inhalt

- Rekapitulation
- Entwicklungssicht
  - Überblick und Ziele
  - Modellgetriebene Entwicklung
  - Code-Organisation
  - Build-Management
- Testsicht
- Bereitstellungssicht
- Andere Ansätze
- Zusammenfassung

Warum sollte die Code-Organisation entworfen werden?

- Bei der Entwicklung eines großen Systems fallen viele Beschreibungen an, die adäquat verwaltet und abgelegt werden müssen.
  - Dies betrifft vor allem die implementierungsnahen Beschreibungen, und hier insbesondere den Quellcode.
  - Bei der modellgetriebenen Entwicklung müssen zusätzlich auch die Modell-Artefakte verwaltet werden.
- Wird die Code-Organisation nicht explizit entworfen, dokumentiert und qualitätsgesichert, so besteht die Gefahr, dass hohe Aufwände für die Einarbeitung neuer Mitarbeiter und für Wartungsarbeiten entstehen oder die Entwicklung irgendwann im Chaos endet.
- Die Automatisierung von Entwicklungsschritten ist nur möglich, wenn die Code-Organisation auf einheitlichen, definierten Strukturen aufbaut.

Fragestellungen der Code-Organisation

1. Welche Konzepte umfasst die Code-Organisation?
   - Welche Elemente werden betrachtet?
   - Welche Beziehungen gibt es zwischen den Elementen?
2. Wie sind die Zusammenhänge zu den Spezifikationssichten?
   - Wie lassen sich Komponenten und ihre Schnittstellen repräsentieren?
   - Wie lassen sich Abhängigkeitsbeziehungen sicherstellen?
3. Wie sind die Code-Elemente organisiert?
   - Wie werden die Elemente abgelegt?
   - Welche Strukturierungsmöglichkeiten gibt es?
   - Wie lassen sich unterschiedliche Versionen verwalten?

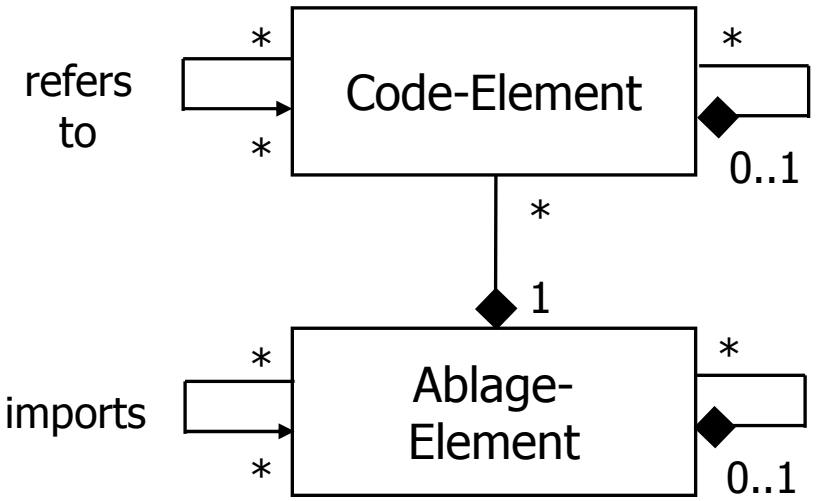
Konzepte der Code-Organisation

- **Code-Elemente** repräsentieren die Beschreibungen, die von den Entwicklern erarbeitet werden, zum Beispiel
  - Modelle unterschiedlicher Art
  - Quellcode, Makros, Skripte
  - Ressource-Blöcke, Icons, Property-Listen

  Der Schwerpunkt liegt dabei wegen ihrer Anzahl und Komplexität auf den implementierungsnahen Beschreibungen.
- **Ablage-Elemente** geben an, wie diese Beschreibungen abgelegt werden. Möglichkeiten sind:
  - Verzeichnisse mit Dateien, beispielsweise
    - Quellcode-Dateien, Modell-Dateien von CASE-Tools
    - Ressource-Dateien und Konfigurationsdateien
  - Repositories mit strukturierten Informationen, beispielsweise
    - Repositories von CASE-Tools

Beziehungen zwischen den Code-Elementen

Metamodell:



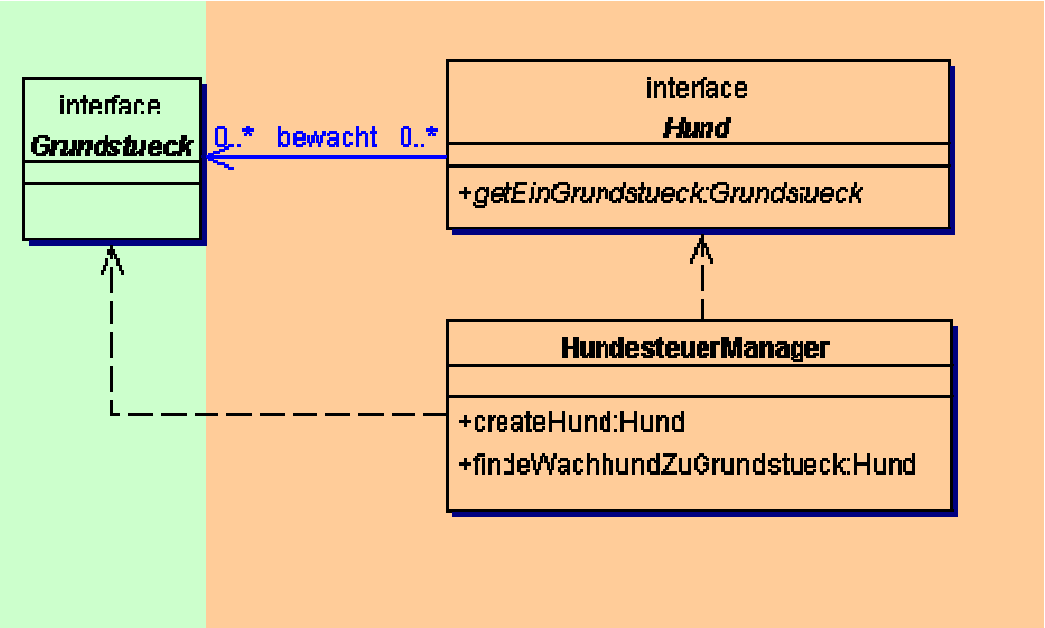
Beispiel:

- Quellcode der Java-Interfaces `A` und `B` mit inherits-Statement.
- Datei `A.java` importiert Datei `B.java`.

Damit sich Code-Elemente in unterschiedlichen Ablage-Elementen aufeinander beziehen können, müssen die entsprechenden Ablage-Elemente einander importieren (und werden voneinander abhängig).

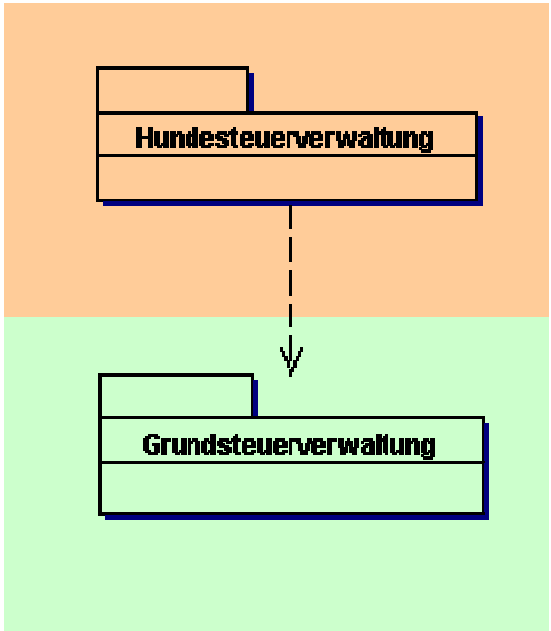
Beispiel eSteuer 2008

Code-Elemente



- Klassen und Assoziationen beziehen sich aufeinander (Nutzung als Parameter, Assoziationsziele)

Ablage-Elemente



- Dateien werden voneinander abhängig

Einfluss der technischen Sicht auf die Code-Organisation

- Die technische Sicht macht bereits Vorgaben für die zu verwendenden Code-Elemente.
  - Moderne objektorientierte Sprachen bieten bereits Standard-Konzepte für die Umsetzung von Komponenten und Schnittstellen.
    - Beispiel: Realisierung von Schnittstellen durch Java-Interfaces
  - Die verwendeten Basismechanismen und Basiskomponenten geben die Realisierung durch Code-Elemente meist bereits vor.
    - Beispiel: Wird als Basistechnik EJB verwendet, so liegen die zu verwendenden Code-Elemente bereits fest.
- Die Ablagestruktur ist in einigen Ansätzen ebenfalls bereits teilweise durch die technische Sicht vorgegeben.
  - Beispiele:
    - Vorgabe, dass jede Klasse in einer eigenen Datei abgelegt wird.
    - In Java spiegelt sich die Package-Hierarchie in der Verzeichnis-hierarchie des Dateisystems wieder.

Technische Realisierung der Ablage

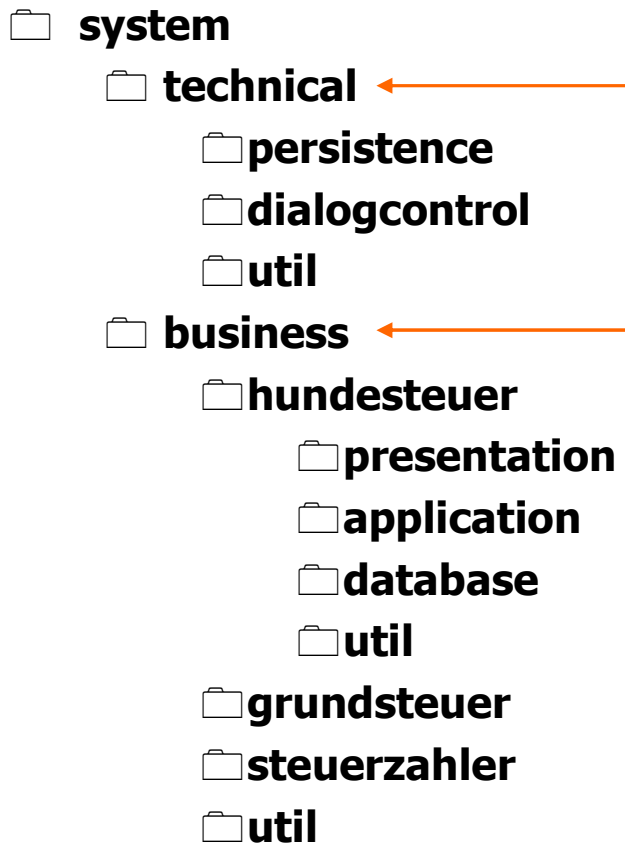
- Dateien
  - Ablage mittels Dateien im Dateisystem, Verzeichnisstrukturen
  - Verarbeitung erfordert Parsen der konkreten Syntax
  - Mechanismen des Dateisystems ausnutzbar (Zugriffsschutz etc.)
  - Basis für Versionsverwaltungssysteme
- Repositories
  - Ablage strukturierter Informationen gemäß erweiterbarem Meta-Modell
  - Direkter Zugriff auf abstrakte Syntax der Code-Elemente möglich
  - Beliebige Zusatzinformationen und Verfahren integrierbar
- Stand der Technik: Allgemein verwendbare Repositories für alle Code-Elemente eines großen Systems existieren noch nicht. Bis auf weiteres ist in den meisten großen Projekten eine Kombination beider Techniken nötig.

In den folgenden Beispielen gehen wir von der Ablage in Dateiform aus.

Beispiele für Fragen bei der Code-Organisation

- Wie ist der Zusammenhang zwischen Code-Elementen und Ablage-Elementen?
  - Wie viele Headerdateien gibt es?
  - Lassen sich gemeinsame Code-Anteile in eigene Dateien auslagern?
  - Wieviele Klassen werden in einer Datei abgelegt?
  - Wofür lassen sich Repositories verwenden? Wie werden sie eingebunden?
- Wie ist die Ablage-Struktur aufgebaut?
  - Wie ist die Verzeichnishierarchie?
  - Wer hat Zugriff auf welche Verzeichnisse und Dateien? Wer ist Eigentümer bestimmter Dateien?
  - Lassen sich Standard-Unterstrukturen definieren?
  - Wie sind die Namenskonventionen für Dateien und Verzeichnisse?
  - Müssen die Dateien an unterschiedlichen Standorten verfügbar sein?

Beispiel: Ablage nach fachlichen Komponenten



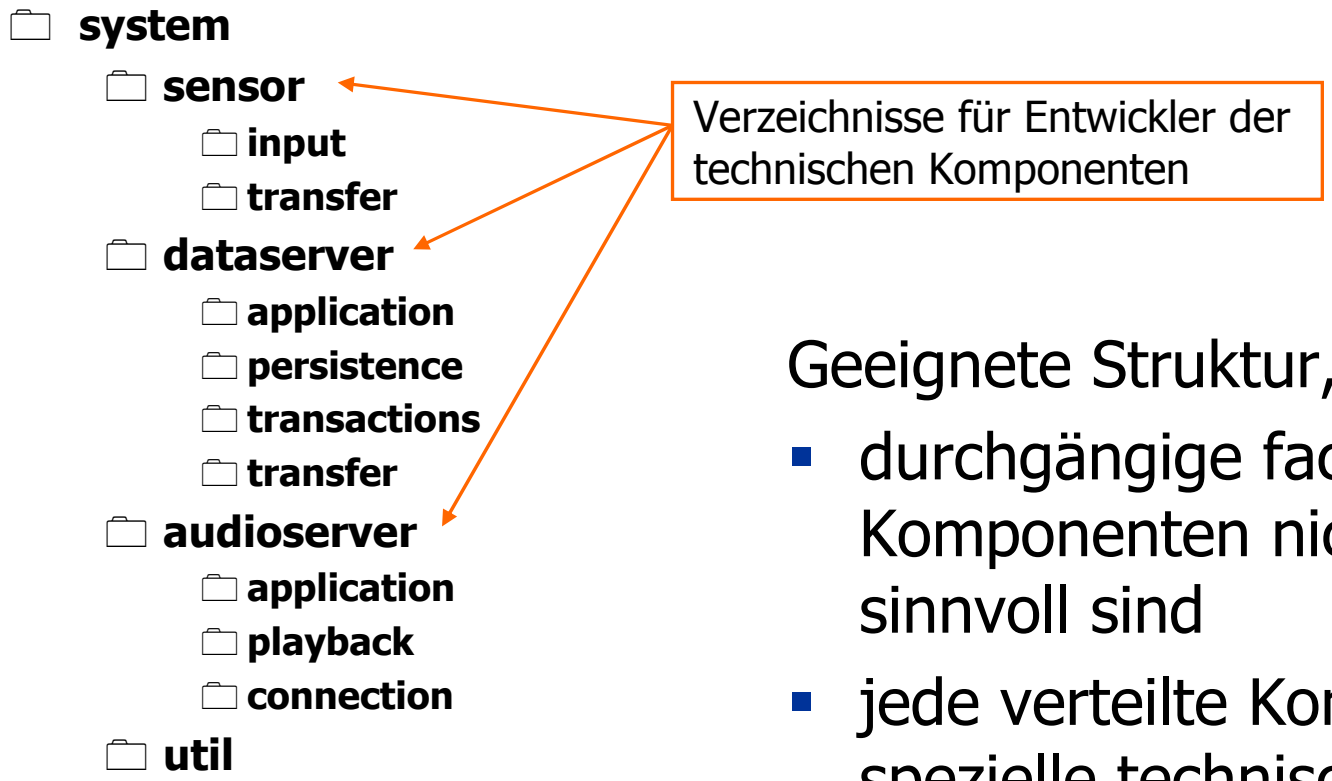
enthält Verzeichnisse für Entwickler der technischen Komponenten

enthält Verzeichnisse für Entwickler der fachlichen Komponenten

Geeignete Struktur, wenn

- durchgängige fachliche Komponenten sinnvoll sind
- eine geeignete Container-Infrastruktur für derartige Komponenten existiert.

Beispiel: Ablage nach technischen Komponenten

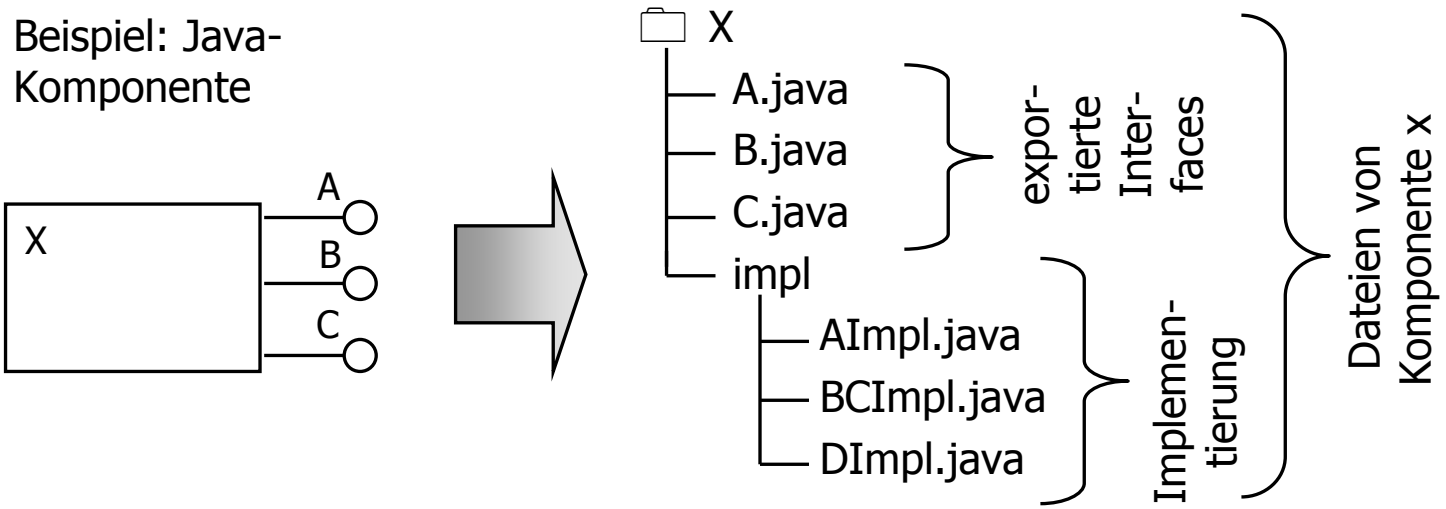


Geeignete Struktur, wenn

- durchgängige fachliche Komponenten nicht sinnvoll sind
- jede verteilte Komponente spezielle technische Anforderungen an die Entwickler stellt.

Ablagestrukturen und exportierte Schnittstellen

- Ziele
  - Entwickler bzw. Teams haben exklusive Verantwortung für bestimmte Komponenten und ihre exportierten Schnittstellen.
  - Andere Teams werden von den Interna entkoppelt und sehen nur die exportierten Schnittstellen dieser Komponenten.
- Standardlösung: jeweils eigenes Export-Verzeichnis



Sicherstellung der Einhaltung von Abhängigkeiten

Einschränkung des Zugriffs von Vornherein

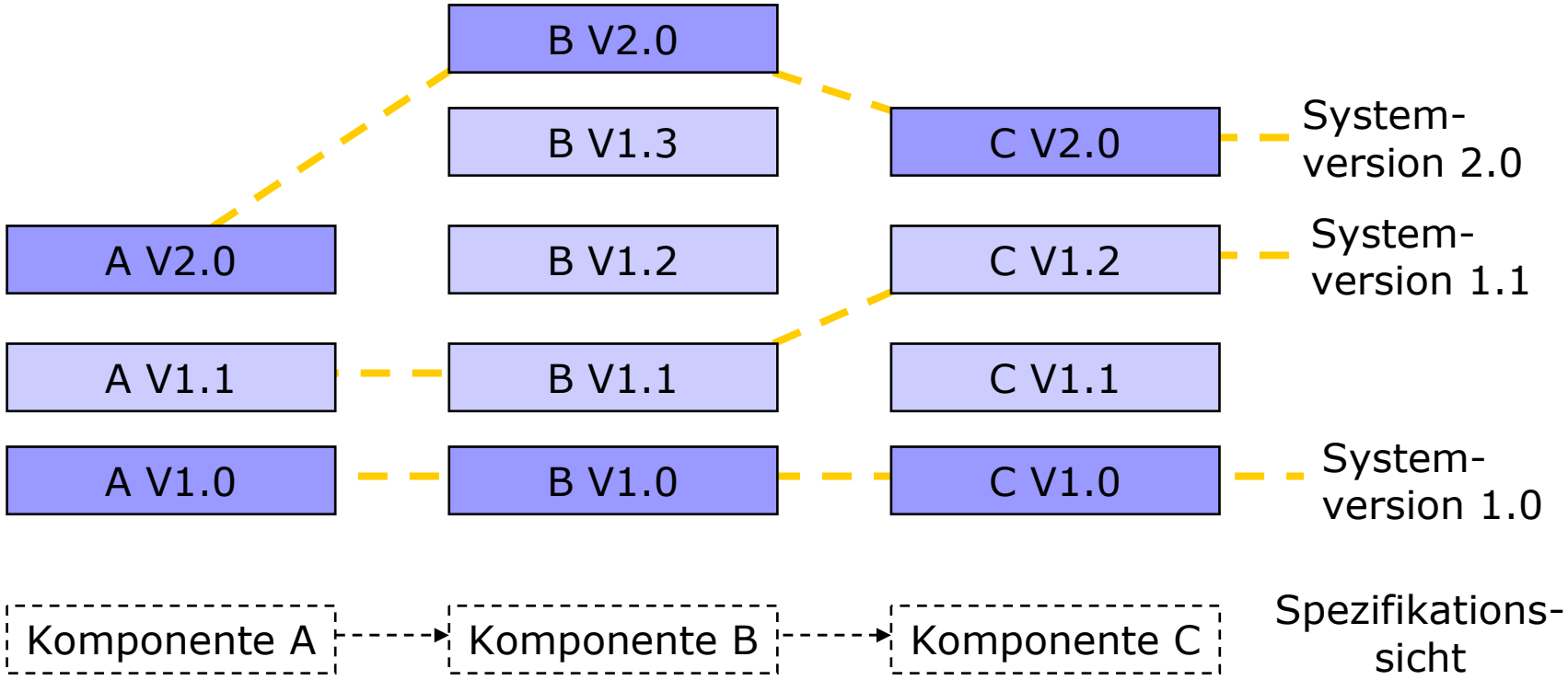
- Die Dateien mit den Code-Elementen einer Komponente werden in einem Verzeichnisbaum abgelegt, auf den nur das zugehörige Team schreibenden Zugriff hat.
- Verzeichnisse mit Code-Elementen der Schnittstellen sind für alle Entwickler von Komponenten lesbar, die die Schnittstellen nutzen.
- Verzeichnisse mit Code-Elementen der interne Implementierung liegen in Verzeichnissen, die nur das zugehörige Team lesen kann.

Überprüfung von Abhängigkeiten im Nachhinein

- Sämtliche Teams haben lesenden Zugriff auf alle Schnittstellen.
- Eine Abhängigkeitsliste dient als Eingabe für ein Werkzeug, das die Abhängigkeiten prüft.
- Die Entwickler erhalten Fehlermeldungen, wenn sie auf unerlaubte Schnittstellen zugreifen.

Versionsmanagement

Sowohl einzelne Komponenten als auch Systeme als Ganzes müssen versioniert werden.



Abhängigkeiten zwischen versionierten Komponenten

- Bei der Betrachtung der Abhängigkeiten zwischen Komponenten müssen die Versionen berücksichtigt werden.
- Beispiel für eine Abhängigkeitsliste mit Versionen:

Komponente	*Version*	*Abhängigkeiten*
A	V2.0	B V1.3
B	V1.3	C V2.0
C	V2.0	-

bei Abwärtskompatibilität sind höhere Versionen ebenfalls erlaubt

- Die Ablage von Versionen kann auf folgende Arten geschehen:
  - explizit über versionsspezifische Unterverzeichnisse für die Ablage-Elemente von Komponenten im Dateisystem
  - durch ein Versionsverwaltungs-Werkzeug

Inhalt

- Rekapitulation
- Entwicklungssicht
  - Überblick und Ziele
  - Modellgetriebene Entwicklung
  - Code-Organisation
  - Build-Management
- Testsicht
- Bereitstellungssicht
- Andere Ansätze
- Zusammenfassung

Motivation für das Build-Management

- Bei großen verteilten Systemen ist die Erzeugung eines Systems ein komplexer Vorgang, bei dem viele Zwischenergebnisse anfallen, verarbeitet und integriert werden müssen.
- Die Build-Prozesse und die verwendeten Werkzeuge müssen sorgfältig entworfen und dokumentiert werden.
- Sind die Build-Prozesse unklar oder nicht genügend automatisiert, so besteht die Gefahr, dass immense Aufwände für die Entwicklung anfallen und neue Systemversionen nur in großen Abständen erstellt werden können.

Fragestellungen beim Build-Management

1. Welche Build-Konzepte gibt es?
   - Welche Elemente werden betrachtet?
   - Welche Beziehungen gibt es zwischen den Elementen?
2. Wie sind die Zusammenhänge zur Code-Organisation?
3. Wie sind die Build-Elemente organisiert?
   - Wie werden die Elemente abgelegt?
   - Welche Strukturierungsmöglichkeiten gibt es?
4. Wie lassen sich Werkzeuge und Prozesse beschreiben?
5. Wie lassen sich komplexe Build-Prozesse entkoppeln?

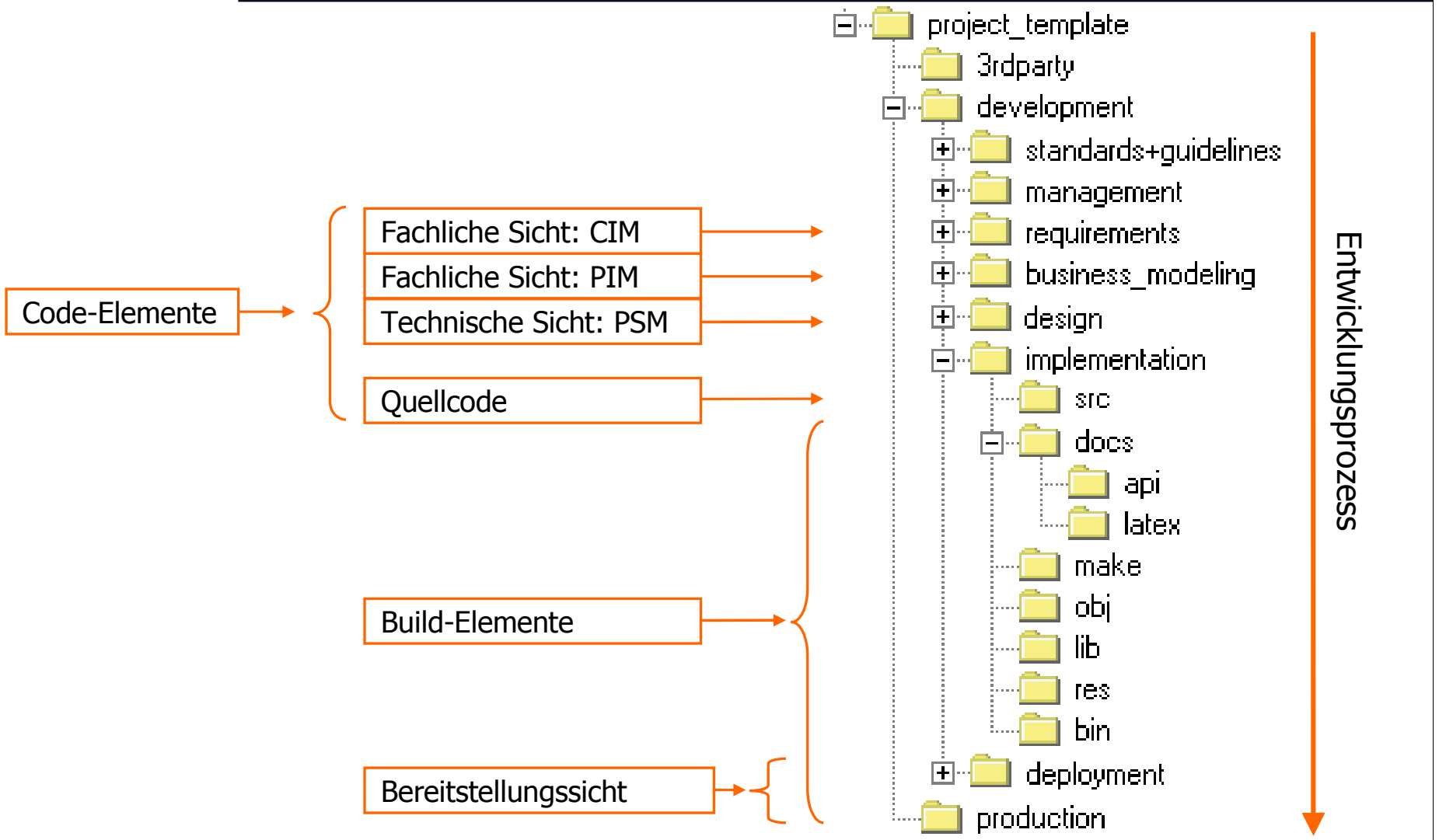
Build-Konzepte

- Das Build-Management beschäftigt sich mit
  - den Ergebnissen und Zwischenergebnissen beim Bau einer lauf- bzw. testfähigen Version einer Komponente bzw. eines Systems
  - den Vorgängen und Entwicklungsprozessen beim Build
  - den dazu verwendeten Werkzeugen
- Beispiele für Build-Elemente
  - generierter Quellcode, generierte Dokumentation (z.B. javadoc), Objektcode, class-Files, Bibliotheken, jar-Files, Project-Files, Executables, Konfigurationsdateien, Datenbank-Image etc.
- Beziehungen zwischen den Build-Elementen
  - generates-Beziehung für (Zwischen-)Ergebnisse (und auch Werkzeuge), parametrisiert durch das verwendete Werkzeug
- Beispiele für Werkzeuge
  - Generatoren, Compiler, Linker, IDEs, makefiles, Skripte etc.

Zusammenhänge zwischen Code- und Build-Elementen

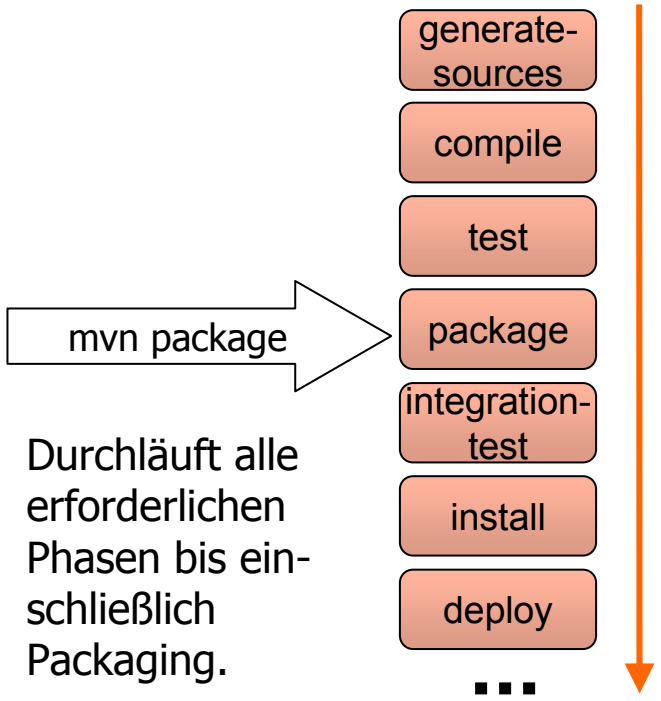
- Die Code-Elemente dienen als Basis für Erzeugung von Build-Elementen.
  - Beispiel: Compilierung von `A.java` zu `A.class`
- Die Ablagestruktur der Build-Elemente ist eine Obermenge der Ablagestruktur der Code-Elemente.
- Die projektspezifischen Werkzeuge werden versioniert.
  - Beispiel: makefiles, compile-Skripte, Skripte zum Starten der Entwicklungs- und Testumgebung
- Spezielle Build-Elemente können ebenfalls versioniert und gespeichert werden.
  - Beispiel: Integrationsversionen von Binärkomponenten könnten versioniert gespeichert werden, um alte Versionen im schnellen Zugriff zu haben und Zeit für ihren Bau zu sparen.

Beispiel für Ablagestruktur: Rational Unified Process [RUP]



Beispiel für Build-Tool: Maven - Standardisierung

- Maven gibt als automatisiertes Build-Tool eine standardisierte Build-Infrastruktur vor.
  - Standardisierte Build-Prozesse (Lifecycles mit Phasen)

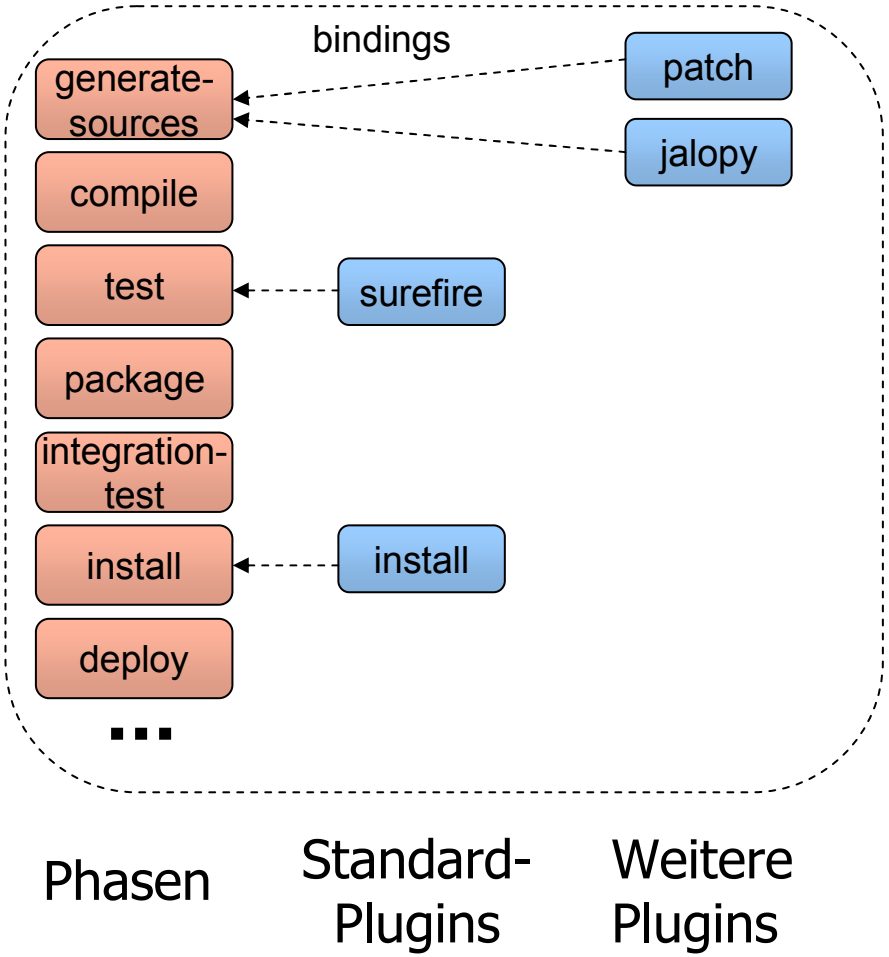


  - Standardisierte Ablagestrukturen

src/main/java	Application/Library sources
src/main/resources	Application/Library resources
src/main/filters	Resource filter files
src/main/assembly	Assembly descriptors
src/main/config	Configuration files
src/main/webapp	Web application sources
src/test/java	Test sources
src/test/resources	Test resources
src/test/filters	Test resource filter files
src/site	Site
LICENSE.txt	Project's license
README.txt	Project's readme

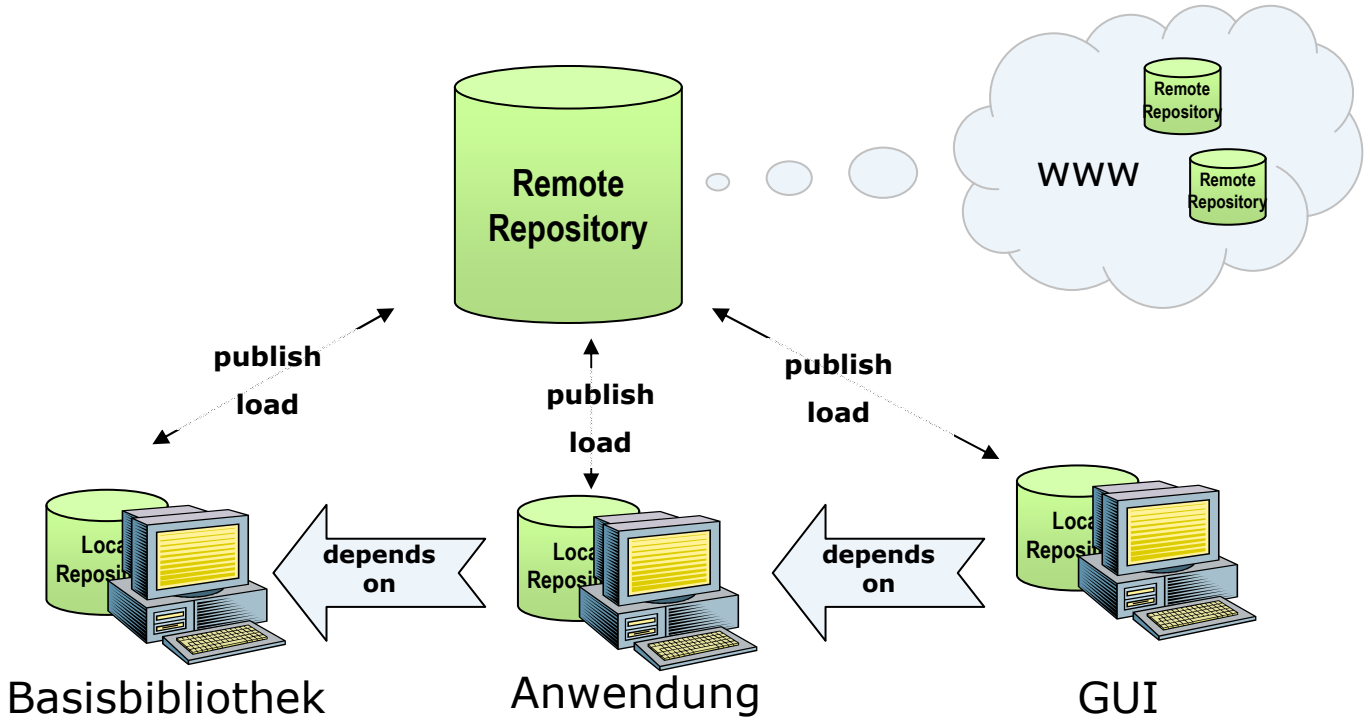
Beispiel für Build-Tool: Maven - Konfiguration

- Jede Phase enthält Ziele, in denen Arbeit verrichtet wird
  - Beispiel: Aufruf des Java-Compilers in Phase *compile*
- Welche Ziele verwendet werden, ergibt sich aus einer formalen Beschreibung des Projekts
  - Project Object Model (POM) in Form einer XML-Datei
- Die Build-Prozesse zu den Zielen lassen sich über konfigurierbare Plugins beschreiben und ausführen.



Beispiel für Build-Tool: Maven – Dependency Management

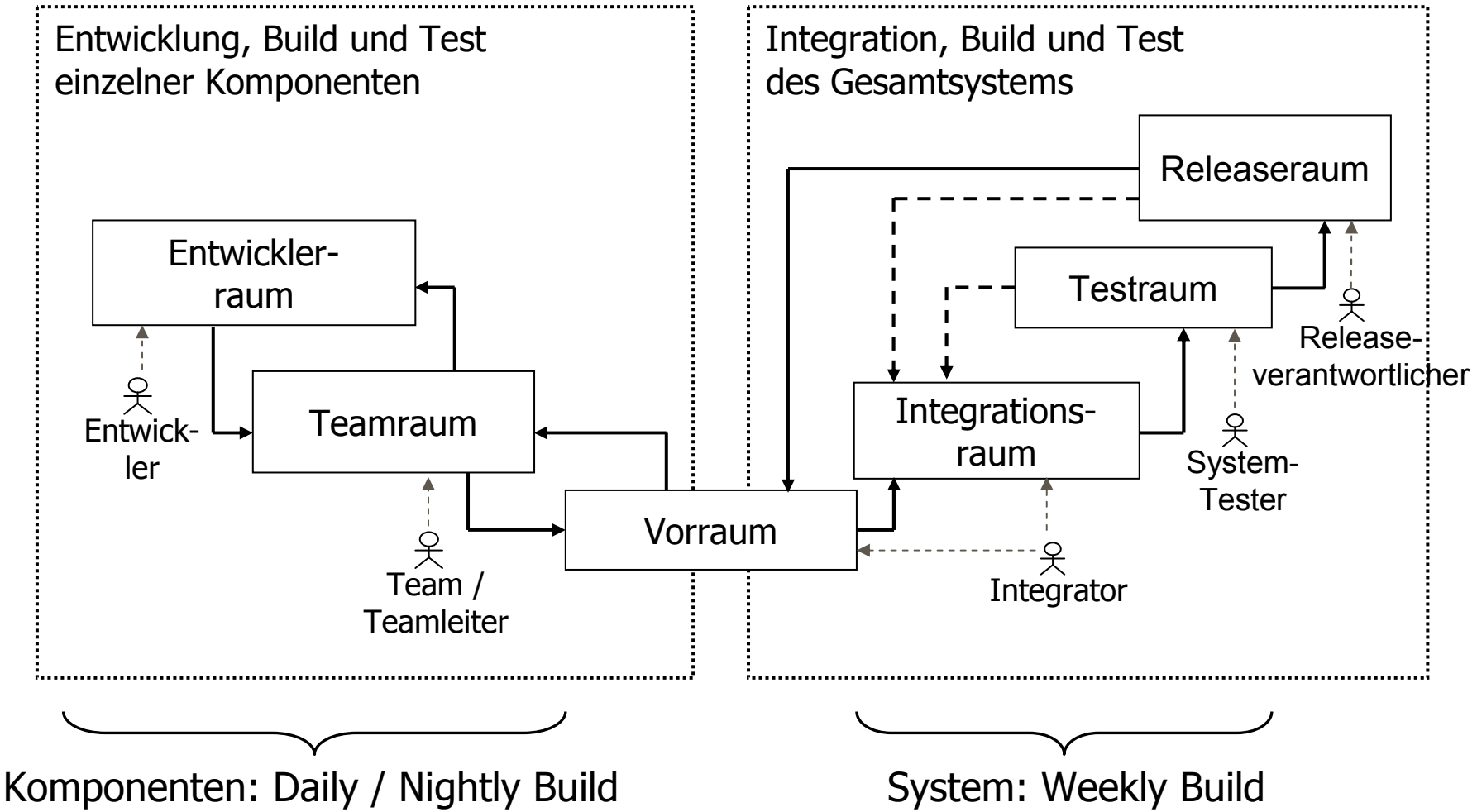
- Maven verwaltet Abhängigkeiten zwischen Code-Elementen (Artefakten).
- Projekte müssen sich nicht darum kümmern, jeweils aktuelle Versionen benötigter Bibliotheken etc. zu beziehen, sondern bekommen sie über gemeinsam bekannte Repositories automatisch.



Entkopplung durch separate Räume

- Beim Bau großer Systeme sind typischerweise mehrere Entwickler in unterschiedlichen Rollen beteiligt.
- Diese können durch Räume voneinander entkoppelt werden.
  - Ein Raum unterstützt einen bestimmten Teil des Build-Prozesses und isoliert den jeweiligen Eigentümer des Raums von der Arbeit anderer Entwickler oder Teams. Änderungen in einem anderen Raum betreffen ihn somit nicht.
  - Die Übergabe von Ergebnissen aus einem Raum in einen anderen muss klar geregelt sein. Gegebenenfalls ist dabei die Integration der entkoppelt entwickelten Ergebnisse nötig (Merge).
- Räume können unterschiedlich realisiert sein:
  - Nutzung separater Verzeichnisse im Dateisystem, Übergabe durch Kopieren bzw. Verschieben von Build-Elementen
  - Nutzung der Mechanismen eines Versionsmanagement-Systems (z.B. Check-In/Out)

Beispiel für Räume



Inhalt

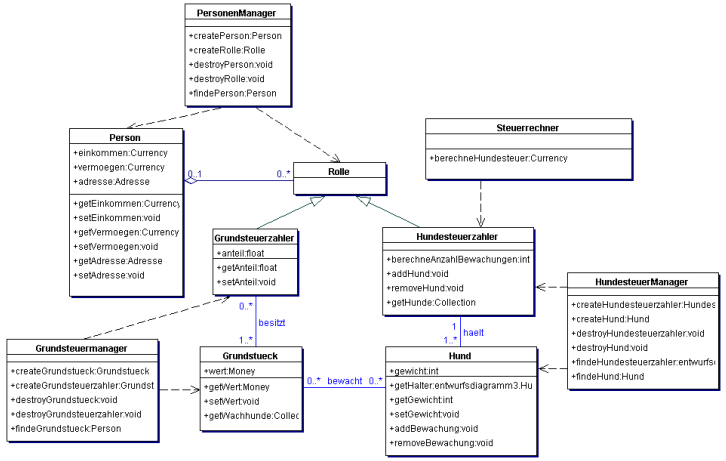
- Rekapitulation
- Entwicklungssicht
  - Überblick und Ziele
  - Modellgetriebene Entwicklung
  - Code-Organisation
  - Build-Management
- Testsicht
- Bereitstellungssicht
- Andere Ansätze
- Zusammenfassung

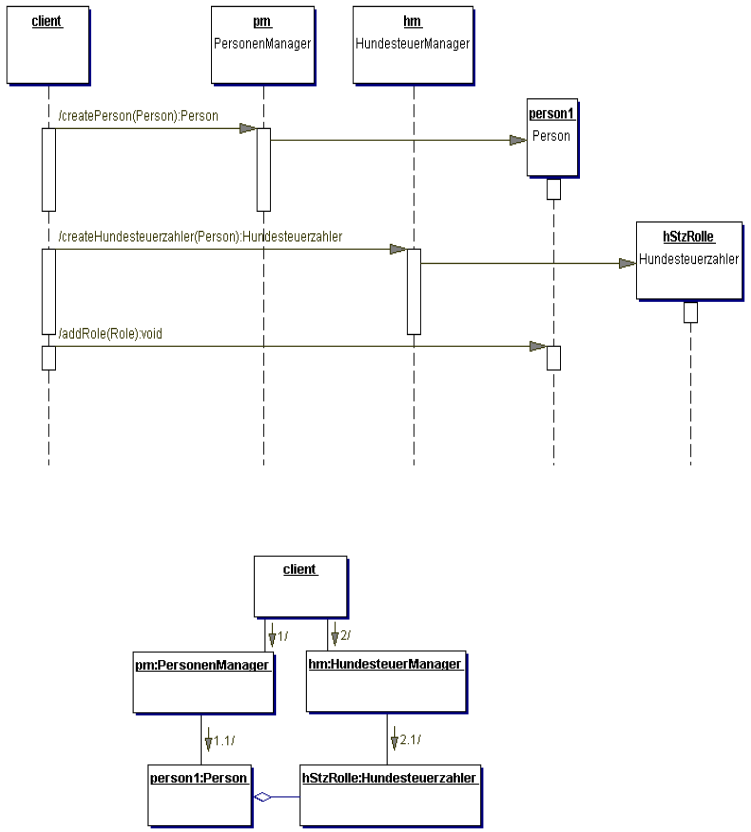
Motivation für die Test-Sicht

- Bei großen verteilten Systemen ist der Test eines Systems ein komplexer Vorgang. Die zwei Hauptaufgaben beim Entwurf der Test-Sicht sind:
  - Entwurf und Realisierung einer umfassenden Test-Suite für das System und seine Komponenten
  - Entwurf einer Testumgebung mit aufeinander abgestimmten Testwerkzeugen und der dazugehörigen Testprozesse
- Die Test-Prozesse und die verwendeten Werkzeuge müssen sorgfältig entworfen und dokumentiert werden.
  - Sind die Test-Prozesse unklar oder nicht genügend automatisiert, so besteht die Gefahr, dass immense Aufwände für Tests anfallen oder die Tests vernachlässigt werden.
- Automatisierte Regressionstests erleichtern Wartung und Weiterentwicklung eines Systemen, da schnell geprüft werden kann, ob eine Änderung auch unerwünschte Nebeneffekte hat.

Entwurf von Testfällen

- Für die Operationen im Klassen-diagramm werden Abläufe durchgespielt und beispielhafte Objektstrukturen gebildet.

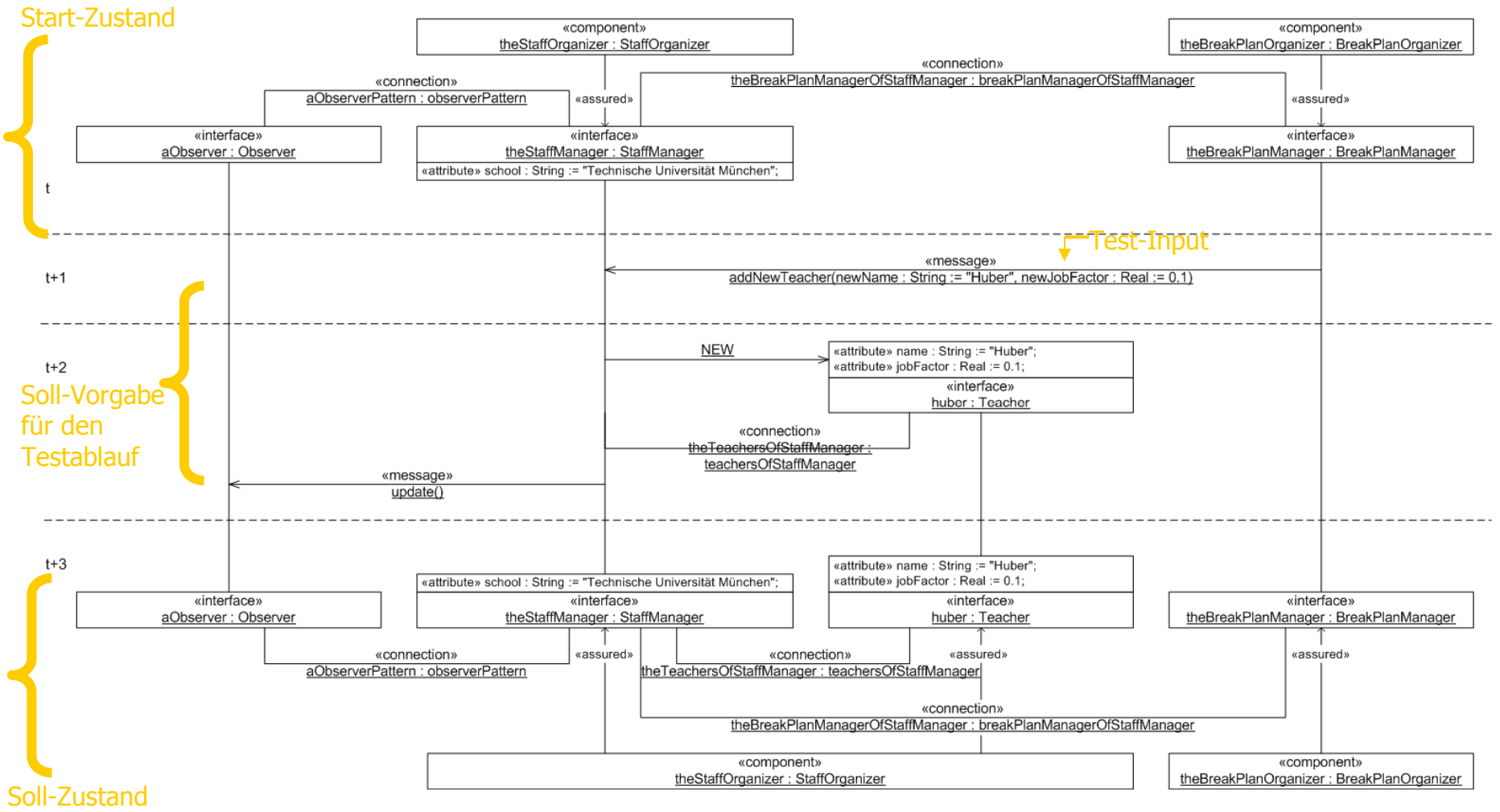




Klassen zur Modellierung

Beispiel-Instanzen zur Validierung

Beispiel: Spezifikation eines Testfalls in UML



Zusammenhänge zwischen Test- und Spezifikationssicht

- Für Black-Box-Tests müssen in der Spezifikationssicht Schnittstellen vorhanden sein, gegen die getestet wird.
- Gegebenenfalls kann es sinnvoll sein, für Testzwecke zusätzliche Schnittstellen in der Spezifikationssicht einzuführen.
  - Beispiel: Bei einem betrieblichen Informationssystem ist der interne Datenbank-Zustand vor und nach Ausführung von Operationen relevant. Will man in Testfällen auf ihn zugreifen, müssen eigene Testmechanismen und Schnittstellen (Dump DB/Load DB) bereit gestellt werden.
- Für die Betrachtung von Abläufen (z.B. im Rahmen von Code-Überdeckungsmessungen) muss in der technischen Sicht ein Tracing-Mechanismus vorgesehen sein.
- Gegebenenfalls müssen für Testzwecke einfachere Varianten des Systems entworfen werden, um Aufwände beim Build bzw. Ressourcen beim Testlauf zu sparen.
  - Beispiel: Zum Test der Geschäftslogik wird ein einfacher Container entwickelt, der einen schnellen lokalen Test erlaubt.

Zusammenhänge zwischen Test- und Entwicklungssicht

- Beziehungen zur Code-Organisation
  - Der Code für die Testfälle sowie Test Inputs und Expected Results müssen standardisiert geschrieben und in Form von Test-Suiten strukturiert abgelegt werden.
- Beziehungen zum Build-Management
  - Die Testfälle müssen übersetzt und abgespielt werden.
  - Die Testumgebung sowie das System (bzw. einzelne zu testende Komponenten) müssen gestartet, initialisiert und wieder entsorgt werden. Gegebenenfalls kann das im Rahmen des Build-Prozesses automatisch geschehen (Nightly Build / Nightly Test).
  - Die Testergebnisse (Teststatus, Testergebnis, zugehöriger Ablauf) müssen als Zwischenergebnis aufbewahrt und den verantwortlichen Entwicklern zugänglich gemacht werden.

Inhalt

- Rekapitulation
- Entwicklungssicht
  - Überblick und Ziele
  - Modellgetriebene Entwicklung
  - Code-Organisation
  - Build-Management
- Testsicht
- Bereitstellungssicht
- Andere Ansätze
- Zusammenfassung

Bereitstellungssicht (Deployment View) - Überblick

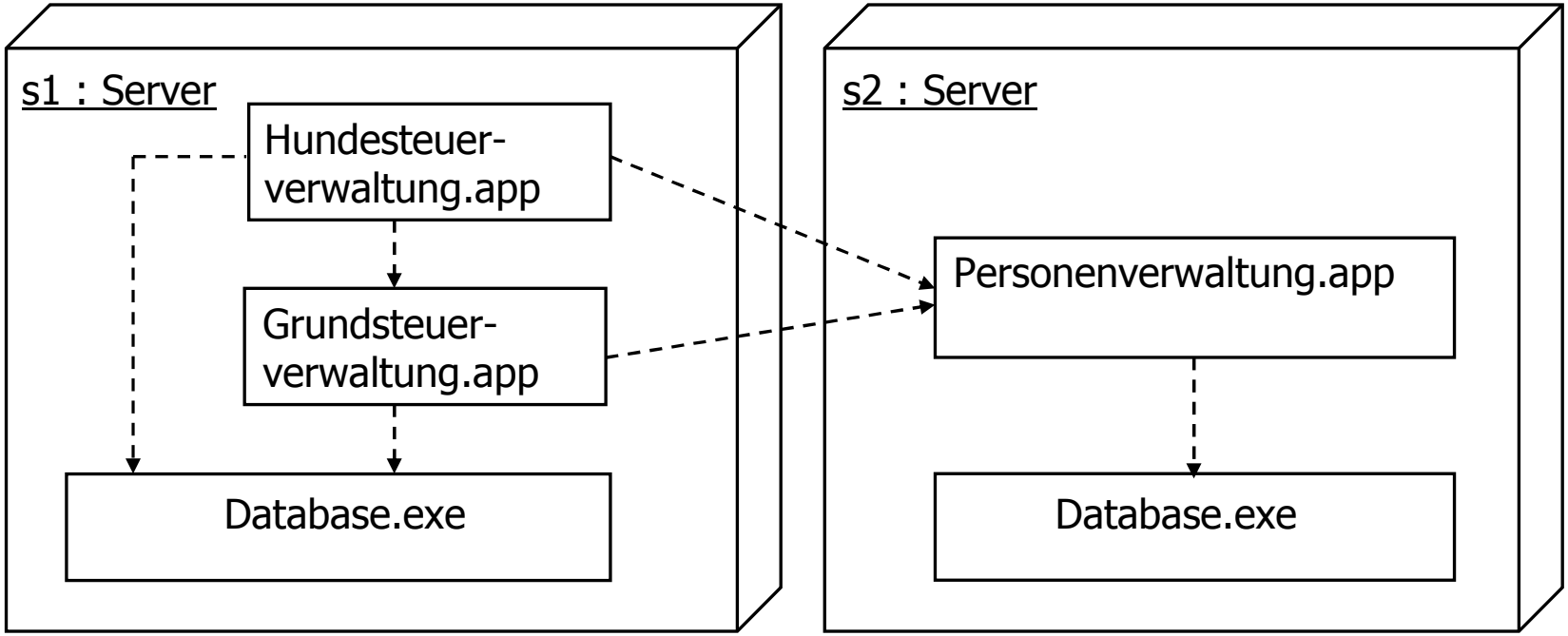
- Zweck
  - Darstellung der auszuliefernden Bestandteile des Systems
- Inhalte
  - ausführbare Beschreibungen des Systems (insbesondere Binärcode)
  - Ablage der Beschreibungen auf Hardware-Komponente
  - Prozesse bei der Konfiguration und Installation
- Beschreibung
  - Deployment-Diagramme von UML
  - Aktivitätsdiagramme für Deployment-Abläufe
  - textuelle Beschreibungen z.B. von Ablagestrukturen und Konfigurationsdateien

Ziele in der Bereitstellungssicht

- Minimierung der Kosten für Installation, Konfiguration und Updates des Systems als wichtiger Anteil an den Total Costs of Ownership (TCO)
- Klärung der Prozesse bei der Auslieferung und Installation von Software
- Entwurf von möglichst komfortablen und zuverlässigen Mechanismen für Installation und Update von Software
  - Möglichkeiten beispielsweise: Installer, Archiv, Quellcode, Netz-Installation, Software-Verteilungstool, Applet, Webstart
- Reibungsloses Zusammenspiel mit dem Betrieb
  - Anbindung an das Asset- und Lizenz-Management
  - Klare Beschreibung als Vorgabe für Administratoren

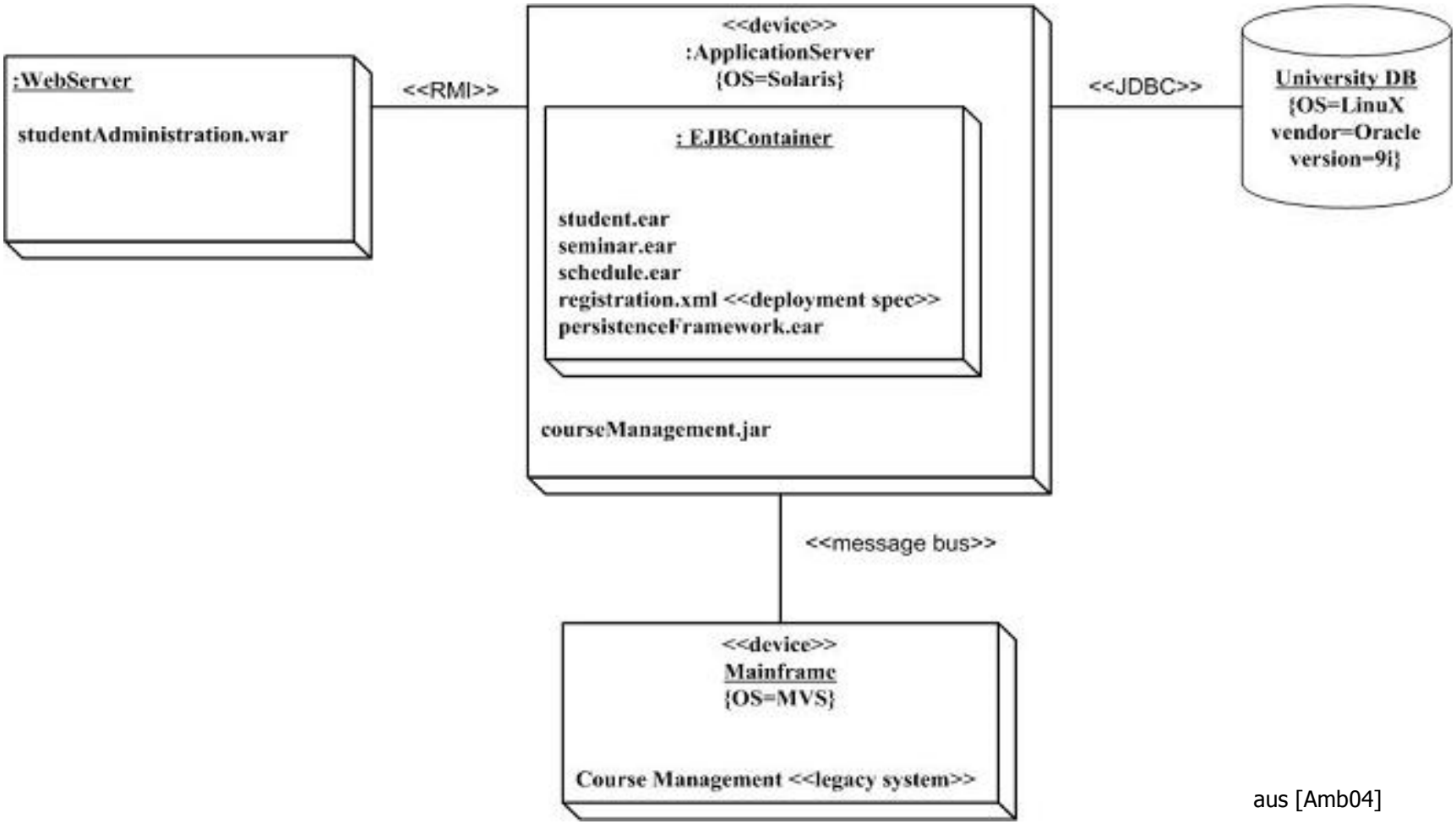
Bereitstellungssicht: Beispiel

- Die Bereitstellungssicht zeigt die Verteilung der Beschreibungen, die für den Ablauf des Systems nötig sind, auf Hardware-Komponenten.
- Beispiel eSteuer 2006:



Im Unterschied zur Verteilungssicht zeigt die Bereitstellungs-Sicht keine Instanzen!

Deployment Diagram in UML2

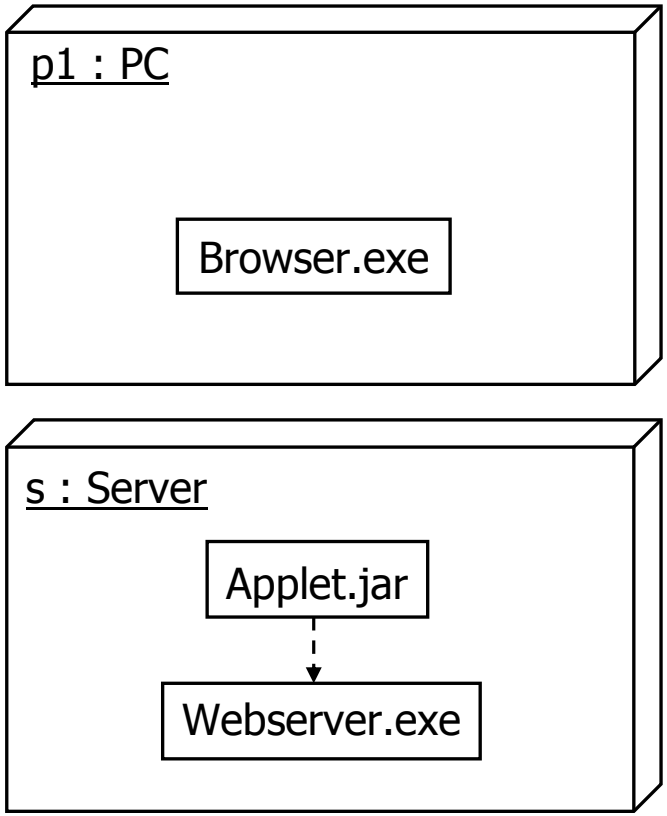


aus [Amb04]

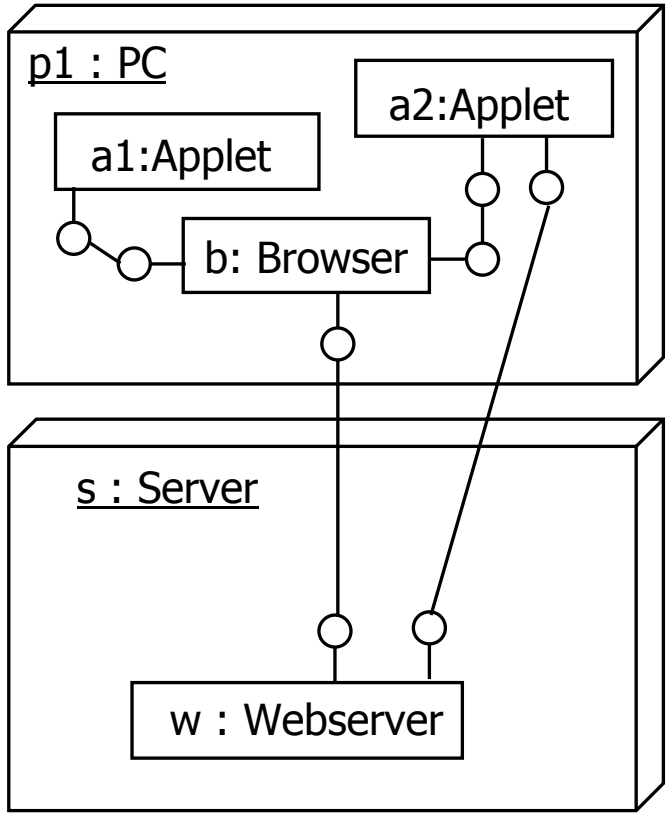
Verteilungs- und Bereitstellungssicht am Beispiel

Beispiel Applet

- Bereitstellungssicht



- Verteilungssicht



Zusammenhänge zu den anderen Sichten

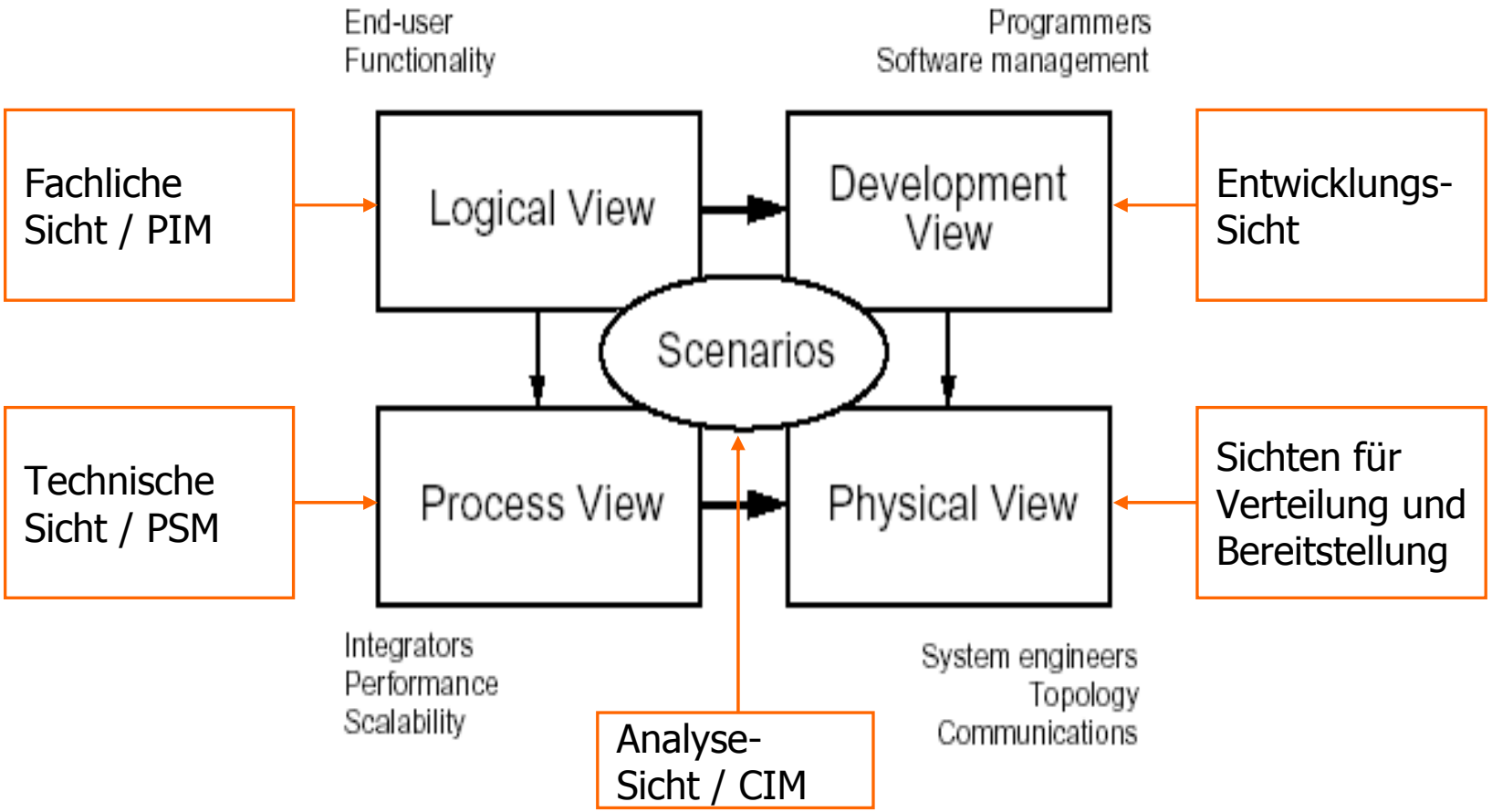
- Die Elemente der Bereitstellungssicht werden als (End-)Ergebnisse der Build-Prozesse erstellt.
- In der technischen Sicht müssen die Mechanismen für Installation und Konfiguration entworfen werden.
- In der Testsicht müssen Testfälle für die Installation und Konfiguration erstellt werden.
- Zum Ablauf von Testfällen sind gegebenenfalls im Rahmen von Test Setup und Test Teardown eigene, angepasste Deployment-Mechanismen erforderlich.

Inhalt

- Rekapitulation
- Entwicklungssicht
  - Überblick und Ziele
  - Modellgetriebene Entwicklung
  - Code-Organisation
  - Build-Management
- Testsicht
- Bereitstellungssicht
- Andere Ansätze
- Zusammenfassung

Andere Klassifikationsschemata

Rational 4+1 Views:



Andere Klassifikationsschemata

- Siemens 4 Views [HNS99]
  - Conceptual View ← Fachliche und Technische Sicht (Instanzen)
  - Module View ← Fachliche und Technische Sicht (Typen)
  - Execution View ← Verteilungssicht
  - Code View ← Entwicklungs- und Bereitstellungssichten

- DSA-Views [CBB02]
  - Module Viewtype ← ≈ Fachliche und Technische Sicht (Typen)
  - Component-and-Connector Viewtype ← ≈ Fachliche und Technische Sicht (Instanzen)
  - Allocation Viewtype ← ≈ Entwicklungs- und Bereitstellungssichten

Inhalt

- Rekapitulation
- Entwicklungssicht
  - Überblick und Ziele
  - Modellgetriebene Entwicklung
  - Code-Organisation
  - Build-Management
- Testsicht
- Bereitstellungssicht
- Andere Ansätze
- Zusammenfassung

Zusammenfassung

- Eine Sicht stellt jeweils bestimmte Eigenschaften eines Systems dar. Es gibt zwei Hauptarten von Sichten: Spezifikationssichten und Erstellungssichten.
- Die Erstellungssichten beschreiben die Ergebnisse, Werkzeuge und Prozesse, die für die Entwickler eines Systems relevant sind.
- Die Elemente der Entwicklungs-, Test- und Bereitstellungssicht müssen ebenso sorgfältig entworfen werden wie das System selbst.

Literaturhinweise

- [Amb04] Scott W. Ambler: *DThe Object Primer 3rd Edition*, Cambridge University Press, 2004.
- [CBB02] Paul Clements, Felix Bachmann, Len Bass, David Garlan, James Ivers, Reed Little, Robert Nord, Judith Stafford: *Documenting Software Architectures – Views and Beyond*, Addison-Wesley, 2002.
- [HNS99] Christine Hofmeister, Robert Nord, Dilip Soni: *Applied Software Architecture*, Addison Wesley – Object Technology Series, 1999.
- [HS99] Peter Herzum, Oliver Sims. Business Component Factory: *A Comprehensive Overview of Component-Based Development for the Enterprise*, John Wiley & Sons, 1999.
- [KWB03] Anneke Kleppe, Jos Warmer, Wim Bast: *MDA Explained*, Addison-Wesley, 2003.

Literaturhinweise

- [MSU03] Stephen J. Mellor, Kendall Scott, Axel Uhl, Dirk Weise: *MDA Distilled*, Addison-Wesley, 2004.
- [OMG05] Object Management Group Website, www.omg.org, 2005.
- [RUP] Rational Software Corporation: *Rational Unified Process*, Rational 2002.
- [TG01] Qiang Tu, Michael W. Godfrey: *The Build-Time Software Architecture View*, Proceedings of 2001 Intl. Conference on Software Maintenance (ICSM 2001), Florenz, November 2001.
- [UML05] Chris Rupp, Jürgen Hahn, Stefan Queins, Mario Jeckle, Barbara Zengler: *UML2 glasklar*, Hanser 2005.