

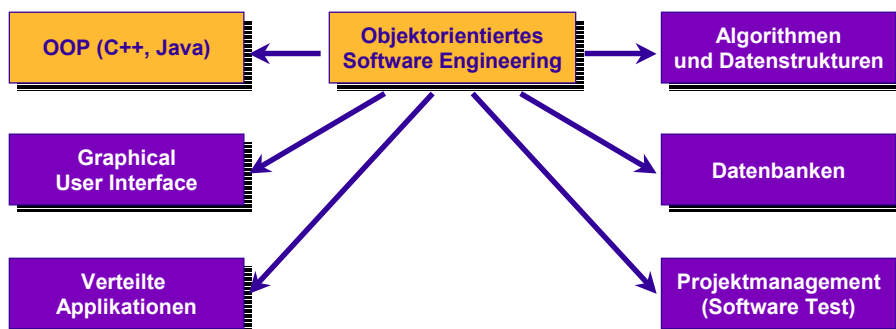
Objektorientiertes Software Engineering

Version 1.3

Eine Einführung mit UML und
Rational Unified Process

1-1

Kurskontext



1-2

Kursinhalt

- **Teil 1 - Einführung (Objektorientierte Konzepte)**
- **Teil 2 - Unified Modeling Language Grundlagen**
- **Teil 3 - Software-Entwicklungsprozess**
- **Teil 4 - Design Patterns**
- **Teil 5 - Schlussbetrachtung (UML Tools)**

1-3

Literatur

- **The Unified Modeling Language User's Guide**
Grady Booch, James Rumbaugh, Ivar Jacobson
ISBN 0-201-57168-4 (Addison Wesley 1999)
- **The Unified Modeling Language Reference Manual**
James Rumbaugh, Ivar Jacobson, Grady Booch
ISBN 0-201-30998-X (Addison Wesley 1999)
- **The Unified Software Development Process**
Ivar Jacobson, Grady Booch, James Rumbaugh
ISBN 0-201-30998-X (Addison Wesley 1999)
- **UML Distilled**
Martin Fowler with Kendall Scott
ISBN 0-201-32563-2 (Addison Wesley 1997)
- **Objektorientierte Software-Entwicklung**
Bernd Oestereich
ISBN 3-486-24787-5 (Oldenburg Verlag 1998)
- **Design Patterns**
Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides
ISBN 0-201-63361-2 (Addison Wesley 1995)

1-4

WWW-Ressourcen

- www.omg.org Object Management Group
- www.rational.com Firma Rational
- www.cetus-links.org Links zu “Objects & Components”
- www.hillside.net/patterns Patterns Home Page
- sunset.usc.edu Center for Software Engineering
- www.esi.es European Software Institute
- www.sei.cmu.edu Software Engineering Institute
- www.saq.ch/software Schweizerische Arbeitsgemeinschaft für Qualitätsförderung (SAQ)

1-5

Teil 1 Einführung

Version 1.1

1-6

Teil 1 - Inhalt

- **Geschichte des Software-Engineering**
- **Strukturiertes Paradigma**
- **Objektorientiertes Paradigma**
- **Objektorientierte Konzepte**

1-7

Software Engineering: Planung von Software

- **Planung der Entwicklung (Software-Entwicklungsmethodik)**
 - Beispiel: Unified Modeling Language (UML)
- **Planung des Vorgehens (Software-Entwicklungsprozess)**
 - Beispiel: Rational Unified Process (RUP)
- **Nutzen einer Software Planung:**
 - Gemeinsame Sprache aller Beteiligten
 - Vereinfachte Darstellung komplexer Systeme
 - Bewältigung zunehmender Systemkomplexität
 - Methodisches Vorgehen erlaubt eine Ressourcen- und damit Kosten-Abschätzung
 - Laufende Dokumentation der Software

Geschichte des Software Engineering

I

- Die Geschichte des Software Engineering ist eng verknüpft mit der Entwicklung der Programmiersprachen und umgekehrt.
- **Die Anfänge der Software-Entwicklung**
 - Verwendung keiner Methodik
 - Implementationsorientierte Sichtweise (Bottom-Up)
 - Keine Abstraktion (**Assembler**-Programmierung)
 - Bessere Abstraktion durch die Entwicklung von "höheren" Programmiersprachen
 - Systementwicklung durch direkte Implementation („goto“ Spaghetti-Code)
- **Beispiel**
 - **FORTRAN** (Formula Translation, IBM 1954)

1-9

Geschichte des Software Engineering

II

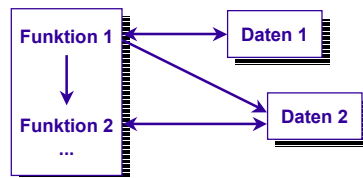
- **Software Crisis**
 - Erreichen der Komplexitätsgrenze dieses unmethodischen Vorgehens (NATO Software Conference, 1968)
 - Problem der Zeit- und Kostenüberschreitung sowie mangelnder Software-Qualität
 - Notwendigkeit zur Verwendung von Ingenieurprinzipien bei der Software-Entwicklung
 - Entstehung des Software-Engineering
 - Entwicklung neuer **strukturierter Programmiersprachen** mit entsprechender Software-Entwicklungsmethodik
- **Beispiele**
 - **PASCAL** (N. Wirth, ETHZ, 1971)
 - **C** (D. Ritchie, 1972)

1-10

Strukturiertes Paradigma

- **Strukturierte Methode der Software-Entwicklung**

- Funktions- und datenorientierte Sichtweise (Top-Down)
- Gute Abstraktion durch Einführung von abstrakten Datentypen
- Systementwicklung durch funktionale Dekomposition und schrittweise Verfeinerung von Algorithmen und Datenstrukturen
- Modellierung der realen Welt mittels linearer Kausalitätsketten



- **Beispiele**

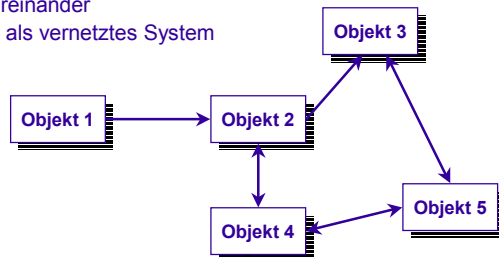
- Jackson Structured Programming (1975)
- Yourdon Structured Analysis and Design (1979)

Objektorientiertes Paradigma

I

- **Objektorientierte Methode der Software-Entwicklung**

- Entwicklung objektorientierter Programmiersprachen
- Ganzheitliche Sichtweise (keine Trennung von Funktionen und Daten)
- Weiterentwicklung abstrakter Datentypen (Klassenkonzept)
- Systementwicklung durch Modellierung mittels Objekten und deren Interaktionen untereinander
- Modellierung der realen Welt als vernetztes System



- **Beispiele**

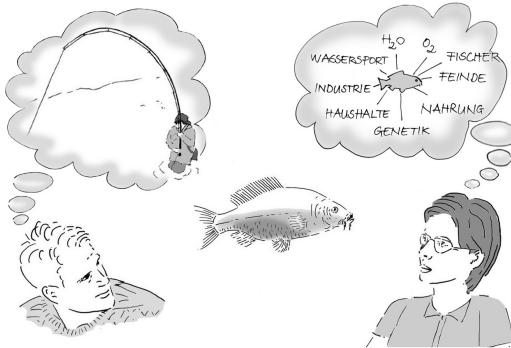
- C++ (B. Stroustrup, 1983)
- Booch Method (1991)
- OMT (Object Modeling Technique, Rumbough, 1994)
- UML (Unified Modeling Language, 1997)

- **Vorteile der objektorientierten Methodik**
 - Objektorientierte Sichtweise lässt sich einfacher in die reale Welt abbilden
 - Bessere Handhabung grosser Systemkomplexität
 - Vereinfachte Kommunikation zwischen Benutzer und Entwickler
 - Durchgängige Modellierung begünstigt Qualität und Konsistenz der Software
 - Objektorientierte Modelle sind stabiler, Änderungen meist lokal begrenzt
 - Modellstabilität vereinfacht Wartung und Erweiterung
 - Komponentenbasierte Entwicklung erhöht die Software-Wiederverwendbarkeit
- **Nachteile der objektorientierten Methodik**
 - Modellierungsbruch bei Datenspeicherung in einer relationalen Datenbank (Notwendigkeit einer Modellkonvertierung oder Verwendung einer OODB)
 - Unter Umständen eine etwas langsamere Runtime-Performance
- **Auch objektorientierte Software-Entwicklung ist keine Garantie für erfolgreiche Software-Projekte!**

Objektorientierte Konzepte

- **Evolutionäre Konzepte als Resultat sukzessiver Erhöhung des Abstraktionsniveaus**
- **Erweiterte Konzepte traditioneller Programmiersprachen**
 - Abstraktion: Klassenbildung und Erzeugung von Objekten
 - Kapselung: Definition der Sichtbarkeit (Zugriffsrechte)
 - Modularisierung: Software-Komponenten
 - Typenprüfung: Prüfung der Klassenkompatibilität
- **Neue Konzepte objektorientierter Programmiersprachen**
 - Generizität: Definition generischer (parametrisierter) Datentypen
 - Hierarchie: Definition hierarchischer Beziehungen zwischen Klassen
 - Vererbung: Übertragung von Eigenschaften an abgeleitete Klassen
 - Polymorphismus: Möglichkeit Funktionen mehrfach zu definieren

OO-Konzept: Abstraktion



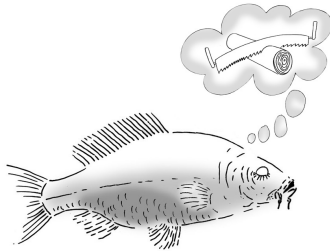
- Unter **Abstraktion** versteht man eine vereinfachte Abbildung der Realität, relativ zur Perspektive des Betrachters.
- Die unterschiedliche Interpretation einer Abstraktion durch verschiedene Personen, kann in einem Informatikprojekt zu grossen Schwierigkeiten führen.
- Die Verwendung eines **Project Dictionary** erleichtert die Definition einer gemeinsamen und genauen Projektsprache.

OO-Konzept: Klasse und Objekt



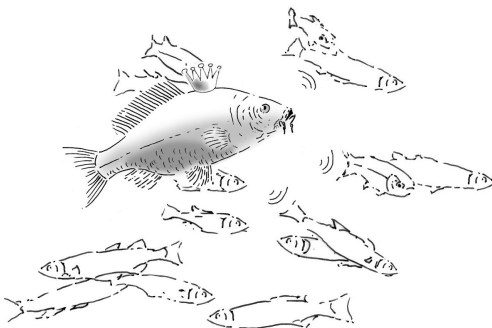
- Eine **Klasse** definiert eine Abstraktion eines Gegenstandes, einer Beziehung oder einer sonstigen Sache ohne reale Manifestation.
- Die Klasse entspricht einem abstrakten Datentyp, welcher aber nicht nur Daten, sondern auch die auf diesen Datentyp anwendbaren Operationen enthält.
- Ein **Objekt** ist die konkrete Realisierung (Instanz) einer Klasse und existiert zur Laufzeit des Programms.
- Ein konkretes Objekt ist definiert durch Objektstatus, Objektidentität und Objektverhalten.

OO-Konzept: Objektstatus



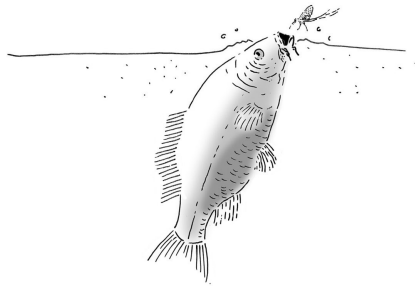
- Der **Status** eines Objektes zu einem bestimmten Zeitpunkt ist definiert durch seinen Informationsinhalt.
- Die Informationen (Daten) eines Objektes werden in **Attributen** abgelegt. Die *momentanen* Werte dieser Attribute definieren den Objektstatus.
- Die Attribute eines Objektes können dabei aus beliebigen Datenstrukturen bestehen (z.B. Text, Listen, Objekte etc.).

OO-Konzept: Objektidentität



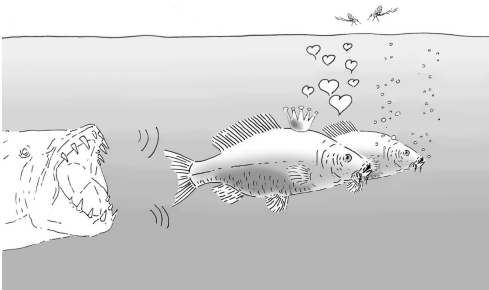
- Unter der **Identität** eines Objektes versteht man die eindeutige Identifikation jedes einzelnen Objektes einer Klasse.
- Die eindeutige Identifikation muss auch dann gewährleistet sein wenn sämtliche Attributwerte zweier Objekte übereinstimmen.

OO-Konzept: Objektverhalten



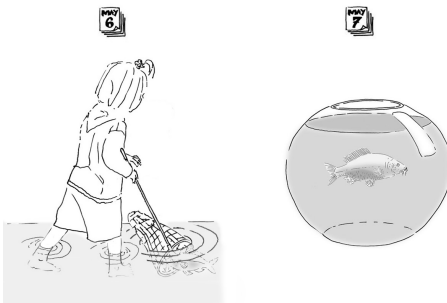
- Das **Verhalten** eines Objektes wird definiert durch seine zur Verfügung gestellten **Operationen** (Funktionen).
- Ein Objekt kann gleichzeitig eine einzelne Funktion (single-tasking) oder mehrere Funktionen (multi-tasking) ausführen.

OO-Konzept: Gleichzeitigkeit



- **Gleichzeitigkeit** (Concurrency) definiert die Möglichkeit der Verrichtung mehrerer Aktionen zum gleichen Zeitpunkt.
- UML erlaubt es Gleichzeitigkeit zu modellieren und bestimmte objekt-orientierte Programmiersprachen (z.B. Java) beinhalten die Möglichkeit diese auch direkt zu implementieren.

OO-Konzept: Persistenz

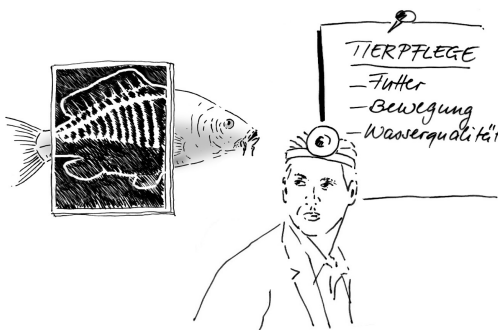


- Unter **Objektpersistenz** versteht man die dauerhafte und vollständige Speicherung (**Serialisierung**) eines Objektes. Dies beinhaltet die Speicherung von Objektidentität, Objektverhalten (Operationen) und Objektstatus (Attribute) zu einem bestimmten Zeitpunkt.

Skript: 2.9

1-21

OO-Konzept: Kapselung

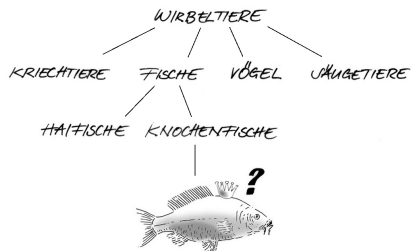


- Unter **Kapselung** (Encapsulation, Information Hiding) versteht man das Verstecken der inneren Funktionalität und Komplexität einer Klasse durch ein definiertes Interface.
- Die Verwendung eines definierten Interfaces für den Zugriff auf Informationen, erlaubt die Entkoppelung der internen Implementation einer Klasse vom übrigen Teil des Programms.
- Daraus resultiert eine verbesserte Wiederverwendung des Codes und eine vereinfachte Erweiterung und Wartung des Programms.

Skript: 2.4

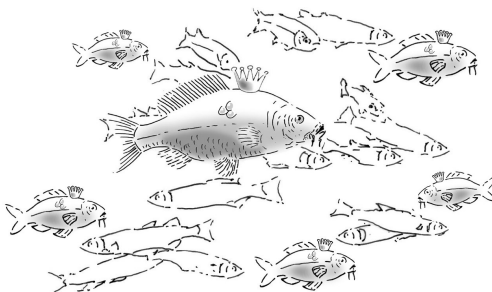
1-22

OO-Konzept: Hierarchie



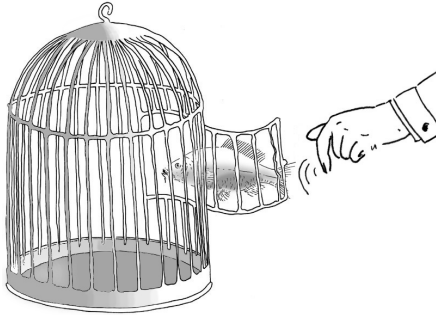
- Beziehungen zwischen Klassen mit gemeinsamen Merkmalen können oft durch eine **Klassenhierarchie** von sogenannten Ober- und Unterklassen beschrieben werden.
- Das Zusammenfassen gemeinsamer Merkmale verschiedener Unterklassen in einer Oberklasse wird als **Generalisierung** bezeichnet.
- Umgekehrt wird die Bildung neuer Unterklassen von einer bestimmten Oberklasse als **Spezialisierung** bezeichnet.

OO-Konzept: Vererbung



- **Vererbung** (Inheritance) ist ein objekt-orientiertes Programmiersprachenkonzept, bei welchem innerhalb einer Klassenhierarchie Funktionen und Attribute einer Oberklasse an eine davon abgeleitete Unterklasse weitergegeben werden.
- Mächtiges Konzept für die vereinfachte Software-Wartung oder um Funktionserweiterungen zu implementieren.

OO-Konzept: Typenprüfung



- Die strikte **Typenprüfung** verhindert Laufzeitfehler weil die Zuweisung inkompatibler Datentypen bereits zur Kompilations-Zeit entdeckt wird.
- Bei objektorientierten Programmiersprachen ist die Typenprüfung erweitert um die Kompatibilität von Objekten unterschiedlicher Klassen zu überprüfen.
- Ein Objekt einer Unterklasse ist dabei immer kompatibel mit einem Oberklassenobjekt der gleichen Klassenhierarchie, aber nicht umgekehrt.

OO-Konzept: Polymorphismus

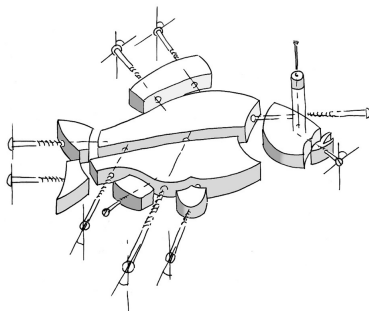
- **Polymorphismus** (griech. Vielgestaltigkeit) erlaubt Funktionen einer Basisklasse in einer abgeleiteten Unterklasse neu zu definieren.
- Polymorphismus ist eines der mächtigsten objektorientierten Konzepte und Voraussetzung um als objektorientierte Sprache zu gelten.
- Bei der **statischen (signaturegebundenen) Polymorphie** ermittelt der Compiler zur Kompilationszeit welche Funktion zur Laufzeit aufgerufen werden muss (static binding).
- Bei der **dynamischen (vererbungsgebundenen) Polymorphie** wird erst zur Laufzeit ermittelt, welche Funktion aufgerufen werden muss (dynamic oder late binding).

OO-Konzept: Generizität

- **Generizität** erlaubt abstrakte Datentypen zu parametrisieren und ermöglicht damit die **generische Programmierung** (v.a. verwendet für Daten-Container).
- In objektorientierten Programmiersprachen wird die strikte Typenprüfung auch auf generische Datentypen angewendet.
- Implementation durch parametrisierte Klassen und Funktionen (Synonyme: generische Klassen, Template-Klassen).
- “Generische Programmierung” mit C erfordert die Verwendung von void-Pointern, wobei keine Typenprüfung möglich ist!

1-27

OO-Konzept: Software-Komponenten



- **Software-Komponenten** sind ausführbare, selbständige funktionale Programm-Module, welche ihre Dienstleistungen zur Laufzeit über ein definiertes Interface zur Verfügung stellen (Software-IC's).
- Unter **Komponentenbasierter Software-Entwicklung** versteht man die Entwicklung (Zusammenbau) eines Programms aus vielen “vorfabrizierten” und getesteten Software-Komponenten (Componentware).
- Weiterentwicklung des Modulkonzeptes zur Erhöhung der Wiederverwendbarkeit (Reuse).
- Die einzelnen Komponenten können dabei auch auf unterschiedlichen Computern laufen.

OO-Konzept: Design Patterns

- Ein Design Pattern (Entwurfsmuster) ist eine (objektorientierte) Musterlösung für häufig auftretende Design-Probleme.
- Dokumentation von Design Know-How und Erfahrung
- Meist sprach- und implementationsunabhängig
- Design-Vokabular für Entwickler

1-29

OO-Konzepte: Zusammenfassung

I

- Hohes problemorientiertes Abstraktionsniveau
- Kapselung von Daten und Funktionen
- Klassen- und Objektkonzept
- Klassenhierarchie
- Vererbungskonzept
- Wiederverwendbarkeit (Software-Komponenten)

1-30

- Objektorientierte Software Entwicklung bietet gegenüber der strukturierten Vorgehensweise erweiterte und sehr mächtige Konzepte, weshalb sich das objektorientierte Paradigma auf breiter Basis etabliert und durchgesetzt hat.
- Die Kunst des objektorientierten Software Engineering besteht darin, objektorientierte Konzepte auf eine reale Problemstellung anzuwenden und damit ein Modell der realen Welt zu beschreiben, so dass ein Programmierer die Lösung implementieren kann.
- Die Art und Weise wie man diese objektorientierten Konzepte am besten anwendet werden im sogenannten Software-Entwicklungsprozess beschrieben.

Teil 2

UML Grundlagen

2-1

Teil 2 - Inhalt

- UML Übersicht
- UML Konzepte
- UML Notation
 - Elemente
 - Beziehungen
 - Diagramme (Klassendiagramm)

2-2

Wozu Software-Modellierung?

I

- Beispiel: Fehlende Stadtplanung



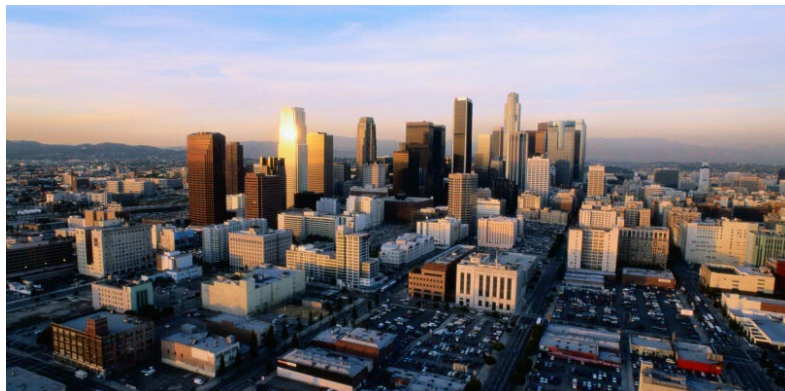
Skript: 1.1

2-3

Wozu Software-Modellierung?

II

- Beispiel: Koordinierte Stadtplanung



Skript: 1.1

2-4

Vorteile einer Software-Modellierung

- Die Verwendung einer Software-Modellierungstechnik erlaubt ein methodisches Problemlösungsvorgehen
- Die durch eine Modellierung erreichte Abstraktion ermöglicht die Reduktion der Systemkomplexität
- Mögliche Lösungsvarianten können im Modell beurteilt werden
- Die Modellierung erlaubt eine Verringerung der Software-Fehler und damit die Reduktion der Entwicklungskosten
- Projekte können schneller realisiert werden (time-to-market)

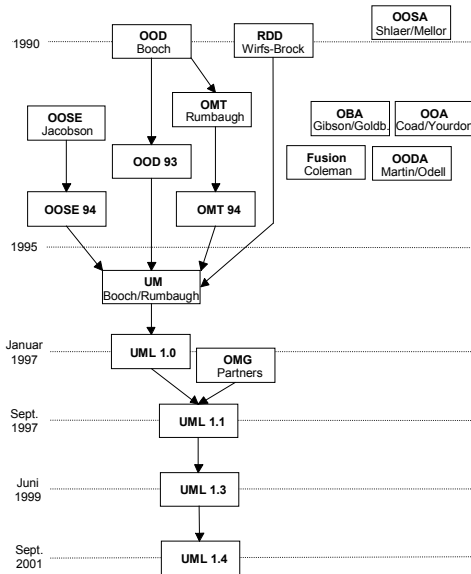
Unified Modeling Language (UML)

- UML ist eine normierte, graphische Sprache zur Visualisierung, Spezifikation, Konstruktion und Dokumentation von objekt-orientierten Systemen
- Verwendung für die unterschiedlichsten Problem-Domains
- Nicht an ein spezielles Vorgehensmodell gebunden
- Standardnotation für verschiedene objektorientierte Methoden
- Offener OMG Standard
- Zunehmend nicht nur für die Modellierung von Software verwendet (z.B. Hardware, Organisation, Business-Modeling, etc.)

Unified ...

- **UML Geschichte:**

- 1981: Gründung Rational
- 1981: Booch bei Rational
- 1994: Rumbaugh bei Rational
- 1995: Jacobson bei Rational
- 1996: UM (Unified Method)
- 1997: UML 1.1 als OMG Standard
- 1999: UML 1.3 Standard
- 2001: UML 1.4 Standard
- 2002: UML 2.0 Draft Standard



Skript: 1.3.2

2-7

... Modeling ...

- Ein Modell ist eine konsistente Vereinfachung der Realität, welche es ermöglicht ein komplexes System (besser) zu verstehen.
- Ein Modell hilft die Struktur und das Verhalten eines Systems zu spezifizieren und wichtige Entscheidungen zu dokumentieren.
- UML ermöglicht es die Funktionsweise und das Design eines Software-Systems in einem Modell zu beschreiben, bevor es implementiert wird.
- Mit UML erfolgt die System-Modellierung durch eine Menge von miteinander interagierender Objekte.

Skript: 1.3.2

2-8

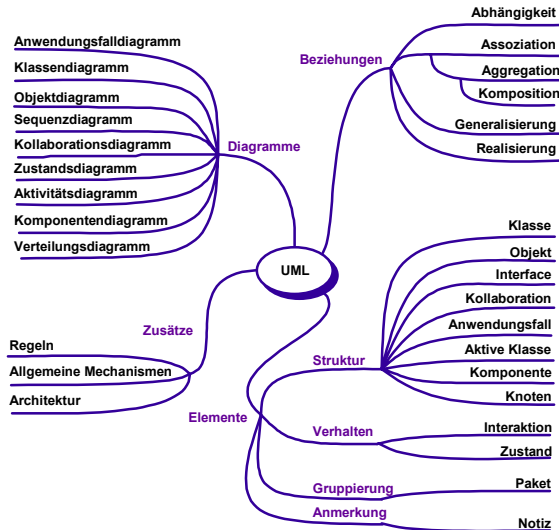
... Language

- UML ist eine allgemein verwendbare visuelle Sprache mit definierter Notation und Semantik
- Standardisierte Sprache für die Software-Modellierung und damit gemeinsame Kommunikationsbasis für Entwickler
- Unabhängig von einer speziellen Implementationssprache oder Plattform
- Unterstützung von Forward- und Reverse-Engineering
- Konzipiert für optimale Unterstützung durch Software-Tools

UML Konzepte

- Durchgängige Verwendung einer einzigen Methode, angefangen bei der Erfassung der Benutzeranforderungen bis zur Implementation
- Modellierung der statischen und dynamischen Eigenschaften eines Software-Systems
- Unterstützt die Modellaufteilung in hierarchische Einheiten (Views)
- Eingebaute Mechanismen für die Erweiterung und Spezialisierung

UML Notationsübersicht



Skript: 3.1

2-11

UML Elemente (Things)

- UML Notations-Elemente (Things) sind die grundlegenden objektorientierten Bausteine für jedes UML-Diagramm.
- UML Elemente lassen sich in vier Kategorien aufteilen:
 - Struktur-Elemente (Structural Things)
 - Anmerkungs-Elemente (Annotational Things)
 - Gruppierungs-Elemente (Grouping Things)
 - Verhaltens-Elemente (Behavioral Things)

Skript: 3.2

2-12

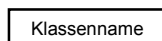
Struktur-Elemente (Structural Things)

- UML Struktur-Elemente werden verwendet um die statischen Aspekte eines Modells darzustellen.
- UML Struktur-Elemente können im Modell sowohl abstrakte als auch real existierende Dinge repräsentieren.
- Zu den UML Struktur-Elementen gehören:
 - Klasse (Class)
 - Objekt (Object)
 - aktive Klasse (Active Class)
 - Anwendungsfall (Use Case)
 - Kollaboration (Collaboration)
 - Komponente (Component)
 - Schnittstelle (Interface)
 - Knoten (Node)

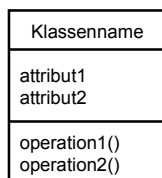
UML Notation: Klasse (Class)

I

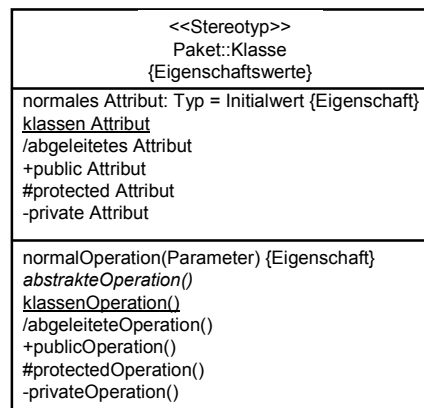
- Einfachste Variante:



- Mittlere Variante:



- Allgemeine Variante:



UML Notation: Klasse (Class)

II

- **UML Klassennotation:**

- Ein **<<Stereotype>>** ist ein Name für einen neuen Typ eines UML-Modellelements, der einem vordefinierten Element eine spezielle Semantik und optional ein spezielles Symbol zuordnet. (z.B. UML vordefiniert: actor, file, executable, etc.)
- Eine **abstrakte Klasse** wird durch kursive Schreibweise des Klassennamens dargestellt und kann nicht instanziiert werden (z.B. verwendet für Interface Definition).
- Ein **Eigenschaftswert (Tagged Value)** kann verwendet werden um zusätzliche Eigenschaften zu spezifizieren: z.B. {Library Version = 2.31}
- Eine **/abgeleitetes Attribut** bzw. eine **/abgeleitete Operation()** lässt sich aus anderen Attributen bzw. Operationen herleiten (z.B. Alter = AktuellesDatum – GeburtsDatum).

UML Notation: Klasse (Class)

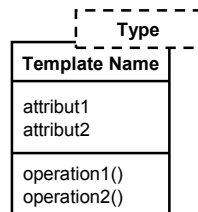
III

- **UML Klassennotation:**

- **Attribut-Eigenschaften (Properties):**
z.B. {changable}, {frozen}, {write once} (entspricht der C++ const Deklaration), etc.
- Ein **Klassen-Attribut** bzw. eine **Klassen-Operation()** wird für alle Instanzen einer Klasse nur einmal verwendet (entspricht einem C++ static Attribut bzw. Operation).
- Attribut- bzw. Operations- **Sichtbarkeit (Visibility):** + (public), # (protected), - (private)
- Eine **abstrakte Operation()** hat keine Implementation, sondern spezifiziert nur die Methoden-Signatur (entspricht einer C++ pure virtual Funktion).

UML Notation: Template-Klasse (Template)

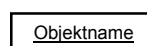
- Eine Template-Klasse (Parametrisierte Klasse) ist eine generische Klasse (Schablone) zum erzeugen von normalen Klassen



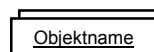
2-17

UML Notation: Objekt (Object)

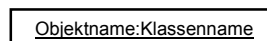
- Nur Objektname:



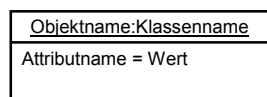
- Multi-Objekt:



- Objektname mit Klassen-zugehörigkeit:

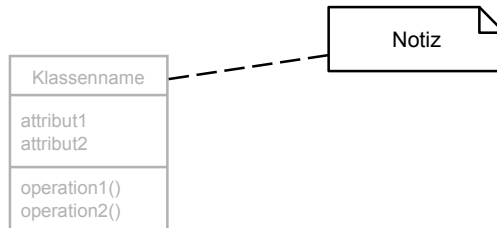


- Objektname mit Klassen-zugehörigkeit und Anzeige von Informationen:



UML Notation: Notiz (Note)

- Eine UML Notiz (Note) wird verwendet um ein beliebiges Modell-element oder ein UML-Diagramm mit Detailerklärungen und Kommentaren zu versehen. (UML Annotational Thing)

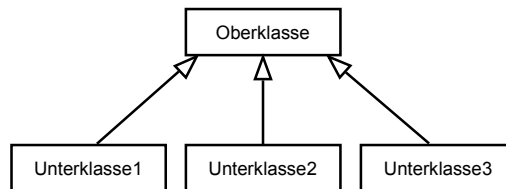


UML Beziehungen (Relations)

- UML Beziehungen werden verwendet um Verbindungen zwischen UML Elementen darzustellen.
- UML kennt vier verschiedene Typen von Beziehungen:
 - Generalisierung, Vererbung (Generalization)
 - Assoziation (Association)
 - Abhängigkeit (Dependency)
 - Realisierung (Realization)

Notation: Generalisierung (Generalization) I

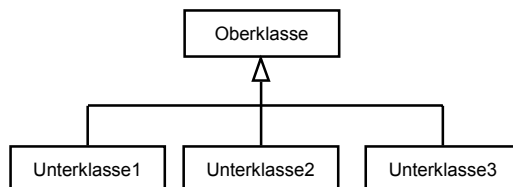
- Eine Generalisierung (Vererbung) wird mit einem Pfeil von der Unterklasse (subclass) in Richtung Oberklasse (base class, superclass) gezeichnet.
- Generalisierung - Darstellung I



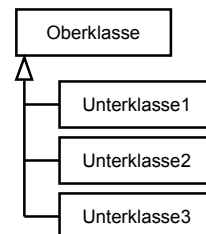
Notation: Generalisierung (Generalization) II

- Generalisierung - Darstellung II
(Pfeile für die Angabe von Vererbungen werden zusammengeführt)

- Baumdarstellung

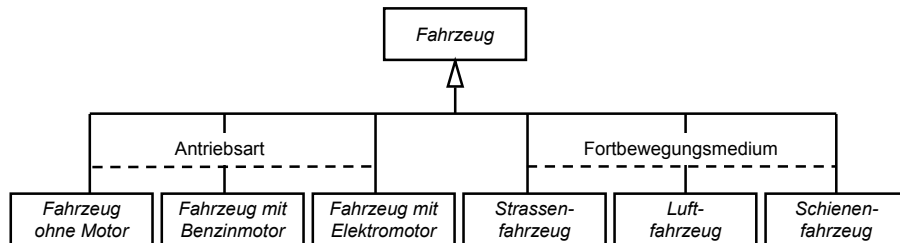


- Bibliotheksdarstellung



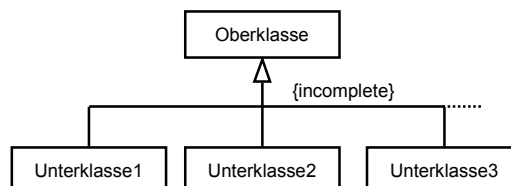
Generalisierung: Diskriminator (Discriminator)

- Ein Diskriminator unterteilt Spezialisierungen aufgrund eines bestimmten Kriteriums
(Notation bei Vererbungslinie oder mittels gestrichelter Linie für mehrere Vererbungslinien)



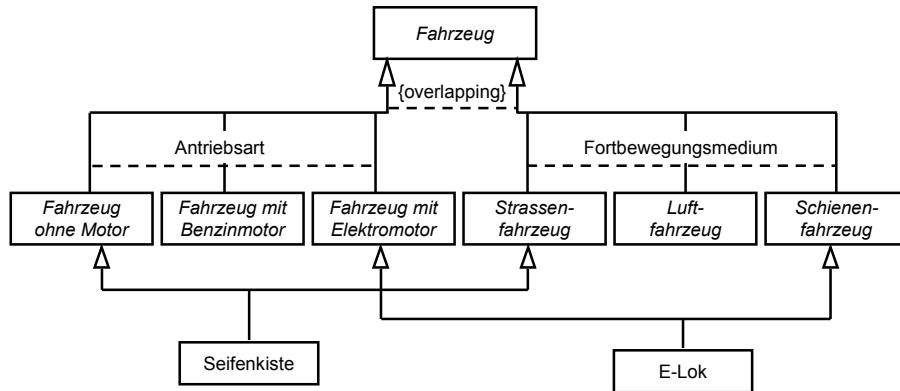
Generalisierung: Zusicherungen (Constraints)

- UML vordefinierte Generalisierungs-Constraints:
 - **incomplete** Nicht alle Klassen sind spezifiziert (weitere Klassen werden erwartet)
 - **complete** Alle Klassen sind spezifiziert



Generalisierung: Mehrfachvererbung

- UML definierte Generalisierungs-Constraints für Mehrfachvererbung:
 - **overlapping** Instanz kann von mehreren Basisklassen erben
 - **disjoint** Instanz kann nur von einer Basisklasse erben (Default)

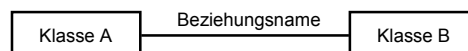


Skript: 3.3.1

2-25

UML Notation: Assoziation (Association) I

- Eine Assoziationsbeziehung beschreibt eine **strukturelle Verbindung** zwischen zwei gleichberechtigten UML-Elementen
- Assoziationsbeziehungen sind **bidirektional d.h. navigierbar in beide Richtungen**.
- **Assoziation - Darstellung I**
(Assoziation mit Beziehungsname)



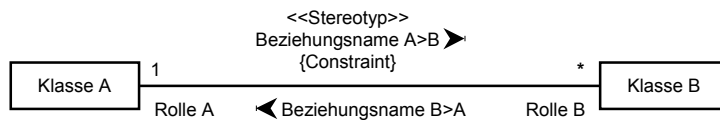
Skript: 3.3.2

2-26

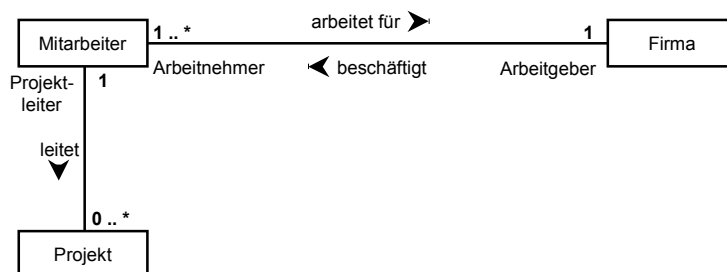
UML Notation: Assoziation (Association) II

- **Assoziation - Darstellung II**

(Allg. Darstellung mit wechselseitig definierten Assoziationsrollen und Beziehungsnamen)



Assoziation: Beispiel



Assoziation: Multiplizität (Multiplicity)

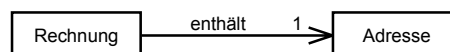
- Die Multiplizität definiert den Mengenbereich einer Assoziation:

Angaben	Bedeutung
1	genau eins
0,1	genau null oder genau eins
0 .. 4	zwischen null (inkl.) und vier (inkl.)
3,7	genau drei oder genau sieben
0 .. *	größer gleich null (implizite default Definition, falls keine explizite Angabe erfolgt)
*	beliebige Anzahl inkl. null (abgekürzte Angabe von 0 .. *)
1 .. *	größer gleich eins
0 .. 3,7,9 .. *	zwischen null (inkl.) und drei (inkl.) oder genau sieben oder größer gleich neun

{ordered}, {unordered}: vordefinierte UML Constraints für Assoziationen mit Multiplizität > 1

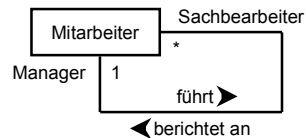
Assoziation: Navigation (Navigability)

- Eine gerichtete (navigierbare) Assoziation beschreibt eine einseitige (unidirektionale) Assoziationsbeziehung.



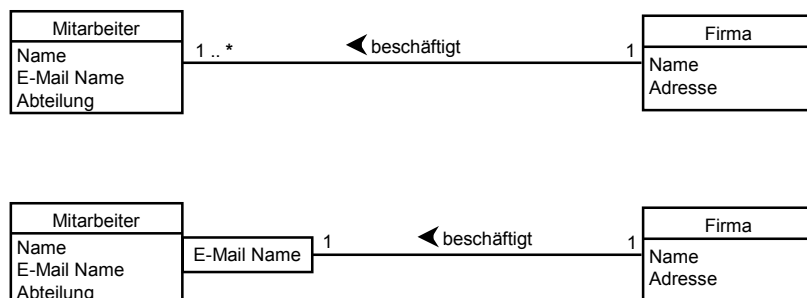
Assoziation: Rekursion (Recursion)

- Eine rekursive Assoziation beschreibt eine strukturelle Beziehung einer Klasse auf sich selber.



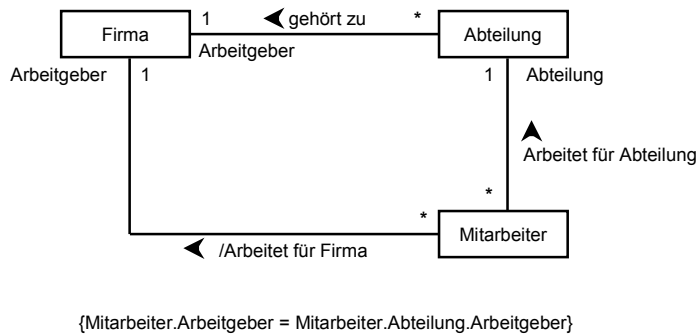
Assoziation: Qualifikation (Qualification)

- Bei einer qualifizierten Assoziation wird die Menge der Objekte durch ein qualifizierendes Attribut (Qualifier) in Partitionen unterteilt.



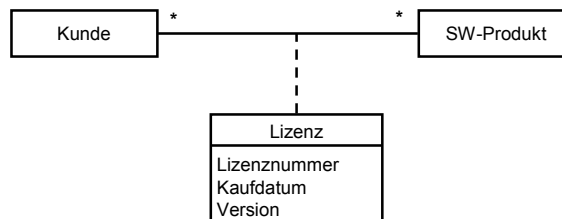
Abgeleitete Assoziation (Derived Association)

- Eine abgeleitete Assoziation entsteht, wenn eine Assoziationsbeziehung über einen andern Beziehungsweg hergeleitet werden kann.



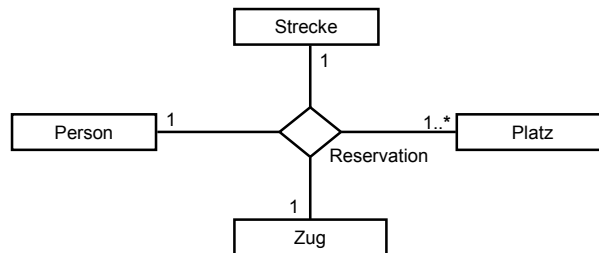
Assoziationsklasse (Association Class)

- Eine Assoziationsklasse wird verwendet um einer Assoziation Attribute zuordnen zu können, welche nicht eindeutig einer der beteiligten Klassen zugeordnet werden können.



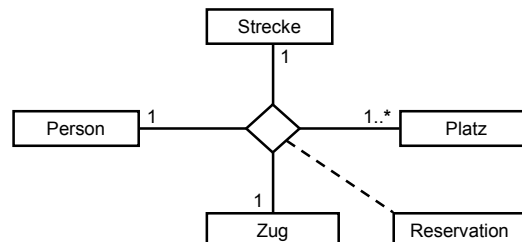
n-gliedrige Assoziation (n-ary Association) I

- Eine mehrgliedrige Assoziation ist eine normale Assoziationsbeziehung zwischen drei oder mehr Klassen (Assoziationsrollen).



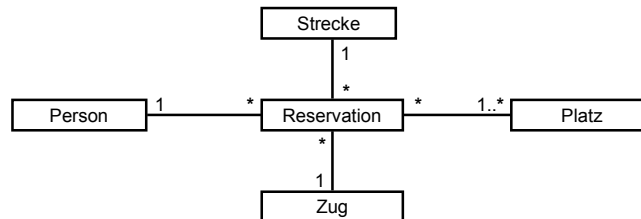
n-gliedrige Assoziation (n-ary Association) II

- Eine mehrgliedrige Assoziation kann in der Regel mit einer attribuierten Assoziation (Assoziationsklasse) dargestellt werden



n-gliedrige Assoziation (n-ary Association) III

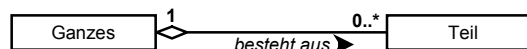
- Eine mehrgliedrige Assoziation kann von einer attributierten Assoziation in mehrere normale Assoziation umgeformt werden.



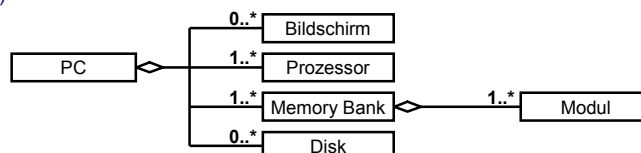
UML Notation: Aggregation (Aggregation)

- Die Aggregation ist eine spezielle Assoziationsbeziehung zur Beschreibung einer Beziehung zwischen einem "Ganzen" und einem "Teil".

- Aggregation - Darstellung I

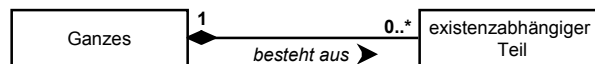


- Aggregation - Darstellung II
(Baumdarstellung)

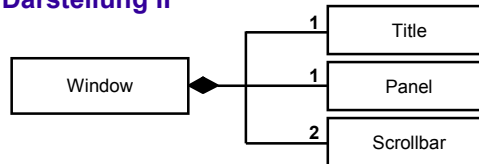


UML Notation: Komposition (Composition) I

- Die Komposition ist eine spezielle Aggregation zur Beschreibung einer "Ganzes/Teil"-Beziehung wobei der Teil existenzabhängig vom Ganzen ist.
- Komposition - Darstellung I



- Komposition - Darstellung II
(Baumdarstellung)

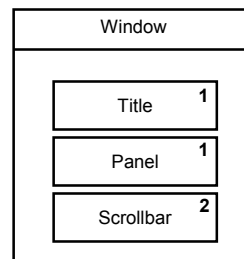
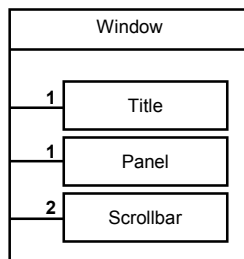


Skript: 3.3.2.2

2-39

UML Notation: Komposition (Composition) II

- Komposition - Darstellung III
(Verschachtelte Darstellung)



2-40

UML Diagramme (Diagrams)

I

- Ein UML Diagramm ist eine grafische Darstellung einer Menge von UML Elementen und ihren Beziehungen untereinander.
- Ein UML Diagramm repräsentiert eine spezifische Perspektive (View) auf das Modell eines Software-Systems.
- Ein einzelnes UML Diagramm kann jeweils nur einen Aspekt in einem kleinen Ausschnitt eines grossen Systems darstellen.

UML Diagramme (Diagrams)

II

- Zur Darstellung von unterschiedlichen Aspekten eines Systems werden unterschiedliche UML-Diagrammtypen verwendet.
- **Diagramme für statische Aspekte (Structural Diagrams):**
 - Anwendungsfalldiagramm (Use Case Diagram)
 - Objektdiagramm (Object Diagram)
 - Klassendiagramm (Class Diagram)
 - Komponentendiagramm (Component Diagram)
 - Verteilungsdiagramm (Deployment Diagram)
- **Diagramme für dynamische Aspekte (Behavioral Diagrams):**
 - Aktivitätsdiagramm (Activity Diagram)
 - Sequenzdiagramm (Sequence Diagram)
 - Kollaborationsdiagramm (Collaboration Diagram)
 - Zustandsdiagramm (Statechart Diagram)

Klassendiagramm (Class Diagram)

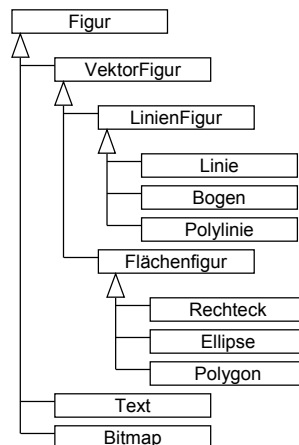
I

- Ein Klassendiagramm zeigt Klassen und ihre Beziehungen untereinander
- Ein Klassendiagramm wird verwendet um die statische Klassenstruktur eines Systems darzustellen
- Häufig auch für Reverse-Engineering eingesetzt

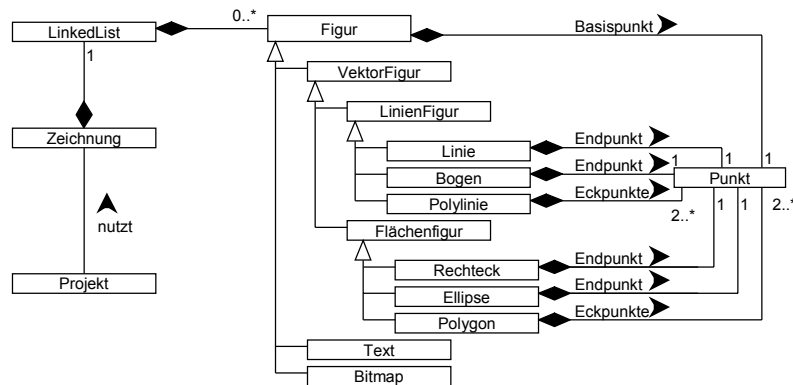
Klassendiagramm (Class Diagram)

II

- Beispiel eines Vererbungsdiagramms (Inheritance Diagram)



- Beispiel eines gemischten Diagramms



Teil 3

Software-Entwicklungsprozess

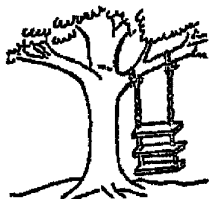
3-1

Teil 3 - Inhalt

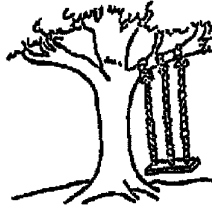
- **Software-Entwicklungsprozess**
- **Software Life Cycle Modelle**
- **Verwendung von UML im Rational Unified Process**
 - UML Use Case Modell
 - UML Analysemodell
 - UML Designmodell
 - UML Prozessmodell
 - UML Implementationsmodell
 - UML Testmodell
 - UML Deployment Modell

3-2

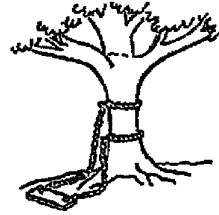
Wozu ein Software-Entwicklungsprozess?



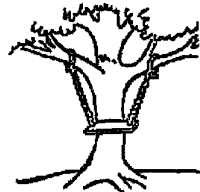
1. Anforderungen des Produktmanagements



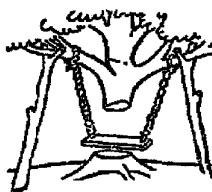
2. Resultat der Anforderungsdefinition



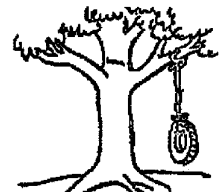
3. Konzept nach Systemanalyse



4. Lösung der Programmierer



5. Beim Kunden installierte Anwendung



6. Ursprüngliche Kundenanforderung

3-3

Software-Entwicklungsprozess

- **Software-Entwicklungsprozess Definition:**
 - Sämtliche notwendigen Aktivitäten, Techniken und Tools um aus Benutzeranforderungen ein konsistentes Software-Produkt mit allen dazugehörigen Artefakten (Modelle, Diagramme, Dokumente etc.) zu produzieren.
 - Beinhaltet sowohl Technik- als auch Managementaspekte

Benutzer-
anforderungen



Software
Entwicklungsprozess



Software-Produkt

Software Life Cycle

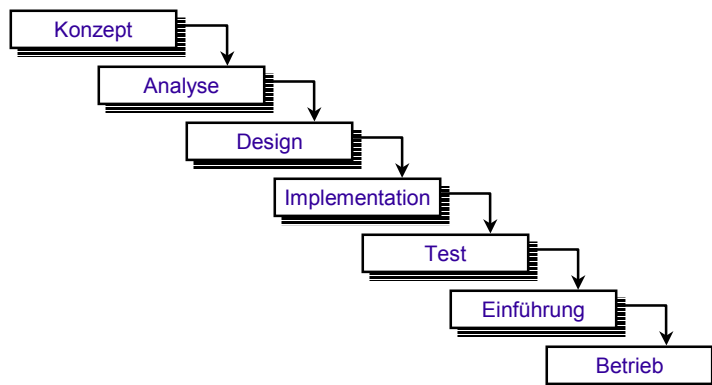
- **Software Life Cycle Definition:**
 - Der Software Life Cycle (Lebenszyklus) beschreibt die verschiedenen Entwicklungsschritte, welche eine Software von ihrem Entstehen bis zum fertiggestellten Produkt durchläuft.
 - Das Software Life Cycle Modell bildet die Grundlage für das Vorgehensmodell des darauf aufbauenden Software-Entwicklungsprozesses.
- **“Build-And-Fix Modell”**
 - Kein richtiges Modell, aber (zu) oft angewendet.
 - Höchstens für kleine Prototypen oder simple Web-Seiten anwendbar.

Skript: 1.4

3-5

Wasserfall Life Cycle Modell

- **Wasserfall Life Cycle Modell (1970)**
 - Streng sequentielles Vorgehen nach der Methode der schrittweisen Verfeinerung
 - Verwendung mit strukturierten Methoden - heute eher veraltet



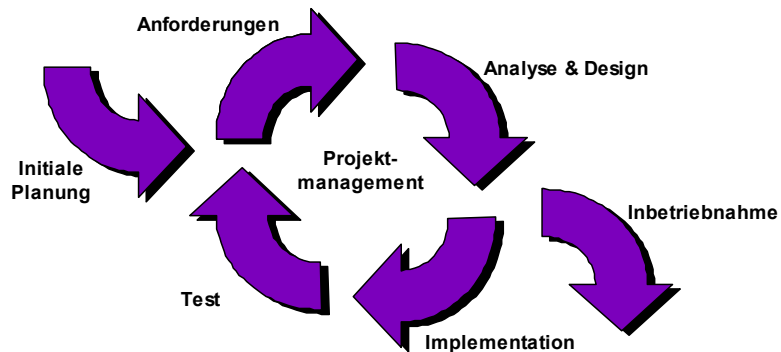
Skript: 1.4.1

3-6

Spiral Life Cycle Modell

- **Spiralmodell (1988)**

- Iteratives Vorgehen, produziert sukzessiv verbesserte Prototypen
- Definitives Software-Produkt wird iterativ und inkrementell entwickelt



Skript: 1.4.2

3-7

Rational Unified Process

- **Rational Unified (Software Development) Process (RUP)**

- Spiral Life Cycle Modell als Grundlage
- Iteratives und inkrementelles Vorgehen
- Anwendungsfall (Use Case) basierend
- Software-Architektur zentriert

- **Vorteile**

- Optimale Unterstützung der Unified Modeling Language
- "Best Practices" aus verschiedenen Prozessmodellen
- Prozess anpassbar auf firmenspezifische Verhältnisse
- Rational Software-Tool Unterstützung

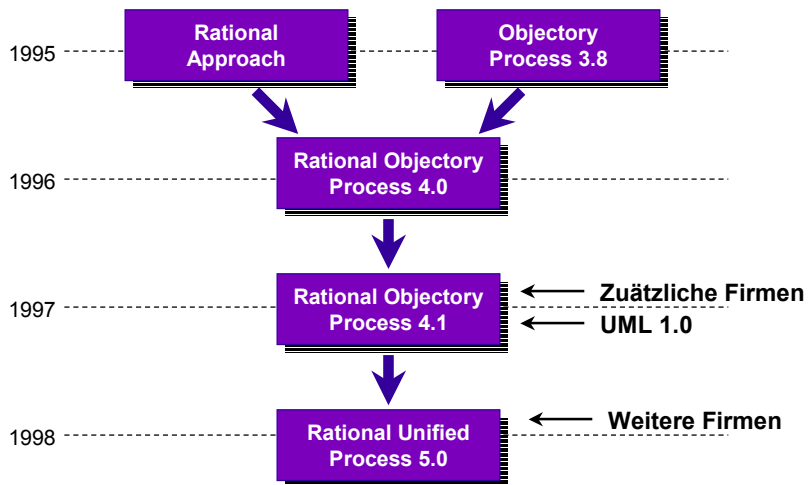
- **Nachteil**

- Proprietärer Prozess der Firma Rational

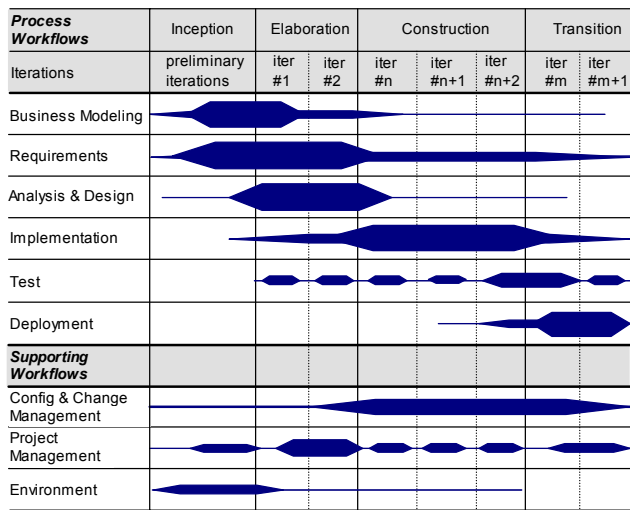
Skript: 1.4.3

3-8

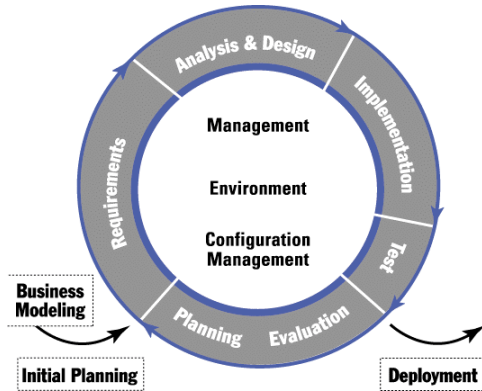
Rational Unified Process Entstehung



Rational Unified Process Übersicht



Iterativer und inkrementeller Prozess



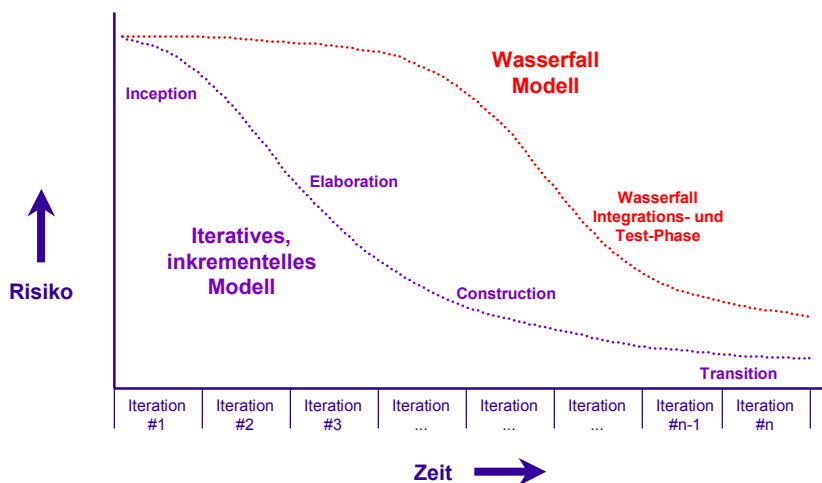
• Vorteile eines iterativen und inkrementellen Vorgehens

- Iterative Fertigstellung neuer (interner) Software-Versionen (**Minor Milestones**)
- Inkrementelle Software-Produktentwicklung (**abgeschlossene Teilprojekte**)
- Berücksichtigung sich ändernder Anforderungen (Requirements)
- Bessere Software-Wiederverwendbarkeit innerhalb des Projektes
- Schrittweise Systemintegration erlaubt kontinuierliches Testen
- Erkennen von kritischen Performance- und Ressourcen-Problemen
- Frühzeitige Minimierung von potentiellen Projektrisiken

Skript: 1.4.3

3-11

Minimierung von Projektrisiken



3-12

Use Case basierter Prozess

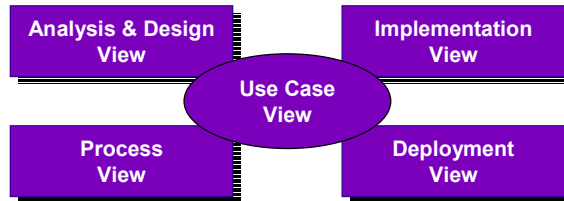
- **Use Cases (Anwendungsfall) als Basis aller Entwicklungsaktivitäten**
- **Erfassung der Benutzeranforderungen (Functional Requirements)**
 - Beschreibung der Systemfunktionalität aus Benutzersicht
 - Grundlage für Pflichtenheft und Projektvertrag
- **Grundlage zur Entwicklung der Benutzerschnittstelle**
 - Anhaltspunkt für Optimierung bezüglich Performance und Usability
- **Basis für inkrementelle Software Release Planung**
- **Grundlage für das Software-Testen (Test Cases)**
- **Vorstufe der Benutzerdokumentation**

Architekturzentrierter Prozess

- **Eine Software-Architektur beschreibt das Entwicklungsziel aus einer (visionären) Gesamtperspektive**
- **Eine Software-Architektur definiert die allgemeinen Systemanforderungen (Nonfunctional Requirements)**
 - Definition von Struktur und Organisation der Software
 - Definition der Systemschnittstellen
 - Definition der zu verwendenden Basistechnologien
 - Berücksichtigung von Standards und Richtlinien
- **Eine Architekturbeschreibung spezifiziert die architekturrelevanten konzeptionellen und physischen Eigenschaften eines Systems (UML-Views)**

UML Architekturmodell

- UML Architekturbeschreibung durch “4+1 View” Modell



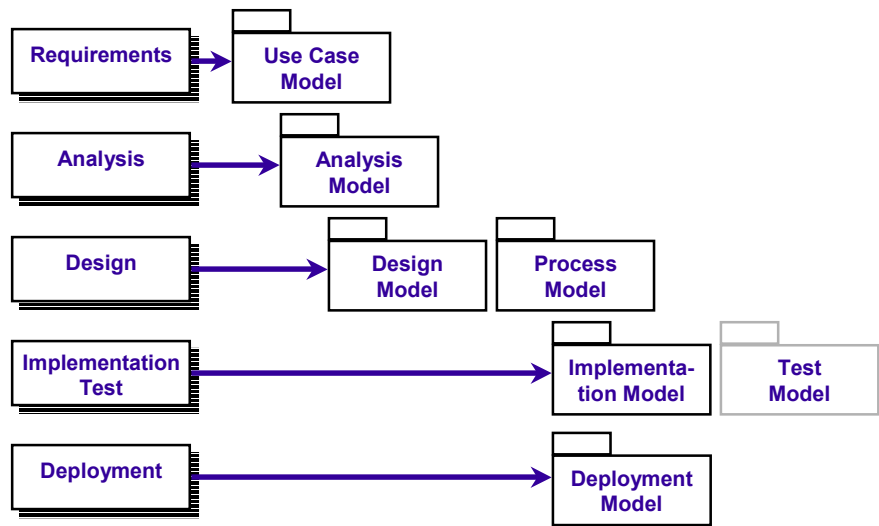
3-15

Process Workflows

- Ein Arbeitsablauf (Workflow) ist eine Abfolge von Arbeitsaktivitäten, aus welchen ein sichtbarer Nutzen resultiert
- Definiert die am Arbeitsablauf beteiligten Arbeiter (Workers) und deren Verantwortlichkeiten
- Beschreibt die aus dem Arbeitsablauf resultierenden Artefakte
- **Rational Unified Process definierte Workflows:**
 - Geschäftsmodellierung (Business Modeling Workflow)
 - Anforderungsanalyse (Requirements Workflow)
 - Analyse und Design (Analysis & Design Workflow)
 - Implementation (Implementation Workflow)
 - Testen (Test Workflow)
 - Verteilung (Deployment Workflow)

3-16

RUP Process Workflows und UML Modelle



Anfangsphase (Inception Phase) I

Process Workflows	Inception	Elaboration		Construction			Transition	
Iterations	preliminary iterations	iter #1	iter #2	iter #n	iter #n+1	iter #n+2	iter #m	iter #m+1
Business Modeling								
Requirements								
Analysis & Design								
Implementation								
Test								
Deployment								
Supporting Workflows								
Config & Change Management								
Project Management								
Environment								

Anfangsphase (Inception Phase)

II

- **Die Anfangsphase umfasst alle notwendigen Aktivitäten um das eigentliche Software-Entwicklungsprojekt zu initiieren:**
 - Integrieren von Projekt und Software-Produkt im Business-Modell (Marktanalyse)
 - Bestimmen der Projekt- und Systemgrenzen
 - Spezifizieren der wesentlichen Use Cases und der dazugehörigen Akteure
 - Beschreiben einer möglichen Systemarchitektur
 - Identifizieren und abschätzen der Projektrisiken
 - Etablieren eines Projektteams
 - Entwickeln eines oder mehrerer Technologie-Prototypen (Proof Of Concept)
 - Erstellen einer provisorischen Projektplanung
 - Abschätzen der erforderlichen Ressourcen, der Projektdauer und der Kosten
 - Abklären der Projektmachbarkeit (Feasibility)
- **Was kann und soll (unter den geg. Bedingungen) realisiert werden.**

Skript: 1.4.3

3-19

Anfangsphase (Inception Phase)

III

- **Je nach Verwendung werden Protoypen in Kategorien eingeteilt:**
- **Exploratory Prototype (Inception Phase)**
 - Wegwerf-Prototyp der nicht weiter verwendet wird und nur zur Evaluation einer einzelnen Funktion oder zur Konzeptvalidierung verwendet wird (z.B. Performance).
- **Evolutionary Prototype (Inception/Elaboration Phase)**
 - Prototyp der inkrementell weiter entwickelt wird, aber entsprechend die üblichen Software-Qualitätsanforderungen erfüllen muss.
- **Behavioral Prototype (Inception/Elaboration Phase)**
 - Verwendet für die Untersuchung von User Interface Varianten, z.B. mit Gui-Builder, Visual Basic etc.. (Exploratory/Evolutionary Prototype).
- **Structural Prototype (Elaboration Phase)**
 - Test der Systeminfrastruktur und Systemarchitektur sowie der Basis-Technologien. (Evolutionary Prototype).

Skript: 1.4.3

3-20

Anfangsphase (Inception Phase)

IV

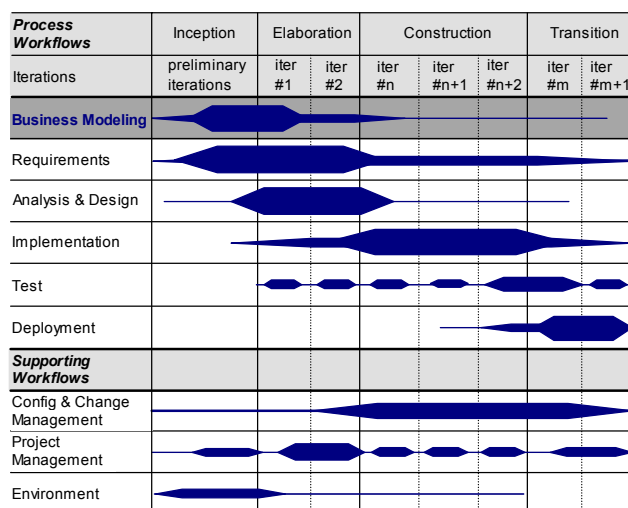
- **Major Milestone:**
Zieldefinition des Entwicklungsprojektes (Life Cycle Objective)
 - Projektentscheid: Abbrechen, Durchführen, Zukaufen, Wiederverwenden ?
- **Output Artefakte:**
 - Vorstudien- und Konzeptdokumente
 - Initiales Use Case Modell
 - Projekt Glossary
 - Ein oder mehrere Prototypen
 - Pflichtenheft (Software Requirements Specification)

Skript: 1.4.3

3-21

Business Modeling Workflow

I



Skript:

3-22

Business Modeling Workflow

II

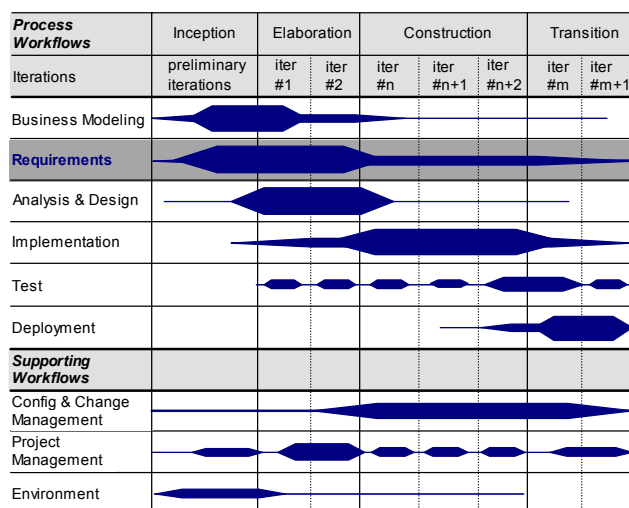
- Ein Geschäftsmodell versucht die Organisation und den Ablauf der einzelnen Arbeitsschritte in einer Firma zu modellieren (Business Process (Re-) Engineering).
- **Business-Anforderungen an (kommerzielle) Software:**
 - Integration des Software-Produktes in das Business-Modell einer Firma
 - Berücksichtigung von Geschäftsfällen (Business Use Cases)
 - Modellierung von “Business Objects”
 - Möglicherweise eine Anpassung des vorhandenen Business-Prozesses

Skript:

3-23

Requirements Workflow

I



Skript:

3-24

Requirements Workflow

II

- Funktionale Beschreibung des Systems aus Benutzersicht
- Erfassen der Benutzeranforderungen (Functional Requirements)
- Erfassen der Architekturanforderungen (Nonfunctional Requirements)
- Spezifikation der Systemfunktionalität
- Etablieren einer gemeinsamen Kommunikationsbasis zwischen Benutzer und Entwickler
- Grundlage zur Entwicklung der Benutzerschnittstelle
- Grundlage zum Abschätzen von Kosten- und Zeitplanung

Skript:

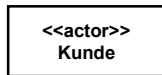
3-25

UML Use Case Modell

- Das Use Case Modell beinhaltet alle zur Beschreibung von Use Cases verwendeten UML Diagramme (Use Case View)
- Modellierung statischer Aspekte von Use Cases:
 - Use Case Diagramme
- Modellierung dynamischer Aspekte von Use Cases:
 - Aktivitätsdiagramme (Zustandsdiagramme)

UML Notation: Akteur Stereotyp (Actor)

- Ein Akteur ist eine Abstraktion eines Benutzers (ausserhalb des Systems) welcher mit dem System in Interaktion tritt.
(Bsp. Person, Maschine, anderes System)



Darstellung als Klassen-Stereotyp



Kunde

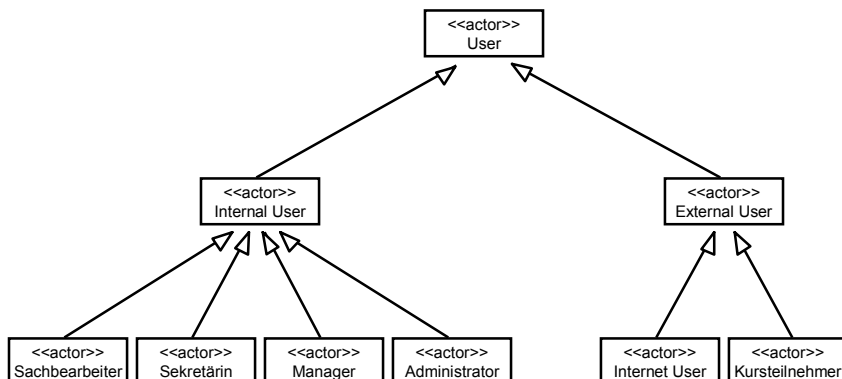
Darstellung als Akteur-Icon

- Stereotyp Definition:**
 - Name für einen neuen Typ eines UML-Modellelements, der einem vordefiniertem Element eine spezielle Semantik und evtl. ein neues Icon zuordnet.
(UML Vordefiniert: actor, file, executable etc.)

Akteurdiagramm (Actor Diagram)

I

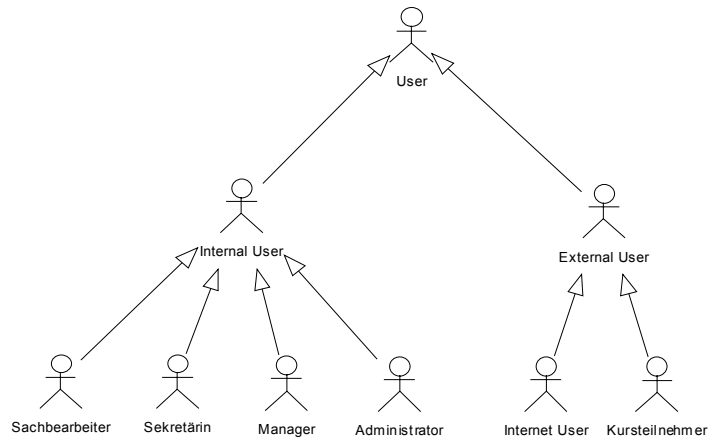
- Ein Akteurdiagramm zeigt die Beziehungen zwischen den Akteuren.
Diese sind als normale Klassen modellierbar (Vererbungsdiagramm).



Akteurdiagramm (Actor Diagram)

II

- Übliche Darstellung mit Icons für Actor Stereotyp

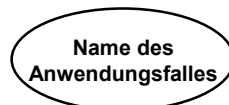


Skript: 3.4.3

3-29

UML Notation: Anwendungsfall (Use Case)

- Ein Anwendungsfall beschreibt eine Folge von Systemaktionen mit denen das System ein für den Akteur sichtbares und nützliches Resultat erzielt.



- Beispiele:
 - Use Cases: Zeichnen einer Linie, Drucken einer Zeichnung, Erfassen eines Kunden
 - Akteure: Hochbauzeichner, Maschinenkonstrukteur

Skript: 3.2.1.5

3-30

Anwendungsfallbeschreibung

Name:	Drucken einer Zeichnung
Nummer:	1
Kurzbeschreibung:	Die Zeichnung wird aus dem laufenden Programm auf dem angeschlossenen Ausgabegerät ausgegeben (print oder plot).
Beteiligte Akteure:	Benutzer und Ausgabegerät
Auslöser /	Zeichnung ist im System geladen
Vorbedingung:	
Ergebnisse /	Zeichnung ist auf Papier übertragen
Nachbedingung:	

Ablauf:

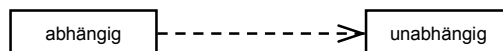
Nr.	Wer	Was
1	Benutzer	Einstellungen Drucker vornehmen
1.1	Benutzer	Drucken auslösen
1.2	System	Plausibilisierung der Einstellungen (Ränder, Farben...)
1.3	System	Ausnahme: Daten nicht i.O
1.4	System	Berechnungen für Ausdruck (Anzahl Seiten)
1.5	System	Effektive Ausgabe
1.6	System	Ausgabe der aktuellen Seite

Ausnahmen, Varianten:

Nr.	Wer	Was
1.3		Daten nicht i.O
1.3.1	System	Fehlermeldung mit Angabe der nicht korrekten Daten, Abbruch des Vorganges
1.3.2	Benutzer	Korrektur der Daten
1.3.3	Benutzer	Erneutes Aufsetzen des Vorganges

UML Notation: Abhängigkeit (Dependency)

- Eine Abhängigkeitsbeziehung beschreibt eine allgemeine (d.h. nicht strukturelle), gerichtete Verbindung zwischen zwei UML Elementen:



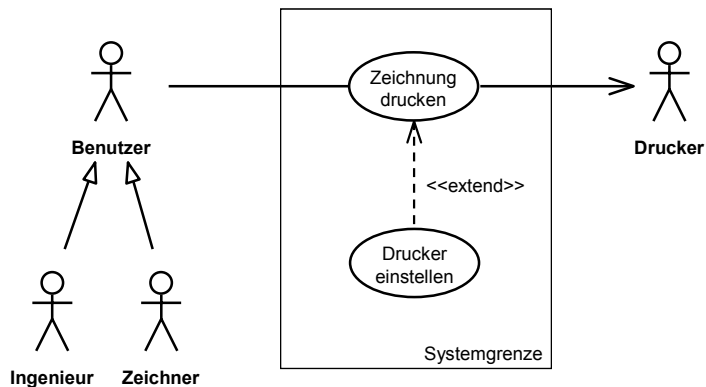
- Nur zu verwenden falls keine strukturelle Beziehung (Assoziation) vorliegt.
- Abhängigkeitsbeziehungen verwenden keine Multiplizitäten.
- Im allgemeinen ist kein Beziehungsname notwendig
- Verwendung hauptsächlich für Klassen, Packages und Use Cases.

- UML vordefinierte Stereotypen für Abhängigkeiten:

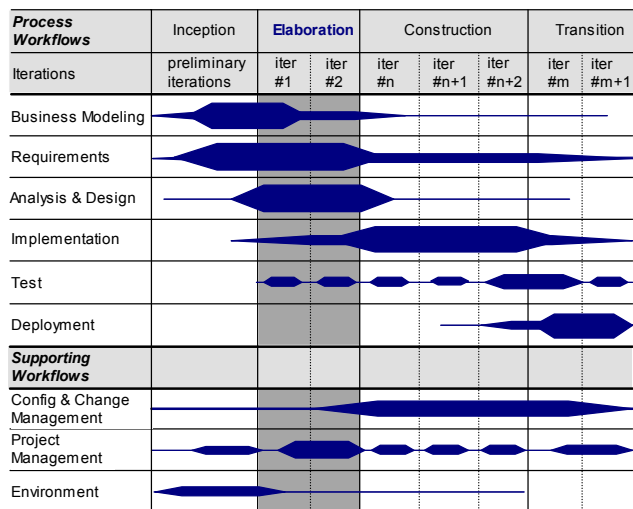
- <<use>> Verwendet zur Verdeutlichung einer "uses" Abhängigkeit
- <<extend>> Verwendet wenn ein Use Case einen anderen erweitert (Variante)
- <<include>> Verwendet wenn ein Use Case einen anderen beinhaltet (Sub-Use-Case)

Anwendungsfalldiagramm (Use Case Diagram)

- Das Anwendungsfalldiagramm zeigt die Anzahl und Abhängigkeit von Use Cases eines Systems aus der Sicht der beteiligten Akteure.



Ausarbeitungsphase (Elaboration Phase) I



Ausarbeitungsphase (Elaboration Phase) II

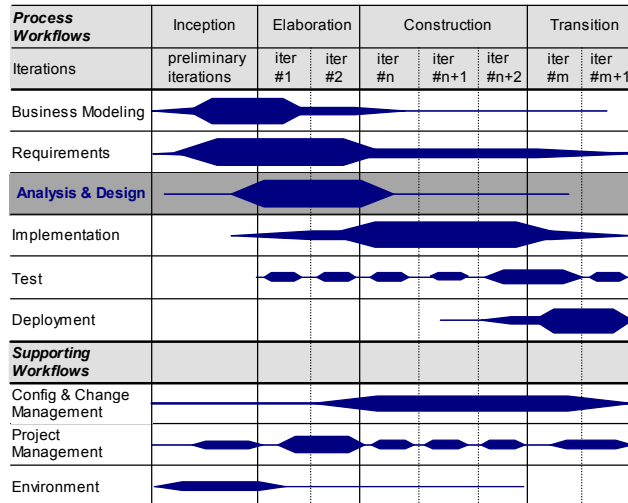
- **Die Ausarbeitungsphase umfasst alle Aktivitäten zur vollständigen Spezifikation von Benutzeranforderungen und Software-Architektur:**
 - Verfeinern und priorisieren der vorhandenen Use Cases
 - Evtl. Spezifizieren von einzelnen weiteren Benutzeranforderungen und Use Cases
 - Ausarbeiten der Analyseklassen und der statischen Software-Struktur
 - Erarbeiten der Designklassen und deren dynamisches Verhalten
 - Identifizieren von Netzwerkknoten und deren Konfiguration für verteilte Applikationen
 - Auswahl der zu verwendenden Software-Komponenten und Tools
 - Etablieren der notwendigen Infrastruktur und einer geeigneten Entwicklungsumgebung
 - Implementation des Architektur-Prototyps
 - Eliminieren der technischen Projektrisiken
 - Konsolidieren und vervollständigen der Architekturbeschreibung
- **Wie soll Lösung realisiert werden.**

Ausarbeitungsphase (Elaboration Phase) III

- **Major Milestone:**
Architekturdefinition des Software-Produktes
 - Beurteilung der Stabilität von Produktdefinition und Software-Architektur
 - Überprüfung der Kosten-, Zeit- und Ressourcen-Planung
- **Output Artefakte:**
 - Umfassendes Use Case Modell mit sämtlichen Akteuren
 - Umfassendes Analysemodell
 - Unvollständiges Designmodell
 - Initiales Verteilungsmodell und Implementationsmodell
 - Komplette Software-Architekturbeschreibung
 - Ausführbarer Architektur-Prototyp
 - Detaillierter Projektplan für den weiteren Projektverlauf

Analysis Workflow

I



Skript:

3-37

Analysis Workflow

II

- Funktionale Beschreibung des Systems aus Entwicklersicht
- Auffinden der zur Use Case Realisierung notwendigen Analyseklassen
- Spezifikation und Unterteilung der Analyseklassen in Stereotype:
 - <<boundary>> Modellierung von Interaktionen zwischen Akteuren und System
 - <<control>> Modellierung von Koordination und Kontrolle mehrerer Objekte
 - <<entity>> Modellierung von (persistenten) Informationen und den dazugehörigen Operationen
- Gruppierung der Analyseklassen
- Vorversion einer Architekturbeschreibung

Skript:

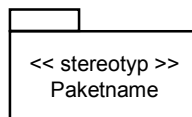
3-38

UML Analysemodell

- **Das Analysemodell umfasst alle UML Diagramme zur Darstellung statischer und dynamischer Aspekte von Analyseklassen.**
- **Methoden zum Auffinden von (Analyse-) Klassen:**
 - CRC-Karten (Class - Responsibility - Collaborator)
Textanalyse (Analysieren von Substantiven, Adjektiven und Verben)
 - Benutzersichtanalyse (Analysieren von Formularen, Listen, Belegen, Eingabemasken)
- **Gruppierung der Analyseklassen mittels UML Packages:**
 - Paketdiagramme
- **Modellierung statischer Aspekte von Analyseklassen:**
 - Klassen- und Objektdiagramme
- **Modellierung dynamischer Aspekte von Analyseklassen:**
 - Use Case Verfeinerung mittels Aktivitätsdiagrammen (Zustandsdiagramme)

UML Notation: Paket (Package)

- Ein UML Paket dient zur beliebigen, logischen Gruppierung von UML Elementen.
- Das Paket ist das einzige UML Gruppierungs-Element (Grouping Thing) und wird für die Modell Dekomposition verwendet.



- Jedes UML Element gehört zu genau einem Package und muss innerhalb diese Package einen eindeutigen Namen haben (entspricht einem C++ Namespace).
- UML definierte Package Stereotypen sind z.B. <<framework>>, <<subsystem>>, etc.

Paketdiagramm (Package Diagram)

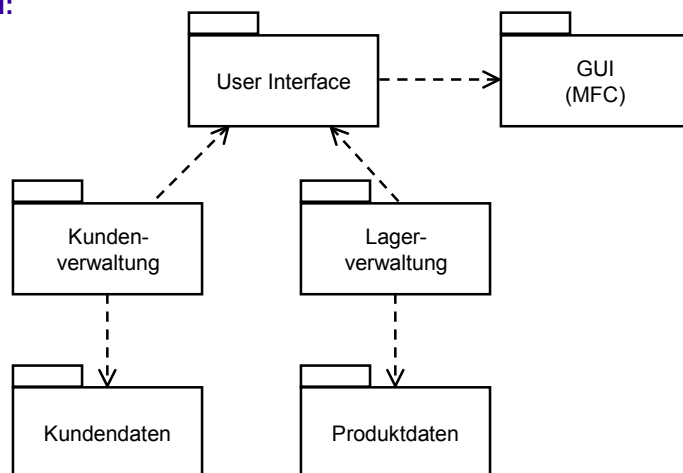
I

- Ein Paketdiagramm zeigt die verschiedenen Pakete (Subsysteme, Module) eines Systems und ihre Abhängigkeit untereinander.
- Ein Paketdiagramm erlaubt die hierarchische Aufteilung und Dekomposition von Modellen.
- Paketdiagramme werden häufig für eine Übersichtsdarstellung in einer Architekturbeschreibung verwendet.

Paketdiagramm (Package Diagram)

II

- Beispiel:



Aktivitätsdiagramm (Activity Diagram)

- Ein Aktivitätsdiagramm zeigt eine State Machine, wobei der Kontrollfluss von Aktivität zu Aktivität hervorgehoben wird.
- Verwendet um dynamische Systemaspekte zu modellieren
- Verwendet für die Analyse von Use Cases
- Verwendet für die Beschreibung von Workflows
- Zur Modellierung von Gleichzeitigkeit und Synchronisation

UML Notation: Aktivität (Activity State)

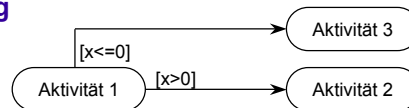
- Eine UML Aktivität ist ein Zustand mit einer intern laufenden Aktion. Bei Abschluss dieser internen Aktion wird automatisch eine Transition (Zustandsübergang) ausgelöst.
- Eine Aktivität ist ein UML Verhaltens-Element (Behavioral Thing) und wird für die Darstellung dynamischer Dinge eines Modells verwendet.



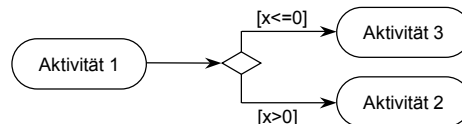
Aktivitätsdiagramm - Verzweigung

- Entscheidungen zwischen Transitionen werden durch Bedingungen unterschieden:

- Normale Darstellung

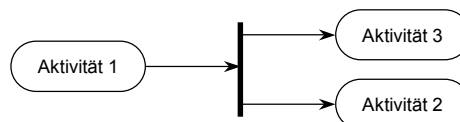


- Branch (bzw. Merge) Darstellung

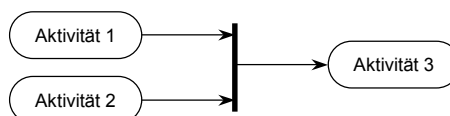


Aktivitätsdiagramm - Fork und Join

- Mit einem Splitting (Fork) werden Transitionen aufgeteilt:
(Ablauf von mehreren gleichzeitigen Aktivitäten)



- Eine Synchronisation (Join) führt Transitionen zusammen:
(Synchronisation von gleichzeitig ablaufenden Aktivitäten)



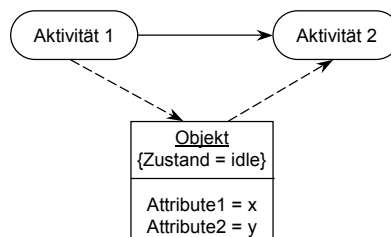
Aktivitätsdiagramm - Start- und Endzustand

- Startzustand (initial state) und Endzustand (final state) einer Aktivität werden mit speziellen Symbolen gekennzeichnet

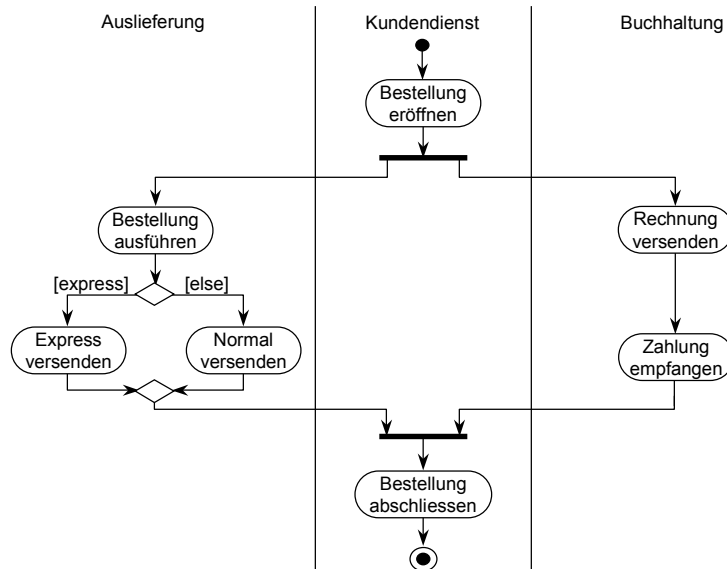


Aktivitätsdiagramm - Objektzustand

- Ein Aktivitätsdiagramm kann zusätzlich Objekte enthalten, um Aktivitäten darzustellen die einen Objektzustand ändern.



Aktivitätsdiagramm - Beispiel



Skript: 3.4.4

3-49

Objektdiagramm (Object Diagram)

I

- Ein Objektdiagramm zeigt die Beziehungen und den Zustand einer Menge von Objekten zu einem bestimmten Zeitpunkt.
- Ein Objektdiagramm wird verwendet um eine statische Momentaufnahme (Instanz) eines Klassendiagramms darzustellen.
- Für ein allgemeines Klassendiagramm existieren beliebig viele Objektdiagramme, deshalb werden meist nur besonders interessante oder prototypische dargestellt (Objekt-Szenario).
- Verwendet zur Darstellung von (prototypischen) Datenstrukturen.
- Häufig für das Reverse-Engineering verwendet.

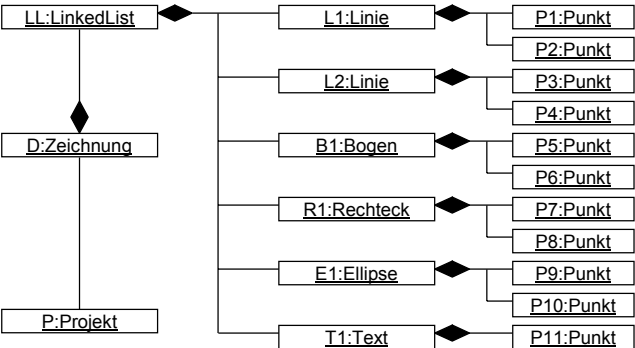
Skript: 3.4.2

3-50

Objektdiagramm (Object Diagram)

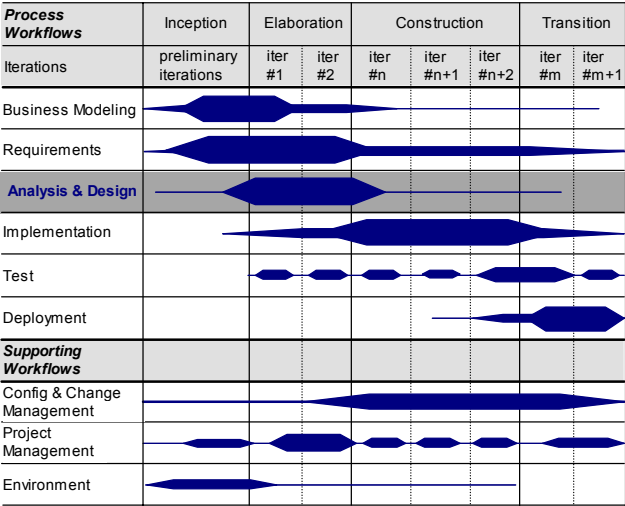
II

- Beispiel eines Objektdiagramms zur Darstellung einer Datenstruktur



Design Workflow

I



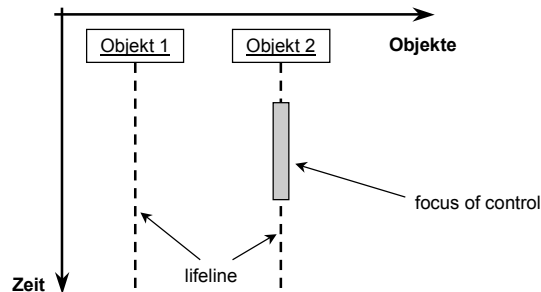
- Systembeschreibung aus Implementationssicht und unter Berücksichtigung von Benutzeranforderungen und Software-Architektur
- Detaillierte Analyse und Spezifikation der Nonfunctional Requirements
- Verfeinerung der Analyseklassen durch Designklassen
- Systemdekomposition in lose gekoppelte Subsysteme
- Interface-Definition für Subsysteme und zukünftige Erweiterungen
- Definition und Spezifikation der verwendeten Software-Architektur

UML Designmodell

- Das Designmodell umfasst alle UML Diagramme zur Darstellung statischer und dynamischer Aspekte von Designklassen
- Verfeinerung und Erweiterung des Analysemodells
- **Modellierung statischer Aspekte von Designklassen:**
 - Verfeinerte Klassen- und Objektdiagramme mit Packages
- **Modellierung dynamischer Aspekte von Designklassen:**
 - Sequenz-, Kollaborations-, Aktivitäts- und Zustandsdiagramme

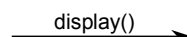
Sequenzdiagramm (Sequence Diagram)

- Ein Sequenzdiagramm zeigt die Interaktionen zwischen einer Anzahl von Objekten, wobei die zeitliche Abfolge der ausgetauschten Nachrichten hervorgehoben wird.



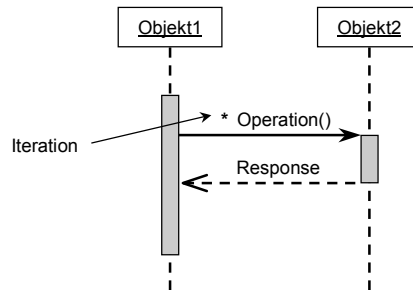
UML Notation: Interaktion (Interaction)

- Eine UML Interaktion ist ein Verhalten (Behavioral Thing), welches den Austausch von Nachrichten zwischen Objekten definiert um ein vorgegebenes Ziel zu erreichen.
- Unter einer Nachricht (Message) versteht man die Übertragung von Informationen von einem Objekt zu einem anderen in der Erwartung eine Aktivität auszulösen.



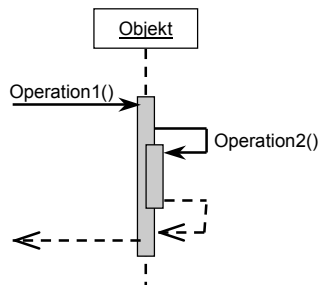
Sequenzdiagramm - Nachrichten

- Beispiel: Objekt1 sendet Objekt2 eine Nachricht (Funktionsaufruf) und erhält nach einer bestimmten Zeit eine Antwort von Objekt2.



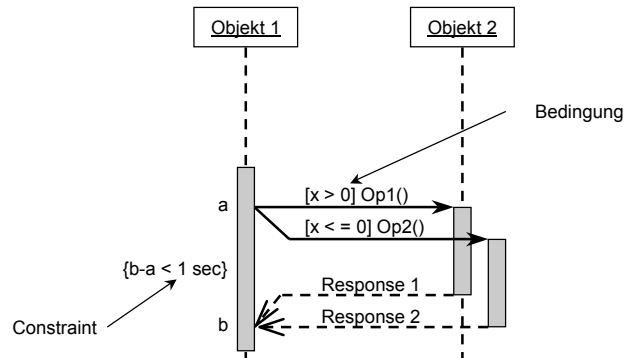
Sequenzdiagramm - Rekursion

- Beispiel: (rekursiver) Aufruf einer eigenen Objektoperation



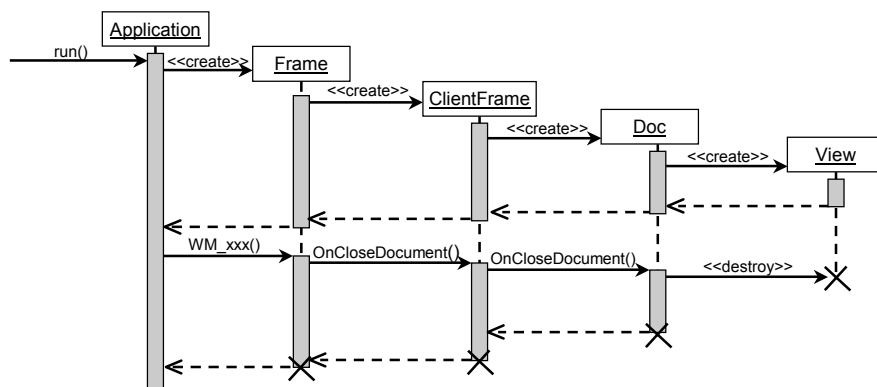
Sequenzdiagramm - Constraints

- Beispiel: Bedingungen und Zusicherungen für das Zeitverhalten



Sequenzdiagramm - Instanziierung

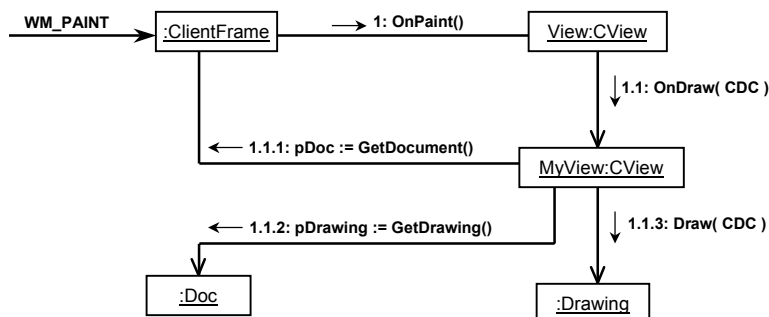
- Beispiel: Dynamische Instanziierung und Zerstörung von Objekten



Kollaborationsdiagramm (Collaboration)

- Ein Kollaborationsdiagramm zeigt, wie das Sequenzdiagramm, die Interaktionen zwischen einer Anzahl von Objekten, wobei die Organisation der beteiligten Objekte hervorgehoben wird.
- Zur Modellierung des Verhaltens mehrerer Objekte verwendet.
- Sequenz- und Kollaborationsdiagramm (Interaktionsdiagramme) sind semantisch gleichwertig:
 - Das Sequenzdiagramm zeigt die zeitliche Abfolge der Interaktionen.
 - Das Kollaborationsdiagramm betont die Organisation der an der Interaktionen beteiligten Objekte.
 - Beide Diagramme können ineinander überführt werden.

Kollaborationsdiagramm - Beispiel



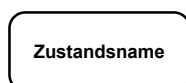
Zustandsdiagramm (Statechart Diagram)

- Ein Zustandsdiagramm zeigt eine State Machine, wobei Zustände und Transitionen (Zustandsübergang) hervorgehoben werden.
- Unter einer Transition versteht man einen Zustandsübergang von einem Ausgangs- Zustand (Source State) in einen weiteren Zustand (Target State) beim Eintreffen eines bestimmten Events. Dabei wird eine definierte Aktion ausgeführt.
- Verwendet um dynamische Systemaspekte zu modellieren
- Modellierung des Zustandes eines Objektes oder eines Systems
- Häufig verwendet zur Modellierung der User Interface Funktionalität

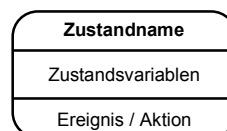
UML Notation: Zustand (State)

- Ein UML Zustand (Behavioral Thing) wird verwendet um einen bestimmten Status einer Zustandsmaschine darzustellen.
- Eine Zustandsmaschine (State Machine) beschreibt das Verhalten eines oder mehrere Objekte durch eine Abfolge von Zuständen die als Antwort auf Ereignisse (Events) durchlaufen werden.

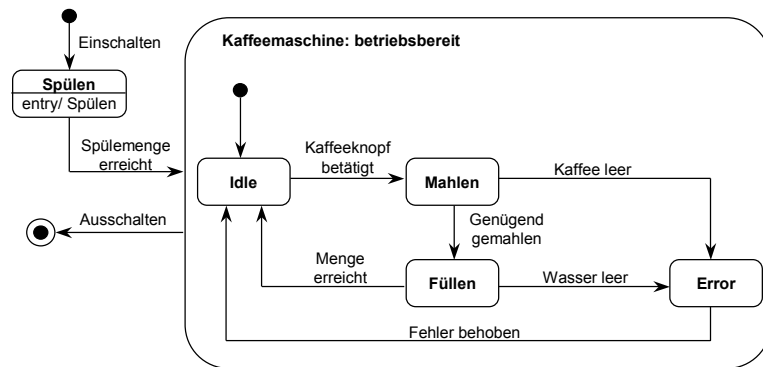
- Einfache Darstellung eines Zustandes:



- Allgemeine Darstellung eines Zustandes:



Zustandsdiagramm - Beispiel



UML Prozessmodell

- Das Prozessmodell umfasst alle UML Diagramme zur Darstellung von Prozessen oder Threads sowie deren Mechanismen für Gleichzeitigkeit (Concurrency) und Synchronisation.
- Modellierung von Prozessen oder Threads:
 - Zustandsdiagramme mit aktiven Klassen oder Objekten
 - Sequenzdiagramme mit aktiven Klassen oder Objekten
 - Kollaborationsdiagramme mit aktiven Klassen oder Objekten
- Modellierung von Gleichzeitigkeit und Synchronisation:
 - Aktivitätsdiagramme

UML Notation: Aktive Klasse (Active Class)

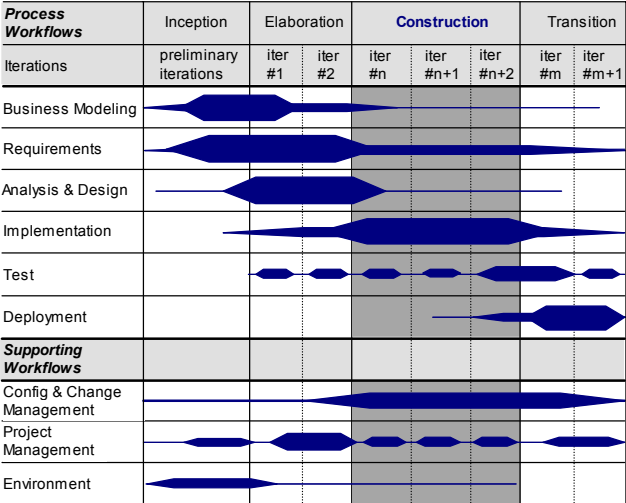
- Eine aktive Klasse ist eine Klasse deren instanzierte Objekte einen oder mehrere Prozesse oder Threads besitzen.

- Aktive Klasse:
- Aktives Objekt:



- Prozesse und Threads werden in UML als Klassen mit den UML vordefinierten Stereotypen <<process>> bzw. <<thread>> dargestellt.

Konstruktionsphase (Construction Phase) I



Konstruktionsphase (Construction Phase) II

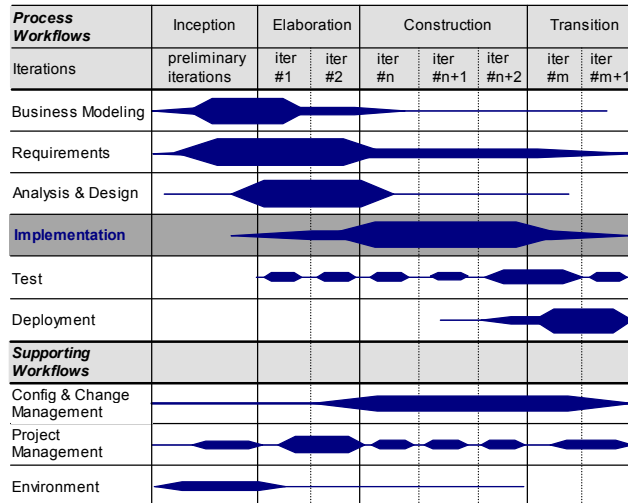
- Die Konstruktionsphase umfasst alle Aktivitäten um eine komplette Vorversion des Software-Produktes fertigzustellen:
 - Erstellen des User Interface
 - Spezifizieren zusätzlicher, architekturkompatibler Use Cases
 - Ausarbeiten und vervollständigen der Designklassen
 - Implementation der Use Cases (Klassen, Subsysteme)
 - Testen der Komponenten und des Gesamtsystems
 - Sichern der Software-Qualität

Konstruktionsphase (Construction Phase) III

- Major Milestone:
Fertigstellung einer Vorversion (Initial Operational Capability)
 - Beurteilung von Stabilität und Vollständigkeit der Vorversion
- Output Artefakte:
 - Software-Produkt Beta Version
 - Beta Release Beschreibung
 - Umfassendes, konsistentes Analyse-, Design-, Implementations- und Verteilungsmodell
 - Projektplanung für die Software-Einführung
 - Vorversion der Benutzerdokumentation

Implementation Workflow

I



Skript:

3-71

Implementation Workflow

II

- Implementation des Designmodells unter Berücksichtigung von Software-Architektur und Komponentenaufteilung (Software-Reuse)
- Software-Test auf Level von Klassen und Operationen (Unit Test)
- Implementation des User Interface
- Inkrementelle Implementation neuer Produkte-Features (Use Cases)
- Inkrementelle Integration neuer Systembestandteile
- Erstellen eines "Make-Scripts" für Software-Build und Integration

Skript:

3-72

UML Implementationsmodell

- Das Implementationsmodell umfasst alle UML Diagramme zur Darstellung physischer Systembestandteile und deren Verteilung
- Modellierung von physischen Systemkomponenten sowie deren Schnittstellen untereinander:
 - Verteilungsdiagramme (Verteilung der Systemkomponenten)
 - Komponentendiagramme (Abhängigkeit und Interfaces zwischen Systemkomponenten)
- Modellierung des Verhaltens von Systemkomponenten:
 - Zustandsdiagramme (Sequenzdiagramme, Aktivitätsdiagramme)

3-73

UML Notation: Komponente (Component)

- Eine Komponente ist ein ausführbarer, physischer Systembestandteil, welcher vordefinierte Schnittstellen zur Verfügung stellt und im allgemeinen austauschbar ist.

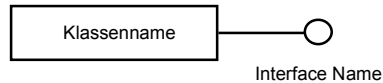


- UML definierte Stereotypen für Komponenten sind z.B. <<executable>>, <<document>>, <<file>> (mit entsprechendem Icon) etc.

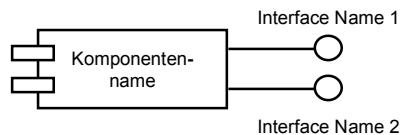
UML Notation: Schnittstelle (Interface) I

- Ein Schnittstelle stellt eine benannte Sammlung von Funktionen dar, welche von einer bestimmten Klasse oder Komponente zur Verfügung gestellt wird.

- Klasse mit einer Schnittstelle:



- Komponente mit zwei Schnittstellen:

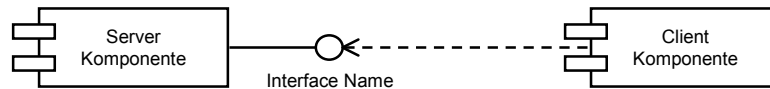


Skript: 3.2.1.3

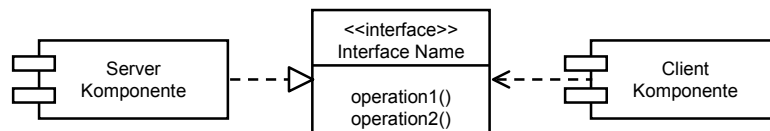
3-75

UML Notation: Schnittstelle (Interface) II

- Darstellung mit Interface Symbol:



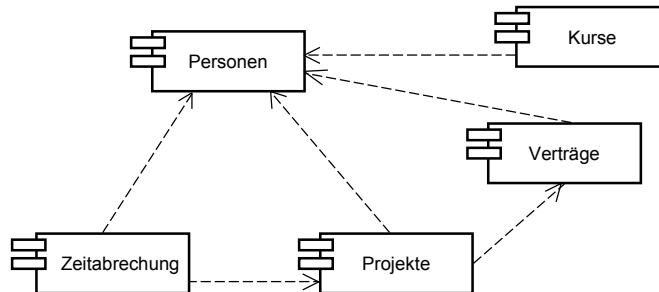
- Erweiterte Darstellung durch Klasse mit <<interface>> Stereotyp:



3-76

Komponentendiagramm (Component Diagram)

- Ein Komponentendiagramm zeigt die Abhängigkeiten zwischen den verschiedenen Komponenten eines Systems



Skript: 3.4.8

3-77

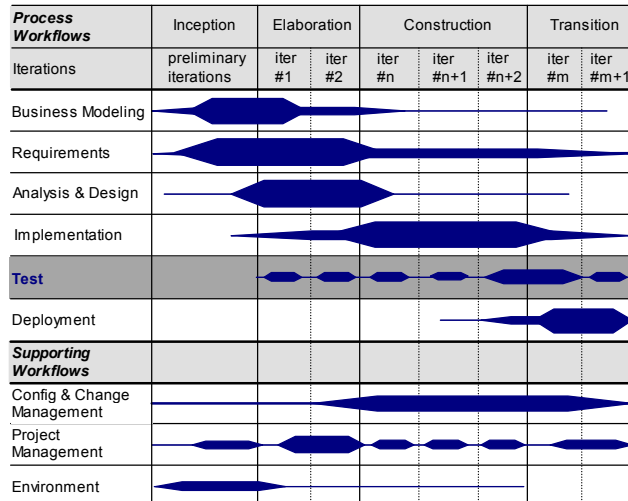
Implementation von UML Diagrammen

- **Implementation von UML Klassen:**
 - Durch ausreichende Verfeinerung der Designklassen können diese direkt implementiert werden. (Klassen, Attribute und Operationen mit Angabe der Sichtbarkeit, abstrakte Implementation, Daten-Typen, etc.)
- **Implementation von UML Klassenbeziehungen:**
 - Generalisierung: Vererbungsmechanismus der Implementationssprache
 - Assoziation: (Gegenseitiger) Pointer zwischen den beteiligten Klassen
 - Aggregation: Im allg. nur ein Pointer von der "Aggregat-" Klasse zur "Teil-" Klasse
 - Komposition: Dynamische Instanziierung und Zerstörung der "Teil-" Klasse oder Deklaration als lokales Objekt
 - Abhängigkeit: Unabhängige Klasse als Parameter einer Klassenoperation
 - Realisierung: Keine direkte Implementation der Beziehung (z.B. Kollaboration)
- **Implementation von UML Beziehungsmultiplizitäten:**
 - 0..1 realisiert mit Pointer, 0..n realisiert mit Liste, 0..i realisiert mit Array der Grösse i

3-78

Test Workflow

I



Skript:

3-79

Test Workflow

II

- Ein vollständiger Systemtest muss gemäss einem definierten Testplan erfolgen und umfasst die folgenden Testaktivitäten:
- **Usability Testing**
 - Zur Sicherstellung der Benutzerfreundlichkeit muss das Testen der Benutzerschnittstelle möglichst frühzeitig während dem Design und der Implementation des User Interface durchgeführt werden (Prototyp).
- **Integration Testing**
 - Inkrementeller Integrationstest für jeden neuen (Minor-) Software-Release
- **Regression Testing**
 - Vollständiger Systemtest aufbauend auf Use Case basierten Test-Cases, Testprozeduren und Test-Skripts sowie den davon erwarteten Testresultaten.
 - Automatisierte Teststrategie, bei welcher zuvor durchgeführte Tests mit einer neuen Software-Version wiederholt werden, um zu verifizieren, dass bekannte Fehler behoben und keine neuen Fehler aufgrund von Nebeneffekten hinzugekommen sind.

Skript:

3-80

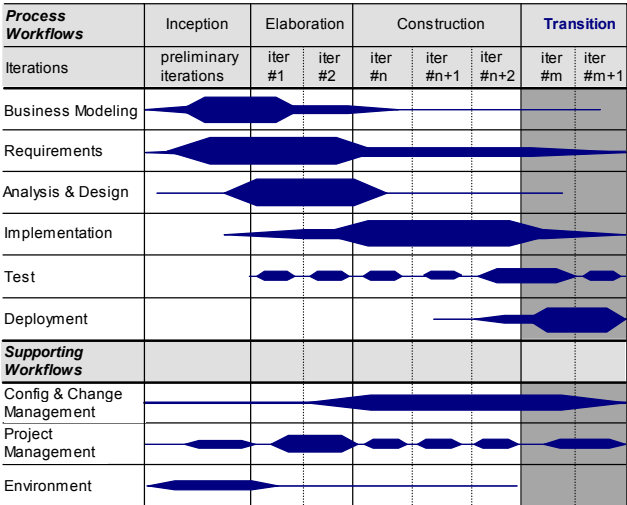
Test Workflow

III

- System Testing**
 - Testen des Gesamtsystems inklusive externen Systemen und Datenbeständen
- Acceptance Testing**
 - Testen des Gesamtsystems beim Endbenutzer um sicherzustellen, dass die Applikation durch den Benutzer erwartungsgemäss eingesetzt werden kann.
 - Dies beinhaltet z.B. eine verständliche Online-Hilfe, ausreichende Performance, keine Kompatibilitätsprobleme mit Fremdapplikationen (DLL Versionen), einfache Integration in bestehende Benutzerumgebung, etc.
 - Verifizieren der vollständigen Implementation sämtlicher Benutzeranforderungen

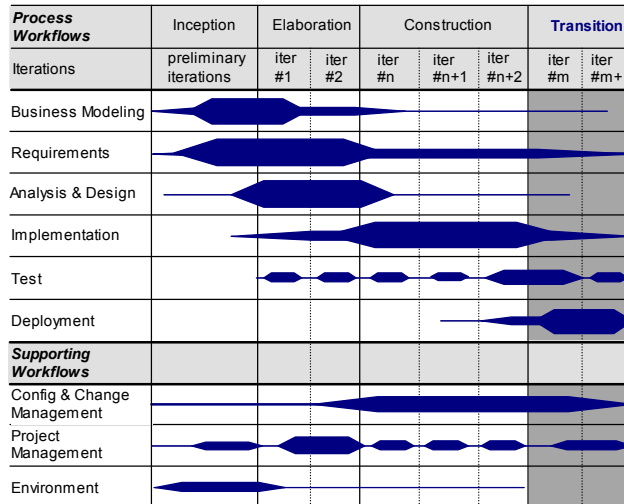
Übergangsphase (Transition Phase)

I



Übergangsphase (Transition Phase)

I



Skript: 1.4.3

3-83

Übergangsphase (Transition Phase)

II

- Die Übergangsphase umfasst alle Aktivitäten um das fertige Software-Produkt an den Endbenutzer zu übergeben:
 - Durchführen von Beta-Tests mit ausgewählten Endbenutzern
 - Durchführen von Tests im Parallelbetrieb mit einem allenfalls bestehenden System
 - Konvertieren von bestehenden Datenbanken oder anderen Datenbeständen
 - Beheben von Fehlern und optimieren der Systemperformance
 - Schulen der Endbenutzer, Administratoren etc.
 - Aufbauen des Benutzer-Supports
 - Überarbeiten und anpassen der Benutzerdokumentation
 - Vorbereiten der Produktion des Software-Produktes
 - Vorbereiten der Aktivitäten zum Marketing, Verkauf und Vertrieb des Produktes
 - Inbetriebnahme und übergeben des Software-Produktes an die Endbenutzer

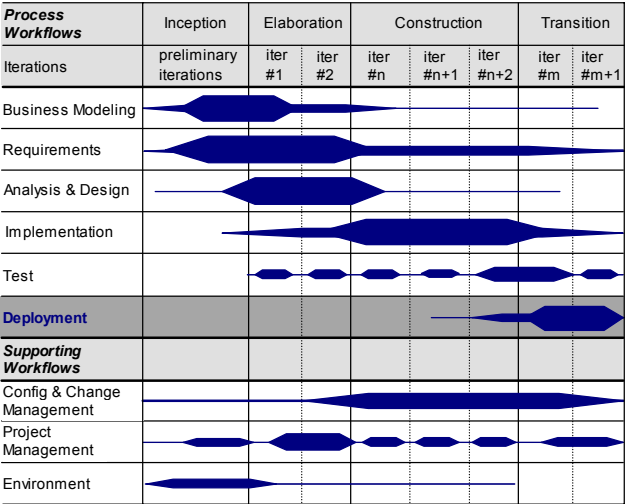
Skript: 1.4.3

3-84

Übergangsphase (Transition Phase)
III

- Major Milestone:**
Produktfreigabe (Product Release)
 - Software-Produktabnahme gemäss spezifizierten Anforderungen
- Output Artefakte:**
 - Vollständiges und stabiles Software-Produkt mit Installationsprogramm
 - Komplette Systemdokumentation (Architektur- und Modellbeschreibungen)
 - Software-Lizenz Dokument
 - Vollständige Benutzerdokumentation

Deployment Workflow
I



Deployment Workflow

II

- Planung der Einführung, Verteilung und Inbetriebnahme des vollständigen Software-Produktes (Beta Version)
- Planung und Durchführung von Software Beta-Tests
- Erstellen eines Installations- bzw. Update-Programms
- Migration von bestehender Software und bestehenden Daten
- Abnahme des Software-Produktes durch den Auftraggeber

Skript:

3-87

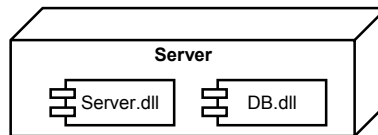
UML Deployment Modell

- Das Deployment Modell umfasst alle UML Diagramme zur Darstellung der Verteilung und Inbetriebnahme eines Softwaresystems.
- **Modellierung der Systemverteilung und Organisation:**
 - Verteilungsdiagramme mit aktiven Klassen und Komponenten
- **Modellierung der dynamischen Aspekte:**
 - Aktivitätsdiagramme, Zustandsdiagramme

3-88

UML Notation: Knoten (Node)

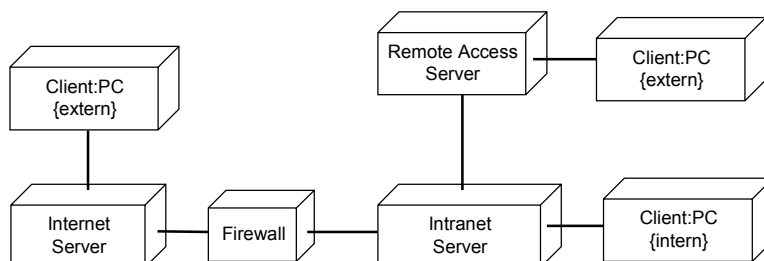
- Ein Knoten beschreibt eine physische Computer-Ressource, welche zur Laufzeit im System existiert (Rechner oder ähnliches Gerät, meist mit CPU und Memory).



- Knoten sind Elemente auf welchen (Software-) Komponenten ausgeführt werden und mittels welchen die physischen Verteilung dieser Komponenten dargestellt werden kann (Verteilte Systeme).
- Knoten werden sehr häufig durch Stereotypen mit entsprechenden Icons gezeichnet (z.B. PC, Terminal, RAID-Subsystem, etc.).

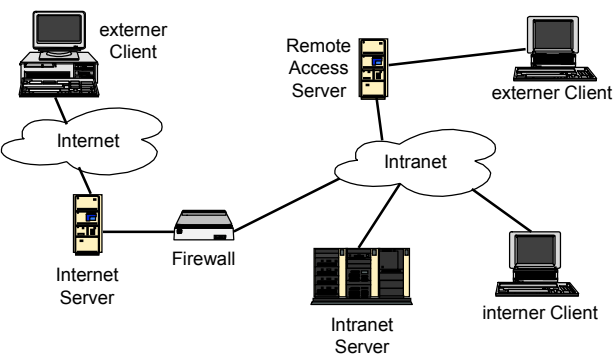
Verteilungsdiagramm (Deployment Diagram) I

- Ein Verteilungsdiagramm zeigt die Hardware-Konfiguration und Topologie auf welcher die zu entwickelnde Software laufen soll.



Verteilungsdiagramm (Deployment Diagram) II

- Zur besseren Illustration werden Verteilungsdiagramme häufig mit Stereotyp Icons dargestellt.



Supporting Workflows I

Process Workflows	Inception	Elaboration		Construction			Transition	
Iterations	preliminary iterations	iter #1	iter #2	iter #n	iter #n+1	iter #n+2	iter #m	iter #m+1
Business Modeling	[Bar]							
Requirements	[Bar]							
Analysis & Design		[Bar]						
Implementation			[Bar]					
Test			[Bar]	[Bar]	[Bar]	[Bar]	[Bar]	[Bar]
Deployment						[Bar]		
Supporting Workflows								
Config & Change Management			[Bar]					
Project Management		[Bar]	[Bar]	[Bar]	[Bar]	[Bar]	[Bar]	[Bar]
Environment	[Bar]							

- **Software Configuration Management Workflow:**
 - Versionsverwaltung sämtlicher wichtiger Artefakte (Spezifikationen, Source Code, etc.)
 - Unterstützung durch entsprechende Software-Tools
- **Change Request Management Workflow:**
 - Definiertes Vorgehen beim Durchführen wesentlicher Änderung bzw. Erweiterung
 - Abschätzung der Konsequenzen einer Änderung bzw. Erweiterung
- **Project Management Workflow:**
 - Phasenplanung (Major Milestones) und Iterationsplanung (Minor Milestones)
 - Frühzeitige Minimierung von potentiellen Projektrisiken
- **Environment Workflow:**
 - Evaluation und Beschaffung von Entwicklungsumgebung und Software-Tools
 - Anpassen von Software-Prozess und Infrastruktur an projektspezifische Bedürfnisse

Teil 4

Design Patterns

4-1

Teil 4 - Inhalt

- Definition von Design Patterns
- Klassifikation von Design Patterns
- Design Patterns Katalog
- Singleton Idiom
- Iterator Design Pattern
- Composite Design Pattern
- Model-View-Controller Architectural Pattern

4-2

Design Patterns - The Beginning

- **Design Patterns - Elements of Reusable Object-Oriented Software**
 - E. Gamma, R. Helm, R. Johnson, J. Vlissides - "Gang of Four" (GOF) 1994
 - Katalog von 23 sehr häufig verwendeten, generischen Software Design Patterns
- **Analogie zu anderen Ingenieur-Disziplinen**
 - C. Alexander's Pattern Language für Architekten
 - Engineering Handbücher mit allgemeinen Lösungen für häufig auftretende Probleme
- **Anfang eines Pattern-Booms in der Informatik**
 - Pattern Mining
 - Analysis Patterns
 - Project Management Patterns
 - Anti-Patterns

Design Patterns - Definition

I

- Ein (Design-) Pattern beschreibt eine häufig auftretende Musterlösung für ein häufig vorkommendes (Design-) Problem
- Dokumentation von Know-How und Erfahrung beim Software-Design
- Universales Design-Vokabular für Software-Entwickler
- Generische und Problem-Domain spezifische Design Patterns
- Verwendung auf unterschiedlichen Abstraktionsebenen
 - Idioms (Implementationsebene)
 - Design Patterns (Software Designebene)
 - Architectural Patterns (Software Architekturebene)

Design Patterns - Definition

II

- **Idiom Definition:**

- Ein Idiom ist ein meist sprachabhängiges low-level Pattern, welches die Implementation von bestimmten Aspekten einer Software-Komponente beschreibt.

- **Design Pattern Definition:**

- Ein Design Pattern (Entwurfsmuster) beschreibt eine häufig auftretende Struktur von miteinander interagierenden Objekten zur sprachunabhängigen Lösung eines allgemeinen Design-Problems in einem spezifischen Kontext.

- **Architectural Pattern Definition:**

- Ein Architectural Pattern (Architekturmuster) ist ein sprachunabhängiges high-level Pattern, welches die grundlegende Struktur eines Software-Systems und die Beziehungen der dazugehörigen Subsysteme beschreibt.

Design Patterns - Vorteile

- **Vorteile von Design Patterns:**

- + Erhöhen den Software-Reuse
- + Erlauben die generische Lösung von spezifischen Design-Problemen
- + Erleichtern die Erweiterung und Anpassung an neue Anforderungen
- + Können helfen Design-Fehler zu vermeiden
- + Verbessern die Code Dokumentation
- + Erleichtern das Reverse-Engineering und das Verstehen von fremdem Code

- **Nachteile von Design Patterns:**

- Können die Komplexität erhöhen
- Können das Laufzeitverhalten verschlechtern

Design Patterns - GOF Klassifikation

- Eine Klassifikation der grossen Anzahl von Patterns erleichtert die Suche eines geeigneten Patterns für ein konkretes Problem.
- **Creational Design Patterns (Beispiel Singleton)**
 - Verwendet für das kontrollierte Instanzieren von neuen Objekten
 - Kapselung der konkret erzeugten Objekte
 - Verbessern die Unabhängigkeit bei der Erzeugung von komplexen Objektstrukturen
- **Behavioral Design Patterns (Beispiel Iterator)**
 - Verwendet für die Zuordnung von Verhaltensabläufen zu Objekten
 - Muster für die Kommunikation und den Kontrollfluss zwischen Objekten
- **Structural Design Patterns (Beispiel Composite)**
 - Verwendet um komplexen Objektstrukturen aufzubauen oder zu verbinden
 - Erlauben den dynamischen Aufbau von Objektstrukturen (zur Laufzeit)

GOF Katalog - Creational Patterns Beispiele

- **Singleton Idiom**
 - Ensure a class only has one instance, and provide a global point of access to it.
- **Abstract Factory**
 - Provide an interface for creating families of related or dependent objects without specifying their concrete classes.
- **Factory Method Idiom**
 - Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.
- **Builder**
 - Separate the construction of a complex object from its representation so that the same construction process can create different representations.

GOF Katalog - Behavioral Patterns Beispiele

- **Iterator**

- Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

- **Chain of Responsibility**

- Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.

- **Command**

- Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

- **Observer**

- Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

Skript: 7.1.4

4-9

GOF Katalog - Structural Patterns Beispiele

- **Composite**

- Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

- **Adapter**

- Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

- **Facade**

- Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.

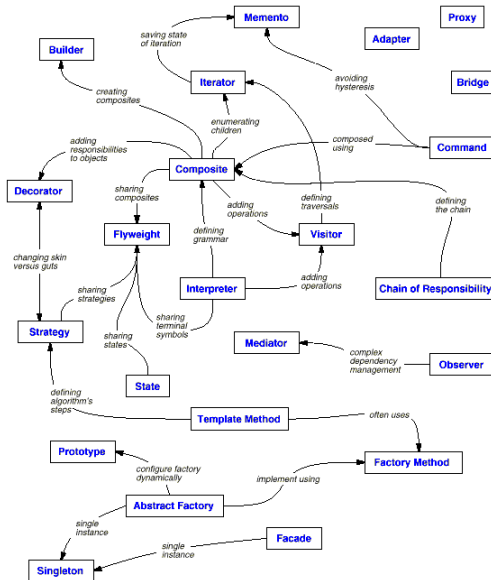
- **Flyweight**

- Use sharing to support large numbers of fine-grained objects efficiently.

Skript: 7.1.4

4-10

GOF Katalog - Patterns Beziehungen



4-11

Architectural Patterns Beispiele

• Model-View-Controller (MVC)

- Das MVC Pattern kann für die Implementation von interaktiven GUI Applikationen verwendet werden.

• Layers

- Das Layers Architectural Pattern hilft eine Applikation zu strukturieren, die in verschiedene Gruppen von Aufgaben unterteilt werden kann, wobei jede dieser Gruppen Aufgaben eines unterschiedlichen Abstraktionslevels übernimmt.

• Pipe-And-Filters

- Das Pipe-And-Filters Pattern kann für Applikationen verwendet werden, deren Hauptaufgabe in der schrittweisen Verarbeitung eines Datenstroms besteht.

• Microkernel

- Das Microkernel Pattern kann für Software-Systeme verwendet werden, welche aus einer Vielzahl von Funktionen bestehen, die alle auf einer kleinen Kernfunktionalität aufbauen.

Singleton Idiom

I

• Zweck

- Sicherstellung dass von einer Klasse nur eine einzige Instanz erzeugt wird.
- Bereitstellung eines globalen Zugriffspunktes auf diese Instanz.

• Anwendung

- Klassen, die ein Gerät oder eine Hardware-Schnittstelle bedienen.
- Klassen, die eine applikationsweite Servicefunktion übernehmen.
- Beispiel: Window Manager in einem GUI.

• Konsequenzen

- + Kontrolle der Instanzierung liegt bei der Singleton-Klasse und nicht beim Client.
- + Keine globale Variable für ein einzelnes Objekt notwendig.
- + Zeitpunkt der Instanzierung ist frei wählbar (im Gegensatz zu globalen Objekten).
- + Durch Vererbung kann diese Funktionalität an abgeleitete Klassen weitergegeben werden.

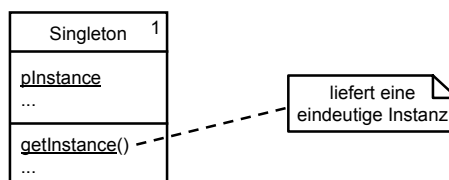
Skript: 7.2

4-13

Singleton Idiom

II

• Struktur



• Varianten

- Es soll nicht nur eine einzige Instanz einer Klasse erzeugt werden sondern eine genau definierte Anzahl.

Skript: 7.2

4-14

Singleton Idiom

III

- Implementation - Deklaration einer Singleton Klasse in C++

```
class Singleton
{
    private:
        static Singleton* pInstance;
    protected:
        Singleton();
    public:
        static Singleton* getInstance();
};
```

Skript: 7.2

4-15

Singleton Idiom

IV

- Implementation - Member-Funktion getInstance() in C++

```
// implement static member to hold pointer to instance
Singleton* Singleton::pInstance = 0;

// implementation of class method to instantiate object
Singleton* Singleton::getInstance()
{
    if ( pInstance == 0 )
    {
        pInstance = new Singleton;
    }
    return pInstance;
}
```

- Implementation - Client-Zugriff auf Singleton-Klasse

```
Singleton* pUniqueInstance = Singleton::getInstance();
```

Skript: 7.2

4-16

Iterator Design Pattern

I

• Zweck

- Erlaubt einen sequentiellen Zugriff auf eine Aggregatsstruktur (z.B. Daten-Container) ohne die zugrundeliegende Datenstruktur kennen zu müssen.
- Die Grundidee dieses Patterns besteht darin die Verantwortlichkeit für Traversierung und Zugriff auf eine Datenstruktur nicht bei der Aggregatsklasse zu implementieren, sondern an eine Iterator-Klasse zu delegieren

• Anwendung

- Unterstützt die Verwendung unterschiedlicher Algorithmen für die Traversierung
- Einheitliches Interface für die Traversierung unterschiedlicher Aggregatsstrukturen

• Konsequenzen

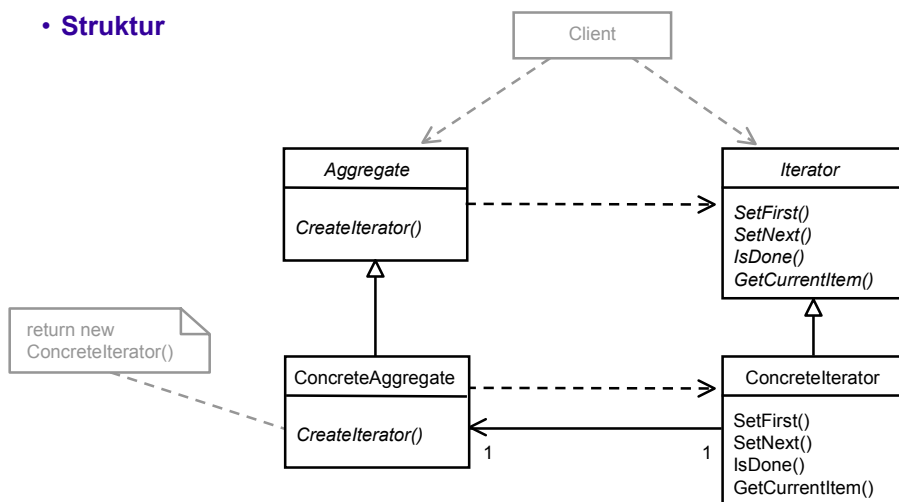
- Vereinfacht das Interface für den Zugriff auf eine Aggregatsstruktur
- Erlaubt die Erzeugung von Iteratoren für die unterschiedliche Traversierung einer Aggregatsstruktur (z.B. ForwardIterator bzw. ReverseIterator).
- Eine Aggregatsstruktur kann gleichzeitig von mehreren (auch unterschiedlichen) Iteratoren durchlaufen werden.

4-17

Iterator Design Pattern

II

• Struktur



4-18

Composite Design Pattern

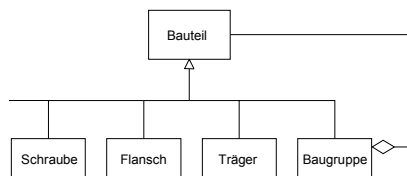
I

• Zweck

- Erlaubt den Aufbau von hierarchischen Objektstrukturen mit Ganzes-Teil Beziehung
- Transparenter Client-Zugriff auf einfache Objekte und Container-Objekte

• Anwendung

- Aggregation von Gruppen gleichartiger Teile
- Graphische Elemente in einem Zeichnungsprogramm
- Files in einer verschachtelten Directory-Struktur
- Beispiel: Stücklistenbeschreibung



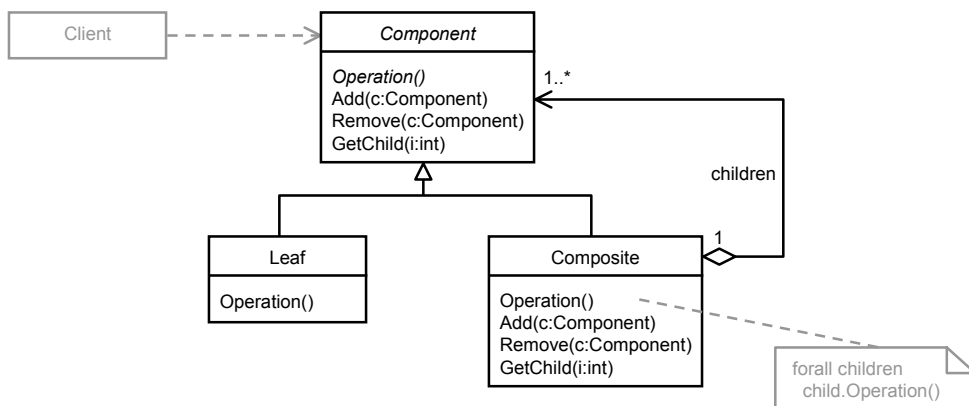
Skript: 7.3

4-19

Composite Design Pattern

II

• Allgemeine Struktur



Skript: 7.3

4-20

• Konsequenzen

- + Einfacher Aufbau und lose Kopplung des Clients mit der Objektstruktur
 - + Einfache Erweiterung der Leaf-Klassenhierarchie ohne Änderung des Clients
 - + Einfache Implementation rekursiver Funktionen für die gesamte Objekthierarchie
 - + Einfache Objekte können zu komplexen Objektstrukturen zusammengesetzt werden
 - + Ein Client muss nur die Root-Komponente kennen um den Baum zu traversieren
-
- Gegenseitige Abhängigkeiten einzelner Komponenten sind nur schwer zu gewährleisten
 - Aufbau der Baumstruktur wird durch das Pattern nicht gelöst

• Varianten

- Komposition anstelle von Aggregation
- Art und Anzahl der Operationen und Attribute

• Zweck

- Architekturmodell für interaktive (GUI-) Applikationen
- Entkoppelung von User-Interface und Kernfunktionalität der Applikation

• Anwendung

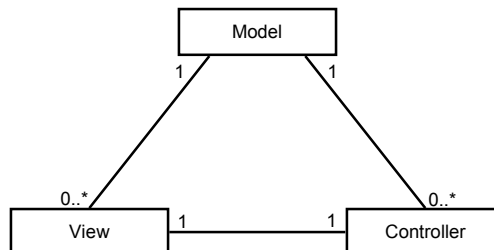
- Geeignetes Pattern für interaktive Systeme
- Erlaubt die einfache Änderung und Erweiterung der Benutzeroberfläche
- Erhöht Portierbarkeit auf mehrere GUI-Systeme
- Einfachere Erweiterbarkeit der Applikationskernfunktionalität

Model-View-Controller Pattern

II

• Struktur

- **Model:** kapselt Daten und Kernfunktionalitäten, unabhängig von speziellen, interaktiven Ausgabeformen oder Eingabemöglichkeiten (Verarbeitung).
- **View:** zeigt dem Benutzer die Informationen (Daten) des Models in einer oder mehreren Ansichten (Ausgabe).
- **Controller:** behandelt Benutzereingaben (Maus, Tastatur) und leitet diese an Model und/oder View weiter wo sie verarbeitet werden (Eingabe).



Skript: 7.4

4-23

Model-View-Controller Pattern

III

• Konsequenzen

- + Erlaubt mehrere (auch dynamisch erzeugte) Ansichten eines (Daten-) Modells.
- + Update-Mechanismus erlaubt eine korrekte Synchronisation aller Views und Controller.
- + Kernfunktionalität der Applikation ist GUI unabhängig und lässt sich einfach portieren.
- + Erweiterbar zu einem (GUI-) Applikations-Framework.
- Erhöhte Komplexität für einfache Applikationen.
- Unter Umständen erfolgt eine übermäßige Anzahl von Update-Messages.
- Enge Kopplung zwischen View und Controller.
- Langsames (Re-) Display aufgrund eines ineffizienten Zugriffs auf Modelldaten.

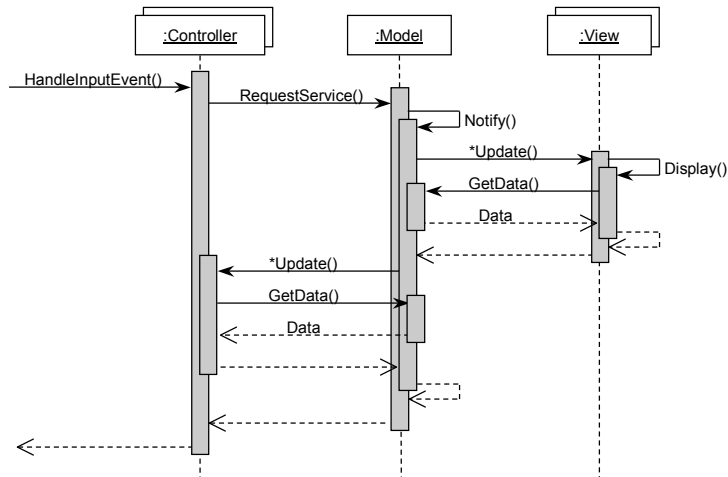
• Varianten

- Document-View Pattern: Trennung von View und Controller wird aufgehoben. Das Pattern besteht nur noch aus Model (Document) und View.

Skript: 7.4

4-24

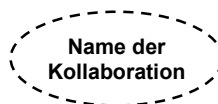
• Implementation - Sequenzdiagramm



4-25

UML Notation: Kollaboration (Collaboration)

- Eine Kollaboration definiert eine Ansammlung von Klassen, Interfaces und anderen UML-Elementen, welche zusammen eine übergeordnete Aufgabe realisieren.

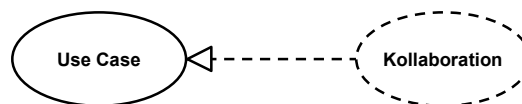


• Anwendung

- Darstellung der Klassen für eine Use Case Realisierung
- Darstellung der zu einem Design Pattern dazugehörigen Klassen

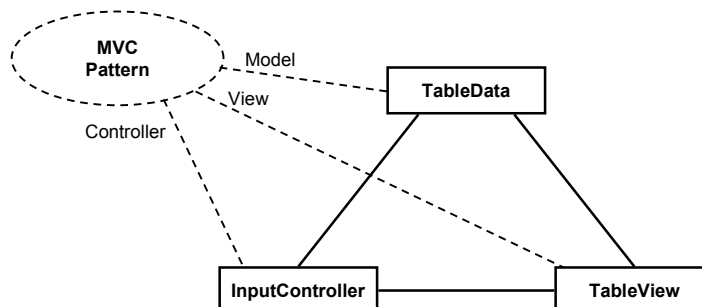
UML Notation: Realisierung (Realization)

- Die Realisierungsbeziehung beschreibt eine semantische Verbindung zwischen einer Spezifikation und ihrer Implementation
- Beispiele:
 - Realisierung eines Interface durch eine Klasse oder Komponente
 - Realisierung eines Use Case durch eine Kollaboration



Kollaboration - Design Pattern

- Anwendung - Darstellung von Design Patterns
 - Beispiel: Model-View-Controller Pattern



Teil 5

Schlussbetrachtung

5-1

Teil 5 - Inhalt

- Entwicklungsprozess und Software-Qualität
- RUP Zusammenfassung
- UML Zusammenfassung
- UML Ausblick
- Übersicht UML Tools (Übung: Rational Rose)

5-2

Entwicklungsprozess und Software-Qualität

- **Software-Qualität ist stark abhängig vom Software-Prozess**
- **Die Verbesserung der Software-Qualität erfordert die Verbesserung des Software-Prozesses**
- **Standards für Software-Prozessverbesserung**
 - ISO-9000 Qualitätsmanagement
 - CMM (Capability Maturity Models)
 - SPICE (Software Process Improvement Capability dEtermination)

5-3

ISO-9000 Qualitätsmanagement

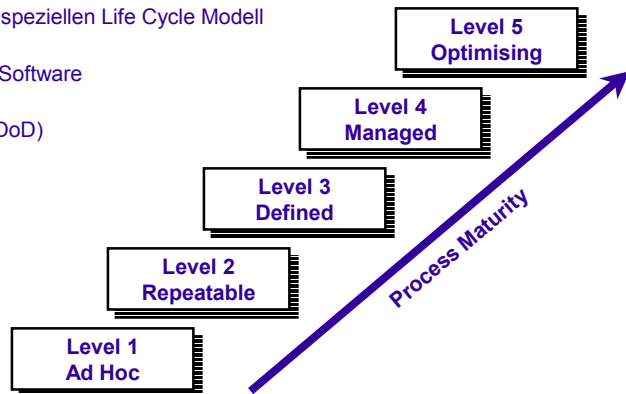
- **International Standards Organisation ISO-9000**
 - Allgemeiner ISO Standard für Qualitätsmanagement (nicht Software spezifisch)
 - ISO-9001 Qualitätsstandard auch auf Software anwendbar (ISO-9000-3, 1991)
 - Der Standard definiert welche Qualitätsanforderungen erfüllt sein müssen, aber nicht wie diese zu realisieren sind (firmen- bzw. branchenspezifisch).
 - Erfordert umfassende Dokumentation des Entwicklungsprozesses und ständige Weiterbildung der Mitarbeiter
 - Zertifizierbar durch SAQ (Schweizerische Arbeitsgemeinschaft für Qualitätsförderung)
 - Grosse Verbreitung in Europa, etwas weniger in den USA

5-4

Capability Maturity Models

- **Capability Maturity Models (CMMs)**

- SEI (Software Engineering Institute) Richtlinien zur Verbesserung von Software- und Managementprozessen
- Unabhängig von einem speziellen Life Cycle Modell
- SW-CMM Standard für Software
- In den USA verbreitet (DoD)



5-5

SPICE

- **Software Process Improvement Capability dEtermination (SPICE)**

- EU Initiative zur Verbesserung des Software-Prozesses (ca. 24 Länder)
- SPICE erweitert die bestehenden Software-Prozess Richtlinien von ISO-9000 und CMM durch die Definition von allgemeinen Verfahren zur Beurteilung und Bewertung von Software-Prozessen (Software Process Assessment).
- Etablierung als internationaler Standard (ISO/IEC 15504, 1997)

5-6

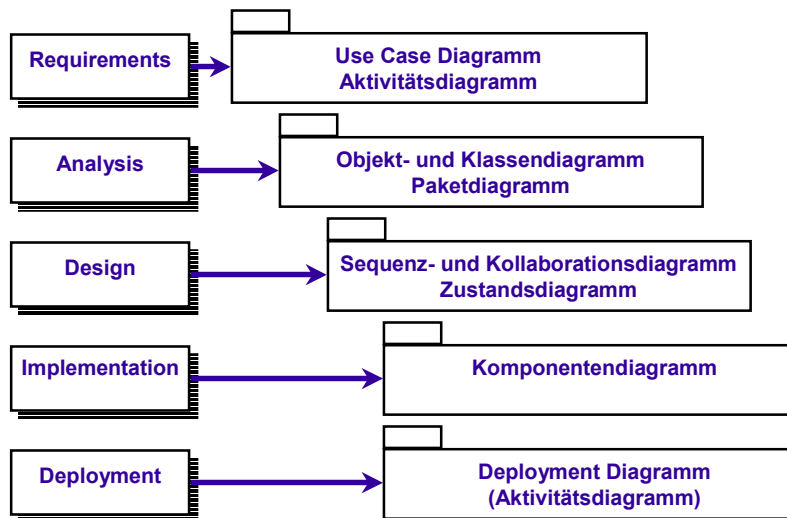
RUP Zusammenfassung

- **Best Practices bestehender Prozessmodelle**
 - Use Case basierend
 - Software-Architektur zentriert
 - Iteratives und inkrementelles Vorgehen
- **Gute Unterstützung von UML**
- **RUP Prozesseinführung als Pilot-Projekt organisieren**
- **Notwendige Anpassung des RUP**
 - Firmenspezifische Anpassungen (Reuse, Problem-Domain, Qualitätsmanagement, ...)
 - Projektspezifische Anpassungen (Projektgrösse, neues oder bestehendes Projekt, ...)

Skript:

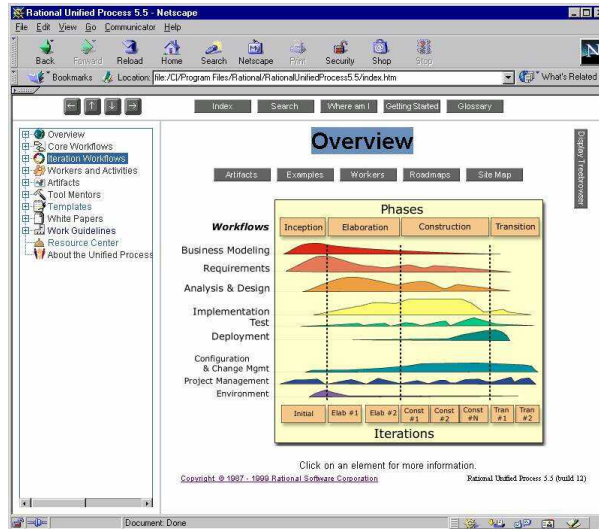
5-7

RUP Workflows und UML Diagramme



5-8

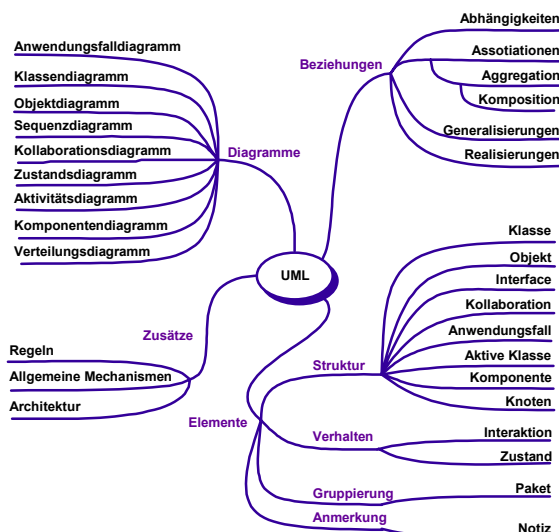
RUP Produkt



Skript:

5-9

UML Zusammenfassung



Skript: 3.1

5-10

UML Erweiterungsmechanismen

I

- **UML Erweiterungsmechanismen (Extensibility Mechanisms)**
ermöglichen es UML an projektspezifische Bedürfnisse anzupassen.
- **Eigenschaftswerte (Tagged Values)**
 - Beispiele {version = 1.0}, {location = 192.23.234.12}, {abstract}
- **Zusicherung, Bedingung (Constraints)**
 - Beispiele {complete}, {incomplete}, {readonly}
 - OCL (Object Constraint Language)

5-11

UML Erweiterungsmechanismen

II

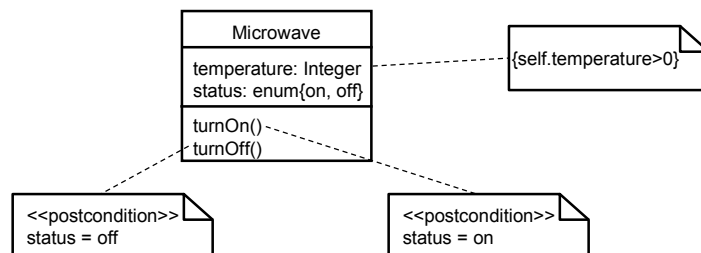
- **Stereotypen (Stereotypes)**
 - UML Stereotypen werden verwendet um vordefinierte UML Symbole um eine spezifische Semantik zu erweitern oder um neue Symbole oder Icons mit entsprechender Semantik in UML einzuführen.
- **Beispiele UML vordefinierter Stereotypen**

Stereotyp:	Verwendet für:	Bedeutung:
<<actor>>	Klasse	System externer Benutzer eines Use Case
<<document>>	Komponente	Allgemeines Dokument
<<exception>>	Klasse	Ausnahme-Event von einer Operation ausgelöst
<<file>>	Komponente	Datei eines Dateisystems
<<requirement>>	Kommentar	Anforderungsspezifikation
<<subsystem>>	Paket	Gruppierung von Elementen zu einem Teilsystem
<<thread>>	Klasse	Modellierung von Threads
<<utility>>	Klasse	Modellierung von Klassen mit (static) Hilfsfunktionen

5-12

UML Object Constraint Language (OCL)

- OCL ist eine formale Sprache zur exakten Spezifikation von Bedingungen in einem UML Diagramm bzw. Modell und Teil des UML Standards.
- Umfassende Sprache mit Kontrollstrukturen und Datentypen
- Beispiel: Spezifikation von Pre- oder Post-Conditions einer Operation



5-13

UML Ausblick

I

- Erweiterungen von UML Version 1.4 (September 2001):
- Spezifikation des UML Metamodells
 - Formale Definition von UML Modellen
 - Erlaubt eine bessere Verifikation von UML Modellen durch Software Tools
 - Erleichtert die automatische Programcode-Generierung
- Spezifikation von UML Profiles
 - Ein UML Profile ist eine Sammlung von Erweiterungen (Stereotypen) mit spezifischer Semantik für eine bestimmte Applikations- oder Problem-Domain.
 - Beispiel: UML Erweiterungen für die Modellierung von Real-Time Applikationen.

- **Erweiterungen von UML Version 2.0 (2003):**

- **Architektur:** Genauere UML Metamodel Spezifikation (UML Kernel / UML Profiles)
- **Beziehungen:** Definition einer genaueren Semantik für verschiedene Beziehungen
- **Erweiterbarkeit:** Detaillierung der Richtlinien für die UML Profile Spezifikation
- **Komponenten:** Bessere Unterstützung für Component-based Software Development
- **Model Management:** Bessere Unterstützung bei der Entwicklung grosser und komplexer Systeme
- **Allgemeine Mechanismen:** Spezifikation eines XML kompatiblen File-Formates für den Modellaustausch

Überblick UML Tools

- **Umfassende Listen von UML Tools:**

- B. Oestereich: <http://www.oose.de/uml> (Tool-Liste)
- Objects By Design: http://www.objectsbydesign.com/tools/umltools_byCompany.html

- **UML Tool Evaluationskriterien:**

- Plattform- und Programmiersprachen-Unterstützung?
- Konfigurations- und Versions-Management?
- Multi- oder Single-User?
- Roundtrip-Engineering und Reverse-Engineering Support / Code Generierung?
- Integration vorhandener Entwicklungs-Tools?

- **Beispiele:**

- RationalRose - Firma Rational Software (<http://www.rational.com>)
- Together - Firma Borland (<http://www.togethersoft.com>)
- Objecteering - Firma Softeam (<http://www.objecteering.com>)
- ArgoUML - Open Source Projekt (<http://argouml.tigris.org>)