Softwaretechnologie

Vorlesung von Dr. Kniesel
Konzept[*]

Version 2[†]

Matiouk Svetlana

WS 2008/09

Prüfungsrelevante Themen im WS 2008/09 sind ***Kapitel 0-10 und 14-15***.

1 Allgemeines zu SE

[1]

- Modellierung
  - der Anwendungsdomäne (Problemmodell[2])
  - des Systems (Lösungsmodell[3])

  ⇝ Vorteile der OO Modellierung

- Problemlösung
  Methoden zu Modellbewertung:
  - Participatory Design
  - Analysis Review
  - Design Review
  - Code Review
  - Project Management

- Wissensaquisition
  Prozessmodelle:
  - Risk-based Development[4] – Erkennung und frühzeitiges Einplanen der mit hohem Risiko behafteten Komponenten und Aktivitäten

---

[*] Mit diesem Dokument wurde ein Versuch unternommen, eine Sammlung der in der Vorlesung angesprochenen Begriffe sowie der während des Übungsbetriebes oft aufgetretenen Fragen/Verständnisschwierigkeiten zu erstellen. Und üblicherweise gehört an diese Stelle folgender **Hinweis!** Dieser Dokument kann nur als Vorbereitungshilfe angesehen werden und bietet somit keine Gewährung der vollständigen Abdeckung der in einer SWT-Klausur gestellten Fragen.

[†] Stand vom 24. März 2009, nicht vollendet.

[1] Software Engineering wird synonym zu Softwaretechnologie verwendet.

[2] Problem DOM: Objekte + Beziehungen

[3] Solution DOM: Objekte + Beziehungen + Interaktionen

[4] Risikoorientierte Softwareentwicklung

  - Issue-based Development[5] – parallele (statt linearer) Ausführung von Aktivitäten
    Grundlage: jede Aktivität beeinflusst jede andere
    → Reaktion auf Änderungen leichter, aber schwierigeres Koordinieren

- Rationale Management

  Motivation: Dynamische Problem- und Lösungs-Domänen

  Erfassung von

  - Problemen
  - Lösungsalternativen
  - Argumenten Für und Wider der Alternativen
  - getroffene Entscheidung
  - Begründung für die Entscheidung

Grundbegriffe

- Teilnehmer
- Rolle
- System
- Modelle
- Arbeitsprodukte (interne und auszuliefernde)
- Aktivität
- Aufgabe
- Ressourcen
- Projektplanung
- Ziel
- Anforderungen (funktionale und nicht-funktionale)
- Randbedingungen
- Notationen
- Methoden
- Methodologien

Software-Qualitäten Sichten, Interessenvertreter, Anwendungsbereiche

Engineering-Prinzipien ⇐ qualitätsorientiertes SE

- Rigor and Formality
- Trennung der Belange nach
  - Zeit
  - Qualitäten
  - Größe

  Gegenwart: OO Dekomposition
  Weiterentwicklung: Aspekt-Orientierte Softwareentwicklung

[5] Problemorientierte Softwareentwicklung

- Modularität
  - Dekomposition
    - Top-Down
  - Komposition
    - Bottom-Up
  - Kohäsion → *Zusammenhangskraft* von Elementen einer Gruppe
  - Kopplung → Maß für *Abhängigkeiten* zwischen Modulen

  ⇝ hohe Kohäsion + geringe Kopplung

- Abstraktion
  die wichtigen Aspekte herausfinden unter Vernachlässigung der Details
- Erwartung von Änderungen
- Allgemeingültigkeit
- Inkrementelle Vorgehensweise

2 Konfigurationsmanagement mit SVN

[6]

Sammelt und verwaltet Informationen zur Erstellung, Verwaltung und Erweiterung von Software

Begriffe

- Arbeitskopie
- Repository
- Konfigurationsmanagement-Systeme:
  - Concurrent Versions System (CVS)
  - Subversion (SVN)
    Buch-Homepage
    Paar Clients für SVN:
    - *Subversive* Plug-In für Eclipse
    - *TortoiseSVN*
  - ClearCase
- Project Sharing
- CheckIn
- CheckOut
- Commit
- Update Problem: Änderungen werden nicht verifiziert. Besser: „Synchronize“ benutzen.

[6] Software Configuration Management (SCM)

- Synchronize – das Projekt synchronisieren
  Konfliktauflösung benötigt den Detailvergleich auf Dateilevel.

  Möglichkeiten der Konfliktlösung:

  – Manuelles *Merging* der Änderungen
  – Overriding:
    „Overwrite and Update“
    „Overwrite and Commit“

3 OO Konzepte

Begriffe

- Objekte – Einheiten von Daten und Code
  – Variablen → Zustand
  – Operationen → Verhalten
  – Kapselung

  ⇝ Wartbarkeit, Zugriffs-Synchronisation

  – Objekt-Referenz (ObjectIDentifier)
  – Sharing / Aliasing ← Objekt-Identität
- Nachricht
  – Dynamisches Binden (Dynamic Binding) und sein Umsetzen (Nachrichtenbasiert ← Methodentabelle des Empfänger-Objektes)

  ⇝ Vermeidung von Typ-Fehlern, Wiederverwendbarkeit, Erweiterbarkeit
- Interface (Schnittstelle)
- Klasse
  – Dynamisches Binden und sein Umsetzen (Klassenbasiert ← Methodentabelle einer Klasse gilt für alle Instanzen)
- Vererbung
  – Einfachvererbung
  – Mehrfachvererbung
- Typen

  in Java:
  – Primitive Datentypen
  – Interfaces
  – Klassen

  ⇝ **Ersetzbarkeit**

  – Designprinzip:
    Typdeklaration ⇝ Interface
    Objekterzeugung ⇝ Klasse

- Typ-Polymorphismus
  - Ad hoc Polymorphismus
  - Subtyp-Polymorphismus
  - Parametrischer Polymorphismus

Dynamisches Binden und Vererbung:

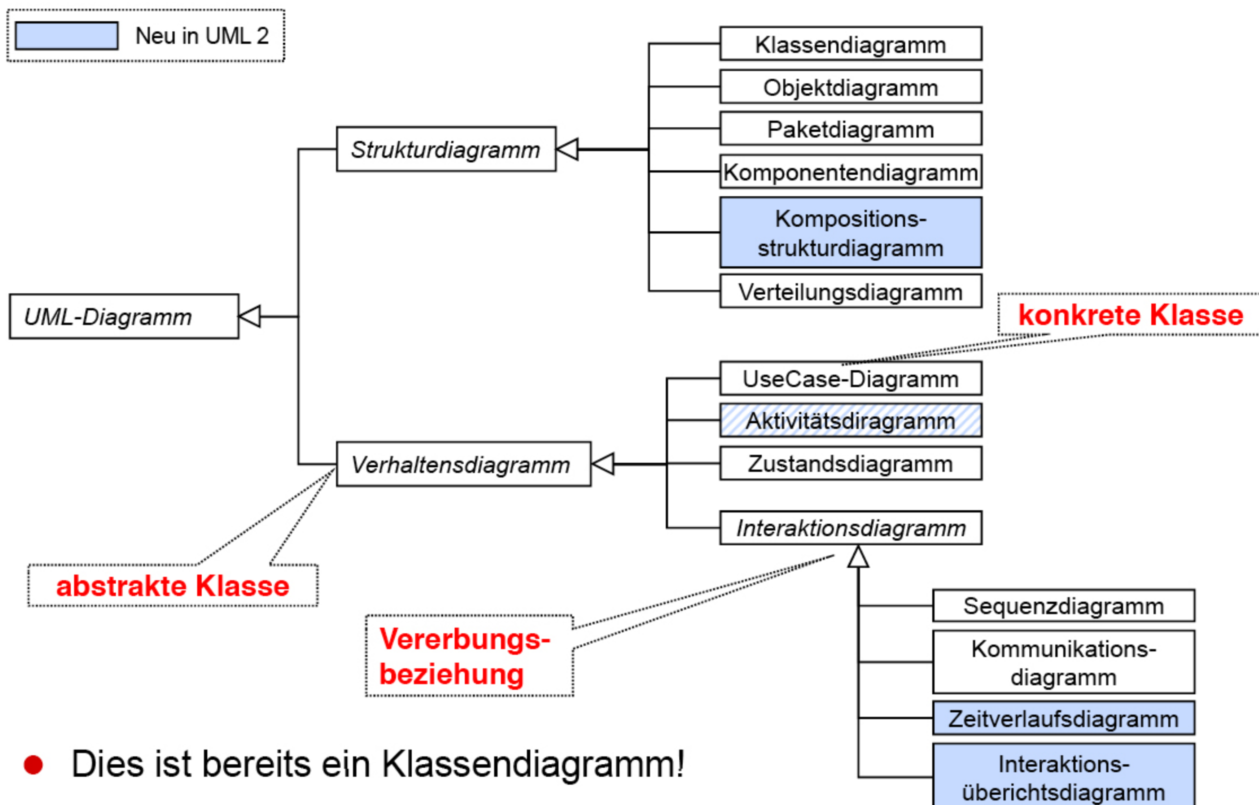
- Overloading
- Overriding

4 UML

- System
- Modell
- Sicht

Objekt-orientierte Modellierung – Entwicklung von Abstraktionen

- Ding
- Konzept (Name, Intention, Instanzen)
- Abstraktion



Strukturmodellierung Die Konzepte des *Klassendiagramms* und des *Objektdiagramms* entstammen der konzeptuellen Datenmodellierung und der objektorientierten Softwareentwicklung. Sie dienen zur Spezifikation der Daten- bzw. Objektstrukturen des Systems.

Klassendiagramm (*Class Diagram*)

basiert auf den Konzepten *Klasse*, *Generalisierung* und *Assoziation*.

- Klasse
- Attribut
- Operation
- Assoziation – beschreibt die gemeinsame Struktur einer Menge von i.d.R. langfristigen[7] Beziehungen zwischen Objekten.[HKKR05]
  - qualifizierte Assoziation
  - Assoziationsklasse
  - N-äre Assoziation

  zusätzliche Angaben:
  - Name (*benannte Assoziation*) und evtl. Leserichtung

  bei Assotiationsenden:
  - Rollenname
  - Multiplizität (fehlende Angabe bedeutet *1..1* bzw. *1*)
  - vordefinierte Eigenschaften, die jeweils an einem mehrwertigen Assotiationsende annotiert werden, z.B. {*ordered*} – um die "geordnete"[8] Beziehung zu Partnerobjekt zu spezifizieren, oder {*unique*} – um bei mehreren Partnerobjekten deren Eindeutigkeit anzugeben.
  - Navigationsrichtung (*navigierbare* Assotiationsenden)[9] – hat die gleiche Semantik wie ein Attribut der Klasse am gegenüberliegenden Assoziationsende ⇒ alle Eigenschaften von Attributen spezifizierbar.
  - Nicht-Navigierbarkeit (*nicht-navigierbare* Assotiationsenden)
- ≫Teil-von≪-Beziehung (≫Teile-Ganzes≪-Beziehung, *component-of* oder *part-of relationship*)[10]
  - Aggregation – asymmetrische, transitive Assoziation zwischen nicht gleichwertigen Partnern (*master-slave*-Verhältnis).
    - Keine zusätzliche Semantik – die teilnehmenden Objekte sind unabhängig von einander.
    - Abhängigkeiten können über Multiplizitätangaben zum Ausdruck gebracht werden (z.B. ≫*1*≪).[HKKR05]
    - ≫*schwache Aggregetion*≪
  - Komposition – strenge Form der Aggregation (≫*strake Aggregetion*≪)
    - ein Teil darf "Kompositionsteil" höchstens *eines* Genzen sein ⇒
    - Einschränkung der Multiplizitäten auf Kompositionsrauten auf:

[7] Mit der Eigenschaft "langfristig" soll angedeutet werden, dass eine solche Beziehung typischerweise einzelne Operationen überdauert.

[8] Gemeint ist die Existenz einer bestimmten Ordnungsrelaton als wesentliches Merkmal der entsprechenden Assoziation. Hinweise über die gewünschte Ordnungsrelation können per Kommentar oder per Einschränkung angegeben werden.[HKKR05]

[9] Ungerichtete Kanten signalisieren "keine Angabe über Navigationsmöglichkeit" und zeigen somit *undefiniert navigierbare* Assoziationen an.[HKKR05]

[10] Mit Aggragation und Komposition wird in UML jedoch keine exakte Semantik und Pragmatik für Einordnung in die in der Literatur ausführlich diskutierte *part-of*-Beziehungsklasse festgelegt.[HKKR05] Gemeint sind vor allem die zwei charakteristischen Eigenschaften: die Abhängigkeit und die Exklusivität, wobei man zwischen folgenden Arten der Komponentenobjekte unterscheidet:

- *Abhängige*,
- *Unabhängige* und
- *Exklusive*,
- *Nicht exklusive*.

        - ≫0..1≪ –Teil kann auch unabhängig vom Ganzen existieren, d.h. kann auch in anderen Kontexten Verwendung finden.
        - ≫1≪ – Teil existiert höchstens so lange wie sein Ganzes, es sei denn, er wird vor Ende der Lebensdauer des Ganzen an ein anderes Ganzes weitergeleitet.
    - das Ganze ist für Verwaltung seiner Teile verantwortlich.
        - Propagierung von Operationen (typischerweise Kopier- und Löschoperationen)

- Einschränkungen
- Generalisierung
- Klassifikationstyp
- Abhängigkeit
- Rollen

Objektdiagramm (*Object Diagram*)[11]

zeigt einen konkreten, beliebig komplexen Ausschnitt des System-*Zustands* zu einem bestimmten Ausführungszeitpunkt des Systems auf Basis der im Klassendiagramm spezifizierten Strukturen. [HKKR05]
⇒ Für Objekte können den Attributen Werte zugeordnet werden.

- Link – Instanz einer Assoziation
    - keine Multiplizitätsangabe
    - keine Zyklen

Paketdiagramm (*Package Diagram*)

- Paket (*package*)
    - ausschließlich als Gruppierungsmechanismus angesehen
    - Referenz
    - Import
    - besitzt keinerlei Klasseneigenschaften[12] [HKKR05]

Kpmponentendiagramm (*Component Diagram*)

- Komponente (*component*) – ist ein modularer Teil eines Systems, der zur Abstraktion und Kapselung einer beliebig komplexen Struktur dient und nach außen wohldefinierte Schnittstellen zur Verfügung stellt. [HKKR05]
    - kann im gesamten Softwareelbenszyklus modelliert und zunehmend verfeinert werden
    - kann sowohl physische als auch logische Modellierungsaspekte abdecken
    - (darüber hinaus) ist Sonderform einer Klasse
        - ⇒ darf alle Klasseneigenschaften aufweisen
        - darf instanziiert werden
    - 2 Sichten bei Modellierung:
        - extern (*black-box view*) – Spezifikation
        - intern (*white-box view*) – Implementierung

[11] Die modellierten Instanzen sind genau genommen *Instanzspezifikationen*, da sie die Objekte, die zur Laufzeit existieren und verwendet werden, *spezifizieren*, diese aber nicht sind.[HKKR05]

[12] Die Kombination als Gruppierungsmechanismus und Klasseneigenschaften bleibt in UML2 den *Komponenten* vorbehalten. Damit einhergehend sind nun (ab UML2) auch *Subsysteme* nicht mehr spezielle Pakete, sondern spezielle Form von Komponenten.

Kompositionsstrukturdiagramm (*Composite Structure Diagram*)

- hierarchische Dekomposition der verschiedenen Systembestandteile
- *kontextabhängige* Modellierung ⇝ höherer Detaillierungsgrad

Verteilungsdiagramm (*Deployment Diagram*)

- zeigt die eingesetzte Hardwaretopologie sowie das zugeordnete Laufzeitsystem
- Hardware:
  - Knoten
  - Kommunikationsbeziehungen
- Laufzeitsystem:
  - Artefakte – logische Teile des Modells, wie z.B. Komponenten, die in Form einer Implementierung manifestieren und auf Knoten verteilt werden. [HKKR05]

Verhaltensmodellierung

Anwendungsfalldiagramm (*Use-Case Diagram*)[13]

beschreibt die Funktionalität des zu entwickelten Softwaresystems aus Benutzersicht;

kann Grenzen des Systems bzw. den Kontext der Anwendungsfälle aufzeigen.

- System
- Akteure
  - stehen klar außerhalb des Systems
    - initiieren die Ausführung von Anwendungsfällen
    - oder werden vom System benutzt, indem sie selbst Funktionalität zur Realisierung einzelner Anwendungsfälle zur Verfügung stellen. [HKKR05]
- Anwendungsfälle
- Beziehungen
  - *include*-Beziehung
    der benutzte Anwendungsfall ist ***unbedingt notwendig***, um die Funktionalität des benutzenden Anwendungsfalls sicherzustellen;
  - *extend*-Beziehung
    der zu erweiternde Anwendungsfall kann vom erweiterten Anwendungsfall übernommen werden, muss aber ***nicht notwendigerweise*** übernommen werden.
  - Generalisierungsbeziehung (*generalization*)
    sollte hauptsächlich für die Beziehung zwischen konkreten und abstrakten Anwendungsfällen reserviert bleiben, weil bei freizügigem Einsatz konzeptionelle Konflikte mit der *extend*-Beziehung zu erwarten sind. [HKKR05]
- Erweiterungsstelle (*extension point*)

[13] Dem UML2 Standard entsprechend gehört das Anwendungsfalldiagramm zu den Verhaltensdiagrammen, obwohl es auch statische Aspekte aufdeckt.

Aktivitätsdiagramm (*Activity Diagram*)

ist ein gerichteter Graph, bestehend aus Aktivitätsknoten und Aktivitätskanten.

- Knoten
  - Aktionsknoten
  - Kontrollknoten
    - Start und Ende von Abläufen
      - Initialknoten ●
      - Aktivitätsendknoten ⊙[14] erzwingt die Beendigung der Ausführung aller Abläufe einer Aktivität
      - Ablaufendknoten ⊗ beendet nur einen bestimmten Ablauf
    - Alternative Abläufe
      - Entscheidungsknoten
      - Vereinigungsknoten
    - Nebenläuige Abläufe
      - Parallelisierungsknoten
      - Synchronisierungsknoten
  - Datenknoten
    - Verwendung als Ein- und Ausgabeparameter
    - oder als Pufferfunktion für Aktionen
- Kanten

Zustandsdiagramm (*State Diagram*)

beschreibt das Verhalten (eines Teils) eines Systems, indem sie die möglichen Folgen von Zuständen definieren, die Elemente des Systems – i.A. Objekte einer bestimmten Klasse –

- während ihres "Lebenslaufs"
- während der Ausführung einer Operation oder Interaktion

der Reihe nach einnehmen können. [HKKR05]

Interaktionsdiagramme (*Interaction Diagrams*)

Modellierung von Interaktionen

- Interaktion[15] spezifiziert die Art und Weise wie, Nachrichten und Daten über die Zeit hinweg zwischen verschiedenen *Interaktionspartnern* in einem bestimmten *Kontext* ausgetauscht werden, um eine bestimmte Aufgabe zu erfüllen.
- Interaktionspartner
- Kontext ⟶

[14] der innere schwarze Kreis des Aktivitätsendknotens ist dabei vom selben Radius wie der Initialknoten.

[15] Begriff der **Interaktion** wird als eine begriffliche Abstraktion von der Komplexität des zugrunde liegenden Nachrichten- und Datenaustauschs verwendet. Also "eine Interaktion" liefert keine Aussage darüber, ob es sich um das Versenden einer einzelnen Nachricht oder um eine Menge von Nachrichten und Interaktionspartnern handelt.

Sequenzdiagramm (*Sequence Diagram*)

dient zu Modellierung von Interaktionen. Eine *Interaktion*[16] spezifiziert die Art und Weise, wie Nachrichten und Daten über die Zeit hinweg zwischen verschiedenen *Interaktionspartnern* in einem bestimmten *Kontext* ausgetauscht werden, um eine bestimmte Aufgabe zu erfüllen. [HKKR05]

- Interaktionspartner
- zwei Darstellungsdimensionen:
  - vertikale – Zeit
  - horizontale – Interaktionspartner
- Ereignisspezifikationen
  - Ausführungsspezifikation – ist definiert durch
    - Start-Ereignisspezifikation
    - End-Ereignisspezifikation
- Nachrichtenübermittlung
  - synchron
  - asynchron
  - Antwortnachricht
- Zeiteinschränkung
- Zustandsinvariante – stellt eine Zusicherung dar, dass eine bestimmte Bedingung zu einem bestimmten Zeitpunkt des Interaktionsablaufs erfüllt sein muss.
  In UML 2 existieren 3 Notationsalternativen:
  - in geschwungener Klammern, z.B. $\{0 < z\}$, direkt auf der Lebenslinie
  - in einem Notizsymbol, das direkt mit der Ereignisspezifikation auf der Lebenslinie verbunden wird.
  - in einem Zustandssymbol
- Kombinierte Interaktionsfragmente
  - zur Kontrollflusssteuerung

Kommunikationsdiagramm (*Communication Diagram*)

(in UML1 als *Kollaborationsdiagramm* bezeichnet)

Im Gegensatz zu Sequenzdiagrammen kann auch der struktureller Zusammenhang der Interaktionspartner dargestellt werden. Dafür aber entfällt die Darstellung der Zeitachse und wird die weniger übersichtliche Sequenznummerierung verwendet.

5 Anforderungserhebung

Funktionale Anforderungen beschreiben die Interaktion des Systems mit seiner Umgebung unabhängig von seiner Implementierung.

[16] hier wird der Begriff der *Interaktion* als eine Art begriffliche Abstraktion von der Komplexität des zugrunde liegenden Nachrichten- und Datenaustausches verwendet.

Nichtfunktionale Anforderungen beschreiben Aspekte des Systems, die nicht unmittelbar in Bezug zu seiner Umgebung stehen.

Kritärien im FURPS+-Modell (*Functionality, Usability, Reliability, Performance, Supportability*) [BD04]

Qualitätsanforderungen:

- Bedienbarkeit
- Zuverlässigkeit
- Leistungsanforderungen
- Unterstützbarkeit
  - Änderbarkeit
  - Wartbarkeit
  - Portabilität

+ **Pseudoanforderungen** / Beschränkungen :

- Implementierungsanforderungen
- Schnittstellenanforderungen
- Betriebliche Anforderungen
- Verpackungsanforderungen
- Rechtliche Anforderungen

6 Anforderungsanalyse

7 Design Patterns

[17] [18]

(**alle** aus der VL ! und nicht nur die aus den Übungen)

Schemata für die Realisation von Subsystemen oder Komponenten eines Softwaresystems.
Ziele:

- Dokumentation von Lösungen Probleme wiederkehrender Probleme.
- Schaffung einer gemeinsamen Sprache, um über Probleme und ihre Lösungen zu sprechen.
- Bereitstellung eines standardisierten Katalogisierungsschemas um erfolgreiche Lösungen aufzuzeichnen.

Bestandteile:

- Name
- Kontext – die Vorbedingungen unter denen das Pattern benötigt wird.
- Problem bzw. Absicht – Ziel und gewünschte Eigenschaften die in einem bestimmten Kontext mit bestimmten Randbedingungen/Kräften erreicht werden sollen.
- Randbedingung / Kräfte (Forces)
  - die relevanten Randbedingungen und Einschränkungen
  - wie diese miteinander interagieren und im Konflikt stehen

[17] Entwurfsmuster

[18] Dieses und nachfolgende Abschnitte wurden noch nicht an die Inhalte des aktuellen Semesters angepasst! Sie stammen aus WS 2007/08.

  - die daraus entstehenden Kosten
  - typischerweise durch ein motivierendes Szenario illustriert

- Die Lösung wird beschrieben durch die Teilnehmer, ihre Rollen, ihre statischen Beziehungen und dynamische Zusammenarbeit.
  - Teilnehmer – die Typen (Interfaces und Klassen)
  - Rollen
  - Statische Beziehungen und dynamische Zusammenarbeit
- Beispiele
- Erläuterung (rationale), warum das Pattern funktioniert
- Bezug anderen zu Patterns
- Bekannte Verwendungen
- optional: Zusammenfassung (auch: „pattern thumbnail”)

Liste

- Observer ← Stellt eine 1-zu-n Beziehung zwischen Objekten her, Propagierung von Änderungen
  muss erweiterbar sein; muss skalierbar sein
- Composite ← rekursive Aggregations-Strukturen darstellen („ist Teil von”), Aggregat und Teile einheitlich behandeln
  komplexe Struktur; muss variable Tiefe und Breite haben
- Strategy ← Kapselung einer Familie von Algorithmen mit der Möglichkeit, sie beliebig auszutauschen.
  muss Funktionalität X in unterschiedlichen Ausprägungen bereitstellen können
- Facade ← Menge von Interfaces eines Subsystems zusammenfassen
  muss mit einer Menge existierender Objekte zusammenarbeiten; stellt ...-Dienst bereit
- Singleton ← Beschränkung der Anzahl von Exemplaren zu einer Klasse
- Adapter ← Schnittstelle existierender Klasse an Bedarf existierender Clients anpassen
  muss mit einem existierenden Objekt zusammenarbeiten
  Pluggable Adapters
  - mit Vererbung und Abstrakten Operationen
  - mit „Delegate Objects“
- Proxy ← Stellvertreter für ein anderes Objekt, bietet Kontrolle über Objekt-Erzeugung und -Zugriff
  muss ortstransparent sein
- Bridge ← Schnittstelle und Implementierung trennen, ... unabhängig variieren
  muss sich um die Schnittstelle zu unterschiedlichen Systemen kümmern, von denen einige erst noch entwickelt werden; ein erster Prototyp muss vorgeführt werden
- Command ← Objekte mit auszuführenden Aktion parametrisieren
  Anfragen an unbekannte Objekte richten, d.h. das Objekt, das eine Anfrage auslöst, muss nur wissen, wie er diese anstößt; es muss nicht wissen, wie die Anfrage ausgeführt wird.
  ⇒ Flexibilität beim Entwurf von Benutzungsschnittstellen
  Kontextsensitive Menüeinträge
  Zusammengesetzte Befehle (Ausführung einer Abfolge von Befehlen)

Operationen können rückgängig gemacht werden (Befehle müssen umkehrbar sein)
Lebensdauer des Befehlsobjekts unabhängig von der Anfrage
Aufreihung und Ausführung der Anfragen zu unterschiedlichen Zeitpunkten
Gemeinsame Schnittstelle für alle Transaktionen sowie Erweiterung um neue Transaktionen

- Factory Method ← Entscheidung über konkreter Klasse neuer Objekte verzögern
- Abstract Factory ← zusammengehörige Objekte verwandter Typen erzeugen ohne deren Klassenzugehörigkeit fest zu codieren
  Herstellerunabhängig; Geräteunabhängig; muss eine Produktfamilie unterstützen
- Visitor ← Repräsentation von Operationen auf Elementen einer Objektstruktur, Neue Operationen definieren, ohne dass die Klassen dieser Objekte zu ändern
  Element-Klassen müssen unabhängig von den auf sie anzuwendenden Operationen sein und neue Operationen müssen unabhängig von den Elementen hinzugefügt werden.

8 Systementwurf

9 Objektentwurf

10 OO Modellierung

11 Testen

12 Automatische Modultests (JUnit)

13 Refactoring

- Grundlage: Verletzung von Designprinzipien erkennen („*Bad smells*“)
- Entsprechende *verhaltenserhaltende Restrukturierung* des Codes
  - Manuell
  - Automatisiert

Liste der „*Bad smells*“:

- Lange Parameterliste
- Änderungsanfälligkeit („Divergente Änderungen“)
- Verteilte Änderungen
- Neid: „Begehre nicht deines Nächsten Hab und Gut!“
- Daten-Klumpen
- Fixierung auf primitive Datentypen
- Fallunterscheidungen (Switch-Statements)
- Faule Klasse
- Spekulative Allgemeinheit
- Temporäre Felder
- Daten Klassen

- Verweigertes Vermächtnis
- Kommentare im Methoden-Rumpf

Die folgenden Refactoring-Patterns wurden in der VL genannt bzw. behandelt:

- Move Method
- Pull Up Method
- Replace Conditional with Polymorphism (Fallunterscheidung durch Polymorphimus ersetzen)
- Replace Data Value with Object
- Replace Type Code with State / Strategy
- Replace Type Code with Class
- Replace Type Code with Subclasses
- Pull Up Constructor Body
- Extract Method
- Inline Method
- Replace Temp with Query
- Inline Temp
- Split Temporary Variable
- Remove Assignments to Parameters
- Replace Method with Method Object
- Encapsulate Field
- Substitute Algorithm
- Collapse Hierarchy
- Extract Class
- Inline Class
- Introduce Parameter Object
- Preserve Whole Object
- Remove Parameter
- Introduce Null Object
- Replace Parameter with explicit Methods
- Replace Parameter with Method

Martin Fowlers Refactoring-Homepage bietet viel Information zum Thema, unter anderem einen Katalog.

14 Softwareentwicklungsprozessmodelle

15 Agile Softwareentwicklung

16 Glossar zu Eclipse

- *Views*
- *Perspectives*

- *Editors*

- `workspace` – ist der default-Verzeichnis, wo standardmäßig alle Resourcen (Projekte, Verzeichnisse, Dateien) gehalten werden. Letztendlich kann man den Namen sowie den default-Ort umändern, oder auch mehrere Arbeitsverzeichnisse anlegen (z.B. für unterschiedliche Benutzer), es ist aber üblich umgangssprachlich den konkreten Arbeitsverzeichnis als *Workspace* zu bezeichnen.

- `.project`-Files – sind XML-Dateien, die auf der obersten File-Hierarchieebene jedes Projektes sich befinden und Projektspezifische Informationen enthalten, z.B. Verweise auf andere Projekte.

- `.classpath`-Files – sind XML-Dateien, die Java Build Path des aktuellen Projekts angeben, sowie anderer Projekte, soweit solche hinzugefügt wurden, oder Bibliotheken.

Literatur

[BD04] Bernd Brügge and Allen H. Dutoit. *Objektorientierte Softwaretechnik mit UML, Entwurfsmustern und Java.* Pearson Studium, 2004.

[HKKR05] Martin Hitz, Gerti Kappel, Elisabeth Kapsammer, and Werner Retschitzegger. *UML@Work: Objektorientierte Modellierung mit UML 2.* dpunkt-Verlag, Heidelberg, 3 edition, 2005.