www.uni-frankfurt.de

Modul: Programmierung B-PRG
Grundlagen der Programmierung 1

V16 Software Engineering – OO-Analyse und Design

Prof. Dr. Detlef Krömker
Professur für Graphische Datenverarbeitung
Institut für Informatik
Fachbereich Informatik und Mathematik (12)

Unsere heutigen Lernziele

Einordnung der Entwurfsmethoden

Kennenlernen: Objektorientierte Analyse und Design -- Was ist das?

Kennenlernen: Der Begriff "Software-Architektur"

Grafische Notationen - Elemente von UML kennenlernen

Vorgehensmodelle im Software Engineering (kommt kommenden Freitag)

Übersicht

- **Rückblick:** SWE: Was ist das?
- **Rückblick:** Objektorientierung und deren Realisierung in Python
  Strukturierungsmittel in Python: Klassen und Module

- **Einordnung der Entwurfsmethoden**
  - Klassische Entwurfsmethoden
  - UML: Was ist das? – Eine Übersicht
  - Software-Architekturen

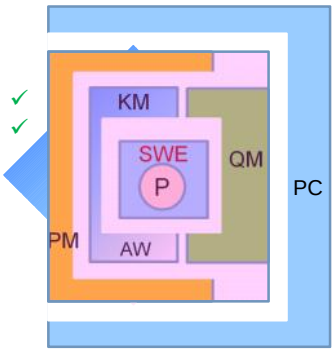
- **Objektorientierung (Übersicht)**
  - Analyse
  - Design
  - Wartung

UML (= Hilfsmittel)
Use Cases
Klassendiagramme
Sequenzdiagramme

Rückblick: SWE - Was ist das?

- V00: Die Rolle der Programmierung

- V08: Module, Kommentare, Docstring ✓
- V09/10 SW Tests ✓



Rückblick: SWE - Software Engineering

in Deutsch meist **Softwaretechnik** genannt:

Nach Balzert: „Zielorientierte Bereitstellung und systematische Verwendung von Prinzipien, Methoden und Werkzeugen für die arbeitsteilige, ingenieurmäßige Entwicklung und Anwendung von umfangreichen Softwaresystemen."

Balzert, Lehrbuch der Software-Technik. Bd.1. (2001),

War eine Reaktion auf die 1. Softwarekrise (1965)

Rückblick: SWE - SWEBOK

Guide to the **Software Engineering Body of Knowledge** (**SWEBOK**) ist ein Dokument der IEEE Computer Society.

Umfasst 10 Wissensgebiete (engl. *Knowledge Areas*, *KA*).

1. *Software requirements*: **Anforderungsanalyse**
2. *Software design*: **Softwareentwurf**
3. *Software construction*: Programmierung
4. *Software testing*: **Softwaretest**
5. *Software maintenance*: Softwarewartung
6. *Software configuration management*: Konfigurationsmanagement
7. *Software engineering management*: Projektmanagement
8. *Software engineering process*: **Vorgehensmodell**
9. *Software engineering tools and methods*: Entwicklungswerkzeuge und -methoden
10. *Software quality*: Softwarequalität

Als 11. Wissensgebiet werden verwandte Wissenschaften genannt

Rückblick: SWE - Vorgehensmodell

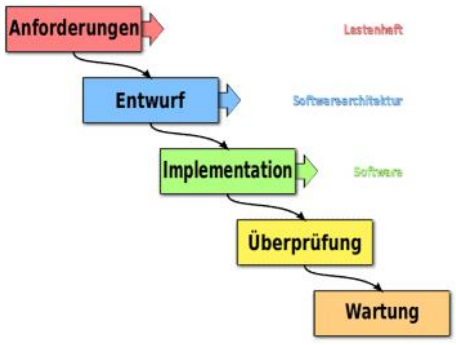
- *allgemein organisiert einen Prozess in verschiedene, strukturierte Abschnitte, denen wiederum entsprechende Methoden und Techniken der Organisation zugeordnet sind.*
- *Aufgabe eines Vorgehensmodells ist es, die allgemein in einem Gestaltungsprozess auftretenden Aufgabenstellungen und Aktivitäten in einer sinnfälligen logischen Ordnung darzustellen.*
- *Mit ihren Festlegungen sind Vorgehensmodelle organisatorische Hilfsmittel, die für konkrete Aufgabenstellungen (Projekte) individuell angepasst werden können und sollen … .*

aus Wikipedia: https://de.wikipedia.org/wiki/Vorgehensmodell

- Teil des Projektmanagements

Rückblick: SWE - Vorgehensmodelle im SWE

- (ganz klassisch) **Wasserfallmodell** (Winston W. Royce 1970)



Von Paul Hoadley, Paul Smith and Shmuel Csaba Otto Traian, CC BY-SA 3.0. nach : https://de.wikipedia.org/wiki/Wasserfallmodell

Rückblick: SWE – Weitere Vorgehensmodelle im SWE

- **V-Modell** (Wasserfall +Testen) nach Boehm 1979: 1:1-Gegenüberstellung von Entwurfs- und Teststufen)
- **Spiralmodell** nach Boehm 1986
- **Extreme Programming** (Beck 2000) spezielle Form des Spiral-Modells … Techniken wie PairProgramming
- **Agile Softwareentwicklung (ab 2002)**
  Oberbegriff für den Einsatz von Agilität (*agilis*: *flink; beweglich*) im Gegensatz zu schwergewichtig, z.B.
  Extreme Programming
  Feature Driven Development
  Kanban
  Scrum u.v.a

Rückblick: Die zweite Wurzel: OO-Programmierung

All die Vorkehrungen in Programmiersprachen, speziell

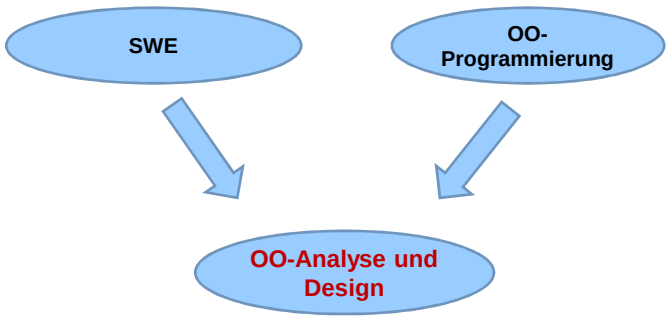
Klassen in Python

- Die class - Anweisung
- Klasseninstanzen (Zugriff auf Attribute und Methoden - Punktnotation)
- Referenzzählung und Zerstörung von Instanzen
- Vererbung
- Datenkapselung
- Überladen von Operatoren
- Introspektion
- Namenskonventionen

Rückblick:
Grundzüge der OO –Programmierung, siehe V12

- Objekt-orientierte Techniken sind im Rahmen von Programmier-sprachen eingeführt worden, um „Beiträge“ zur Lösung der Softwarekrisen zu leisten (vor allem der 2. Krise).
- **Simula** (1973) von Nygaard und Dah war die erste objekt-orientierte Programmiersprache ➔ Modellierung für **Simulationen**
- **Smalltalk** wurde Ende der 70er Jahre bei Xerox PARC entwickelt und 1983 von Goldberg publiziert: Erste **radikale** objekt-orientierte Programmiersprache. Alles sind Objekte einschließlich der Klassen ➔ graphische Benutzeroberfläche der neuen Arbeitsplatzsysteme
- C++ (erste Anfänge in 1979, nannte sich zunächst ‘‘C with Classes”) von Stroustrup

Unser Vorgehen:



Möglichkeiten zur Strukturierung von Programmen Klassen, Module (und Pakete)

- Der größte(stärkste) Feind einer/s jeden Softwareentwicklerin/s ist die **Fehlerausbreitung, insbesondere bei größeren Programmen.**
- Durch geeignete Strukturierung des Quellcodes kann man dies minimieren oder Fehler zumindest offensichtlicher machen.
- **Nur durch Strukturierung** kann die inhärente Komplexität von Modellen realer Systeme vermindert werden.
- Einfache (und schon wirkungsvolle) Hilfsmittel sind: **Blöcke und Prozeduren.**
- **Hinzu kommen Klassen und Module.**

Strukturierung in Python durch Blöcke und Funktionen / Methoden

Blöcke	Funktionen / Methoden
Keine Kapselung – keine Beschränkung der Gültigkeit	Haben Ihren eigenen Gültigkeitsbereich (*scope*) ➔ Kapseln die Namen
	aber: nicht absolut gekapselt: siehe Name-Mangling für Namen beginnend mit __ (zwei _)
	Kapselung kann durch das Schlüsselwort `global` aufgehoben werden
Blöcke werden **nur** zusammen mit Verzweigungs- oder Schleifen-konstrukten genutzt. Einrückung.	Konstrukte (`def()`)können immer genutzt werden.

Strukturierung in Python durch Klassen und Module

Klassen	Module
sind beides **Konstrukte, die definierte Elemente kapseln.**	
aber: nicht absolut gekapselt: siehe Name-Mangling für Namen beginnend mit __ (zwei _)	**aber**: nicht absolut gekapselt. siehe Name-Mangling
Klassen könnnen instanziert werden ➔ Objekte (Instanzen) dieser Klasse und Objektvariablen	Instanzierung gibt es bei Modulen nicht!
Vererbung	Vererbung gibt es nicht!
Klassen werden in einem Modul (= einer Datei)	Module können in einem Paket zusammengefasst werden

Hierarchie der Strukturierungsmittel in Python

Unqualifizierte Namen unterliegen **lexikalischen Gültigkeitsregeln**.
Zuweisungen binden solche Namen an den lokalen Gültigkeitsbereich (*scope*), es sei denn, sie sind als `global` deklariert. Details siehe:
http://sebastianraschka.com/Articles/2014_python_scope_and_namespaces.html#1-lg---local-and-global-scopes

Kontext	Lokaler Bereich	Auflösung
Modul = File des Source Codes	das Modul selbst	**wie lokal**, das Modul selbst
Funktion, Methode	Funktions**aufruf** Methoden**aufruf**	**LEGB -** Local, Enclosed, Global, Built-in
Klasse	class-Anweisung **Achtung** Klassenattribute	**LEGB -** Local, Enclosed, Global, Built-in
Skript, interaktiver Modus	modul `__main__`	wie lokal

Wie haben wir bisher strukturiert?

- … nach Gefühl oder Zwang (Aufgbenstellung!).
- Meist nicht … ein Programm = ein Modul
  bei den Klassen haben wir schon Schlüsselfragen kennengelernt
  Ist ein? ➔ Unterklasse
- Geht das auch systematischer?

Was ist das UML? (1)

UML = Unified Modeling Language

Aktuell: Version 2.5 (2015) viele Werkzeuge noch für 2.3 (oder gar 2.0)

Siehe http://www.omg.org/spec/UML/2.5/PDF/
Aktuell ist die Spezifikation 752 Seiten lang

Gepflegt von der OMG ® (Object Management Group ®):
Ein internationales **Technologie Standards** Consortium (1989)

Was ist das UML? (2)

*"The objective of UML is to provide system architects, software engineers, and software developers with tools for **analysis, design, and implementation** of software-based systems as well as for **modeling business and similar processes**." …*

UML meets the following requirements: ….

- *A detailed explanation of the semantics of each **UML modeling concept**. The semantics define, in a **technology-independent manner**, how the UML concepts are to be realized by computers.*
- *A specification of the **human-readable notation elements** for representing the individual UML modeling concepts as well as rules for combining them into **a variety of different diagram types** corresponding to different aspects of modeled systems."*

Mikro-Einführung in UML: Unified Modeling Language

- Mit der Einführung verschiedener objekt-orientierter Programmiersprachen in den achtziger Jahren entstanden auch mehr oder weniger formale (graphische) Sprachen für OO-Designs.
- Popularität genossen unter anderem die graphische Notation von Grady Booch aus dem Buch ‘‘Object-Oriented Analysis and Design”, OMT von James Rumbaugh (Object Modeling Technique), die Diagramme von Bertrand Meyer in seinen Büchern und die Notation von Wirfs-Brock et al in ‘‘Designing Object-Oriented Software”.
- Später vereinigten sich Grady **Booch**, James **Rumbaugh** und Ivar **Jacobson** in Ihren Bemühungen, eine einheitliche Notation zu entwerfen. Damit begann die Entwicklung von UML Mitte der 90er Jahre.

Mikro-Einführung in UML: Unified Modeling Language

- Anders als die einfacheren Vorgänger vereinigt UML **eine Vielzahl einzelner Notationen** für verschiedene Aspekte aus dem Bereich des OO-Designs und es können deutlich mehr Details zum Ausdruck gebracht werden.
- Somit ist es üblich, sich **auf eine Teilmenge von UML zu beschränken**, die für das aktuelle Projekt ausreichend ist.
- Wir können nur das Wichtigste betrachten!
- Es gibt diverse unterstützende Werkzeuge.

UML Diagramme in der Übersicht (1)

- Strukturdiagramme:
  - **Klassendiagramm**
  - Objektdiagramm
  - Anwendungsfalldiagramm
  - Paketdiagramm
  - Profildiagramm
  - Architekturdiagramme:
    - Kompositionsdiagramm
    - Komponentendiagramm
    - Subsystemdiagramm
    - Einsatz- und Verteilungsdiagramm

UML Diagramme in der Übersicht (2)

- Verhaltensdiagramme
  - Aktivitätsdiagramm
  - Zustandsdiagramme
    - Prozessautomat
  - Interaktionsdiagramme
    - **Sequenzdiagramm**
    - Kommunikationsdiagramm
    - Zeitdiagramm
    - Interaktionsübersicht

Übersicht

- **Einordnung der Entwurfsmethoden**
  - Rückblick Objektorientierung und deren Realisierung in Python
  - SWE: Was ist das?
  - Strukturierungsmittel in Python: Klassen und Module
  - UML: Was ist das? – Eine Übersicht
- **Objektorientierung**
  - Analyse
  - Design
  - Implementierung
  - Tests (und Werkzeuge: Debugger)
  - Wartung

Heute nur oberflächlich betrachtet!

UML (= Hilfsmittel)
Use Cases
Klassendiagramme
Sequenzdiagramme

Objektorientierung - Problem-Analyse

- Die **Analyse ist ein fortdauernder Prozess**, der auf den weiteren Verlauf eines Software-Projekts andauernd Einfluss ausübt.
- Die Analyse eines Problems erfolgt unabhängig von den später verwendeten objekt-orientierten Techniken und auch den OO-Modellierungsansätzen. (Entsprechend ist der häufig verwendete Begriff ”OO Analyse” nicht wirklich präzis.)
- Die Ergebnisse der Problem-Analyse sollten in einer schriftlichen und **allseits verständlichen** Form vorliegen.

1. Problem-Analyse

- Im Rahmen einer Analyse entstehen **"Use Case"** Szenarien, die die künftigen **Abläufe aus Benutzersicht** beschreiben, und Modelle, die eine Gesamtsicht geben.
- Die Modelle sind anschließend sehr nützlich, um zu einem OO-Design zu gelangen und die **"Use Case"** helfen, die richtigen Schnittstellen zu finden.

1. Problem-Analyse -- 2. OO-Design

Im OO-Design sind **als erstes** die **„Hauptsorten“** an Objekten zu identifizieren. Sie stehen typischerweise in Verbindung mit den Daten, die zu verwalten sind.

Beispiel:

Im Falle eines Herstellers von Gütern könnten dies (nebst vielen weiteren Dingen) Geschäftskunden, private Kunden, Zulieferer, Teile und hergestellte Güter sein.

Hinzu kommen Prozesse mit der Buchhaltung, die sich z.B. um einen Fall kümmern, bei dem die Ware bereits geliefert, jedoch noch nicht gezahlt worden ist.

2. OO-Design

Die Objekte sind hierarchisch zu klassifizieren anhand ihrer Gemeinsamkeiten.

Beispiel:

So haben beispielsweise Geschäfts- und Privatkunden viele Gemeinsamkeiten und es kann davon ausgegangen werden, dass der Unterschied für viele Vorgänge keine Rolle spielt.

2. OO-Design

Drittens: Die Beziehungen zwischen den Klassen sind zu modellieren.

Beispiel:

So sollte ein Kunde solange nicht gelöscht werden, solange es noch einen offenen Prozess gibt, der sich auf den Kunden bezieht.

2. OO-Design

- Problem: 1/3 aller Softwareprojekte werden **nicht abgeschlossen!**
- Was tun?

- Um das Rad nicht immer neu zu erfinden, ist es beim OO-Design wichtig, nach bereits vorhandenen **Komponenten, Frameworks und Bibliotheken** zu suchen, die dazu dienen könnten den Entwicklungsaufwand zu reduzieren.

- Jenseits des aktuellen Stands der Analyse und der Anforderungen könnte es sich lohnen, die Klassen „allgemein genug“ zu entwerfen, so dass spätere Änderungswünsche leichter berücksichtigt werden können und auch möglicherweise künftige Projekte davon profitieren. (Achtung: Aber nicht zu weit gehen! – Angemessen: Erfahrung!)

2. OO-Design

- Die Kunst liegt darin, die richtige Balance zu finden zwischen Entwürfen, die kaum geeignet sind, spätere Änderungswünschen zu überleben, und Entwürfen, die so allgemein und abstrakt sind, dass sie weder im Rahmen der zur Verfügung stehenden Ressourcen zu realisieren sind noch überschaubar bleiben.
- Die Suche nach Einfachheit gewinnt fast immer (**KISS-Prinzip** (**keep it simple and smart**)).
- Viele ehrgeizige Projekte endeten als unglaublich komplizierte Ungetüme, bei denen die meisten Teile selten oder nie Verwendung fanden.
- Extrem komplexe Entwürfe können auch erhebliche Performance-Probleme in der Implementierung mit sich bringen.

OO-Implementierung

- Die Struktur der Implementierung sollte sich direkt ableiten lassen von dem OO-Design. Dazu ist es notwendig, dass die ausgewählte Programmiersprache das verwendete OO-Modell unterstützt.
- Es ist dabei nicht ungewöhnlich, sich bei der Implementierung auf eine Teilmenge der zur Verfügung stehenden Sprachmittel zu beschränken. Dies kann sinnvoll sein, um die Portabilität und die Wartbarkeit zu erhöhen.
- Viele Programmiersprachen laden jedoch zu einem bestimmten Programmierstil ein und sind mit etablierten Konventionen verbunden. (➔ Programmierrichtlinien). Es ist nicht klug, davon in extremer Weise abzuweichen, da sonst das Resultat schwer lesbar und wartbar sein könnte.

1. Problem-Analyse -- 2. OO-Design -- 3. Implementierung

- **Programmtext** wird typischerweise nur einmal geschrieben (modulo späterer Korrekturen und Erweiterungen), **jedoch vielfach gelesen**. Entsprechend sollte der **Lesbarkeit** hohes Gewicht beigemessen werden.
- Wenn immer möglich, sollte der Programmtext sich selbst dokumentieren, indem genügend **Kommentare** und auch formelle Dokumentationstexte direkt in den Text eingebettet werden (Beispiele: Docstring, Help, Javadoc).

  Sonst besteht die Gefahr, dass die separat erstellte Dokumentation nicht mehr mit dem tatsächlichen Stand des Programmtexts übereinstimmt.

1. Problem-Analyse -- 2. OO-Design – 3. OO-Implementierung

- In diesem Zusammenhang gibt es **Werkzeuge**, die diese Dokumentation zusammen mit den formalen Klassenbeziehungen aus den Programmtexten extrahieren können.
- Ansonsten haben wir uns mit der Implementierung (= Programmieren) ja schon beschäftigt.

Was immer dazugehört: Tests (1)

- Für jeden Testfall sollten die zu erwartenden Resultate im Voraus ermittelt werden, typischerweise aus der Spezifikation.
- Das Ziel besteht darin, soviel wie möglich Fehler zu finden, bevor das Auftreten von Fehlern zu teuer wird. **Dies ist eine destruktive Tätigkeit**.
- Tests sollten reproduzierbar sein.

Tests (2)

- Es ist sowohl sinnvoll, Tests für einzelne Module bei der Entwicklung mit zu erstellen (am besten zuerst den Test!) und zusammen mit dem Programmtext für ein Modul zu verwalten
- oder auch getrennte Teams mit der Entwicklung von Testfällen zu beauftragen. Letzteres stellt auch sicher, dass das korrekte Verständnis der Spezifikation mit getestet wird.
- Testsuiten sind sehr hilfreich, um sich gegen neu eingeführte Fehler zu schützen und um bei Portierungen frühzeitig Fehler zu erkennen.

Was immer dazugehört: Auch Wartung

Alle Entwicklungsprozesse und Kosten nach der Auslieferung eines Software-Produkts gehören zur Wartungsphase.
Es wird häufig übersehen, dass in vielen Fällen **die Wartungsphase kostspieliger** als alle vorangegangen Phasen sein kann:

- **Änderungswünsche des Kunden** (41.8%).
- Änderungen bei der Repräsentierung von Daten (17.6%).
- Notfall-Reparaturen (12.4%).
- Gewöhnliche Korrekturen (9%).
- Hardware-Änderungen (6.2%).
- Dokumentation (5.5%).
- Performance-Verbesserungen (4%).
- Andere Ursachen (3.4%).

Wartung

- Diese Prozentzahlen geben die relativen Wartungskosten an, die bei einer Studie von Lientz und Swanson 1980 aus 487 Software-Projekten ermittelt wurden.
- Ungeachtet des Alters der Studie, gibt es keine signifikanten Änderungen zu heutigen Software-Projekten.

  Quelle: Bertrand Meyer, ‘‘Object-Oriented Software Construction”, ‘‘About Software Maintenance”

Zwischen-Zusammenfassung

- Die Objektorientierung bedeutet eine Modularisierung der Programme und eine klare Beschreibung ihrer Zusammenarbeit.
- Bei der Planung einer Software helfen Benutzungsszenarien (usecases) weiter. Sie beschreiben Akteure und ihre Interaktionen.
- Die Beziehungen zwischen Objekten können mit UML-Diagrammen verdeutlicht werden
- Wichtige Beispiele sind Klassendiagramme und Sequenzdiagramme … aber die besprechen wir erst Freitag.

Ausblick ... nächsten Freitag

UML detaillierter

... und, danke für Ihre Aufmerksamkeit!