Prof. Dr. Jörg Desel, Maren Stephan

Software Engineering

LESEPROBE

Fakultät für
Mathematik und Informatik

FernUniversität in Hagen

Kapitel 1

Softwareengineering

[...]

1.1 Entwicklung des Softwareengineering

[...]

1.2 Gremien und Standards im Bereich Softwareengineering

[...]

1.3 Softwareengineering heute

1.3.1 Was ist Softwareengineering?

Softwareengineering wurde erstmalig bedeutsam, als Software breitentauglicher (Einsatz auch außerhalb von Forschungseinrichtungen) werden sollte und gleichzeitig die Einsatzfelder von Software zu kritisch waren (z.B. Verteidigungssysteme, medizinische Geräte), als dass man „ungeplant“ hätte entwickeln können. Aus den bisherigen Darstellungen sollte deutlich geworden sein, dass es im Laufe der letzten rund fünfzig Jahre sich verändernde, ergänzende, aber teilweise auch konkurrierende Vorstellungen gab, welche Aspekte Softwareengineering beinhalten und in welche Richtung es sich weiterentwickeln soll. Wenn man heute in Lehrbüchern nach Definitionen zum Softwareengineering sucht, findet man immer noch viele verschiedene Vorschläge. Zum einen unterscheiden sie sich darin, welche Prozesse aus dem Bereich der Erstellung von Software dem Softwareengineering zugerechnet werden und welche nicht: Gehören beispielsweise die Kostenkalkulation des Softwareprojekts, die Zusammenstellung des Entwicklerteams oder die Wahl

des Vorgehensmodells schon in den Bereich Softwareengineering oder beginnt Softwareengineering erst **danach**? Wann endet Softwareengineering: Inwiefern gehören Wartung, Weiterentwicklung, aber auch Vertrieb der Software dazu? Zum anderen gehen die Meinungen, inwieweit Softwareengineering eine (echte) ingenieurwissenschaftliche Disziplin ist, auch heute noch auseinander (dasselbe gilt im Übrigen auch für die gesamte Informatik). Die Gründe für die unterschiedlichen Vorstellungen über das Gebiet Softwareengineering sind die gleichen wie vor fünfzig Jahren: Menschen mit unterschiedlichen (beruflichen) Hintergründen oder unterschiedlichen aktuellen Arbeitsgebieten definieren Softwareengineering und vor allem auch das Aufgabengebiet von Softwareingenieuren und -ingenieurinnen unterschiedlich.

Prinzipien des Ingenieurwesens für die Entwicklung von Software nutzen

Nichtsdestotrotz besteht heute Einigkeit darüber, dass Softwareengineering sich dadurch auszeichnet, dass Prinzipien des Ingenieurwesens auf die Entwicklung von Software angewendet werden. Dazu gehören die systematische Entwicklung und Verwendung von Methoden, Standards und Werkzeugen und die Entwicklung von Maßsystemen zur Messung und Qualitätsbeurteilung von Softwareeigenschaften genauso wie die intensive Nutzung von Erfahrungswerten. Des Weiteren sind folgende Punkte Konsens:

- Softwareengineering ist mehr als nur Programmierung, beinhaltet also noch weitere Prozesse.
- Softwareengineering beschäftigt sich mit der **systematischen** Entwicklung von **komplexer** Software. Es geht also zum einen darum, geplant zu entwickeln, statt „irgendwie zu programmieren“. Und zum anderen liegt der Fokus auf den größeren, komplexeren Softwaresystemen und nicht auf dem kleinen, einfachen Programm zur automatischen Bewässerung der eigenen Zimmerpflanzen.
- Softwareengineering beruht neben theoretischen (mathematischen) Grundlagen auch auf heuristischen Techniken und Methoden (Best Practices).

Der schon erwähnte Sevocab-Standard definiert Softwareengineering dementsprechend als:

Definition Softwareengineering

„systematic application of scientific and technological knowledge, methods, and experience to the design, implementation, testing, and documentation of software.”

Eintrag „software engineering“ bei www.computer.org/sevocab

Die Sevocab Softwareengineering-Definition stellt die theoretisch fundierten Erkenntnisse und die Erkenntnisse aus praktischen Erfahrungen gleichwertig nebeneinander. Wie Sie im Laufe des Kurses noch sehen werden, dominieren aber in manchen Prozessen des Softwareengineering – vor allem in der Anforderungsermittlung und im Entwurf – heute die Best Practice Techniken. Eine theoriegeleitete Beschäftigung mit Softwareengineering existiert nichtsdestotrotz auch heute noch, und dies hat auch eine starke Berechtigung. Einerseits werden hier Grundlagen entwickelt, die in der Zukunft die Softwareentwicklung revolutionieren könnten. Zum Anderen unterscheiden sich Softwaresysteme unter anderem darin, welche Konsequenzen Fehler haben. Stark sicherheitskritische Anwendungen, zum Beispiel in der Raumfahrt oder in der Medizintechnik, erfordern ein weit

höheres Maß an Vertrauen in die Korrektheit von Software als andere Anwendungen; hier spielen formale Methoden weiterhin eine große Rolle.

formale Methoden vs. Best Practices
Schon lange vor der Entstehung von agilen Entwicklungsansätzen, die in den 1990er Jahren eine bis heute bestehende Konfliktlinie zwischen Befürwortern und Gegnern agiler Softwareentwicklung eröffneten (s. Kap. 2.3), existierte im Softwareengineering eine ältere Konfliktlinie zwischen zwei wettstreitenden Fraktionen: diejenigen, die an formale Methoden und Korrektheitsbeweise für Software glaubten, und diejenigen, die all diese mathematisch-logisch orientierten Methoden für nicht praktikabel und viel zu aufwändig hielten. Beide Seiten hielten sich gegenseitig mangelhaftes ingenieurmäßiges Denken vor: Die formale Fraktion verwies auf die formalen Grundlagen in den anderen Ingenieurwissenschaften (z.B. Statikberechnungen), die andere Seite verwies auf die Orientierung an Best Pratices und die Notwendigkeit, Methoden für aktuell existierende Probleme und aktuelle Softwareentwicklungsprojekte zu erstellen, statt Grundlagen für irgendeine Zukunft zu legen. Formale Spezifikationen zu Projektbeginn bei der Formulierung von Anforderungen können in diesem Zusammenhang als ein Kompromiss betrachtet werden. Die Vision der formalen Fraktion war und ist, dass Software eigentlich gar nicht mehr (von Menschen) entwickelt werden muss, sondern automatisch aus hinreichend präzisen Anforderungen generiert wird und dann per Konstruktion diese Anforderungen erfüllt (natürlich muss dafür auch diese Generierungssoftware fehlerfrei sein). Mit diesem Ziel wurden und werden eigene Sprachen für die Spezifikation und auch neue Programmiersprachen entwickelt. Man könnte diese Forschungsrichtungen ebenfalls Softwareengineering nennen, tatsächlich verteilen sie sich aber über andere Teildisziplinen der Informatik, und werden auch an der FernUniversität in anderen Modulen gelehrt. Deshalb gehen wir auch in diesem Kurstext den sehr formalen Ansätzen nicht intensiver nach. Bemerkenswert bleibt aber, dass die Ursprünge des Softwareengineering durchaus auch auf eher formalen Ansätzen beruhten und beide genannten Fraktionen anfangs beteiligt waren.

Softwareengineering ist ein Fachgebiet, das sich weiterhin im Wandel befindet. Anhand der Herausforderungen neuer Softwarekonzepte oder neuer Arten von Softwaresystemen entstehen neue, erweiterte oder veränderte Programmiersprachen, Notationen, Vorgehensmodelle, Methoden und Techniken in allen Bereichen des Softwarelebenszyklus - und das meist außerhalb des wissenschaftlichen Bereichs. Jenseits aller Standardisierungsbemühungen und verschiedener Versuche, der Disziplin ein stärkeres wissenschaftliches Fundament zu geben, wird sich Softwareengineering auch weiterhin in erster Linie dadurch definieren, welche Fähigkeiten ein Softwareingenieur in der Praxis benötigt und was er in seiner täglichen Arbeit tut. Die universitäre Lehre (und ein Kursskript zum Thema Softwareengineering) stellt dies vor große Herausforderungen. Wenn wir uns zu sehr auf die neuesten oder aktuell beliebtesten Entwufsmethoden, Frameworks oder Entwicklungswerkzeuge konzentrieren, könnte auch das, was wir heute als neues-

te Errungenschaften lehren, morgen schon wieder veraltet sein. Es könnte sich auf der anderen Seite in der Praxis (noch) nicht durchgesetzt haben. Um das beliebte Beispiel der Vorgehensmodelle noch einmal zu bemühen: Wir können sowohl Scrum als auch das Wasserfallmodel vorstellen (und werden das auch tun), aber aus Lehrperspektive wichtiger ist die Vermittlung, was ein Vorgehensmodell auszeichnet, warum man es im Softwareengineering einsetzt und welche Vor- und Nachteile unterschiedliche Kategorien von Vorgehensmodellen haben. Insofern werden wir uns in diesem Kurs auf solche übergreifenderen Aspekte fokussieren.

Heute gibt es sehr viele unterschiedliche Arten von Softwaresystemen, für die unterschiedliche Softwareengineering-Techniken erforderlich sind. So werden sich das Softwareengineering für eine stand-alone Fotobearbeitungssoftware, für ein eingebettetes System eines Autos und für eine Webanwendung eines Onlineshops in den eingesetzten Programmiersprachen, Vorgehensmodellen und Entwurfstechniken stark unterscheiden. Die unterliegenden Ideen und Konzepte des Softwareengineering lassen sich aber für alle Softwaresysteme anwenden. Für jede industriell gefertigte Software müssen in irgendeiner Art und Weise Anforderungen spezifiziert werden, die in irgendeiner Notation dokumentiert werden müssen. Jede Software durchläuft irgendwann einen oder mehrere Entwurfsprozess(e), muss implementiert, verifiziert oder getestet, an die Kunden ausgeliefert und gewartet werden. Und alle diese Aktivitäten müssen innerhalb gegebener Zeit-, Kosten-, und Personalbudgets in hoher Qualität erfolgreich abgeschlossen werden. Dementsprechend beschäftigt sich dieser Kurs schwerpunktmäßig mit den grundlegenden Aspekten des Softwareengineering, die unabhängig von der Art des zu entwickelnden Softwaresystems eine Bedeutung haben.

1.3.2 Rollen, Prozesse, Aktivitäten – Begriffe des Softwareengineering

Stakeholder

An einem Softwareentwicklungsprojekt sind in der Regel verschiedene Personengruppen beteiligt, die im Englischen und mittlerweile zunehmend häufiger auch in deutschsprachiger Literatur mit dem Begriff *Stakeholder* (Interessenvertreter) bezeichnet werden. Zu den Stakeholdern eines Softwareentwicklungsprojekts zählen zum Beispiel Softwareentwickler, Softwarearchitekten, Tester, Qualitätssicherungsexperten, Projektleiter, Domänenexperten, Auftraggeber, (zukünftige) Nutzer. Je nach Projektgröße werden unterschiedlich viele Personen am Projekt beteiligt sein. Gerade bei kleineren Projekten muss ein konkretes Teammitglied oft mehrere Aufgaben wahrnehmen, zum Beispiel könnten die Aufgaben der Softwarearchitektin von einer der Softwareentwicklerinnen mit übernommen werden oder der Auftraggeber gleichzeitig der Domänenexperte sein. In großen Projekten könnten mehrere Teammitglieder identische oder ähnliche Aufgabengebiete besitzen oder ein Aufgabengebiet von wechselnden Personen übernommen werden. Wenn man über das Team spricht, das an einem Softwareentwicklungsprojekt beteiligt ist, abstrahiert man daher von konkreten Personen und spricht stattdessen von sogenannten *Rollen*. Jeder Rolle sind bestimmte Aufgaben zugeordnet. Zudem ist sie mit spezifischen Kenntnissen und Fähigkeiten verknüpft, die benötigt werden, um die der Rolle zugeordneten Aufgaben ausführen zu können. Im konkreten Projekt nimmt jedes Teammitglied

Rollen im Softwareentwicklungsprojekt

eine oder mehrere Rollen ein und führt die der Rolle/den Rollen zugeordneten Tätigkeiten aus.

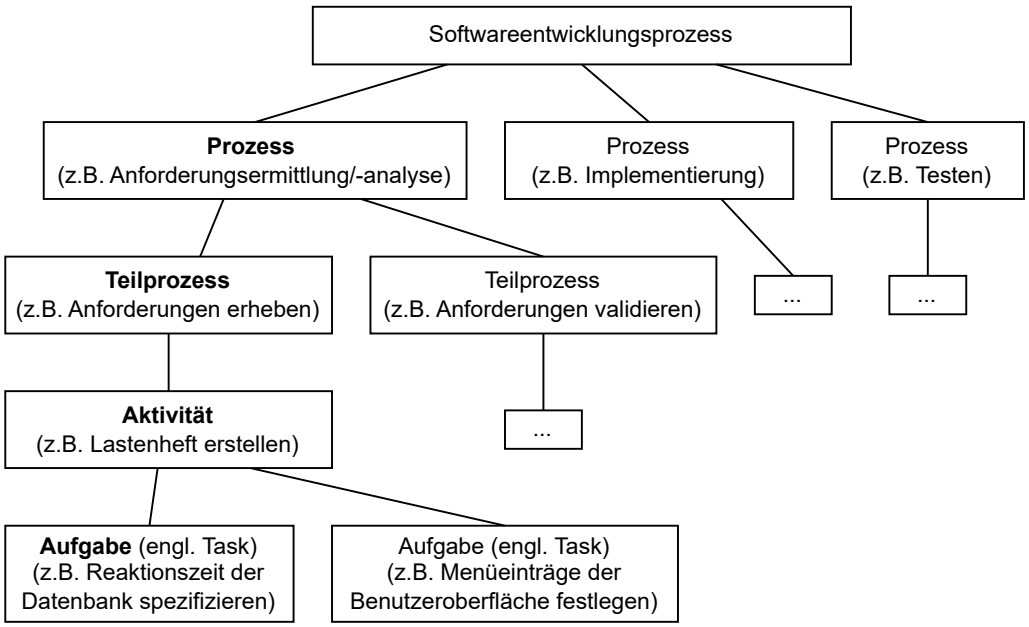


Abbildung 1.1: Zusammenhang zwischen Prozessen, Aktivitäten und Aufgaben

Die innerhalb eines Softwareentwicklungsprojekts durchgeführten einzelnen Tätigkeiten (z.B. das Lastenheft schreiben, den Testplan erstellen, eine bestimmte Funktion implementieren) kann man verschiedenen großen Bereichen des Softwareentwicklungsprozesses zuordnen, wie zum Beispiel dem Bereich der Implementierung oder dem Bereich des Testens. Diese Teilbereiche des Softwareentwicklungsprozesses bezeichnen wir als *Prozesse*. Abbildung 1.1 zeigt den Zusammenhang der in diesem Kurs verwendeten Begriffe: Ein Prozess, wie zum Beispiel der Prozess der Anforderungsermittlung, kann in *Teilprozesse* weiter unterteilt werden. Jeder Teilprozess besteht aus einer Menge von *Aktivitäten*, wie zum Beispiel „Lastenheft erstellen“ oder „Module der Software bestimmen“. Aktivitäten wiederum können in noch kleinere Einheiten, sogenannte *Aufgaben* (engl. Tasks), unterteilt werden, die schlussendlich von Rollen ausgeführt werden. Ein konkreter Softwareentwicklungsprozess ist somit eine Menge aus einzelnen Aktivitäten zusammengesetzter Prozesse. Dabei können sich die Prozesse oder Teilprozesse auch überschneiden, sodass es häufig nicht möglich ist genau zu bestimmen, an welcher Stelle ein Prozess bzw. Teilprozess endet und ein anderer beginnt. Ebenso lässt sich nicht immer eindeutig bestimmen, welchem Teilprozess eine konkrete Aufgabe zugeordnet ist.

Prozess, Aktivität, Aufgabe

Der Software Engineering Body of Knowledge (SWEBOK) definiert einen ingenieurwissenschaftlichen Prozess als

„a set of interrelated activities that transform one or more inputs into outputs while consuming resources to accomplish the transformation“. [swe14, S. 8–1]

Bezogen auf den Gesamtprozess der Entwicklung eines Softwareprodukts besteht der Input aus der Menge der Anforderungen an die zu erstellende Software und der Output aus dem fertigen Softwareprodukt. Die einzelnen Prozesse innerhalb des Softwareentwicklungsprozesses transformieren unterschiedliche Arten von Inputs in Outputs, wobei die Outputs eines Prozesses oft die Inputs eines oder mehrerer Folgeprozesse sind. So könnte zum Beispiel der Output eines Implementierungsprozesses ein Stück Programmcode sein und Letzteres ein Teil des Inputs für den Prozess des Testens. Bei den verbrauchten Ressourcen für die Erstellung des Softwareprodukts handelt es sich in erster Linie um die Arbeitszeit der an der Entwicklung beteiligten Personen. Ressourcen können aber zusätzlich auch Softwareressourcen (z.B. fertige Softwarekomponenten, die in das Produkt eingebunden werden, oder für die Entwicklung verwendete Tools) und Hardwareressourcen (z.B. Entwicklungsinfrastruktur) sein.

Hinweis

unterschiedliche Begriffsverwendung

Beachten Sie, dass Teile der Literatur die großen Bereiche des Softwareentwicklungsprozesses wie Anforderungsermittlung/-analyse, Implementierung oder Testen nicht als **Prozesse**, sondern schon als **Aktivitäten** bezeichnen. Für konkrete Tätigkeiten, wie zum Beispiel für die Tätigkeit „das Lastenheft erstellen“ innerhalb der Anforderungsermittlung/-analyse werden dann Begriffe wie Unteraktivitäten oder Sub-Aktivitäten verwendet.

Kernprozesse, unterstützende Prozesse

Bei den angesprochenen Prozessen des Softwareengineering unterscheidet man *Kernprozesse* (engl. primary processes, development processes, implementation processes) von *unterstützenden Prozessen* (engl. support processes). Letztere werden häufig auch als Softwaremanagementprozesse bezeichnet. Seit Mitte der 1980er Jahre besteht relative Einigkeit darüber, welche **Kernprozesse** dem Softwareengineering zuzurechnen sind, auch wenn die Benennungen, die konkreten Aktivitäten und die Abgrenzungen zwischen den Prozessen je nach Blickwinkel differieren können. Die Kernprozesse des Softwareengineering sind:

- die Anforderungsermittlung/-analyse
- der Softwareentwurf
- die Implementierung
- das Testen bzw. allgemeiner die Qualitätssicherung sowie
- die Wartung, ggf. ergänzt um die Weiterentwicklung der Software

Mittlerweile ist unbestritten, dass neben den Kernprozessen auch Softwaremanagementprozesse Teil des Softwareengineering sind. Welche Managementprozesse dies genau betrifft, variiert aber stark in der Literatur. Der Schwerpunkt unseres Kurses liegt auf den Kernprozessen des Softwareengineering, die unterstützenden Prozesse werden an inhaltlich passenden Stellen thematisiert. Zum Beispiel behandeln wir in Kapitel 2 das Thema der Vorgehensmodelle.

Aufbau des Kursskripts

Lassen Sie uns an dieser Stelle die inhaltliche Ebene verlassen und auf die noch ausstehende Darstellung zum Aufbau des Kursskripts zurückkommen. Kapitel 2 in dieser ersten Kurseinheit behandelt das Thema der Vorgehensmodelle und deren Zusammenhang zu den Prozessen des Softwareengineering.

Wie erwähnt legt der Kurs den Schwerpunkt auf grundlegende Aspekte des Softwareengineering, die weitestgehend unabhängig von der Art des (zu entwickelnden) Softwareprodukts sind. Einige Einschränkungen müssen wir allerdings doch treffen, um den Umfang des Kurses im Rahmen zu halten: Der Kurs beschäftigt sich nur mit objektorientierter Softwareentwicklung und aus der Menge der möglichen Notationssprachen wird für diesen Kurs die Unified Modeling Language (UML) gewählt, die die bei Weitem größte Verbreitung genießt. Dementsprechend wird sich Kurseinheit 2 zum einen mit den Prinzipien objektorientierter Softwareentwicklung und zum anderen mit den Themen Modellierung und UML beschäftigen. Die detaillierte Beschäftigung mit den Kernprozessen des Softwareengineering erfolgt ab Kurseinheit 3.

1.4 Kommentierte Literatur

Naur/Randell (1969). Software Engineering: Report on a Conference 1968 [nau69]

Der Abschlussbericht zur Softwareengineering-Konferenz in Garmisch 1968. Die Diskussionen auf der Konferenz wurden detailliert protokolliert und zu großen Teilen sogar auf Band aufgenommen, sodass der Bericht neben der Zusammenfassung des Diskussionsverlaufs auch Originalredebeiträge der Teilnehmer aus den Diskussionen darstellen kann. Eine redaktionelle Bearbeitung durch die Herausgeber des Berichts erfolgte dabei nur insofern, dass die Redebeiträge unabhängig von ihrer chronologischen Reihenfolge thematisch zugeordnet sowie inhaltlich passende Passagen aus den auf der Konferenz vorgestellten Working Papers der Darstellung des Diskussionsverlaufs hinzugefügt wurden. Zur Folgekonferenz in Rom im Jahr 1969 existiert ebenfalls ein Abschlussbericht [bux70].

Grier (2011). Software Engineering: History [gri11]

Sechsseitiger überblicksartiger Artikel zur Geschichte des Softwareengineering aus der Enzyklopädie des Softwareengineering [lap11] unter dem Blickwinkel, durch welche Entwicklungen sich Softwareengineering zu einer eigenen Disziplin entwickelt hat. Die Meilensteine der frühen Jahre werden chronologisch dargestellt. In der Folge beleuchtet der Artikel dann systematisch, welche Entwicklungen seit den 1970er Jahren bis Ende der 1980er Jahre bezüglich der Prozesse Spezifikation, Entwurf, Implementierung, Test und Wartung von Software wichtig waren. Abschließend beschäftigt sich der Artikel mit der Frage, welche der Merkmale einer Disziplin (Fachgesellschaften, Curricula, Handbücher etc.) Softwareengineering heute (2011) aufweist und inwiefern sich Softwareengineering von den klassischen Ingenieurwissenschaften unterscheidet.

Díaz-Herrera/Freeman (2014). Discipline of Software Engineering: An Overview [dia14]

Ein umfangreicher Artikel aus dem Handbuch Computer Science and Software Engineering [gon14], der einen sehr detaillierten Überblick über die wichtigen Entwicklungen im Softwareengineering von den Anfängen bis zur Gegenwart (2014) gibt. Der Artikel verweist dabei auch intensiv auf die historischen Dokumente zur Softwareentwicklung (wie z.B. die Arbeiten von Dijkstra und Parnas). Er ist daher sehr gut als Ausgangspunkt geeignet, um sich noch intensiver in die Geschichte des Softwareengineering zu vertiefen. Gleichzeitig beschäftigt sich der Artikel mit der Frage, was das ingenieurmäßige an Softwareengineering ist und inwiefern sich Softwareengineering andererseits von den klassischen Ingenieurwissenschaften unterscheidet. Der Artikel stellt außerdem die Diskussion vor, ob und inwiefern Softwareengineering eine Disziplin ist. Wer sich noch intensiver mit dem Thema Softwareengineering als Disziplin befassen möchte, sei zusätzlich auf [sei14] und [wan00] verwiesen.

Booch (2018). The History of Software Engineering [boo18]

Inhaltlich sehr dichter (Aufzählung vieler Personen und Ereignisse) siebenseitiger Artikel zum 50jährigen Jubiläum des Softwareengineering über dessen Entwicklung, geschrieben

von einem der Pioniere der objektorientierten Softwareentwicklung und der UML. Im Gegensatz zu der anderen hier vorgestellten Literatur beschäftigt sich der Artikel auch mit frühen Ereignissen der Computerhistorie des späten 19. und frühen 20. Jahrhundert (Babbage, ENIAC, Turing etc.). Zudem listet er Errungenschaften des Softwareengineering ab den späten 1990er Jahren auf, was die anderen hier vorgestellten Artikel zur Geschichte des Softwareengineering aufgrund ihrer Fokussierung auf die Disziplinfrage etwas vernachlässigen. Der Fokus des Artikels liegt dabei immer auf der Darstellung von technologischen und gesellschaftspolitischen Gegebenheiten der jeweiligen Jahrzehnte und deren Auswirkungen auf die Ausrichtung des Softwareengineering.

del Águila/Palma/Túnez (2014). Milestones in Software Engineering and Knowledge Engineering History: A Comparative Review [del14]

Der Zeitschriftenartikel vergleicht die Meilensteine in der Entwicklung des Softwareengineering mit denen in der Entwicklung des Knowledge Engineering[1]. Für die Darstellung der Entwicklungen im Softwareengineering verwenden die Autorinnen und Autoren eine Kategorisierung von [end97] aus dem Jahr 1996, die die Geschichte des Softwareengineering in wenige große zeitliche Phasen einteilt, und erweitern diese bis in die Gegenwart (2014). Dadurch zeigt der Artikel stärker als die bisher erwähnte Literatur die größeren Linien in der Geschichte des Softwareengineering, ist gleichzeitig aber auch deutlich weniger detailliert bezüglich der einzelnen Errungenschaften. Der eigentliche Fokus des Artikels liegt auf der Frage, wie Softwareengineering und Knowledge Engineering voneinander lernen und sich weiterentwickeln können.

Mahoney (2004). Finding a History for Software Engineering [mah04]

Ein etwas anderer Ansatz die Entwicklung des Softwareengineering darzustellen. Der Wissenschaftshistoriker Michael Mahoney beleuchtet in seinem Zeitschriftenartikel die akademische und berufliche Sozialisation der Teilnehmer der 1968er-Konferenz, um zu erklären, warum es auf der Konferenz und auch bis in die Gegenwart (2004) so unterschiedliche Ansichten darüber gibt, in welche Richtung sich Softwareengineering entwickeln soll. Er identifiziert drei Gruppen [s. S. 3]. Im Rahmen der Vorstellung dieser drei Gruppen führt der Artikel wichtige Meilensteine der Entwicklung des Softwareengineering auf.

Tanenbaum/Austin (2014). Rechnerarchitektur: Von der digitalen Logik zum Parallelrechner [tan14]

Ein auch für Anfänger sehr gut verständliches Lehrbuch mit umfangreichen Informationen zu den verschiedenen Computergenerationen, zu Compilern, Assemblern und vielen weiteren hardwarenahen Themen.

[1] in Deutsch: Wissensmodellierung. Das Themenfeld Knowledge Engineering in der Informatik beschäftigt sich mit der Erforschung und Entwicklung wissensbasierter Systeme. Es ist ein Teilgebiet des Bereichs künstliche Intelligenz.

Wirth (2008). A Brief History of Software Engineering [wir08]

Ein auf sieben Seiten sehr persönlich geprägter Blick auf die Geschichte des Softwareengineering von Niklaus Wirth, der heute vor allem für die Entwicklung der Programmiersprache Pascal bekannt ist, aber auch an der Entwicklung verschiedener anderer Programmiersprachen beteiligt war. Der Artikel beschäftigt sich schwerpunktmäßig mit der Implementierungsebene und den dortigen Entwicklungen der 1960er bis 1980er Jahre.

Sommerville (2018). Software Engineering [som18]

Die Entwicklung des Softwareengineering ist in der neuesten Auflage auf die Website zum Buch (https://software-engineering-book.com/web/history/) ausgelagert worden und wird nur sehr knapp dargestellt. Im ersten Kapitel des Buchs beschäftigt sich der Autor unter anderem mit den heute existierenden unterschiedlichen Arten von Softwaresystemen und der Frage, warum die Fundamente des Softwareengineering trotzdem für alle gelten (und gelehrt werden können). Die verschiedenen informationstechnischen und organisatorischen Prozesse des Softwareengineering werden ausführlich und jeweils mit Bezug zu vier Fallstudien, die im ersten Kapitel des Buchs vorgestellt werden, behandelt. Dieses Lehrbuch wird Ihnen in den kommentierten Literaturlisten der weiteren Kurskapitel wieder begegnen.

Mühlbauer (2015). Kurze Einführung in die Normung [mue15]

Vom Deutschen Institut für Normung e.V. (DIN) herausgegebene, sehr informative, kleine Broschüre zur Entstehung von Normen, dem Unterschied zwischen Normen und Standards, dem Zusammenspiel zwischen DIN und ISO und weiteren Themen des Bereichs Normung.

McCall (2011). IEEE Computer Society [mcc11]

Zehnseitiger Artikel über die IEEE Computer Society aus der Enzyklopädie des Softwareengineering [lap11] mit detaillierten Informationen zu ihren Mitgliedern, Gremien, Standardisierungsaktivitäten, Fachzeitschriften und Zertifikaten. Der Artikel stellt auch SWEBOK vor, allerdings die Vorgängerversion von 2010, die sich von der 2014er-Version im Aufbau unterscheidet. Der Artikel ist sehr informativ, hat aber einen leicht werbenden Charakter - die Autorin war (2011) Corporate Communications Manager der IEEE Computer Society.

Bourque/Fairley (Hrsg.). (2014). SWEBOK V3.0 [swe14]

Kompendium des Softwareengineering [s. S. 3], unter www.computer.org/web/swebok als PDF verfügbar. Kapitel 8 des SWEBOK beschäftigt sich umfassend mit Softwareengineering-Prozessen und ist Grundlage für die hier im Kurstext in Kapitel 1.3 dargestellten Inhalte.

ISO/IEC/IEEE 24765: (2017). Systems and Software Engineering - Vocabulary [iso17]

Die aktuelle, schriftlich veröffentlichte, Sevocab-Norm [s. Kap. 1.2], online kostenpflichtig verfügbar unter www.iso.org/standard/71952.html. Die Sevocab-Datenbank ist unter www.computer.org/sevocab kostenfrei verfügbar.

Shafer (2011). Process [sha11]

Der Artikel aus der Enzyklopädie des Softwareengineering [lap11] bietet eine sehr umfassende Darstellung des Prozessbegriffs im Bereich Softwareengineering. Neben der Darstellung, wie sich ein Prozess des Softwareengineering definiert, was er beinhaltet und wie er sich verbessern lässt, behandelt der Artikel auch den Zusammenhang zwischen den Softwareengineeringprozessen und Vorgehensmodellen, mit denen wir uns in Kapitel 2 dieser Kurseinheit beschäftigen. Zudem werden Qualitätssicherungsprozesse thematisiert (die wir in Kurseinheit 6 behandeln werden) und verschiedene Reifegradmodelle vorgestellt. Mit Reifegradmodellen, die die Fähigkeit eines Unternehmens bewerten, Softwareentwicklungsprojekte erfolgreich durchzuführen, beschäftigen wir uns im Rahmen dieses Kurses nicht. Der Artikel von Shafer stellt zudem SWEBOK (in der Version von 2004) und andere internationale Standards vor, die Softwareengineering-Prozesse behandeln.

Dumke (2003). Software Engineering [dum03]

Im Unterschied zu den meisten anderen Büchern zum Thema wird Softwareengineering hier aus einer Ingenieurperspektive statt aus einer Informatikerperspektive betrachtet. Deutlich stärker als in anderer Literatur richtet sich der Blickwinkel daher auf die Frage, was eigentlich das ingenieurmäßige am Softwareengineering ist. Für das Themenfeld dieses ersten Kapitels des Kurstexts relevant sind vor allem die Abschnitte 1.1 und 1.2 des Buchs. In Abschnitt 1.1 stellt der Autor grundlegende Begriffe des Softwareengineering vor – in ausführlicherem Umfang, als wir es im Rahmen dieses Kapitels getan haben. In Abschnitt 1.2 werden auf knapp 80 Seiten die Kernprozesse des Softwareengineering anhand von fünf kleinen Softwareproduktbeispielen beschrieben. Dieser Abschnitt wird Ihnen auch in späteren kommentierten Literaturlisten wieder begegnen.

Kapitel 2

Vorgehensmodelle im Softwareengineering

[...]

2.1 Vorgehensmodelle – Ziele und Abgrenzungen

[...]

2.2 Kategorien von Vorgehensmodellen

Konkrete Vorgehensmodelle unterscheiden sich mindestens darin, in welcher Granularität sie die durchzuführenden Tätigkeiten in einem Softwareentwicklungsprojekt vorgeben. Doch deutlich stärkere Unterschiede zwischen Vorgehensmodellen finden sich, wenn diese unterschiedlichen *Paradigmen* folgen. Ein Paradigma ist eine grundsätzliche Denkweise/Lehrmeinung, man findet als Synonyme auch die Begriffe Weltanschauung oder Weltbild. Möglicherweise ist Ihnen der Begriff des Paradigmas in der Informatik aus dem Bereich der Programmierung bekannt. Programmiersprachen folgen einem Programmierparadigma (wie zum Beispiel dem objektorientierten Paradigma), wenn sie bestimmte im Paradigma festgelegte Prinzipien einhalten.

Paradigma

Die ersten im Softwareengineering eingesetzten Vorgehensmodelle folgten einem Paradigma, das man als *plangesteuert* bezeichnen kann. Diesem plangesteuerten Paradigma liegt die Einschätzung zugrunde, dass Softwareentwicklungsprojekte nur dann erfolgreich abgeschlossen werden können, wenn die durchzuführenden (Teil)Prozesse des Softwareengineering und ihr kausal-zeitlicher Ablauf im Vorfeld des Projekts systematisch geplant werden und spätere (Teil)Prozesse immer erst bei Vorliegen von vollständigen, qualitätsgesichertern und dokumentierten Ergebnissen vorhergehender Prozesse starten. Vorgehensmodelle, die dem plangesteuertem Paradigma folgen, gehören zur Kategorie der sogenannten *sequentiellen Modelle* oder Phasenmodelle. Der bekannteste Repräsentant sequentieller Modelle ist das Wasserfallmodell.

sequentielle Modelle

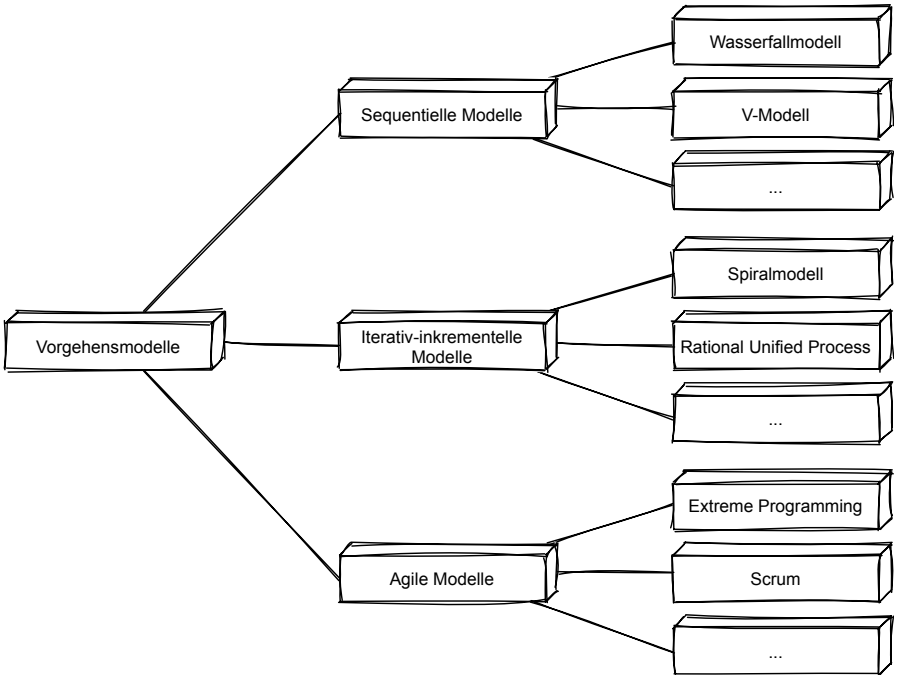


Abbildung 2.1: Kategorien von Vorgehensmodellen

agile Modelle

Im Unterschied zum plangesteuerten Paradigma wird im sogenannten agilen Paradigma, das seit Ende der 1990er Jahre verstärkt propagiert wird, die Einschätzung vertreten, dass ein Softwareentwicklungsprojekt nur dann erfolgreich abgeschlossen werden kann, wenn im Projektverlauf Änderungen zugelassen werden können (im Besonderen Änderungen der Anforderungen) und schon ab frühen Zeitpunkten im Projektverlauf lauffähiger (aber auch wieder änderbarer) Programmcode erzeugt wird. Vorgehensmodelle, die dem agilen Paradigma folgen, nennt man *agile Modelle*. Bekannte Repräsentanten sind Extreme Programming und Scrum.

Sowohl sequentielle als auch agile Modelle werden heute im Softwareengineering eingesetzt. Konkrete Vorgehensmodelle – sowohl diejenigen, die wir in diesem Kurs vorstellen als auch die vielen anderen, die wir hier nicht thematisieren – passen in der Regel nicht hundertprozentig in genau eine Kategorie, da sie zusätzlich oft auch Kennzeichen anderer Kategorien aufweisen. In der Praxis gilt dies umso mehr, je stärker die Grundform eines Vorgehensmodell individuell an unternehmensspezifische Belange angepasst wird (s. Kap. 2.3)

Wir werden in den folgenden Abschnitten sowohl allgemeiner die Kennzeichen sequentieller Modelle (Kap. 2.2.1) und agiler Modelle (Kap. 2.2.3) als auch konkrete Vorgehensmodelle als Repräsentanten dieser Kategorien von Vorgehensmodellen vorstellen. Kapitel 2.2.2 thematisiert zwischen der Vorstellung der sequentiellen und der Vorstellung der agilen Modelle eine dritte Kategorie von Vorgehensmodellen, die *inkrementellen und iterativen Modelle*, die in den späten 1980er und frühen 1990er Jahren erstmalig

inkrementelle und iterative Modelle

vorgestellt wurden und damit auch in ihrer Entstehungszeit zwischen den sequentiellen und den agilen Modellen liegen. Deren zugrundeliegendes Paradigma betont den Stellenwert der fachlichen Aspekte (Strukturen, Geschäftsprozesse etc.) des Einsatzgebiets des zu entwickelnden Softwareprodukts – und kritisiert damit auch die in der Regel sehr technisch-orientierte Sichtweise von sequentiellen Vorgehensmodellen. Zum anderen beinhaltet es die Einschätzung, dass für erfolgreich durchzuführende Softwareentwicklungsprojekte lauffähiger Programmcode nicht erst am Ende des Projekts vorliegen darf – hier wurde die Basis für die Programmcode-Fokussierung der späteren agilen Modelle gelegt.

Unterscheidungsmerkmale zwischen Vorgehensmodellen

Sequentielle, iterativ-inkrementelle (synonym: inkrementell-iterativ) und agile Vorgehensmodelle unterscheiden sich vor allem in folgenden Aspekten, die wir bei der Vorstellung der drei Kategorien in den folgenden Kapiteln jeweils im Detail betrachten werden:

1. In welcher Weise werden die einzelnen (Teil)Prozesse zum Softwareentwicklungsprozess zusammengestellt?
2. Wie wird mit neuen oder veränderten Anforderungen während der Entwicklung umgegangen?
3. Inwieweit werden Auftraggeber und zukünftige Nutzer des zu erstellenden Softwareprodukts in die Entwicklung einbezogen?
4. Zu welchen Zeitpunkten liegen auslieferungsfähige Produkte bzw. Teilprodukte vor?
5. Welche Formen von Artefakten (z.B. Programmcode, Dokumente, Modelle) entstehen im Laufe des Softwareentwicklungsprozesses?

2.2.1 Sequentielle Modelle

[...]

2.2.2 Inkrementelle und iterative Modelle

[...]

2.2.3 Agile Modelle

[...]

2.3 Vorgehensmodelle – Vergangenheit, Gegenwart und Zukunft

[...]

Einleitung zur Kurseinheit

Sie haben in Kurseinheit 1 gelernt, dass sich ein Softwareentwicklungsprozess aus einzelnen Prozessen (wie Anforderungsermittlung oder Implementierung) zusammensetzt, in deren Rahmen verschiedene Aktivitäten ausgeführt werden. Sie wissen zudem, dass man diesen Entwicklungsprozess auf unterschiedliche Arten strukturieren kann, indem man Vorgehensmodelle einsetzt. — das letzte Mal

In dieser Kurseinheit wird wieder der Begriff **Modell** fallen. Wir werden uns mit dem Modellbegriff im Softwareengineering, mit der Realweltorientierung als zentraler Idee der Objektorientierung und darauf aufbauend mit der *objektorientierten Modellierung* beschäftigen. Letztere zeichnet sich dadurch aus, dass objektorientierte Prinzipien durchgängig in den Modellierungsprozessen der Anforderungsermittlung und -analyse und des Entwurfs eingesetzt werden und nicht nur bei der Implementierung eine objektorientierte Programmiersprache verwendet wird. Auf dieser Grundlage betrachten wir die Modellierung von Realweltzusammenhängen – in der Terminologie des objektorientierten Softwareengineering als *Domänenmodellierung* bezeichnet. Wir zeigen, wie sich mithilfe der Modellierungssprache UML Objekte der Realwelt, ihre Eigenschaften und Beziehungen modellieren lassen und wie die für die Objektorientierung wichtigen Klassen damit in Zusammenhang stehen. Diese Kurseinheit ist im Unterschied zu den folgenden noch nicht einem spezifischen Prozess des Softwareengineering gewidmet, da die Themenbereiche Objektorientierung und Modellierung alle Prozesse betreffen. Im Zuge der Domänenmodellierung werden Sie aber schon auf Verbindungen zum Prozess der Anforderungsermittlung und -analyse treffen, der den Schwerpunkt der nächsten Kurseinheit bildet. — dieses Mal

Diejenigen von Ihnen, die schon Erfahrung mit Objektorientierung und insbesondere mit objektorientierter Programmierung haben, werden in dieser Kurseinheit viele Aspekte vermissen. Warum reden wir über Klassen, aber nicht über Interfaces? Warum fehlen Themen wie Sichtbarkeit, Polymorphie und Überschreiben? Es ist eine Frage der Perspektive! Wir fokussieren unseren Blick in dieser Kurseinheit auf die Modellierung von Realweltstrukturen. Dafür benötigen wir nur wenige basale Konzepte. Die weiteren, sehr mächtigen Konzepte der Objektorientierung, die die UML auch abbilden kann, spielen für Realweltmodellierungszwecke kaum eine oder gar keine Rolle. Sie werden manche von ihnen im weiteren Verlauf des Kurstexts kennenlernen. — Realweltfokus

Es bleibt die Frage, warum wir uns im Rahmen eines Softwareengineering-Kurses über-

haupt mit Modellen beschäftigen, wenn das Anliegen der Softwareentwicklung die Erstellung von funktionierendem Programmcode ist. Mal abgesehen davon, dass auch Programmcode ein Modell ist, nämlich das Modell des ausführbaren Programms – zunächst

der Sinn von Modellen

ein pragmatischer Grund: An einem Softwareentwicklungsprojekt sind sehr viele unterschiedliche Personen und Personengruppen beteiligt, von denen die meisten in der Regel nicht programmieren können. Diskussionen oder Dokumentationen auf der Grundlage von Programmcode sind daher problematisch. Es braucht andere Formen für die Kommunikation und Zusammenarbeit der heterogenen Beteiligten, und im Softwareengineering – wie in vielen anderen Bereichen – sind das Modelle. Ein weiterer Grund ist die Mächtigkeit von Modellen: mit Modellen kann man frei wählbare Perspektiven auf das existierende oder zu erstellende Original (z.B. das Softwareprodukt, die Realwelt oder den Softwareentwicklungsprozess) gestalten und damit auch die Komplexität der Softwareentwicklung reduzieren, indem man ganz gezielt nur einen bestimmten Aspekt oder eine bestimmte Blickrichtung berücksichtigt und alles andere zunächst ignoriert.

Es gibt sehr viele unterschiedliche Möglichkeiten, Modelle im Softwareengineering einzusetzen. Wichtig für ein Softwareentwicklungsprojekt ist, das Modellieren nicht zum Selbstzweck werden zu lassen, sondern Modelle immer als Schritte auf dem Weg zum zu erstellenden Softwareprodukt zu sehen und somit sowohl quantitativ als auch qualitativ **zielgerichtet** zu modellieren. Das betrifft auch die Realweltmodellierung, die den Schwerpunkt dieser Kurseinheit bildet: Man erstellt Modelle der Realwelt nicht, um ein hübsches Bild von den Zuständen der Wirklichkeit zu erhalten, sondern weil man ein Softwareprodukt entwickeln möchte, das **in** dieser Wirklichkeit **oder mit** dieser Wirklichkeit arbeitet.

Fallbeispiel

Zur Einführung in die Domänenmodellierung stellen wir in dieser Kurseinheit ein Fallbeispiel vor, das Sie auch in den weiteren Kurseinheiten begleiten wird. Es dient dazu, aus einem praktischeren Blickwinkel ein paar Schlaglichter auf typische Tätigkeiten im Softwareengineering zu werfen. Die Simulation eines kompletten Softwareentwicklungsprozesses anhand eines Fallbeispiels ist im Rahmen eines solchen Kurses allerdings nicht möglich. Für das Fallbeispiel haben wir den Realweltbereich des Zoos gewählt und entschuldigen uns schon einmal im Vorfeld bei allen Zoo-Spezialistinnen und Zoo-Spezialisten für die Halbwahrheiten und erfundenen Informationen zur Lebenswelt Zoo in diesem natürlich für unsere Lehrzwecke so konstruierten Fallbeispiel.

Bezug zu den Vorgehensmodellen

Schlagen wir zum Schluss dieser Einleitung noch kurz den Bogen zurück zu den Vorgehensmodellen aus der letzten Kurseinheit. Wir haben in der aktuellen Kurseinheit – und stärker noch in den weiteren Kurseinheiten – die Situation, dass manche Aspekte des Softwareengineering, die wir darstellen, in einigen Vorgehensmodellen sehr wichtig sind und in anderen gar keine Berücksichtigung finden. Den starken Fokus auf die objektorientierte Modellierung mithilfe der UML zum Beispiel, den wir hier setzen, findet man in agilen Softwareentwicklungsprojekten häufig so nicht. In Projekten, die nach wasserfallartigem Vorgehen arbeiten, trifft man dagegen nicht selten auf den Fall, dass zwar eine objektorientierte Programmiersprache eingesetzt wird, das zentrale Konzept der Realweltorientierung der Objektorientierung (und damit auch die Domänenmodellierung)

aber in den der Implementierung vorgeschalteten Prozessen vernachlässigt wird. Wir versuchen bei Aspekten, in denen sich verschiedene Arten von Vorgehensmodellen sehr stark unterscheiden, diese Problematik explizit zu machen. Sie sollten sich aber insgesamt bewusst sein, dass nicht alle Methoden, die Sie hier im Kurs kennenlernen, in jedem praktischen Softwareentwicklungsprojekt eingesetzt werden. Und das hat ausnahmsweise mal nicht nur damit zu tun, dass die universitäre Ausbildung teilweise andere Schwerpunkte setzt als man sie in betrieblichen Ausbildungs- und Arbeitszusammenhängen findet.

Kapitel 3

Modelle im Softwareengineering

3.1 Der Modellbegriff

[...]

3.2 objektorientierte Modellierung

[...]

3.2.1 Objektorientierung

[...]

3.2.5 Realweltstrukturen modellieren

Im objektorientierten Softwareengineering versucht man das (zukünftige) Softwareprodukt so zu gestalten, dass es sich an den Objekten und Strukturen der Domäne orientiert. Die objektorientierte Modellierungssprache UML bietet mit dem sogenannten Objektdiagramm die Möglichkeit, (Abstraktionen der) Objekte der Realwelt, ihre Eigenschaften und ihre Beziehungen zueinander in einer von UML vorgegebenen Syntax zu modellieren. (Mit dem Objektdiagramm kann man im Übrigen auch Objekte modellieren, die keine Realweltentsprechungen haben, aber das ignorieren wir in dieser Kurseinheit noch.)

Objektdiagramm

Objektdiagramme können unterschiedlich umfangreich sein, auch eine Menge von Objekten ohne Verbindungen zueinander (wie in Abbildung 3.4) und selbst ein einziges aufgezeichnetes Objekt (wie in Abbildung 3.3) stellen Objektdiagramme dar. In der praktischen Anwendung im Softwareengineering beinhalten Objektdiagramme in der Regel aber mehrere verbundene Objekte.

3.2.5.1 Objekte und Klassen

Objekt

Zu modellierende Realwelt-Objekte können Dinge sein, die man sehen oder anfassen kann, wie zum Beispiel eine Katze, ein Tisch, eine Person oder ein Stern, aber auch immaterielle Dinge, wie zum Beispiel ein Konto, eine Reise oder eine Vorlesung. In der UML-Notation werden Objekte als rechteckige Kästen dargestellt, die (mindestens, s.u.) einen Objektnamen enthalten (Abb. 3.3).

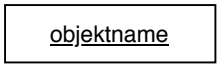


Abbildung 3.3: Ein Objekt in UML-Darstellung

Nach UML-Konvention wird der Objektname unterstrichen dargestellt. Üblich – obwohl die UML diesbezüglich keine Vorgaben macht – ist außerdem, dass Objektnamen mit einem Kleinbuchstaben beginnen und zentriert dargestellt werden. Innerhalb eines Diagramms müssen die Objektnamen eindeutig gewählt werden. Sollten innerhalb eines Diagramms trotzdem zwei Kästchen denselben Namen enthalten – die UML verbietet dies nicht –, so ist mit beiden Kästchen dasselbe Objekt gemeint. Die doppelte Darstellung eines Objekts wird vor allem in handschriftlich erstellten oder sehr umfangreichen Objektdiagrammen aus Lesbarkeitsgründen verwendet.

Hinweis

CamelCase-Schreibweise

Bezeichner (Namen) in Programmcode dürfen häufig keine Leerzeichen enthalten. Die sogenannte CamelCase-Schreibweise, bei der das erste Wort kleingeschrieben wird und alle folgenden jeweils mit einem Großbuchstaben beginnen, ist eine Möglichkeit, auch ohne Trennzeichen ausdrucksstärkere Bezeichner als z.B. `auto1` und `auto2` zu verwenden. Die CamelCase-Schreibweise findet man auch in Domänenmodellen häufig, obwohl Domänenmodelle eigentlich noch von Implementierungsaspekten abstrahieren sollten und hier die Verwendung von Leerzeichen Realwelt-näher wäre. Die UML selber macht keine Vorgaben oder Einschränkungen bezüglich der Schreibweise der Namen, spezifische UML-Werkzeuge tun dies allerdings teilweise schon.

Abbildung 3.4 zeigt weitere Objekte in UML-Darstellung. Bei manchen wird aus dem Namen (relativ) deutlich, welches konkrete Realwelt-Objekt gemeint ist. So ist `dasAutoVonPetra`, sofern man Petra kennt und sie nicht mehr als ein Auto besitzt, einem konkreten Realwelt-Auto zuordenbar. Für `reiseNr5` bedarf es dagegen schon der zusätzlichen Kenntnis des Kontexts (z.B. die Auflistung von Reisen in einem Katalog), um eine Realwelt-Reise mit diesem Namen zu verbinden. Für Objekte wie `einAlgorithmus` oder

`eineZugfahrt` und auch `dieKatze` lassen sich die gemeinten Realweltentsprechungen nicht bestimmen.

Abbildung 3.4: Ein Objektdiagramm mit sieben unverbundenen Objekten

Zumindest könnte man aber anhand der Namen der Objekte vielleicht auf die **Art** des Realwelt-Objekts schließen? So sollte es sich bei `dieKatze` doch wohl um eine Katze und nicht um einen Hund handeln? Doch vielleicht trägt mein Realwelt-Meerschweinchen – aus welchem Grund auch immer – den Namen `dieKatze` und meine Realwelt-Katze heißt stattdessen `pünktchen`. Der modellierten Abstraktion des Realwelt-Objekts kann man den Typ des Objekts in der bisher gewählten Darstellungsform also nicht ansehen.

Um den Typ eines modellierten Realwelt-Objekts anzugeben, wird in der UML-Darstellung des Objekts der Objektname um die Angabe des Namens der *Klasse* ergänzt (Abb. 3.5 rechts). Die Objektdarstellung ohne zusätzlichen Klassennamen (wie in Abb. 3.5 links und den vorherigen Abbildungen) ist nach UML-Regeln zulässig, sollte aus semantischen Gründen aber nur dann verwendet werden, wenn der Zielgruppe des Modells der Typ des Objekts bekannt ist.

Klasse

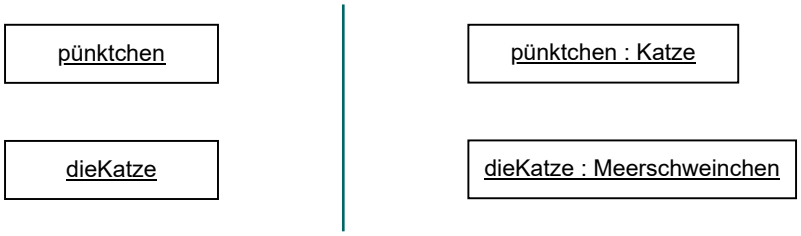


Abbildung 3.5: Ein Objekt namens `pünktchen` und ein Objekt namens `dieKatze` (links). Ein Katzen-Objekt namens `pünktchen` und ein Meerschweinchen-Objekt namens `dieKatze` (rechts). Der senkrechte Strich in der Abbildung trennt die linke und die rechte Seite dieser Abbildung. Er ist nicht Bestandteil eines UML-Objektdiagramms.

Abbildung 3.6 zeigt weitere Katzen-Objekte. Wichtig ist, dass jeweils die Angabe `:Katze` bestimmt, dass es sich um ein Objekt vom Typ Katze handelt (das gilt auch für `hund`). Die beiden unteren Objekte in der Abbildung sind sogenannte *anonyme Objekte*. Diese Form der Darstellung wird verwendet, wenn man nicht ein konkretes, mit einem Na-

anonymes Objekt

men versehenes, Katzen-Objekt modellieren möchte, sondern **irgendein** Objekt vom Typ Katze. Beachten Sie, dass auch ein anonymes Objekt nur genau **ein** Objekt ist. Es steht nicht stellvertretend für beliebig viele Katzen-Objekte. Im Unterschied zu benannten Objekten handelt es sich bei mehreren anonymen Objekten derselben Klasse im selben Objektdiagramm nach UML-Definition um **unterschiedliche** Objekte. Die beiden anonymen Katzen-Objekte in Abbildung 3.6 modellieren daher zwei unterschiedliche Realwelt-Katzen, bei denen es aber für den Modellierungszweck irrelevant ist, um welche konkreten Realwelt-Katzen es sich handelt. Das Objekt mit Namen `eineKatze` ist dagegen kein anonymes Objekt, sondern modelliert genau diejenige Realwelt-Katze, die den Namen `eineKatze` trägt.

Abbildung 3.6: vier benannte und zwei anonyme Objekte der Klasse Katze

Das Konzept der Klasse ist ein zentraler Bestandteil der objektorientierten Softwareentwicklung – auch wenn es einige wenige objektorientierte Programmiersprachen gibt (z.B. JavaScript), die keine Klassen, sondern ausschließlich Objekte kennen. Sie kennen aus der objektorientierten Programmierung sicher die Definition einer Klasse als Bauplan bzw. Schablone für gleichartige Software-Objekte. Dabei wird die Klasse aus dem Blickwinkel der Programmcodeerstellung betrachtet. Aber was ist eigentlich eine Klasse, wenn wir mit dem Fokus der Realweltmodellierung hinsehen?

Abbildung 3.7 greift das Beispiel mit Herrn Müller aus Abschnitt 3.2.1 (S. **??**) wieder auf. Aus dem Realwelt-Objekt Herr Müller wird durch entsprechende (unterschiedliche) Abstraktion das modellierte Realwelt-Objekt Lehrer Müller oder das modellierte Realwelt-Objekt Fußballer Müller. Eine Klasse beschreibt genau diese Abstraktion zwischen dem Realwelt-Objekt und der Modellierung des Realwelt-Objekts. Die Klasse Lehrer definiert, welche Merkmale des Realwelt-Objekts für die Modellierung als Lehrer Müller relevant sind. Die Klasse Fußballer beschreibt diejenigen Merkmale des Realwelt-Objekts, die für die Modellierung als Fußballer Müller relevant sind.

Eine Klasse ist die Beschreibung einer Abstraktion

Zwei Aspekte zum Konzept der Klasse müssen wir an dieser Stelle noch ergänzen. Erstens gilt die in den Klassen Lehrer und Fußballer beschriebene Abstraktion natürlich nicht nur für das konkrete Realwelt-Objekt Herrn Müller, sondern für alle Realwelt-Objekte (Frau Schulze, Herr Özdemir, Frau Kinsombi, . . .), die modellierte Realwelt-Lehrer oder Realwelt-Fußballer werden sollen. Und zweitens beschreibt eine Klasse die Abstraktion zwischen Realwelt-Objekt und Modellierung eines solchen Realwelt-Objekts auch dann,

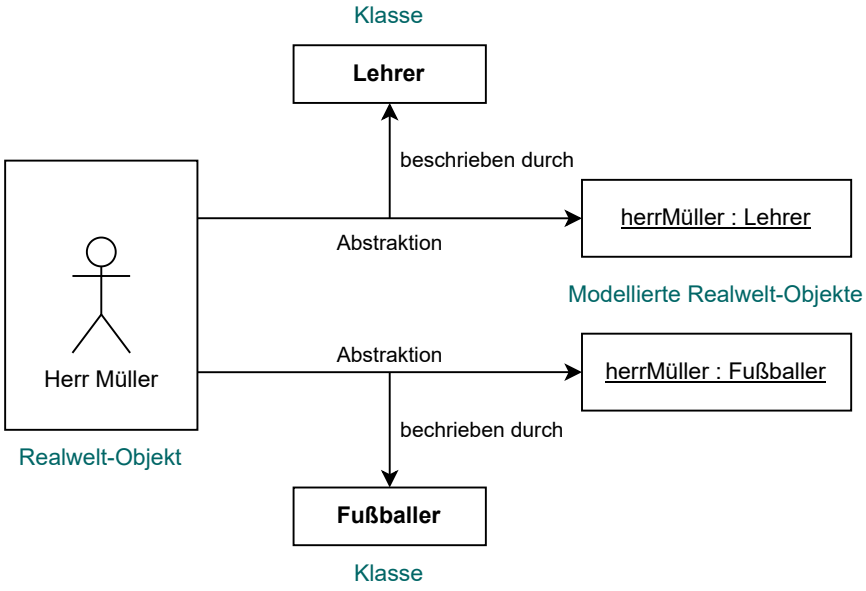


Abbildung 3.7: Der Klassenbegriff aus Sicht der Realweltmodellierung

wenn man (noch) gar kein modelliertes Objekt hat. So kann zum Beispiel eine Klasse Katze existieren, die definiert, wie man von einer Realwelt-Katze zu einer modellierten Realwelt-Katze kommen würde, ohne dass es ein modelliertes Katzen-Objekt gibt. Eine Klasse ist somit unabhängig von der Existenz der modellierten Objekte – umgekehrt gilt dies jedoch nicht.

Die UML-Darstellung einer Klasse ist sehr ähnlich zu der Darstellung eines Objekts. Es handelt sich ebenfalls um ein Rechteck, in dem (mindestens) der Klassenname eingetragen ist.

Abbildung 3.8: Eine Klasse mit Namen `Katze` (links) und ein Objekt dieser Klasse mit Namen `pünktchen` (rechts)

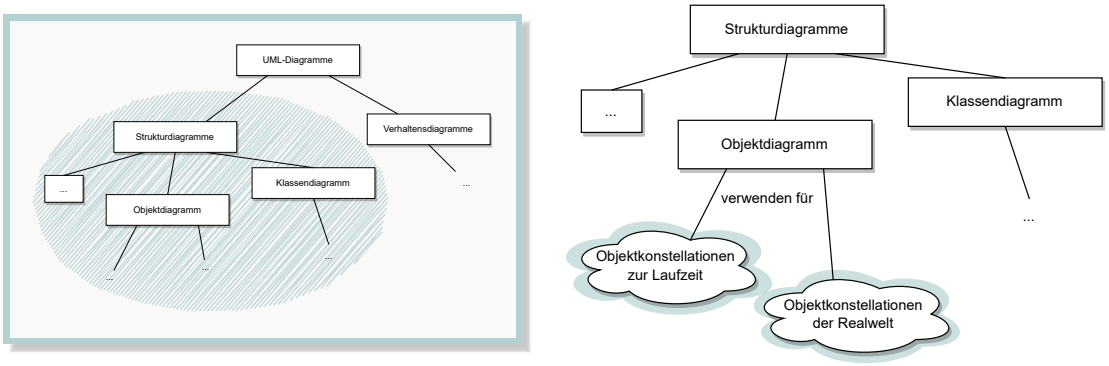
Abbildung 3.8 zeigt links eine Klasse `Katze` und rechts ein modelliertes Katzen-Objekt namens `pünktchen`. Im Unterschied zu Objekten wird bei der Darstellung einer Klasse der Name nicht unterstrichen. Zudem ist es üblich, den Klassennamen mit einem Großbuchstaben zu beginnen und ihn zentriert und fett gedruckt zu setzen. Der Klassenname ist üblicherweise ein Substantiv im Singular und nicht im Plural.

Wenn man Softwareprodukte, wie zum Beispiel die erwähnte Schulverwaltungssoftware

entwickeln möchte, sind die konkreten Objekte der Realwelt wie Herr Müller meistens weniger interessant. Entscheidender ist, dass die Software später mit beliebigen Objekten eines bestimmten Typs (z.B. Lehrer-Objekt) umgehen kann. Die Informationen zu den Merkmalen von Objekttypen finden sich in der objektorientierten Softwareentwicklung aber in den Klassen und nicht in den Objekten. Daher interessieren für die Softwareentwicklung vor allem die Klassen.

Einsatzgebiete von Klassen- und Objektdiagrammen

Für die Modellierung von Klassen stellt die UML ein eigenes Diagramm, das *Klassendiagramm*, zur Verfügung. Es gehört wie das Objektdiagramm zu den Strukturdiagrammen der UML und stellt Klassen und ihre Beziehungen zueinander dar. Das Klassendiagramm ist für das Softwareengineering eine der wichtigsten Diagrammarten der UML, da es in fast allen Prozessen des Softwareengineering eingesetzt werden kann. Objektdiagramme dagegen werden in der Praxis seltener eingesetzt. Man kann sie verwenden, um bestimmte Situationen zur Laufzeit des Softwareprodukts zu veranschaulichen (welche Software-Objekte existieren zu einem bestimmten Zeitpunkt und wie stehen sie miteinander in Verbindung). Für die Lehre eignen sich Objektdiagramme zudem ganz gut, da sie für Anfänger im Bereich der Objektorientierung die Kluft zwischen den Objekten der Realwelt und den Klassen der objektorientierten Programmierung überbrücken helfen.



Es gibt in der Praxis Situationen, in denen es sinnvoll sein kann, im Objektdiagramm auch Klassen oder im Klassendiagramm auch Objekte aufzuführen. Für diese Modellierung bietet die UML für beide Diagramme eine spezielle textuelle Ergänzung (`«instantiate»`) zur grafischen Darstellung an. Im Fallbeispiel in Kapitel 3.3 wird Ihnen eine solche Konstruktion begegnen. [...]

3.3 Fallbeispiel Zoo – Ein erstes Domänenklassendiagramm

[...]

Literatur

[boo18] Grady Booch. „The History of Software Engineering“. In: *IEEE Software* 35.5 (2018), S. 108–114. DOI: 10.1109/MS.2018.3571234.

[bux70] J. N. Buxton und B. Randell, Hrsg. *Software Engineering Techniques: Report on a Conference Sponsored by the NATO Science Committee, Rome, Italy, 27th to 31st October 1969*. Brüssel, 1970.

[del14] Isabel M. del Águila, José Palma und Samuel Túnez. „Milestones in Software Engineering and Knowledge Engineering History: A Comparative Review“. In: *The Scientific World Journal* (2014), Article ID 692510. DOI: 10.1155/2014/692510.

[dia14] Jorge Díaz-Herrera und Peter A. Freeman. „Discipline of Software Engineering: An Overview“. In: *Computing Handbook*. Hrsg. von Teofilo Gonzalez, Jorge Díaz-Herrera und Allen Tucker. Boca Raton: CRC Press, 2014, 72-1 bis 72–20.

[dum03] Reiner Dumke. *Software Engineering: Eine Einführung für Informatiker und Ingenieure; Systeme, Erfahrungen, Methoden, Tools*. 4., überarb. und erw. Aufl. Wiesbaden: Vieweg, 2003.

[end97] Albert Endres. „A Synopsis of Software Engineering History: The Industrial Perspective“. In: *History of Software Engineering*. Hrsg. von Andreas Brennecke und Reinhard Keil-Slawik. Dagstuhl-Seminar-Report. Saarbrücken: Geschäftsstelle Schloss Dagstuhl, 1997, S. 20–24.

[gon14] Teofilo Gonzalez, Jorge Díaz-Herrera und Allen Tucker, Hrsg. *Computing Handbook: Computer Science and Software Engineering*. 3. Auflage. Boca Raton: CRC Press, 2014.

[gri11] David Alan Grier. „Software Engineering: History“. In: *Encyclopedia of Software Engineering*. Hrsg. von Phillip A. Laplante. Bd. 2. Boca Raton: CRC Press, 2011, S. 1119–1126.

[iso17] ISO/IEC/IEEE. *Systems and software engineering - Vocabulary*. Vernier, Genf, 2017. URL: https://www.iso.org/standard/71952.html.

[lap11] Phillip A. Laplante, Hrsg. *Encyclopedia of Software Engineering*. Boca Raton: CRC Press, 2011.

[mah04] Michael S. Mahoney. „Finding a History for Software Engineering“. In: *IEEE Annals of the History of Computing* 26.1 (2004), S. 8–19. DOI: 10.1109/MAHC.2004.1278847.

[mcc11] Margo McCall. „IEEE Computer Society“. In: *Encyclopedia of Software Engineering*. Hrsg. von Phillip A. Laplante. Bd. 1. Boca Raton: CRC Press, 2011, S. 404–413.

[mue15] Holger Mühlbauer. *Kurze Einführung in die Normung: Das Wesentliche zu DIN, CEN und ISO*. 2. Auflage. Beuth kompakt. Berlin, Wien und Zürich: Beuth, 2015.

[nau69] Peter Naur und Brian Randell, Hrsg. *Software Engineering: Report on a conference sponsored by the Nato Science Committee, Garmisch, Germany, 7th to 11th October 1968*. Brüssel, 1969.

[sei14] Stephan B. Seidman. „Professionalism and Certification“. In: *Computing Handbook*. Hrsg. von Teofilo Gonzalez, Jorge Díaz-Herrera und Allen Tucker. Boca Raton: CRC Press, 2014, 73-1 bis 73–10.

[sha11] Linda Shafer. „Process“. In: *Encyclopedia of Software Engineering*. Hrsg. von Phillip A. Laplante. Bd. 2. Boca Raton: CRC Press, 2011, S. 684–703.

[som18] Ian Sommerville. *Software Engineering*. 10., akt. Aufl. Hallbergmoos: Pearson, 2018.

[swe14] Pierre Bourque und Richard E. Fairley, Hrsg. *SWEBOK V3.0: Guide to the Software Engineering Body of Knowledge*. 2014. URL: https://www.computer.org/web/swebok.

[tan14] Andrew S. Tanenbaum und Todd Austin. *Rechnerarchitektur: Von der digitalen Logik zum Parallelrechner*. 6., aktualisierte Aufl. Always Learning. Hallbergmoos: Pearson, 2014.

[wan00] Yingxu Wang und Dilip Patel. „Editors' introduction: Comparative software engineering: Review and perspectives“. In: *Annals of Software Engineering* 10.1 (2000), S. 1–10. DOI: 10.1023/A:1018931531464.

[wir08] Niklaus Wirth. „A Brief History of Software Engineering“. In: *IEEE Annals of the History of Computing* 30.3 (2008), S. 32–39. DOI: 10.1109/MAHC.2008.33.