Media Engineering

Skript zur Vorlesung

Kapitel 1 bis 3. Die weiteren Kapitel erscheinen jeweils nach der Vorlesung.

René Weller

Arbeitsgruppe Computergraphik und Virtuelle Realität

Universität Bremen

Wintersemester 2026/27

Inhaltsverzeichnis

Zu diesem Skript **4**

1 Motivation **6**
1.1 Wenn Software scheitert . . . 6
1.1.1 Fehler im Code . . . 6
1.1.2 Fehler in der Benutzeroberfläche . . . 7
1.1.3 Projekte, die ohne Programmfehler scheitern . . . 8
1.2 Was bedeutet Scheitern? . . . 9
1.3 Ein Problem nur der IT? . . . 10
1.4 Was ist Software, und was macht sie besonders? . . . 12
1.4.1 Arten von Software . . . 12
1.4.2 Besondere Eigenschaften von Software . . . 12
1.5 Digitale Güter und digitale Märkte . . . 13
1.6 Warum Projekte scheitern . . . 14
1.7 Softwarekrise und Software Engineering . . . 15
1.7.1 Die Softwarekrise . . . 15
1.7.2 Die Antwort: Software Engineering . . . 15
1.7.3 Software Engineering ist mehr als Programmieren . . . 17
1.7.4 Meilensteine . . . 17
1.8 Media Engineering . . . 19
1.9 Der Software Development Lifecycle . . . 19
Zusammenfassung . . . 21
Kontrollfragen . . . 21
Aufgaben . . . 22
Lernziele überprüfen . . . 23
Zum Weiterlesen . . . 23

2 Game Engines **24**
2.1 Der Medienmarkt in Deutschland . . . 24
2.1.1 Wie viel Zeit, wie viel Geld? . . . 24
2.1.2 Der Spielemarkt . . . 25
2.1.3 Game Engines außerhalb von Spielen . . . 26
2.2 Von der Platine zur Engine . . . 26
2.3 Was ist eine Game Engine? . . . 27
2.4 Ein Frame Pac-Man: die Module einer Engine . . . 27
2.4.1 Eingabe . . . 28
2.4.2 Spiellogik und künstliche Intelligenz . . . 29
2.4.3 Physik . . . 29
2.4.4 Animation . . . 29
2.4.5 Sound . . . 30
2.4.6 Rendering . . . 30
2.4.7 Was nicht in der Schleife liegt . . . 31
2.5 Architektur und Größe . . . 31

2.6 Welche Engine? . . . 32
2.6.1 Kaufen oder selbst bauen? . . . 32
2.6.2 Unreal, Unity und Godot im Vergleich . . . 34
Zusammenfassung . . . 35
Kontrollfragen . . . 35
Aufgaben . . . 36
Rückblick . . . 36
Lernziele überprüfen . . . 37
Zum Weiterlesen . . . 37

3 Requirements Engineering 38
3.1 Warum braucht man Anforderungen? . . . 39
3.2 Was sind Anforderungen? . . . 40
3.2.1 Von wem stammen Anforderungen? . . . 41
3.2.2 Arten von Anforderungen . . . 41
3.2.3 Nichtfunktionale Anforderungen und Normen . . . 42
3.3 Wie findet man Anforderungen? . . . 43
3.3.1 Fragen . . . 43
3.3.2 Ausprobieren . . . 44
3.3.3 Szenarien . . . 45
3.3.4 Schwierigkeiten . . . 45
3.4 Wie hält man Anforderungen fest? . . . 46
3.4.1 Lastenheft und Pflichtenheft . . . 46
3.4.2 Game Design Document und User Stories . . . 47
3.4.3 Anforderungen formulieren . . . 48
3.4.4 Priorisieren . . . 49
3.4.5 Das Begriffslexikon . . . 50
3.4.6 Wozu der Aufwand? . . . 50
Zusammenfassung . . . 51
Kontrollfragen . . . 52
Aufgaben . . . 52
Rückblick . . . 53
Lernziele überprüfen . . . 53
Zum Weiterlesen . . . 54

A Formelsammlung 55

B UML-Notation auf einen Blick 57

Literatur 59

Stichwortverzeichnis 66

Zu diesem Skript

Dieses Skript begleitet die Vorlesung *Media Engineering*. Die Folien der Vorlesung enthalten bewusst wenig Text; hier steht das Gesagte zum Nachlesen. Mehrere Arten von Kästen kehren in allen Kapiteln wieder:

Definition 0.1: So sehen Definitionen aus

Begriffe, die Sie kennen und erklären können sollten, stehen in einem roten Kasten und sind je Kapitel durchnummeriert.

Merke

Graue Kästen fassen das Wichtigste eines Abschnitts in ein bis zwei Sätzen zusammen.

Beispiel: Fallbeispiele

Beispiele aus der Praxis stehen in einem Kasten mit grauer Kopfzeile.

Blaue Randmarkierungen verweisen auf spätere Kapitel, in denen ein Thema vertieft wird.

Jedes Kapitel beginnt mit seinen *Lernzielen*, an denen Sie nach dem Lesen prüfen können, ob Sie das Wesentliche mitgenommen haben.

Mythos und Befund: Was alle wissen

In diesen Kästen geht es um Behauptungen, die man in Vorträgen, Büchern und Kaffeeküchen oft hört und die bei genauerem Hinsehen falsch oder stark verkürzt sind. Wir stellen sie dem gegenüber, was die Forschung tatsächlich gefunden hat.

Im Projektchen

Hier steht, was ein Kapitel für das Projekt der Übungen bedeutet: konkrete Hinweise, die Ihnen im eigenen kleinen Spiel Zeit und Ärger ersparen sollen.

Zwei kleinere Elemente unterbrechen den Text gelegentlich.

Bevor Sie weiterlesen: Vor manchen Ergebnissen steht eine kurze Frage. Legen Sie sich auf eine Antwort fest, bevor Sie weiterlesen, am besten auf eine Zahl. Das kostet zehn Sekunden, und wer sich vorher festgelegt hat, behält das Ergebnis besser (Sana et al. 2020). Ob Sie richtig liegen, ist dabei nicht wichtig.

Am Rande So gesetzte Absätze erzählen, woher ein Begriff stammt oder was sonst noch dazu zu sagen wäre. Für das Verständnis des Kapitels und für die Klausur brauchen Sie sie nicht; beim ersten Lesen können Sie sie überspringen.

An einigen Stellen wird außerdem ein Verfahren unter der Überschrift *Schritt für Schritt* vollständig vorgeführt, etwa das Berechnen eines Netzplans oder der Weg von einem Szenario zum Sequenzdiagramm. Decken Sie dort nach den ersten Schritten den Rest ab und versuchen Sie, selbst weiterzukommen.

Am Ende jedes Kapitels finden Sie unter *Zum Weiterlesen* zwei oder drei Bücher und Texte für den Einstieg. Viele Abbildungen stammen aus den Folien der Vorlesung; sie haben einen dünnen grauen Rahmen, damit Sie sie beim Nacharbeiten leicht wiedererkennen. Im Anhang stehen eine Formelsammlung und eine Übersicht über die UML-Notation, am Schluss folgen das Literaturverzeichnis mit allen im Text genannten Quellen und ein Stichwortverzeichnis.

Am Ende jedes Kapitels stehen Kontrollfragen und Aufgaben. Die *Kontrollfragen* prüfen, ob Sie die Begriffe und Zusammenhänge des Kapitels verstanden haben. Die *Aufgaben* gehen einen Schritt weiter: Sie wenden das Gelernte auf einen neuen Fall an, rechnen, zeichnen, suchen Fehler, wägen ab oder übertragen es auf Ihr eigenes Spiel. Abgeben müssen Sie davon nichts; die praktische Arbeit findet im Projekt der Übungen statt. Beantworten Sie die Fragen zuerst aus dem Gedächtnis und schlagen Sie erst danach nach: Sich selbst abzufragen hilft dem Behalten deutlich mehr als wiederholtes Lesen (Roediger & Karpicke 2006).

Ab dem zweiten Kapitel folgt darauf ein *Rückblick* mit drei Fragen zu früheren Kapiteln. Das hat zwei Gründe. Was man nach ein paar Wochen noch einmal aus dem Gedächtnis holen muss, sitzt fester als das, was man nur in der Woche gelernt hat, in der es dran war. Und in der Klausur steht über keiner Aufgabe, zu welchem Kapitel sie gehört. Den Schluss jedes Kapitels bildet die Tabelle *Lernziele überprüfen*. Sie nennt zu jedem Lernziel vom Kapitelanfang die Fragen und Aufgaben, an denen Sie feststellen können, ob Sie es erreicht haben.

In der Klausur dürfen Sie ein DIN-A4-Blatt verwenden, das Sie selbst beidseitig von Hand beschrieben haben. Definitionen, Aufzählungen und Formeln müssen Sie also nicht auswendig lernen; die Kontrollfragen und die Formelsammlung im Anhang sind ein guter Ausgangspunkt für Ihr Blatt, und schon das Zusammenstellen ist eine gute Wiederholung. Die meisten Klausuraufgaben verlangen aber wie die Aufgaben in diesem Skript, das Gelernte auf einen *neuen* Fall anzuwenden. Dabei hilft kein Blatt, sondern nur Übung.

1 Motivation: Shit happens . . . aber nicht notwendigerweise

Warum sollte man sich mit Software Engineering beschäftigen, wenn man eigentlich Spiele, Apps oder interaktive Installationen bauen will? Die kurze Antwort lautet, dass erstaunlich viele Software- und Medienprojekte scheitern, und zwar meist nicht an der Technik. In diesem Kapitel sehen wir uns zunächst einige Beispiele an, bei denen etwas gründlich schiefgegangen ist. Danach klären wir, was „Scheitern“ eigentlich heißt, warum gerade Software so anfällig dafür ist und welche Antwort die Informatik darauf gefunden hat. Am Ende steht der Fahrplan für den Rest der Vorlesung.

Lernziele

Nach diesem Kapitel können Sie

- an Beispielen erklären, auf welche verschiedenen Arten Software- und Medienprojekte scheitern können,
- Projekterfolg und Produkterfolg unterscheiden,
- die Besonderheiten von Software und digitalen Gütern nennen, die Projekte schwer planbar machen,
- den Begriff Software Engineering definieren und erläutern, warum er mehr umfasst als Programmieren,
- die Phasen des Software Development Lifecycle benennen.

1.1 Wenn Software scheitert

Dem amerikanischen Ingenieur Edward A. Murphy wird der Satz zugeschrieben: „Anything that can go wrong, will go wrong.“ Gemeint war das nicht als Klage, sondern als Arbeitsauftrag: Wer etwas konstruiert, soll vorher durchdenken, auf welche Arten es schiefgehen kann, vor allem auf welche katastrophalen. Die folgenden Beispiele zeigen, was passiert, wenn das unterbleibt. Jedes der vier steht für eine eigene Art von Fehler, und jede dieser Arten greifen wir in einem späteren Kapitel wieder auf.

1.1.1 Fehler im Code

Beispiel: CrowdStrike (2024)

Am 19. Juli 2024 verteilte die Sicherheitssoftware *Falcon* der Firma CrowdStrike automatisch ein Konfigurations-Update. Eine Vorlage im Programm sah 21 Eingabefelder vor, der zugehörige Code lieferte aber nur 20 Werte. Das neue Update griff erstmals auf das 21. Feld zu. Das Programm las damit hinter dem Ende eines Arrays, und Windows stürzte ab, auch nach jedem Neustart. Betroffen waren rund 8,5 Millionen Rechner in Flughäfen, Krankenhäusern, Banken und Fernsehsendern. Kein Test hatte die Abweichung zwischen

21 und 20 geprüft, und das Update ging an alle Kunden gleichzeitig.

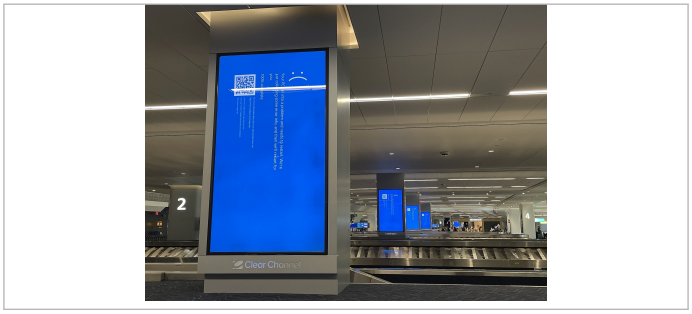


Foto: Wikimedia Commons („CrowdStrike BSOD at LGA“), Flughafen LaGuardia, 19. Juli 2024

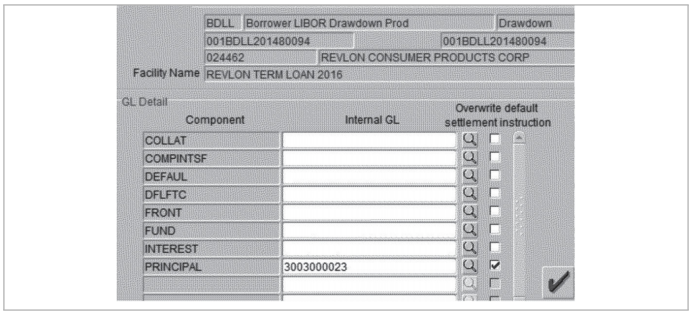
Neu ist diese Art von Fehler nicht. Am 4. Juni 1996 zerstörte sich die europäische Trägerrakete Ariane 5 rund 37 Sekunden nach dem Start selbst. Eine 64-Bit-Gleitkommazahl wurde in eine 16-Bit-Ganzzahl umgewandelt, für die sie zu groß war. Der Programmteil stammte unverändert aus dem Vorgängermodell, das langsamer flog und bei dem der Wert immer gepasst hatte. Obwohl zwischen den beiden Fällen 28 Jahre liegen, haben sie denselben Kern. In beiden Fällen liegt ein Wert außerhalb des Bereichs, mit dem die Entwickler gerechnet haben, und niemand fängt das ab.

> Wie man solche Fehler systematisch sucht, etwa mit Grenzwert- und Schnittstellentests, ist Thema des Kapitels *Testen*.

1.1.2 Fehler in der Benutzeroberfläche

Beispiel: Citibank (2020)

Im August 2020 wollte die Citibank im Auftrag des Kosmetikkonzerns Revlon 7,8 Millionen Dollar Zinsen an dessen Kreditgeber überweisen. Tatsächlich überwies sie rund 900 Millionen Dollar, nämlich den gesamten Kredit, und zwar aus eigenem Geld. In der verwendeten Banksoftware hätten für die geplante Zahlung drei Kästchen angekreuzt sein müssen. Angekreuzt war nur eines. Drei Personen hatten die Zahlung geprüft und freigegeben, keiner fiel der Fehler auf. Rund 500 Millionen Dollar bekam die Bank zunächst nicht zurück; erst ein Berufungsgericht entschied 2022 zu ihren Gunsten.



Originalmaske aus dem Urteil *In re Citibank August 11, 2020 Wire Transfers*, S.D.N.Y. 2021

Man könnte den drei Prüfern einfach Unaufmerksamkeit vorwerfen. Damit macht man es sich aber zu leicht, denn die Oberfläche hat den Fehler nahegelegt und nichts getan, um ihn zu verhindern. Wenn drei Menschen nacheinander denselben Fehler übersehen, liegt das Problem vermutlich nicht nur bei den Menschen.

Wie man Oberflächen so gestaltet, dass sie Fehler unwahrscheinlich machen, behandelt das Kapitel *GUI-Design*.

1.1.3 Projekte, die ohne Programmfehler scheitern

Bis hierher ging es um Software, die nicht tat, was sie sollte. Projekte können aber auch scheitern, wenn die Software tadellos läuft.

Beispiel: Concord (2024): das falsche Produkt

Das Studio Firewalk entwickelte für Sony rund acht Jahre lang den Online-Shooter *Concord*. Am 23. August 2024 erschien das Spiel, am 6. September 2024 wurden die Server abgeschaltet. Es war 14 Tage auf dem Markt. Technisch funktionierte es; es wollte nur kaum jemand spielen. Die Käufer erhielten ihr Geld zurück, das Studio wurde geschlossen.



Bild: Sony Interactive Entertainment / Firewalk Studios

Beispiel: Star Citizen (seit 2012): nie fertig

Das Weltraumspiel *Star Citizen* sammelte ab Oktober 2012 über Kickstarter Geld ein und sollte 2014 erscheinen. Im Mai 2026 überschritt die Summe der Unterstützerbeiträge eine Milliarde Dollar. Ein fertiges Spiel gibt es weiterhin nicht. Mit dem Geld wuchsen die Versprechen, und mit den Versprechen wuchs der Umfang. Dieses Muster heißt **Feature Creep**: Ein Projekt bekommt immer neue Funktionen und wird deshalb nie fertig.



Bild: Cloud Imperium Games (Pressematerial)

Die beiden Fälle sind in gewisser Weise Gegenstücke. Concord wurde fertig, aber kaum jemand wollte das Ergebnis haben. Star Citizen möchten sehr viele Menschen haben, aber es wird nicht fertig.

Was Nutzer wirklich wollen, klärt das *Requirements Engineering*. Wie man Umfang, Zeit und Kosten plant und steuert, behandeln die Kapitel *Prozessmodelle* und *Projektmanagement*.

Tabelle 1.1 fasst die Beispiele zusammen.

Beispiel	Wofür es steht	Kommt wieder in
CrowdStrike (und Ariane 5)	Programmierfehler; ungeprüfte Werte und ungeprüfte Wiederverwendung	Testen
Citibank	Oberfläche, die Fehler nahelegt	GUI-Design
Concord	Am Nutzer vorbei entwickelt	Requirements Engineering
Star Citizen	Umfang und Zeit außer Kontrolle	Projektmanagement, Prozessmodelle

Tabelle 1.1: Die Beispiele dieses Kapitels und die Fehlerarten, für die sie stehen.

1.2 Was bedeutet Scheitern?

Die klassische Antwort betrachtet drei Größen: was geliefert wird, was es kostet und wie lange es dauert. Diese drei Größen hängen voneinander ab und werden deshalb gern als Dreieck gezeichnet (Abbildung 1.1).

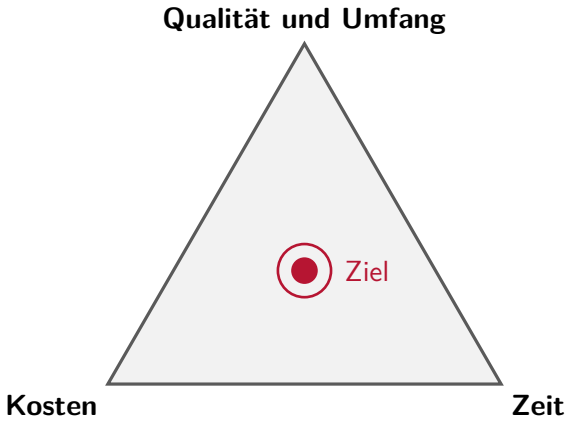


Abbildung 1.1: Das magische Dreieck. Wer an einer Ecke zieht, verschiebt die anderen beiden.

Definition 1.1: Magisches Dreieck

Das magische Dreieck beschreibt den Zusammenhang zwischen **Qualität und Umfang**, **Kosten** und **Zeit** eines Projekts. Keine der drei Größen lässt sich ändern, ohne mindestens eine der anderen zu beeinflussen.

Daraus ergibt sich die klassische Definition des Scheiterns.

Definition 1.2: Projekterfolg

Ein Projekt ist im Sinne des Projektmanagements erfolgreich, wenn es das versprochene Ergebnis in der geplanten Zeit und zu den geplanten Kosten liefert. Es gilt als gescheitert, wenn es nicht wie versprochen liefert, mehr kostet als veranschlagt oder länger dauert als geplant.

Diese Definition reicht für Medienprodukte nicht aus. Concord hat ein funktionierendes Spiel geliefert und ist trotzdem gescheitert. Umgekehrt gibt es Produkte, die viel zu spät und viel zu teuer fertig wurden und dennoch ein großer Erfolg waren. Wir brauchen deshalb einen zweiten Begriff.

Definition 1.3: Produkterfolg

Ein Produkt ist erfolgreich, wenn es für seine Nutzer den erhofften Wert hat: wenn es gebraucht, gern benutzt und im kommerziellen Fall auch gekauft wird.

Merke

Projekterfolg heißt, das Produkt *richtig* zu bauen. Produkterfolg heißt, das *richtige* Produkt zu bauen. Bei Unterhaltungsmedien entscheidet am Ende der Produkterfolg.

1.3 Ein Problem nur der IT?

Die Beispiele aus Abschnitt 1.1 sind keine Einzelfälle. Die Liste gescheiterter IT-Großprojekte ist lang: Der Lebensmittelhändler Lidl stoppte 2018 nach sieben Jahren und rund 500 Millionen Euro die Einführung eines neuen Warenwirtschaftssystems. Die britische BBC stellte 2013 ihre digitale Produktionsplattform ein, nachdem sie 98 Millionen Pfund gekostet hatte. Die Stadt Birmingham plante für eine neue Verwaltungssoftware 19 Millionen Pfund ein und hat inzwischen mehr als 140 Millionen ausgegeben.

Bevor Sie weiterlesen: Auch Flughäfen, Bahnhöfe und Konzerthäuser werden teurer als geplant. Was vermuten Sie: Überziehen IT-Großprojekte ihr Budget im Mittel eher wie Straßen und Brücken oder eher wie Olympische Spiele?

Nun scheitern auch Bauprojekte, wie der Flughafen Berlin Brandenburg, der Bahnhof Stuttgart 21 oder die Elbphilharmonie zeigen. Der Vergleich lässt sich mit Zahlen unterlegen. Der Planungsforscher Bent Flyvbjerg hat Daten von über 16 000 Großprojekten gesammelt und ausgewertet, um wie viel die tatsächlichen Kosten im Mittel über den geplanten lagen (Abbildung 1.2).

IT-Projekte liegen mit durchschnittlich 73 Prozent Kostenüberschreitung weit oben, deutlich schlechter als Tunnel, Brücken oder Straßen. Hinzu kommt eine zweite Beobachtung: Bei IT-Projekten gibt es besonders extreme Ausreißer. Die meisten überziehen mäßig, einige wenige aber um ein Vielfaches. Am besten schneiden Projekte ab, die aus vielen gleichen Teilen bestehen, etwa Solaranlagen und Windparks. Wer hundertmal dasselbe baut, lernt dabei und kann beim nächsten Mal genauer planen. Software ist dagegen fast immer ein Einzelstück.

Merke

Scheitern ist kein reines IT-Problem. IT-Projekte gehören aber zu den Projektarten mit den höchsten Kostenüberschreitungen und den extremsten Ausreißern.

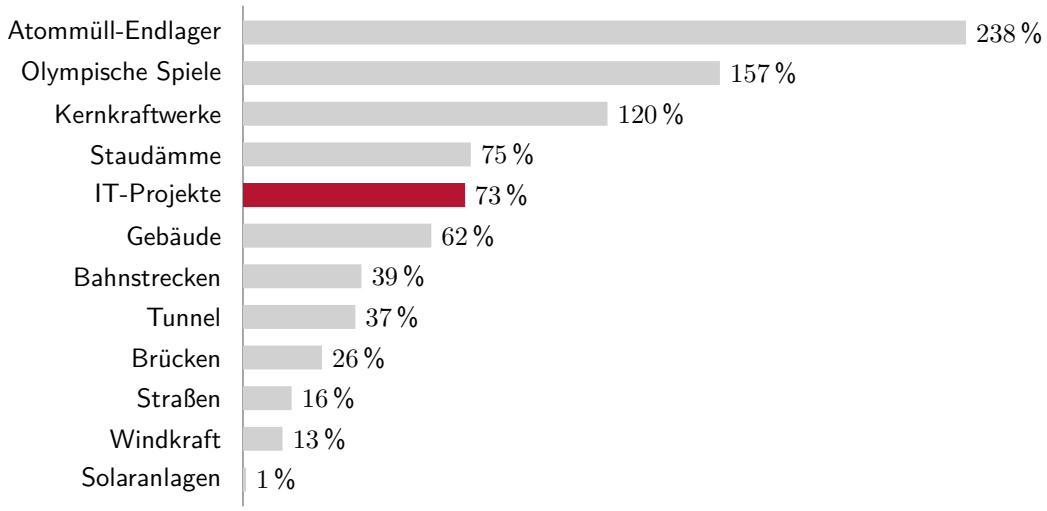


Abbildung 1.2: Mittlere Kostenüberschreitung gegenüber dem Plan nach Projektart (Auswahl; Daten: Flyvbjerg und Gardner 2023).

Bevor Sie weiterlesen: Studierende, die gerade an ihrer Abschlussarbeit schrieben, sollten angeben, wann sie realistisch fertig sein würden und wann im schlimmsten Fall. Wo lag die tatsächliche Dauer: unter der realistischen Schätzung, zwischen den beiden Angaben oder über dem schlimmsten Fall?

Warum verschätzen sich selbst erfahrene Planer so gründlich? Ein Teil der Antwort liegt in uns selbst. Roger Buehler, Dale Griffin und Michael Ross baten 1994 Studierende, die gerade an ihrer Abschlussarbeit schrieben, um drei Schätzungen: wann sie realistisch fertig sein würden, wann im besten Fall und wann im schlimmsten Fall. Im Mittel rechneten sie mit 33,9 Tagen, im besten Fall mit 27,4 und im schlimmsten Fall mit 48,6 Tagen. Tatsächlich brauchten sie 55,5 Tage, also länger als in ihrem eigenen schlimmsten Fall (Abbildung 1.3). Wer schon einmal eine Hausarbeit „locker bis Freitag“ fertig haben wollte, wird das Ergebnis nicht ganz überraschend finden.

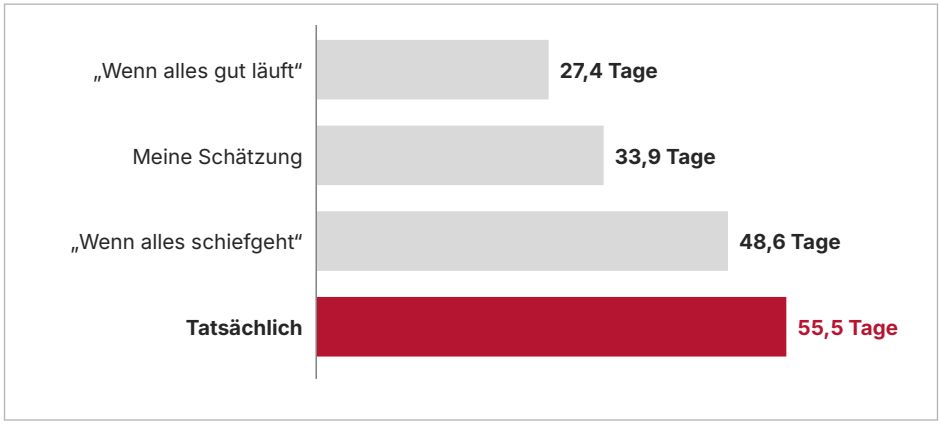


Abbildung 1.3: Geschätzte und tatsächliche Dauer von Abschlussarbeiten (Daten: Buehler, Griffin und Ross 1994).

Definition 1.4: Planungsfehlschluss

Der **Planungsfehlschluss** (planning fallacy) bezeichnet die Neigung, die Dauer und die Kosten eigener Vorhaben systematisch zu unterschätzen, selbst wenn man weiß, dass ähnliche Vorhaben früher länger gedauert haben.

1.4 Was ist Software, und was macht sie besonders?

Definition 1.5: Software

Software besteht aus **Programmen**, den zugehörigen **Daten** und der **Dokumentation**.

Die Definition ist breiter, als man zunächst denkt. Zum Programm gehört der ausführbare Code. Zu den Daten gehören Konfigurationen und Inhalte. Bei einem Spiel sind das 3D-Modelle, Texturen, Klänge und Level, und sie machen den größten Teil des Produkts aus. Die Dokumentation richtet sich an Benutzer und an Entwickler. Wer bei Software nur an Code denkt, übersieht den Großteil der Arbeit.

1.4.1 Arten von Software

Software ist nicht gleich Software. Üblich ist folgende grobe Einteilung:

- **Eigenständige Programme**: Büroanwendungen, Bild- und Videobearbeitung, Computerspiele.
- **Interaktive, transaktionsbasierte Systeme**: Webshops, Online-Banking.
- **Eingebettete Systeme**: Software in Autos, Smartphones, Haushaltsgeräten und Industrierobotern.
- **Stapelverarbeitung**: Systeme, die große Datenmengen ohne Zutun eines Benutzers abarbeiten, etwa Lohn- oder Telefonabrechnungen.
- **Unterhaltungssysteme**: vor allem Spiele; hier ist die Interaktion mit dem Benutzer besonders wichtig.
- **Datenerfassungssysteme**: Systeme, die Messwerte von Sensoren sammeln und weiterverarbeiten.

Jede dieser Arten stellt andere Anforderungen. Die eine richtige Methode für alle gibt es deshalb nicht.

1.4.2 Besondere Eigenschaften von Software

Trotz dieser Vielfalt hat Software einige Eigenschaften, die sie von fast allen anderen Produkten unterscheiden und die erklären, warum sie so schwer zu planen ist.

- Software ist **immateriell**. Bei einem Haus sieht jeder, wie weit der Bau ist. Den Fortschritt eines Programms sieht man nicht.

- Software unterliegt **keinen physikalischen Grenzen**. Alles scheint machbar, und entsprechend viel wird versprochen.
- Software ist **schwer zu vermessen**. Wie groß ein Programm ist, wie fertig und wie gut, lässt sich nur mühsam in Zahlen fassen.
- Software ist **leicht zu ändern**. Das verführt dazu, auch spät im Projekt noch grundlegende Änderungen zu verlangen.
- Software **nutzt sich nicht ab, altert aber trotzdem**. Die Umgebung ändert sich: neue Betriebssysteme, neue Geräte, neue Erwartungen.
- Einzelne Teile lassen sich nicht austauschen, ohne das Ganze zu verstehen. Ein „Ersatzteil“ wie bei einer Maschine gibt es nicht.
- Fehler können **sehr lange unentdeckt** bleiben, manchmal über Jahre.

1.5 Digitale Güter und digitale Märkte

Software teilt eine wirtschaftliche Besonderheit mit allen digitalen Medien.

Definition 1.6: Digitales Gut

Ein digitales Gut ist ein Produkt, das vollständig in digitaler Form vorliegt und über Netze ausgeliefert werden kann, zum Beispiel Musik, Filme, Spiele, E-Books und Software.

Bei einem digitalen Gut verursacht die erste Kopie praktisch die gesamten Kosten. Jede weitere Kopie kostet fast nichts, und auch die Auslieferung über das Netz ist nahezu kostenlos. Lager und Transport entfallen. Daraus folgt, dass der Preis kaum noch an die Herstellungskosten des einzelnen Stücks gebunden ist. Abonnements, Streaming und kostenlose Spiele mit Käufen im Spiel sind nur deshalb möglich.

Wenn Kopien nichts kosten, funktionieren auch die Märkte anders als bei herkömmlichen Gütern. Vier Begriffe beschreiben die wichtigsten Effekte.

Definition 1.7: Netzwerkeffekt

Ein Netzwerkeffekt liegt vor, wenn der Nutzen eines Produkts für den Einzelnen davon abhängt, wie viele andere es ebenfalls nutzen. Steigt der Nutzen mit der Zahl der Nutzer, spricht man von einem positiven Netzwerkeffekt.

Ein Mehrspieler-Spiel ohne Mitspieler ist wertlos, eine Plattform ohne Inhalte ebenso.

Definition 1.8: Preisdifferenzierung

Preisdifferenzierung bedeutet, dasselbe oder nahezu dasselbe Produkt an verschiedene Kundengruppen zu verschiedenen Preisen zu verkaufen, etwa nach Region, Zeitpunkt oder Personengruppe.

Definition 1.9: Dynamische Preisgestaltung

Bei dynamischer Preisgestaltung ändert sich der Preis eines Produkts laufend, abhängig von Nachfrage, Auslastung oder Angebot.

Definition 1.10: Disintermediation

Disintermediation bezeichnet den Wegfall von Zwischenstufen in der Wertschöpfungskette. Hersteller erreichen ihre Kunden direkt, Zwischenhändler verlieren an Bedeutung.

Für die Entwicklung haben diese Effekte eine unangenehme Folge. Wegen der Netzwerkeffekte gewinnt in digitalen Märkten oft der Erste oder der Größte fast alles. Das erzeugt enormen Zeitdruck, und Zeitdruck ist ein verlässlicher Weg zu schlechter Qualität.

1.6 Warum Projekte scheitern

Fasst man die Beispiele und die besonderen Eigenschaften von Software zusammen, ergibt sich eine Liste typischer Gründe für das Scheitern:

- unrealistische oder unklar formulierte Ziele,
- falsch eingeschätzte Ressourcen und unterschätzte Komplexität,
- schlecht definierte Anforderungen,
- fehlende Kommunikation zwischen Entwicklern, Auftraggebern und Nutzern,
- kaum Risikomanagement,
- Einsatz unfertiger Technologien,
- schlampige Programmierung und ungenügendes Testen,
- fehlende Dokumentation,
- kommerzieller Druck und Entscheidungen von außen,
- schlechtes Projektmanagement.

Auffällig ist, wie wenige dieser Gründe technischer Natur sind. Die meisten betreffen Planung, Kommunikation und Organisation und damit letztlich Menschen. Für die Spielebranche lässt sich das recht genau beziffern, weil viele Studios nach der Veröffentlichung eines Spiels einen Rückblick veröffentlichen, ein sogenanntes **Postmortem**. Politowski und Kollegen (2021) haben 200 solcher Postmortems aus den Jahren 1997 bis 2019 ausgewertet und darin 927 Probleme gefunden. 46 % davon betrafen die Produktion, also Technik, Game Design, Werkzeuge und Tests, und 45 % das Management, also Team, Planung, Kommunikation und Zeitdruck. Die restlichen 9 % waren geschäftliche Probleme wie Marketing (Abbildung 1.4).

Merke

Software- und Medienprojekte scheitern selten an der Technik. Sie scheitern an unklaren Zielen, an schlechter Kommunikation und an falscher Planung.

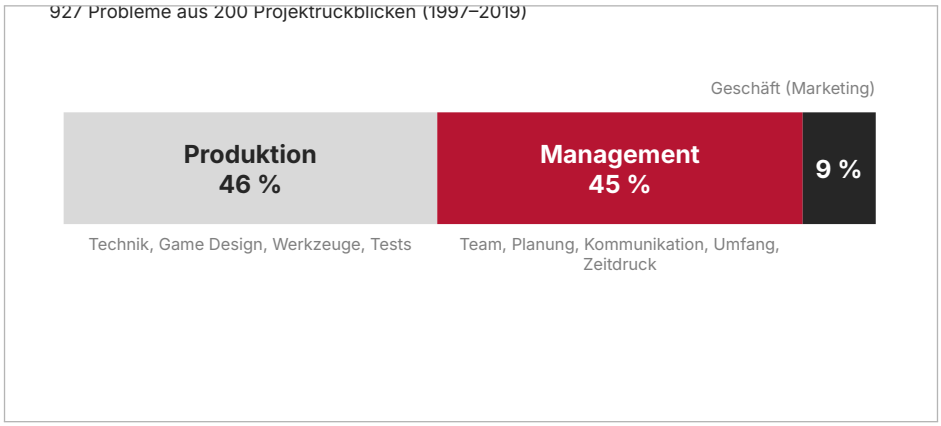


Abbildung 1.4: Art der Probleme in 200 Postmortems der Spielebranche (Daten: Politowski et al. 2021).

1.7 Softwarekrise und Software Engineering

1.7.1 Die Softwarekrise

Neu ist diese Einsicht nicht. In den 1950er-Jahren war die Hardware der teure Teil eines Computersystems, die Programme waren Nebensache. Mit jeder Generation wurden die Rechner leistungsfähiger, die Programme größer und die Projekte unübersichtlicher. Mitte der 1960er-Jahre kostete die Software in großen Projekten bereits mehr als die Hardware, und ein wachsender Teil davon entfiel nicht auf die Entwicklung, sondern auf Pflege und Wartung. Projekte wurden zu spät fertig, sprengten ihr Budget oder funktionierten nicht. Für diesen Zustand bürgerte sich Ende der 1960er-Jahre das Wort **Softwarekrise** ein.

Der Informatiker Edsger W. Dijkstra beschrieb die Ursache 1972 so:

> *As long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem, and now we have gigantic computers, programming has become an equally gigantic problem.*
>
> — E. W. Dijkstra, The Humble Programmer, 1972

Seither sind die Programme nicht kleiner geworden. Der Kern des Betriebssystems Linux umfasste 1994 rund 176 000 Zeilen Quellcode, im Jahr 2025 waren es mehr als 40 Millionen (Abbildung 1.5). Ein solches System kann kein einzelner Mensch mehr vollständig überblicken. Die wachsende **Komplexität** gilt deshalb als die eigentliche Ursache der Softwarekrise, und an ihr hat sich bis heute nichts geändert.

1.7.2 Die Antwort: Software Engineering

Im Jahr 1968 trafen sich auf Einladung der NATO Fachleute in Garmisch, um über die Krise zu beraten. Der Titel der Konferenz war als Provokation gemeint: *Software Engineering*. Software sollte künftig so gebaut werden, wie Ingenieure Brücken und Maschinen bauen. Das war damals ein Wunsch und keine Beschreibung der Wirklichkeit. Die bis heute gebräuchlichste Definition stammt vom Berufsverband IEEE.

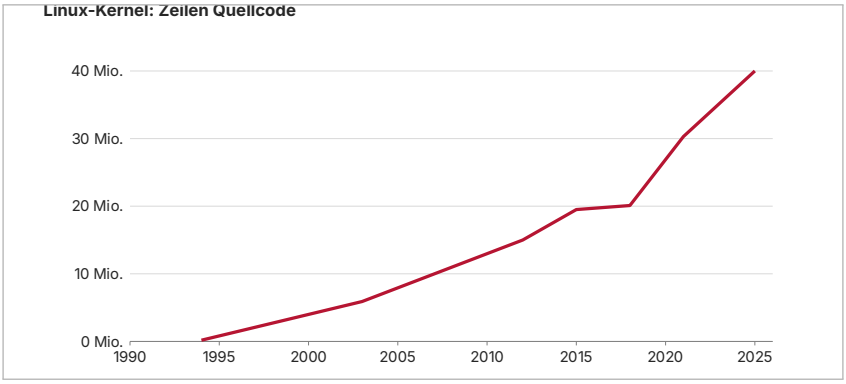


Abbildung 1.5: Umfang des Linux-Kernels in Zeilen Quellcode.

Definition 1.11: Software Engineering

„The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software.“ (IEEE Standard Glossary of Software Engineering Terminology, 1990)

Auf Deutsch: die Anwendung eines **systematischen**, **disziplinierten** und **quantifizierbaren** Vorgehens auf Entwicklung, Betrieb und Wartung von Software, also die Anwendung ingenieurmäßigen Arbeitens auf Software.

Drei Dinge fallen an dieser Definition auf. Sie verlangt ein planvolles Vorgehen statt Ausprobieren. Sie verlangt, dass man misst, statt zu schätzen. Und sie umfasst neben der Entwicklung ausdrücklich Betrieb und Wartung.

Was dabei mit „Engineering“ gemeint ist, hat der amerikanische Ingenieurverband ECPD so beschrieben:

> *The creative application of scientific principles to design or develop structures, machines, apparatus, or manufacturing processes [. . .]; or to forecast their behavior under specific operating conditions; all as respects an intended function, economics of operation and safety to life and property.*
>
> — Engineers' Council for Professional Development

Ingenieure wenden also wissenschaftliche Erkenntnisse an, können das Verhalten ihrer Konstruktionen vorhersagen und achten auf Wirtschaftlichkeit und Sicherheit. Gerade das Vorhersagen fällt bei Software schwer, weil sie unsichtbar und schwer zu messen ist.

Andere Autoren setzen andere Schwerpunkte. Barry Boehm betont die wissenschaftliche Grundlage und die Dokumentation:

> *The practical application of scientific knowledge in the design and construction of computer programs and the associated documentation required to develop, operate, and maintain them.*
>
> — B. W. Boehm, 1976

Richard Fairley betont das Management, also Termine und Kosten:

> *Software engineering is the technological and managerial discipline concerned with systematic production and maintenance of software products that are developed and modified on time and within cost estimates.*

— R. E. Fairley, 1985

Merke

Allen Definitionen ist gemeinsam: Software Engineering bedeutet systematisches Vorgehen, umfasst mehr als den Programmcode und behält Zeit und Kosten im Blick.

1.7.3 Software Engineering ist mehr als Programmieren

Programmieren, also der Umgang mit Programmiersprachen, Datenstrukturen und Algorithmen, ist nur ein Teil des Software Engineering. Hinzu kommen:

- die **Spezifikation**: festlegen, was die Software leisten soll,
- der **Entwurf**: festlegen, wie sie aufgebaut ist,
- die **Gestaltung der Benutzeroberfläche**,
- das **systematische Testen**,
- die **Dokumentation**,
- das **Projektmanagement**,
- Überlegungen zur **Hardware**, auf der die Software laufen soll.

Je größer ein Projekt wird, desto kleiner wird der Anteil des Programmierens. In kleinen Projekten macht das Schreiben von Code den größten Teil der Arbeit aus, in großen Projekten überwiegen Fehlersuche, Dokumente, Abstimmung und Management. Wer bisher vor allem kleine Programme für Übungsblätter geschrieben hat, hält das Programmieren deshalb leicht für die Hauptsache.

Bevor Sie weiterlesen: Welchen Anteil ihrer Arbeitszeit verbringen Berufsentwickler damit, vorhandenen Code zu lesen und zu verstehen? Legen Sie sich auf eine Prozentzahl fest.

Mythos und Befund: Entwickler verbringen ihre Zeit vor allem mit Programmieren

Xia und Kollegen (2018) haben bei 78 Berufsentwicklern über mehrere Wochen aufgezeichnet, womit sie ihre Arbeitszeit verbringen. Im Mittel entfielen rund 58 % auf das Verstehen von vorhandenem Code, also auf Lesen, Navigieren und Nachvollziehen (Abbildung 1.6). Das Schreiben von neuem Code war nur ein kleiner Teil der Arbeit. Lesbarer Code und gute Dokumentation sparen deshalb vor allem den Kolleginnen und Kollegen Zeit, und nicht selten dem eigenen Ich ein halbes Jahr später.

1.7.4 Meilensteine

Tabelle 1.2 nennt die wichtigsten Stationen in der Entwicklung des Fachs.

Der jüngste Meilenstein macht das Fach nicht überflüssig. Ein KI-Assistent kann Code erzeugen, aber zu entscheiden, was gebaut werden soll, zu prüfen, ob das Ergebnis stimmt, und die Verantwortung dafür zu tragen, bleibt Aufgabe von Menschen. Genau diese Aufgaben bilden den Kern des Software Engineering. Wie groß der Zeitgewinn durch KI tatsächlich ist,

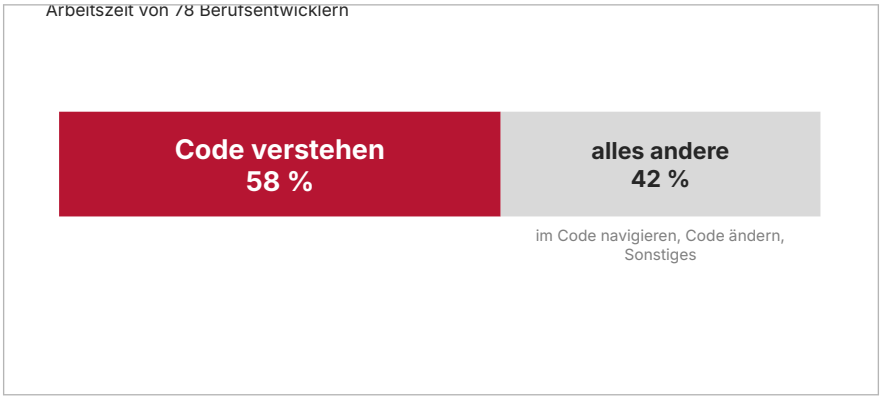


Abbildung 1.6: Anteil der Arbeitszeit, den Berufsentwickler mit dem Verstehen von Code verbringen (Daten: Xia et al. 2018).

Zeit	Meilenstein
1968	NATO-Konferenz in Garmisch macht den Begriff *Software Engineering* bekannt.
1968	Dijkstra kritisiert die Sprunganweisung GOTO. Daraus entsteht die **strukturierte Programmierung**, die nur Abfolge, Auswahl und Wiederholung zulässt.
1970er	**Abstrakte Datentypen** und **Datenkapselung**: Daten werden vor direktem Zugriff von außen verborgen.
1980er	**Integrierte Entwicklungsumgebungen** vereinen Editor, Übersetzer und Fehlersuche in einem Werkzeug.
1990er	Die **objektorientierte Programmierung** setzt sich durch. 1997 wird die Modellierungssprache UML standardisiert.
2001	Das **Agile Manifest** fordert kurze Entwicklungszyklen statt großer Pläne.
2005	Mit **Git** verbreitet sich die verteilte Versionsverwaltung.
2010er	Software wird automatisch gebaut, getestet und ausgeliefert.
2020er	**KI-Assistenten** schreiben Programmcode mit.

Tabelle 1.2: Meilensteine des Software Engineering.

ist übrigens weniger klar, als es oft scheint. In einer Studie von Becker und Kollegen (2025) erwarteten erfahrene Entwickler vorab, mit KI-Werkzeugen 24 % schneller zu sein, und hatten hinterher das Gefühl, 20 % schneller gewesen zu sein. Gemessen wurden 19 % *mehr* Zeit (Abbildung 1.7). Das ist eine Momentaufnahme vom Frühjahr 2025 mit sechzehn Entwicklern, und die Werkzeuge haben sich seitdem stark verändert. Als dieselbe Forschungsgruppe den Versuch Ende 2025 wiederholte, deutete die Schätzung eher auf einen Zeitgewinn. Für belastbar halten die Autoren sie allerdings selbst nicht, unter anderem weil viele Teilnehmer Aufgaben, die sie nicht ohne KI erledigen wollten, gar nicht erst einreichten (METR 2026). Fest steht damit vor allem eines: Das Gefühl, schneller zu sein, ersetzt keine Messung. Wir kommen im Kapitel über Werkzeuge darauf zurück.

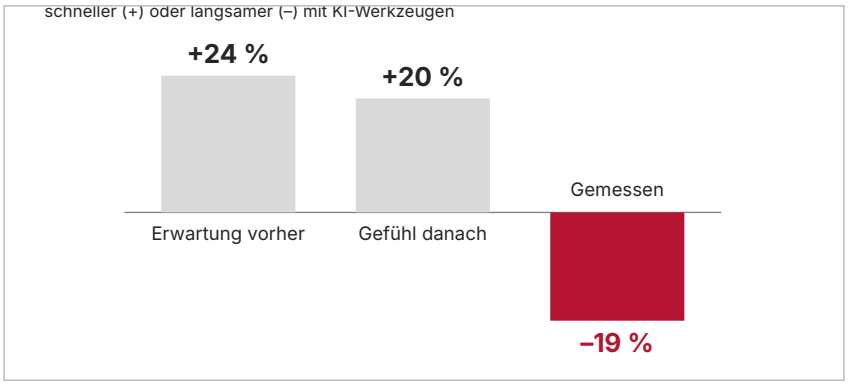


Abbildung 1.7: Erwarteter, gefühlter und gemessener Zeitgewinn erfahrener Entwickler mit KI-Werkzeugen (Daten: Becker et al. 2025).

1.8 Media Engineering

Definition 1.12: Media Engineering

Media Engineering ist die Anwendung der Prinzipien des Software Engineering auf die Entwicklung digitaler Medien.

Digitale Medien sind zum Beispiel Spiele, interaktive Anwendungen, Anwendungen für virtuelle und erweiterte Realität, Webseiten und Apps. Die Prinzipien des Software Engineering gelten auch für sie, aber die Schwerpunkte verschieben sich (Tabelle 1.3).

Klassische Software	Digitale Medien
Oft für den Arbeitsplatz	Eher zur Unterhaltung
Besteht vor allem aus Code	Code und Inhalte: Grafik, Klang, Geschichte, Leveldesign
Oft klar umrissene Benutzergruppe	Schwer vorhersehbar; ein Erfolg kann über Nacht aus hundert Benutzern hunderttausend machen
Anforderungen lassen sich erfragen	„Spaß“ lässt sich nicht erfragen, sondern nur mit Prototypen und Probespielen ermitteln
Meist Maus, Tastatur, Touchscreen	Viele Eingabegeräte: Controller, Touchscreen, Sprache, Gesten, VR-Brillen
Die Gestaltung folgt der Funktion	Die Funktion folgt der Gestaltung

Tabelle 1.3: Unterschiede zwischen klassischer Software und digitalen Medien.

Aus diesen Unterschieden folgt zweierlei. An Medienprojekten arbeiten Menschen aus sehr verschiedenen Fächern zusammen: Programmierer, Grafiker, Sounddesigner und Spieldesigner. Und der Erfolg lässt sich schlechter vorab festlegen, weil er vom Produkterfolg abhängt (Abschnitt 1.2) und nicht davon, ob eine Liste von Funktionen abgearbeitet wurde.

1.9 Der Software Development Lifecycle

Trotz aller Unterschiede folgt die Entwicklung von Software und von digitalen Medien demselben Grundmuster.

Definition 1.13: Software Development Lifecycle

Der Software Development Lifecycle (SDLC, Software-Lebenszyklus) beschreibt die Tätigkeiten, die eine Software von der ersten Idee bis zur Ablösung durchläuft: **Anforderungsanalyse**, **Entwurf**, **Implementierung**, **Test** und **Weiterentwicklung**.

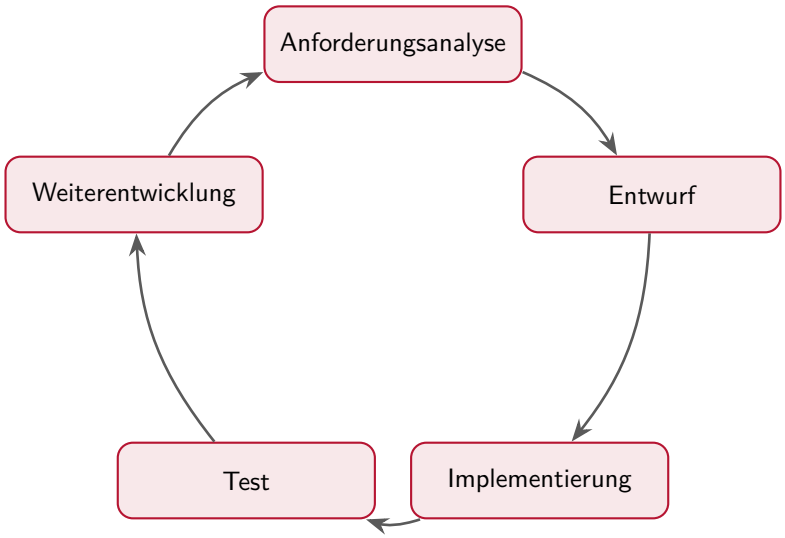


Abbildung 1.8: Der Software Development Lifecycle.

Jede Phase beantwortet eine Frage. Die Anforderungsanalyse klärt, *was* die Software tun soll. Der Entwurf klärt, *wie* sie aufgebaut wird. In der Implementierung wird sie gebaut. Der Test prüft, ob das Gebaute stimmt. Nach der Auslieferung beginnt die Weiterentwicklung, und mit neuen Wünschen beginnt der Kreis von vorn.

Dieser Kreis ist der rote Faden der Vorlesung (Tabelle 1.4). Die ersten Kapitel folgen den Phasen, die letzten behandeln, was alle Phasen betrifft: wie man die Phasen anordnet, wie Menschen zusammenarbeiten und wie man ein Projekt plant. Die Übungen durchlaufen denselben Kreis an einem eigenen kleinen Spiel.

Phase	**Kapitel der Vorlesung**	**Übung**
Werkzeug	Game Engines	Einarbeitung in die Unreal Engine
Anforderungsanalyse	Requirements Engineering	Pflichtenheft
Entwurf	GUI-Design; Objektorientierte Analyse und Design	GUI-Entwurf; UML-Diagramme
Implementierung	Tools	Implementierung des Spiels
Test	Testen	Betatest und Abnahmetest
Übergreifend	Prozessmodelle; Der Mensch im Projekt; Projektmanagement	

Tabelle 1.4: Der Lebenszyklus als Fahrplan der Vorlesung.

Im Projektchen

Bevor Sie mit dem Projekt beginnen, schätzen Sie bitte jede für sich, wie viele Stunden Ihre Gruppe insgesamt für das fertige Spiel brauchen wird, und notieren Sie die Zahl.

Am Ende des Semesters vergleichen wir die Schätzungen mit den tatsächlich gebuchten Stunden. Nach dem, was Sie in diesem Kapitel über den Planungsfehlschluss gelesen haben, ahnen Sie vermutlich, in welche Richtung die Abweichung gehen wird. Ein wenig Hoffnung darf man trotzdem haben.

Zusammenfassung

Das Wichtigste aus Kapitel 1

- Software- und Medienprojekte scheitern häufig, und zwar auf sehr verschiedene Arten: durch Programmierfehler, schlechte Oberflächen, falsche Produkte und ausufernden Umfang.
- Projekterfolg (Zeit, Kosten, Umfang) und Produkterfolg (Wert für die Nutzer) sind zweierlei. Das magische Dreieck beschreibt, wie Qualität, Kosten und Zeit zusammenhängen.
- Software ist immateriell, schwer zu messen und leicht zu ändern. Das macht sie schwer planbar.
- Digitale Güter kosten in der Vervielfältigung fast nichts. Das verändert die Märkte und erzeugt Zeitdruck.
- Die meisten Ursachen des Scheiterns liegen bei Menschen und Organisation, nicht bei der Technik.
- Software Engineering ist die Antwort auf die Softwarekrise: ein systematisches, diszipliniertes und quantifizierbares Vorgehen bei Entwicklung, Betrieb und Wartung. Programmieren ist nur ein Teil davon.
- Media Engineering wendet diese Prinzipien auf digitale Medien an.
- Der Software Development Lifecycle gliedert die Arbeit in Anforderungsanalyse, Entwurf, Implementierung, Test und Weiterentwicklung.

Kontrollfragen

1. Software kann auf sehr unterschiedliche Arten scheitern. Nennen Sie drei Arten und beschreiben Sie zu jeder ein Beispiel.

2. Warum kann ein Projekt scheitern, obwohl die Software keinen einzigen Programmfehler enthält?

3. Erklären Sie das magische Dreieck. Was geschieht mit den anderen Ecken, wenn man an einer zieht?

4. Worin unterscheiden sich Projekterfolg und Produkterfolg? Kann es das eine ohne das andere geben?

5. Woraus besteht Software außer aus Programmcode, und was bedeutet das für ein Spiel?

6. Nennen Sie vier Eigenschaften, die Software von materiellen Produkten unterscheiden, und erläutern Sie, warum jede davon die Planung erschwert.

7. Software ist teuer zu entwickeln, aber fast kostenlos zu kopieren. Welche Folgen hat das für die Märkte digitaler Güter und für den Zeitdruck in der Entwicklung?

8. Was versteht man unter der Softwarekrise, und wie hängt sie mit der Entstehung des Software Engineering zusammen?

9. Was bedeutet es, Software systematisch, diszipliniert und quantifizierbar zu entwickeln? Nennen Sie zu jeder Eigenschaft ein Beispiel aus einem Spieleprojekt.

10. Warum ist Software Engineering mehr als Programmieren, und warum wird das mit wachsender Projektgröße immer deutlicher?

11. Worin unterscheidet sich die Entwicklung digitaler Medien von der Entwicklung klassischer Software?

12. Welche Phasen hat der Software Development Lifecycle, welche Frage beantwortet jede Phase, und warum wird er als Kreis dargestellt?

Aufgaben

A1. Ordnen Sie die vier Fälle vom Kapitelanfang (CrowdStrike, Citibank, Concord, Star Citizen) danach ein, *wo* der Fehler lag (im Code, in der Oberfläche, im Projekt ohne Programmfehler) und ob es sich um ein Scheitern im Sinne des Projekterfolgs, des Produkterfolgs oder beider handelt.

A2. Ihr Team soll ein Spiel in sechs Monaten fertigstellen. Nach drei Monaten verlangt der Publisher, das Spiel zwei Monate früher zu veröffentlichen, weil zum geplanten Termin ein großer Konkurrent erscheint. Welche Möglichkeiten haben Sie nach dem magischen Dreieck, und welche Folgen hat jede?

A3. Schätzen Sie, ohne nachzudenken, wie viele Stunden Sie für das Lesen dieses Kapitels und das Beantworten aller Kontrollfragen brauchen werden. Halten Sie die Zahl fest und messen Sie die tatsächliche Zeit. Welche Abweichung erwarten Sie nach Buehler, Griffin und Ross (1994), und welches Vorgehen würde die Schätzung verbessern?

A4. Welcher der Begriffe Netzwerkeffekt, Preisdifferenzierung, dynamische Preisgestaltung und Disintermediation beschreibt jeweils die folgende Beobachtung? (a) Ein Spiel kostet auf Steam in der Türkei deutlich weniger als in Deutschland. (b) Ein Fahrdienst verlangt bei Regen und zu Stoßzeiten mehr. (c) Ein Indie-Studio verkauft sein Spiel selbst über eine Online-Plattform, ohne Publisher und ohne Handel. (d) Ein Chatdienst ist umso nützlicher, je mehr Freunde ihn verwenden.

A5. Eine Gruppe beschreibt ihr letztes Projekt so: „Wir haben sofort angefangen zu programmieren. Getestet haben wir am Abend vor der Abgabe. Dass der Auftraggeber eine Bestenliste wollte, haben wir erst bei der Präsentation erfahren. Wie viele Stunden wir gebraucht haben, weiß keiner so genau.“ Welche Phasen des Software Development Lifecycle fehlten oder kamen zu kurz? Ordnen Sie jedem der vier Sätze eine der Eigenschaften systematisch, diszipliniert und quantifizierbar zu, gegen die er verstößt. Was hätte die Gruppe mit dem geringsten Aufwand anders machen können?

Lernziele überprüfen

Können Sie ...	**Kontrollfragen**	**Aufgaben**
an Beispielen erklären, auf welche Arten Projekte scheitern?	1, 2	A1
Projekterfolg und Produkterfolg unterscheiden?	3, 4	A1, A2
die Besonderheiten von Software und digitalen Gütern nennen, die Projekte schwer planbar machen?	5, 6, 7, 11	A3, A4
Software Engineering definieren und erläutern, warum es mehr umfasst als Programmieren?	8, 9, 10	A5
die Phasen des Software Development Lifecycle benennen?	12	A5

Zum Weiterlesen

- Frederick P. Brooks: *The Mythical Man-Month. Essays on Software Engineering.* Addison-Wesley, Jubiläumsausgabe 1995. Der Klassiker, erstaunlich aktuell.
- Bent Flyvbjerg, Dan Gardner: *How Big Things Get Done.* Currency, 2023. Warum Großprojekte scheitern und was die erfolgreichen anders machen.
- Ian Sommerville: *Software Engineering.* Pearson, 10. Auflage, auch auf Deutsch. Ein verbreitetes Lehrbuch für das gesamte Fach.

2 Game Engines

Im Projekt der Übungen bauen Sie ein kleines Spiel mit der Unreal Engine. In diesem Kapitel sehen wir uns an, was für ein Werkzeug das eigentlich ist. Wir beginnen mit einem Blick auf den Markt, denn bevor man ein Semester in ein Werkzeug steckt, darf man ruhig fragen, ob sich das lohnt. Danach geht es um die Technik, also darum, woher Game Engines kommen, woraus sie bestehen und nach welchen Kriterien man eine auswählt.

Lernziele

Nach diesem Kapitel können Sie

- die wirtschaftliche Bedeutung von Spielen im Medienmarkt einordnen und erklären, warum Marktzahlen nur mit ihrer Abgrenzung vergleichbar sind,
- die Entstehung von Game Engines als Folge der Trennung von Technik und Inhalt beschreiben,
- den Begriff Game Engine definieren,
- die Stationen der Game Loop und die Aufgaben der wichtigsten Module am Beispiel eines Frames erklären,
- Kriterien für die Auswahl einer Engine nennen und die Entscheidung zwischen Kaufen und Selbstbauen begründen.

2.1 Der Medienmarkt in Deutschland

2.1.1 Wie viel Zeit, wie viel Geld?

Bevor Sie weiterlesen: Wie viele Stunden am Tag nutzen Menschen in Deutschland im Durchschnitt Medien, also Radio, Fernsehen, Internet, Spiele und Gedrucktes zusammengenommen?

Im Durchschnitt nutzen Menschen in Deutschland fast elf Stunden am Tag Medien (Abbildung 2.1). Die Zahl ist so hoch, weil parallele Nutzung doppelt zählt: Wer beim Frühstück Radio hört und dabei Zeitung liest, steht mit beiden Medien in der Statistik. Den größten Teil der Zeit nehmen nach wie vor Radio und Fernsehen ein, wobei das Fernsehen seit Jahren verliert und Online-Video gewinnt. Der Wert ist ein Durchschnitt über alle Altersgruppen ab 14 Jahren; bei jüngeren Menschen liegen Online-Video, Musikstreaming und Spiele deutlich weiter vorn.

Gemessen an dieser Nutzungszeit ist der Umsatz der Branche überschaubar. Die sieben Mediengattungen in Tabelle 2.1 setzten 2024 zusammen rund 45 Milliarden Euro um. Zum Vergleich: Der Volkswagen-Konzern allein kam im selben Jahr auf 325 Milliarden Euro, das deutsche Bruttoinlandsprodukt lag bei rund 4 300 Milliarden Euro. Die sieben Gattungen zusammen entsprechen also etwa einem Prozent der Wirtschaftsleistung.

Die Tabelle zeigt drei Entwicklungen. Gedrucktes verliert, bleibt aber groß: Mit Büchern wird immer noch mehr als doppelt so viel umgesetzt wie mit Musikaufnahmen und Kino

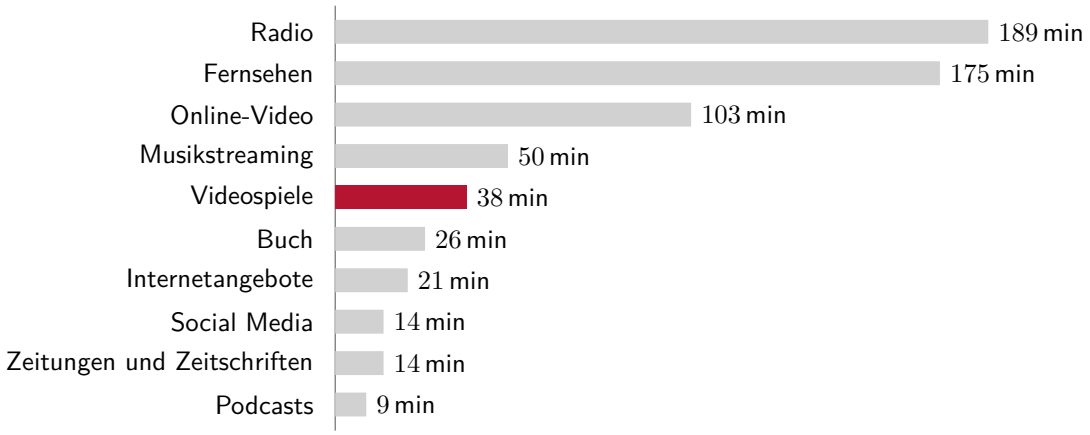


Abbildung 2.1: Durchschnittliche tägliche Mediennutzung in Deutschland 2025 in Minuten (Auswahl; Daten: VAUNET-Mediennutzungsanalyse 2025).

Mediengattung	**2013**	**2024**
Fernsehen	12,9	10,5
Zeitungen und Zeitschriften	13,6	8,7
Bücher	9,5	8,1
Videospiele (ohne Hardware)	1,9	6,0
Internetvideo	–	4,9
Hörfunk (letzter Wert: 2022)	3,5	3,9
Musikaufnahmen	1,5	2,6

Tabelle 2.1: Umsätze ausgewählter Mediengattungen in Deutschland in Milliarden Euro (Daten: PwC German Entertainment & Media Outlook; game – Verband der deutschen Games-Branche).

zusammen. Internetvideo, also Streamingdienste und Videoplattformen, gab es 2013 als eigene Kategorie noch gar nicht. Und der Umsatz mit Videospielen hat sich in gut zehn Jahren verdreifacht.

Merke

Jede Marktzahl hängt davon ab, was man mitzählt. Der Radiomarkt ist je nach Quelle 0,7 Milliarden Euro groß (nur Werbung) oder knapp 4 Milliarden Euro (einschließlich Rundfunkbeitrag). Die Unternehmensberatung PwC kommt für die gesamte Unterhaltungs- und Medienbranche 2024 auf 111,6 Milliarden Euro, weil sie Internetzugänge und Onlinewerbung mitrechnet. Bevor man zwei Zahlen vergleicht, sollte man deshalb nachsehen, wie die Quellen den Markt jeweils abgegrenzt haben.

2.1.2 Der Spielemarkt

Nach den Zahlen des Branchenverbands *game* wurden in Deutschland 2025 rund 9,4 Milliarden Euro mit Spielen, Spiele-Hardware und Online-Diensten umgesetzt. Deutschland ist damit der größte Spielemarkt Europas. Rund 41 Millionen Menschen spielen, ihr Durchschnittsalter liegt bei 38 Jahren. Zwei Beobachtungen sind für alle wichtig, die Spiele entwickeln wollen.

Erstens wird das Geld nicht mehr mit dem Verkauf von Spielen verdient. Auf den Kauf von Spielen entfielen 2025 rund 0,8 Milliarden Euro, auf Käufe *innerhalb* von Spielen (In-

Game- und In-App-Käufe) rund 4,1 Milliarden Euro, also das Fünffache. Die umsatzstärkste Plattform ist das Smartphone, vor Konsole und PC. Das Geschäftsmodell bestimmt damit, was für ein Spiel man überhaupt baut. Das Beispiel *Concord* aus Kapitel 1 ist auch daran gescheitert: Es kostete Geld in einem Genre, in dem die Konkurrenz kostenlos spielbar ist.

Bevor Sie weiterlesen: Von 100 Euro, die in Deutschland für Spiele ausgegeben werden: Wie viel davon bleibt bei Unternehmen aus Deutschland?

Zweitens landet von diesem Geld wenig bei deutschen Entwicklern. Von 100 Euro, die in Deutschland für Spiele ausgegeben werden, bleiben 5,50 Euro bei deutschen Unternehmen (Abbildung 2.2). Der Rest geht an internationale Publisher, Plattformbetreiber und App-Stores.



Abbildung 2.2: Anteil deutscher Unternehmen an den Ausgaben für Spiele in Deutschland (Daten: game – Verband der deutschen Games-Branche, Jahresreport 2026).

Trotzdem gibt es einen Arbeitsmarkt: 2026 zählte der Verband 956 Unternehmen und rund 12 000 Beschäftigte bei Entwicklern und Publishern.

2.1.3 Game Engines außerhalb von Spielen

Kenntnisse über Game Engines sind übrigens nicht nur in der Spielebranche gefragt. Dieselben Werkzeuge werden überall eingesetzt, wo dreidimensionale Inhalte in Echtzeit dargestellt werden: in Trainingssimulatoren der Medizin, bei der Planung von Fabrikanlagen, in Fahrzeugkonfiguratoren, in der Architektur und in virtuellen Fernsehstudios. Beim Film ersetzt die Engine zunehmend den Greenscreen: Bei der *Virtual Production* stehen die Schauspieler vor einer LED-Wand, deren Bild eine Game Engine passend zur Kameraposition berechnet.

2.2 Von der Platine zur Engine

Pac-Man erschien 1980. Ein Team von zehn Leuten arbeitete knapp anderthalb Jahre daran, und das gesamte Programm war für genau eine Platine geschrieben. Das war damals üblich. Die Hardware war langsam, also wurde jedes Programm von Hand an die Maschine angepasst, und beim nächsten Spiel begann man von vorn. Wiederverwendung gab es kaum.

Ab 1983 erschienen die ersten Baukästen, etwa das *Pinball Construction Set*. Mit ihnen ließen sich ohne Programmierung Spiele eines bestimmten Genres zusammenstellen. Sie blieben aber auf dieses eine Genre beschränkt und richteten sich an Hobby-Entwickler.

Der Durchbruch kam mit *Doom* (id Software, 1993), einem Spiel, das heute auf so ziemlich allem läuft, was einen Prozessor hat, einschließlich Taschenrechnern. Das Programm trennte die Technik sauber vom Inhalt: Level, Texturen und Töne lagen in eigenen Dateien. Das hatte zwei Folgen. Spieler konnten eigene Level bauen und tauschen. Und andere Firmen konnten die Technik lizenzieren und mit eigenen Inhalten ein neues Spiel daraus machen. In diesem Zusammenhang entstand der Begriff **Game Engine**.

Aus dem Nebengeschäft wurde ein Geschäftsmodell. Ende der 1990er-Jahre wurden Engines gezielt für die Lizenzierung entwickelt, etwa für *Quake II* (1997) und *Unreal* (1998). Später kehrte sich das Verhältnis um: Bei der CryEngine (2004) war das Spiel vor allem Werbung für die Engine. Unity (seit 2005) stellt schließlich gar keine eigenen Spiele mehr her. Heute teilen sich wenige universelle Engines den größten Teil des Marktes (Abschnitt 2.6).

Merke

Die Geschichte der Game Engines ist eine Geschichte der Wiederverwendung. Software ist teuer zu entwickeln, aber fast kostenlos zu kopieren (Kapitel 1). Also lohnt es sich, die Teile, die jedes Spiel braucht, nur einmal zu bauen.

2.3 Was ist eine Game Engine?

Definition 2.1: Game Engine

Eine **Game Engine** ist eine Sammlung von Modulen und Schnittstellen, die es einem Entwicklungsteam erlaubt, sich auf die Inhalte und Regeln des Spiels zu konzentrieren statt auf die Technik (nach Julian Gold).

Die Definition nennt zwei Bestandteile und ein Ziel. Die **Module** erledigen jeweils eine Aufgabe, die fast jedes Spiel hat, etwa Bilder zeichnen oder Töne abspielen. Über die **Schnittstellen** greift das Spiel auf diese Module zu. Das Ziel ist die Arbeitsteilung: Die Engine liefert die Technik, das Team liefert das Spiel.

Wo die Engine aufhört und das Spiel anfängt, ist nicht scharf festgelegt. Als Faustregel gilt: Je mehr sich an einem Spiel über Daten ändern lässt, ohne das Programm selbst anzufassen, desto eher handelt es sich um eine Engine.

Neben der Wiederverwendung haben Engines einen zweiten Vorteil. Sie verbergen die Unterschiede zwischen den Zielplattformen. Dasselbe Spiel lässt sich mit wenig Aufwand für PC, Konsolen und Smartphones ausliefern.

2.4 Ein Frame Pac-Man: die Module einer Engine

Was die Module einer Engine tun, versteht man am leichtesten, wenn man ihr bei der Arbeit zusieht. Ein Spiel ist im Kern eine Endlosschleife.

Definition 2.2: Game Loop

Die **Game Loop** ist die Hauptschleife eines Spiels. Sie wird ständig wiederholt, solange das Spiel läuft. Jeder Durchlauf liest die Eingaben, berechnet den neuen Zustand der Spielwelt und zeichnet ein Bild. Ein Durchlauf heißt **Frame**.

Bevor Sie weiterlesen: Bis zu welcher Bildrate werden Spieler in einem Ego-Shooter noch messbar besser: bis 30, 60, 90 oder 240 Bilder pro Sekunde?

Bei 60 Bildern pro Sekunde bleiben für einen Durchlauf knapp 17 Millisekunden. Die Zahl ist nicht willkürlich gewählt. In Versuchen mit Ego-Shootern steigt die Leistung der Spieler bis etwa 60 Bilder pro Sekunde steil an und verbessert sich oberhalb von etwa 90 kaum noch (Abbildung 2.3). Das Bild wird zwar bis 500 Bilder pro Sekunde als immer flüssiger empfunden, an der Punktzahl ändert das aber nichts mehr. In den 17 Millisekunden durchläuft die Engine nacheinander mehrere Stationen (Abbildung 2.4). Angenommen, der Spieler drückt in Pac-Man die Taste „nach links“. Bis er das Ergebnis sieht, geschieht Folgendes.

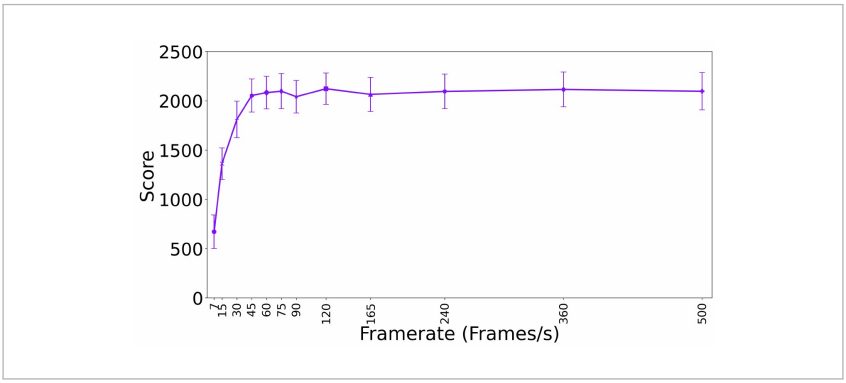


Abbildung 2.3: Punktzahl von Spielern in einem Ego-Shooter in Abhängigkeit von der Bildrate (Daten: Tokey, Boudaoud, Kim, Spjut und Claypool 2025).

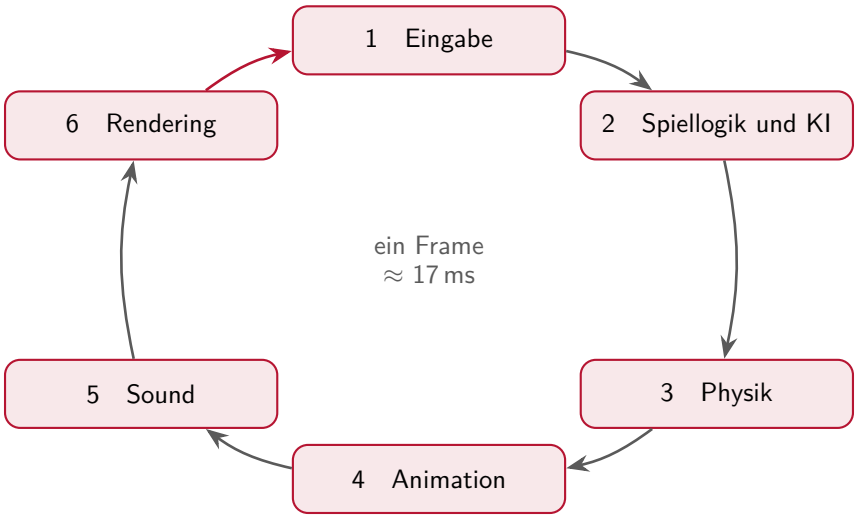


Abbildung 2.4: Die Game Loop: Stationen eines Frames.

2.4.1 Eingabe

Das Eingabemodul fragt die angeschlossenen Geräte ab: Tastatur, Maus, Gamepad, Touchscreen oder Spezialcontroller. Seine wichtigste Aufgabe ist eine Übersetzung. Es bildet die gedrückte Taste auf eine Aktion im Spiel ab, hier „nach links“. Der Rest des Spiels kennt nur die Aktion, nicht das Gerät. Deshalb funktioniert dasselbe Spiel mit Tastatur und Gamepad, und Spieler können die Belegung ändern, ohne dass der Spielcode angefasst werden muss.

2.4.2 Spiellogik und künstliche Intelligenz

Nun wird der neue Zustand der Spielwelt berechnet. Pac-Man bewegt sich ein Stück nach links, und die Geister entscheiden, wohin sie laufen. Dafür ist das KI-Modul zuständig. Es sorgt dafür, dass sich Figuren, die nicht vom Spieler gesteuert werden, glaubhaft verhalten.

Beispiel: Die vier Geister

Die Geister in Pac-Man wirken, als würden sie zusammenarbeiten. Tatsächlich folgt jeder einer einzigen einfachen Regel. Der rote Geist läuft direkt auf Pac-Man zu. Der rosa Geist zielt auf ein Feld vier Schritte vor Pac-Man und schneidet ihm so den Weg ab. Der blaue richtet sich zusätzlich nach der Position des roten, und der orange dreht ab, sobald er Pac-Man zu nahe kommt. Dazu wechseln alle vier zwischen den Zuständen *Jagen*, *Zerstreuen* und *Flüchten*. Für Spiele-KI genügt es also, klug zu *wirken*. Ein Gegner, der wirklich optimal spielt, würde vermutlich auch niemandem Spaß machen.

Heutige Engines liefern dafür fertige Bausteine: Zustandsautomaten, Verhaltensbäume (*Behavior Trees*), Wegfindung und Schwarmverhalten.

> Zustandsautomaten lernen Sie im Kapitel zur objektorientierten Analyse als UML-Zustandsdiagramme kennen.

Regeln wie die der Geister ändern sich während der Entwicklung ständig. Damit dafür nicht jedes Mal die Engine neu übersetzt werden muss, bieten Engines ein **Scripting**-Modul. Die Spielregeln werden in einer einfacheren Sprache geschrieben, getrennt vom Code der Engine. Beispiele sind Lua, C# in Unity, GDScript in Godot und die grafischen *Blueprints* der Unreal Engine. Der Vorteil ist die Geschwindigkeit beim Ändern, und auch Teammitglieder ohne C++-Kenntnisse können Spielregeln anpassen. Der Nachteil ist die geringere Geschwindigkeit beim Ausführen. Üblich ist deshalb eine Arbeitsteilung: rechenintensive Teile in C++, Spielregeln im Skript.

2.4.3 Physik

Das Physikmodul sorgt dafür, dass sich Objekte nach den Gesetzen der Mechanik verhalten: Sie haben Masse, werden beschleunigt, fallen und prallen ab. Um in Echtzeit zu bleiben, rechnen Engines mit stark vereinfachten Modellen.

Pac-Man braucht davon nur einen Teil, die **Kollisionserkennung**. Sie stellt fest, ob sich zwei Objekte berühren: Pac-Man und eine Münze, eine Wand oder einen Geist. Im Original genügte dafür ein Vergleich, ob beide auf demselben Feld des Rasters stehen. In 3D-Spielen wird jedes Objekt zunächst von einem einfachen Körper umhüllt, etwa einer Kugel. Zwei Kugeln berühren sich, wenn der Abstand ihrer Mittelpunkte kleiner ist als die Summe ihrer Radien. Erst wenn sich die Hüllen berühren, lohnt die genaue und teure Prüfung. Die Engine meldet eine Berührung als Ereignis an das Spiel, das dann entscheidet, was passiert.

2.4.4 Animation

Pac-Man hat eine Münze gefressen, und sein Mund soll sich bewegen. Im Original wechselten dazu drei Einzelbilder einander ab. Dreidimensionale Figuren werden meist per **Skelettanimation** bewegt. Die Figur erhält ein Skelett aus Knochen und Gelenken. Animiert

wird nur das Skelett, und die Engine überträgt dessen Bewegung auf das darüberliegende Dreiecksnetz (*Skinning*). Die Bewegungen stammen oft aus Motion-Capture-Aufnahmen. Die Engine blendet zwischen ihnen über, etwa vom Gehen zum Laufen. Bei der *inversen Kinematik* berechnet sie die Gelenkwinkel selbst, damit zum Beispiel ein Fuß genau auf einer Treppenstufe steht.

2.4.5 Sound

Zum Fressen gehört bei Pac-Man ein Geräusch. Das Soundmodul spielt Geräusche ab, wenn ein Ereignis sie auslöst, hier die Kollision mit der Münze. Dazu kommt Hintergrundmusik, die oft je nach Spielsituation überblendet wird. In 3D-Spielen werden Geräusche im Raum platziert und mit Effekten wie Hall versehen, sodass man hört, aus welcher Richtung sie kommen. Damit trägt Ton nicht nur zur Stimmung bei, sondern gibt dem Spieler auch Rückmeldung und gehört so zur Benutzungsschnittstelle.

2.4.6 Rendering

Erst jetzt wird das Bild gezeichnet. Der **Renderer** hat die Aufgabe, die Szene so schnell wie möglich darzustellen. Sie besteht aus geometrischen Grundformen, vor allem aus Dreiecken, dazu kommen Materialien und Lichtquellen. Der Renderer verbraucht den größten Teil der 17 Millisekunden und nutzt dafür die Grafikkarte. Auf sie greift er über standardisierte Schnittstellen zu, heute Direct3D, Vulkan oder Metal.

Zur Darstellung gehören drei weitere Bausteine:

- Der **Szenengraph** ist eine meist baumartige Datenstruktur. Sie verwaltet alle Objekte einer Szene mit ihren Positionen, Materialien, Lichtquellen und Kameras. Mit ihrer Hilfe lässt sich schnell aussortieren, was gerade nicht im Blickfeld liegt und deshalb nicht gezeichnet werden muss (*Culling*).
- **Visuelle Effekte** sind zum Beispiel Partikelsysteme für Feuer, Rauch und Explosionen, dynamische Schatten, Spiegelungen und die Nachbearbeitung des fertigen Bildes, etwa eine Farbkorrektur.
- Die **GUI** liegt als flache Schicht über dem 3D-Bild: Menüs, Inventar und das *Heads-up-Display* mit Punktestand und Leben.

Beispiel: Level 256

Pac-Man hat 255 spielbare Level. Im 256. versinkt die rechte Hälfte des Bildschirms in wirren Zeichen, und das Level lässt sich nicht beenden. Der Levelzähler ist in einem einzigen Byte gespeichert, das nur Werte bis 255 aufnehmen kann. Die Routine, die am unteren Rand für jedes Level eine Frucht zeichnet, rechnet deshalb in Level 256 mit einem falschen Wert und überschreibt den Bildschirm. Es ist derselbe Fehlertyp wie bei der Ariane 5: Eine Zahl passt nicht in den Speicherplatz, der für sie vorgesehen ist.

Mit dem fertigen Bild ist der Frame beendet, und die Schleife beginnt von vorn. Dauert ein Durchlauf länger als vorgesehen, fällt ein Bild aus, und das Spiel ruckelt. Damit sich Figuren trotzdem gleichmäßig bewegen, rechnet man mit der Zeit, die seit dem letzten Frame vergangen ist. Der zurückgelegte Weg ist Geschwindigkeit mal vergangene Zeit. Lässt man das weg, läuft das Spiel auf einem schnelleren Rechner schneller.

2.4.7 Was nicht in der Schleife liegt

Drei Module arbeiten im Hintergrund und tauchen in der Schleife nicht als eigene Station auf.

Der **Ressourcen-Manager** verwaltet die Inhalte des Spiels, die *Assets*: 3D-Modelle, Texturen, Schriften, Animationen, Level und Töne. Er importiert sie aus den Werkzeugen der Gestalter, lädt sie bei Bedarf, sorgt dafür, dass jedes Asset nur einmal im Speicher liegt, und gibt den Speicher wieder frei. Ein aktuelles Spiel belegt oft über 100 Gigabyte. Davon ist nur ein verschwindender Teil Programmcode, fast alles sind Assets. Auch deshalb ist Spieleentwicklung ein Medienprojekt und kein reines Softwareprojekt.

Das **Netzwerkmodul** unterstützt Mehrspieler-Spiele. Seine Aufgabe ist es, auf allen beteiligten Rechnern denselben Zustand der Spielwelt sicherzustellen, obwohl Nachrichten verzögert ankommen oder verloren gehen. Davon zu unterscheiden sind Online-Spiele mit Tausenden gleichzeitigen Spielern in einer Welt, die zusätzlich große Serveranlagen brauchen.

Das **Core System** enthält die Hilfsfunktionen, auf denen alle anderen Module aufbauen: Mathematik-Bibliothek, Speicherverwaltung, Dateizugriff, Zufallszahlen, Unterstützung für mehrere Prozessorkerne und Werkzeuge zur Fehlersuche.

Tabelle 2.2 fasst die Module zusammen und nennt die Stelle, an der Sie ihnen in der Unreal Engine begegnen.

Modul	Aufgabe	In der Unreal Engine
Eingabe	Tasten und Knöpfe auf Aktionen des Spiels abbilden	Enhanced Input
KI	glaubhaftes Verhalten computergesteuerter Figuren	Behavior Tree, NavMesh
Scripting	Spielregeln getrennt vom Engine-Code festlegen	Blueprints
Physik	Bewegung nach den Gesetzen der Mechanik, Kollisionserkennung	Chaos, Collision Components
Animation	Figuren bewegen	Animation Blueprint
Sound	Geräusche und Musik abspielen	Sound Cue, MetaSound
Rendering	Szene zeichnen, einschließlich Effekten und GUI	Renderer, Niagara, UMG
Ressourcen	Assets importieren, laden und freigeben	Content Browser
Netzwerk	gemeinsamen Zustand in Mehrspieler-Spielen sichern	Replication
Core System	Hilfsfunktionen für alle Module	Core-Modul

Tabelle 2.2: Die Module einer Game Engine.

2.5 Architektur und Größe

Die Module einer Engine stehen nicht gleichberechtigt nebeneinander. Sie sind in Schichten angeordnet. Ganz unten liegen Hardware, Treiber und Betriebssystem. Darüber folgt eine Schicht, die die Unterschiede zwischen den Plattformen verbirgt, dann das Core System, die Ressourcenverwaltung und die Module aus Abschnitt 2.4. Ganz oben liegt das eigentliche Spiel. Jede Schicht benutzt nur die Schichten unter ihr. Kaum eine Engine erfindet dabei alles neu. Für Teilaufgaben wie Physik oder Sound werden häufig fertige Bibliotheken anderer Hersteller eingebunden.

Wie groß eine Engine wird, zeigt Tabelle 2.3. Der Umfang der Engines von id Software hat sich von einer Generation zur nächsten ungefähr verdoppelt. Die Unreal Engine 4 umfasste

2014 bereits knapp zwei Millionen Zeilen. Das ist derselbe Trend wie beim Linux-Kernel in Kapitel 1.

Engine	Quellcode freigegeben	Codezeilen
id Tech 1	1999	79 000
id Tech 2	2001	138 000
id Tech 3	2005	329 000
id Tech 4	2011	586 000
Unreal Engine 4 (Version 4.6)	2014	1 964 000

Tabelle 2.3: Umfang von Game Engines in Codezeilen (eigene Zählung, nur C- und C++-Dateien).

Mythos und Befund: Mehr Codezeilen bedeuten mehr Leistung

Codezeilen sind ein brauchbares Maß für den Umfang einer Software. Als Maß für Qualität oder für die Produktivität der Entwickler taugen sie nicht, und die Versuche, sie trotzdem so zu verwenden, sind gut dokumentiert. Bei Apple sollten die Entwickler des Lisa-Rechners Anfang 1982 jede Woche auf einem Formular angeben, wie viele Zeilen sie geschrieben hatten. Bill Atkinson hatte in dieser Woche die Routinen für die Bildschirmdarstellung von Grund auf überarbeitet und dabei schneller und deutlich kürzer gemacht. Er trug „−2000“ ein. Nach dem Bericht seines Kollegen Andy Hertzfeld musste er das Formular danach nicht mehr ausfüllen. Edsger Dijkstra schlug 1988 vor, Codezeilen nicht als „produziert“, sondern als „verbraucht“ zu verbuchen, also auf der Kostenseite.

2.6 Welche Engine?

2.6.1 Kaufen oder selbst bauen?

Niemand schreibt zwei Millionen Zeilen freiwillig selbst. Entsprechend selten sind eigene Engines geworden. Von den Spielen, die 2024 auf der Plattform Steam erschienen sind, wurde rund die Hälfte mit Unity gebaut und gut ein Viertel mit der Unreal Engine (Abbildung 2.5). 2012 entstanden noch etwa 70 Prozent der Titel mit einer eigenen Engine.

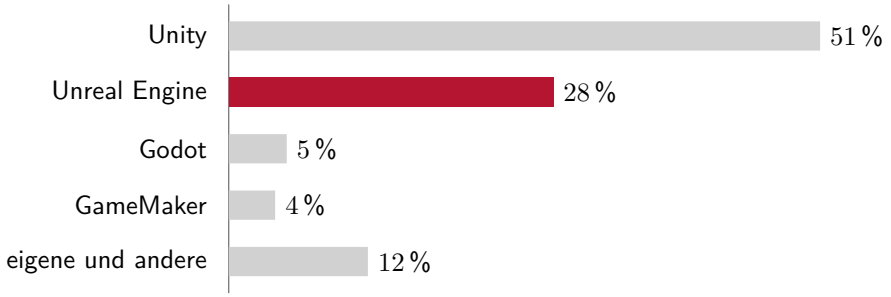


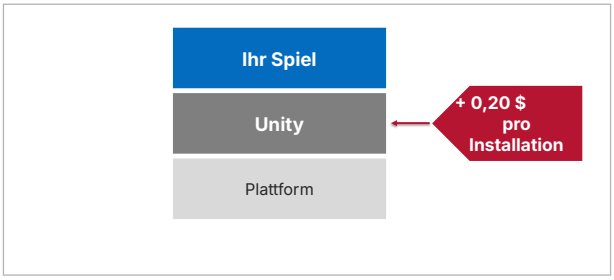
Abbildung 2.5: Anteil der Engines an den Neuerscheinungen auf Steam 2024 (Daten: Video Game Insights 2025).

Beim Umsatz sieht das Bild anders aus. Dort liegen Spiele mit eigener Engine und Spiele mit der Unreal Engine vorn. Viele kleine Spiele entstehen mit Unity, die wenigen sehr großen Produktionen mit einer eigenen Engine oder mit Unreal.

Rund die Hälfte aller neuen Spiele hängt damit an einem einzigen Hersteller. Was das bedeuten kann, zeigte sich im Herbst 2023.

Beispiel: Unitys Runtime Fee (2023)

Am 12. September 2023 kündigte Unity eine neue Gebühr an. Ab Januar 2024 sollten Studios für jede Installation ihres Spiels bis zu 20 US-Cent zahlen, sobald das Spiel in zwölf Monaten mehr als 200 000 Dollar eingenommen hatte und 200 000-mal installiert worden war. Die Regel sollte auch für Spiele gelten, die längst fertig und veröffentlicht waren. Der Protest war heftig. Garry Newman, Chef des Studios Facepunch (*Rust*), schrieb am Tag danach, das Vertrauen sei weg, und kündigte an, das nächste Spiel nicht mehr mit Unity zu entwickeln. Viele Studios überlegten öffentlich, auf andere Engines wie Godot umzusteigen. Unity schwächte die Regel nach gut einer Woche ab, im Oktober trat der Vorstandschef John Riccitiello zurück, und im September 2024 strich sein Nachfolger die Gebühr ganz.
Für viele kleine Studios war das eine unangenehme Erinnerung daran, dass die Wahl einer Engine auch die Wahl eines Geschäftspartners ist, dessen Bedingungen sich ändern können, während das eigene Spiel schon darauf aufgebaut ist.



Liegt es da nicht nahe, die Engine gleich selbst zu bauen? Politowski und Kollegen haben 2021 124 Entwickler quelloffener Engines gefragt, warum sie das tun (Abbildung 2.6). Am häufigsten nannten sie die volle Kontrolle über Umgebung und Quellcode und den Wunsch, zu lernen, wie man eine Engine baut. Dass vorhandene Engines zu wenig können oder zu teuer sind, spielte nur eine kleine Rolle. Bei Hobbyprojekten ist das Selbstbauen also oft eher das Ziel als ein Mittel zum Zweck, was völlig legitim ist, solange man nicht nebenbei auch noch ein Spiel fertigstellen möchte.

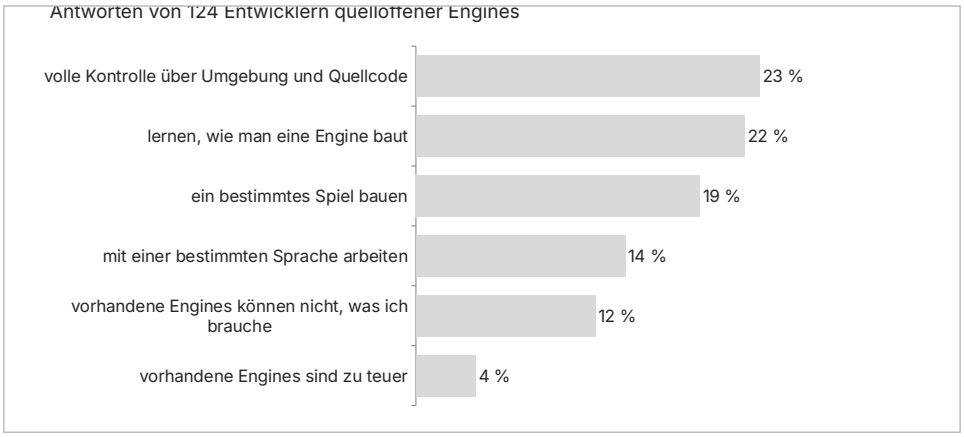


Abbildung 2.6: Gründe für eine eigene Engine aus einer Befragung von 124 Entwicklern quelloffener Engines (Daten: Politowski et al. 2021).

Für eine eigene Engine kann sprechen, dass keine vorhandene die technischen Anforderungen erfüllt, dass die Engine genau zum Spiel passt und keinen Ballast mitbringt und dass man von keinem Hersteller abhängig ist. Dagegen sprechen Kosten und Zeit. Außerdem fehlen einer eigenen Engine zunächst die Werkzeuge, vor allem der Editor, und neue Teammitglieder müssen erst eingearbeitet werden. In fast allen Fällen ist es teurer, selbst zu bauen. Die Frage „selbst bauen oder kaufen“ (*make or buy*) stellt sich im Software Engineering nicht nur bei Engines, sondern bei jeder Komponente.

2.6.2 Unreal, Unity und Godot im Vergleich

Tabelle 2.4 vergleicht die drei Engines, die für kleine Teams in Frage kommen. Die Lizenzbedingungen ändern sich häufig, wie die Runtime Fee gezeigt hat; maßgeblich sind die aktuellen Angaben der Hersteller.

	Unreal Engine	**Unity**	**Godot**
Sprache	C++ und Blueprints	C#	GDScript, C#
Kosten	frei bis 1 Mio. Dollar Umsatz je Spiel, danach 5 % Beteiligung	frei bis 200 000 Dollar Jahresumsatz, danach Abonnement	frei (MIT-Lizenz)
Quellcode	einsehbar	nur gegen Aufpreis	Open Source
Typisch für	große 3D-Produktionen	Handyspiele, kleine Studios	kleine Teams, 2D-Spiele

Tabelle 2.4: Drei verbreitete Game Engines im Vergleich (Stand 2026).

Weitere Kriterien für die Auswahl sind die unterstützten Zielplattformen, der Umfang der mitgelieferten Module, die Qualität der Dokumentation und die Größe der Community, in der man Hilfe findet.

In den Übungen verwenden wir die Unreal Engine. Sie bietet einen grafischen Editor und zugleich vollen Zugriff auf den Quellcode. Programmiert wird in C++, der in der Spieleentwicklung üblichen Sprache, mit der sich die objektorientierten Konzepte der späteren Kapitel direkt anwenden lassen. Und sie lässt sich in vollwertige Entwicklungsumgebungen einbinden.

Merke

Ziel der Übungen ist nicht, eine bestimmte Engine bis ins Detail zu beherrschen. Alle Game Engines sind ähnlich aufgebaut, und wenn Sie das Prinzip an einer verstanden haben, finden Sie sich in der nächsten schnell zurecht.

Im Projektchen

Sehen Sie sich zu Beginn des Projekts an, welche Module aus Tabelle 2.2 Ihr Spiel tatsächlich braucht, und probieren Sie jedes einmal in einem kleinen Testlevel aus. Zwei Dinge sparen später viel Ärger. Fragen Sie im Code keine Tasten ab, sondern Aktionen (in Unreal über *Enhanced Input*), und multiplizieren Sie jede Bewegung mit der Zeit seit dem letzten Frame. Sonst läuft Pac-Man auf dem Rechner mit der besten Grafikkarte schneller, was die Geister vermutlich unfair finden würden.

Zusammenfassung

Das Wichtigste aus Kapitel 2

- Medienmärkte sind groß, aber jede Marktzahl hängt davon ab, was man mitzählt; Spiele sind ein wesentlicher Teil davon.
- Die Geschichte der Game Engines ist eine Geschichte der Wiederverwendung: Mit *Doom* (1993) wurden Technik und Inhalt getrennt, daraus entstanden lizenzierbare Engines.
- Eine Game Engine ist eine Sammlung von Modulen und Schnittstellen, mit der sich ein Team auf Inhalte und Regeln des Spiels konzentrieren kann.
- Die Game Loop liest in jedem Frame die Eingaben, berechnet den neuen Zustand der Spielwelt (Spiellogik, KI, Physik, Animation, Sound) und zeichnet ein Bild.
- Engines sind in Schichten aufgebaut und umfassen Millionen Codezeilen; Codezeilen sagen etwas über den Umfang, nicht über die Qualität.
- Eigene Engines sind selten geworden; die meisten Spiele entstehen mit Unity oder Unreal. Auswahlkriterien sind Sprache, Kosten, Quellcode, Plattformen, Dokumentation und Community.
- Alle Engines sind ähnlich aufgebaut; das Prinzip lässt sich von einer auf die nächste übertragen.

Kontrollfragen

1. Warum lassen sich Zahlen zur Größe eines Marktes nur vergleichen, wenn man weiß, was jeweils mitgezählt wurde?

2. Wie hängen das Geschäftsmodell und die Gestaltung eines Spiels zusammen?

3. Warum ist Wissen über Game Engines auch außerhalb der Spielebranche nützlich?

4. Welche Idee machte aus der Technik eines einzelnen Spiels eine wiederverwendbare Engine, und warum war das wirtschaftlich attraktiv?

5. Was ist eine Game Engine, und was gewinnt ein Team, das eine verwendet, statt alles selbst zu programmieren?

6. Was ist die Game Loop? Nennen Sie die Stationen eines Frames in einer sinnvollen Reihenfolge und begründen Sie die Reihenfolge.

7. Erklären Sie an einem Beispiel, wie mehrere Module einer Engine zusammenarbeiten.

8. Warum trennt eine Engine die gedrückte Taste von der Aktion im Spiel?

9. Welche Vor- und Nachteile hat es, Spielregeln in einer Skriptsprache statt in C++ zu schreiben?

10. Warum ist die Entwicklung eines Spiels eher ein Medienprojekt als ein reines Softwareprojekt?

11. Warum ist eine Engine in Schichten aufgebaut?

12. Nach welchen Kriterien wählt man eine Engine aus? Was spricht für, was gegen eine eigene Engine?

Aufgaben

A1. Pac-Man soll sich mit 8 Feldern pro Sekunde bewegen. Wie weit muss er pro Frame bei 60 und bei 30 Bildern pro Sekunde vorankommen? Ein Programmierer bewegt Pac-Man stattdessen in jedem Frame fest um 8/60 Felder. Wie schnell ist Pac-Man dann auf einem Monitor, der 144 Bilder pro Sekunde darstellt?

A2. Ein Profiler meldet für einen Frame Ihres Spiels: Eingabe 0,2 ms, Spiellogik und KI 2,5 ms, Physik 3,0 ms, Animation 2,0 ms, Sound 0,5 ms, Rendering 9,0 ms. Erreicht das Spiel 60 Bilder pro Sekunde? Wo würden Sie mit dem Optimieren beginnen, und wie viel Zeit stünde bei 120 Bildern pro Sekunde zur Verfügung?

A3. Welche Module der Engine sind an den folgenden Ereignissen in einem Frame beteiligt? (a) Der Spieler drückt auf dem Gamepad nach links. (b) Inky wählt sein Zielfeld. (c) Pac-Man berührt eine Kraftpille. (d) Die Geister werden blau. (e) Ein Ton erklingt. (f) Die Punktzahl ändert sich auf dem Bildschirm.

A4. Welche Engine würden Sie in den folgenden Fällen wählen, und warum? (a) Zwei Studierende wollen in ihrer Freizeit ein kleines 2D-Spiel bauen und kein Geld ausgeben. (b) Ein Team aus erfahrenen C++-Entwicklern baut einen Simulator mit fotorealistischer 3D-Grafik für die Ausbildung in einer Klinik. (c) Ein kleines Studio entwickelt ein Handyspiel und hat bereits Erfahrung mit C#.

A5. In Ihrem Pac-Man stehen das Labyrinth, die Geschwindigkeit der Geister und die Punktwerte fest im C++-Code. Eine Leveldesignerin ohne Programmierkenntnisse soll zehn weitere Level bauen. Was müssen Sie ändern, damit sie das ohne Ihre Hilfe kann? Welche Idee aus der Geschichte der Game Engines wenden Sie damit an, und was kostet Sie der Umbau?

Rückblick

Die folgenden Fragen greifen frühere Kapitel wieder auf. Beantworten Sie sie, ohne zurückzublättern.

R1. Software ist teuer zu entwickeln, aber fast kostenlos zu kopieren. Erklären Sie mit dieser Eigenschaft digitaler Güter, warum es sich für id Software lohnte, die Technik von *Doom* an andere Studios zu lizenzieren.

R2. *Star Citizen* hat über eine Milliarde Dollar eingesammelt und ist nicht fertig. *Concord* war fertig und wurde nach 14 Tagen abgeschaltet. Bei welchem der beiden fehlt der Projekterfolg, bei welchem der Produkterfolg? Zu welchem der beiden Begriffe gehört die Beobachtung aus diesem Kapitel, dass *Concord* Geld kostete, während die Konkurrenz kostenlos spielbar war?

R3. Ein Mitglied Ihrer Gruppe sagt: „In die Unreal Engine arbeite ich mich an einem Wochenende ein.“ Mit welchem Effekt aus Kapitel 1 müssen Sie rechnen, was hat die Studie mit den Abschlussarbeiten dazu gezeigt, und wie kommen Sie zu einer besseren Schätzung?

Lernziele überprüfen

Können Sie ...	**Kontrollfragen**	**Aufgaben**
die wirtschaftliche Bedeutung von Spielen einordnen und erklären, warum Marktzahlen nur mit ihrer Abgrenzung vergleichbar sind?	1, 2, 3	–
die Entstehung von Game Engines aus der Trennung von Technik und Inhalt beschreiben?	4	A5
den Begriff Game Engine definieren?	5	A5
die Stationen der Game Loop und die Aufgaben der Module an einem Frame erklären?	6 bis 11	A1, A2, A3
Kriterien für die Auswahl einer Engine nennen und Kaufen oder Selbstbauen begründen?	12	A4

Zum Weiterlesen

- Jason Gregory: *Game Engine Architecture.* CRC Press. Das Standardwerk zum Aufbau von Engines, von der Game Loop bis zum Renderer.
- Robert Nystrom: *Game Programming Patterns.* Genever Benning, 2014. Auch kostenlos online lesbar; erklärt unter anderem Game Loop und Update-Methode sehr anschaulich.
- Fabien Sanglard: *Game Engine Black Book: Doom.* 2018. Wie *Doom* auf den PCs von 1993 überhaupt laufen konnte, mit vielen Abbildungen.

3 Requirements Engineering

Die erste Phase im Lebenszyklus einer Software ist die Anforderungsanalyse. Sie klärt, *was* gebaut werden soll, bevor irgendjemand darüber nachdenkt, *wie*. Das klingt selbstverständlich, wird in der Praxis aber erstaunlich oft übersprungen, weil sich Programmieren nach Fortschritt anfühlt und Nachdenken nicht. In diesem Kapitel gehen wir vier Fragen nach: Warum braucht man Anforderungen? Was sind Anforderungen überhaupt? Wie findet man sie? Und wie hält man sie so fest, dass alle Beteiligten dasselbe darunter verstehen?

Lernziele

Nach diesem Kapitel können Sie

- begründen, warum Fehler in den Anforderungen besonders teuer sind,
- die Begriffe Anforderung, Stakeholder und Requirements Engineering definieren,
- funktionale und nichtfunktionale sowie harte und weiche Anforderungen unterscheiden und Merkmale nach dem Kano-Modell einordnen,
- Techniken zur Ermittlung von Anforderungen nennen und Szenarien in Tabellenform aufschreiben,
- Anforderungen mit Satzschablone und Sprachregeln prüfbar formulieren, nach MoSCoW priorisieren und ein Begriffslexikon anlegen.

Beispiel: No Man's Sky (2016)

Das kleine britische Studio Hello Games kündigte *No Man's Sky* 2013 an: ein Weltraumspiel mit einem nahezu unendlichen, prozedural erzeugten Universum. In den Jahren bis zur Veröffentlichung sprach Studiochef Sean Murray in vielen Interviews über das Spiel. Dabei entstand bei den Fans ein Bild, das deutlich größer war als das, was das Team bauen konnte, unter anderem die Erwartung, man könne im Universum anderen Spielern begegnen. Als das Spiel im August 2016 erschien, trafen sich zwei Spieler schon in den ersten Tagen gezielt auf demselben Planeten, und keiner konnte den anderen sehen. Die Enttäuschung war groß, die Bewertungen auf Steam stürzten ab, und die britische Werbeaufsicht ASA prüfte nach Beschwerden die Steam-Seite des Spiels. Sie kam im November 2016 zu dem Schluss, dass diese nicht irreführend war. Zu dem Zeitpunkt war der Schaden aber längst angerichtet.

Die Geschichte hat ein bemerkenswertes Ende. Hello Games arbeitete jahrelang weiter und lieferte kostenlose Erweiterungen, 2018 auch einen Mehrspielermodus. Heute gilt das Spiel als eines der bekanntesten Beispiele dafür, wie man ein misslungenes Projekt noch retten kann. Für das Requirements Engineering ist vor allem der Anfang interessant: Was ein Spiel können soll, legen nicht nur das Pflichtenheft und das Team fest, sondern auch die Erwartungen der Menschen, die es kaufen.



Bild: Hello Games (Steam), heutiges Titelbild

3.1 Warum braucht man Anforderungen?

Das Spiel *Concord* aus Kapitel 1 war technisch in Ordnung. Es lief stabil und sah gut aus. Trotzdem wurde es zwei Wochen nach dem Start abgeschaltet, weil kaum jemand es spielen wollte. Man kann ein Produkt richtig bauen und trotzdem das falsche Produkt bauen. Der Fehler lag nicht in der Programmierung, sondern in der Frage, für wen das Spiel gedacht war und warum diese Menschen es spielen sollten.

Dass diese Frage die schwierigste des ganzen Projekts ist, hat Fred Brooks schon 1987 festgehalten:

> *The hardest single part of building a software system is deciding precisely what to build. No other part of the conceptual work is as difficult as establishing the detailed technical requirements [...] No other part of the work so cripples the resulting system if done wrong. No other part is more difficult to rectify later.*
>
> — Fred Brooks, *No Silver Bullet*, 1987

Bevor Sie weiterlesen: Ein Fehler in den Anforderungen wird sofort bemerkt und behoben; das koste eine Einheit. Was kostet es, denselben Fehler zu beheben, wenn er erst nach der Auslieferung auffällt?

Wie schwer sich Fehler in den Anforderungen später korrigieren lassen, zeigt Tabelle 3.1. Ein Fehler, der in den Anforderungen entsteht und sofort entdeckt wird, kostet eine Einheit. Fällt derselbe Fehler erst nach der Auslieferung auf, kostet seine Behebung das Zehn- bis Hundertfache. Der Grund ist einfach: Alles, was inzwischen auf dem Fehler aufgebaut wurde, muss mit geändert werden.

	Fehler wird entdeckt in				
Fehler entsteht in	Anforderungen	Architektur	Entwurf	Test	Betrieb
Anforderungen	1	3	5–10	10	10–100
Architektur	–	1	10	15	25–100
Implementierung	–	–	1	10	10–25

Tabelle 3.1: Relative Kosten für die Behebung eines Fehlers (nach McConnell 2004).

Die Zahlen stammen aus großen Projekten der 1970er- und 1980er-Jahre. In Projekten mit kurzen Entwicklungszyklen fällt der Anstieg deutlich geringer aus. Der Grundgedanke bleibt

aber richtig: Je mehr auf einem Fehler aufbaut, desto teurer wird er.

Woran es in der Praxis hakt, hat eine Befragung von 228 Unternehmen in zehn Ländern untersucht (Abbildung 3.1). Fast jedes zweite Unternehmen nennt unvollständige oder verborgene Anforderungen als Problem, vier von zehn nennen die Kommunikation mit dem Kunden. Auffällig ist, was in der Liste fehlt: Kein einziges der häufigen Probleme ist technischer Natur, alle betreffen die Verständigung zwischen Menschen.

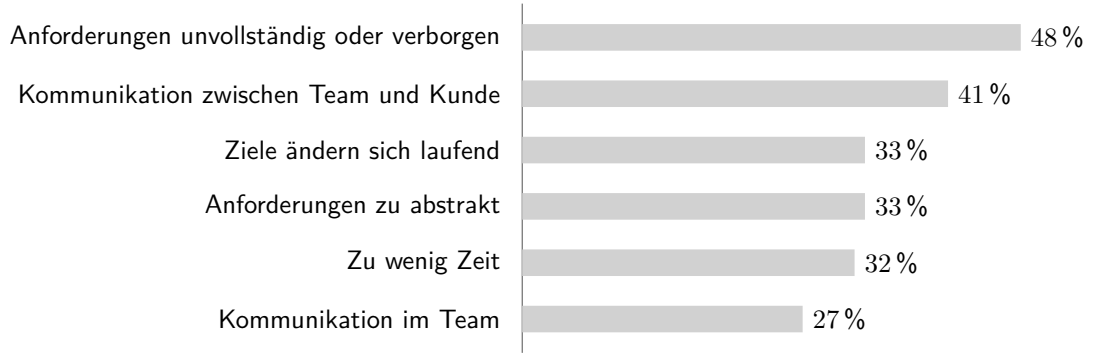


Abbildung 3.1: Die häufigsten Probleme mit Anforderungen: Anteil der befragten Unternehmen, die das Problem nennen (Daten: Méndez Fernández et al. 2017).

Merke

Fehler in den Anforderungen sind die teuersten Fehler eines Projekts, und sie entstehen fast immer durch Missverständnisse, nicht durch Technik.

3.2 Was sind Anforderungen?

Definition 3.1: Anforderung (Requirement)

Eine **Anforderung** ist
(1) eine Bedingung oder Fähigkeit, die ein Benutzer braucht, um ein Problem zu lösen oder ein Ziel zu erreichen,
(2) eine Bedingung oder Fähigkeit, die ein System erfüllen oder besitzen muss, um einen Vertrag, eine Norm oder ein anderes verbindliches Dokument zu erfüllen,
(3) die dokumentierte Darstellung einer Bedingung oder Fähigkeit nach (1) oder (2).
(nach ISO/IEC/IEEE 24765)

Die Definition enthält zwei Sichten: die des Benutzers, der ein Problem lösen will, und die des Systems, das eine Verpflichtung erfüllen muss. Der dritte Punkt ist ebenso wichtig, denn solange eine Anforderung nur in einem Kopf existiert, kann niemand sonst sie prüfen oder sich auf sie berufen.

Definition 3.2: Requirements Engineering

Requirements Engineering ist das systematische Vorgehen, mit dem man von einer Projektidee über Ziele zu einem vollständigen Satz von Anforderungen gelangt. Es umfasst das Ermitteln, Dokumentieren, Prüfen und Verwalten von Anforderungen.

3.2.1 Von wem stammen Anforderungen?

Anforderungen kommen nicht nur von dem, der die Software bezahlt.

Definition 3.3: Stakeholder

Ein **Stakeholder** ist eine Person oder Organisation, die ein Interesse an einem System hat oder von ihm betroffen ist.

Dazu gehören der Auftraggeber, die Benutzer, aber auch Menschen, die das Produkt verkaufen, betreuen oder nur indirekt mit ihm zu tun haben. Bei den Benutzern unterscheidet man primäre (verwenden das Produkt regelmäßig selbst), sekundäre (gelegentliche oder indirekte Benutzer) und tertiäre (von der Einführung betroffen). Bei einem Spiel sind Stakeholder zum Beispiel die Spieler, der Publisher, die Betreiber der Plattformen, auf denen es erscheint, und der Jugendschutz.

Je nachdem, wie eine Anforderung zum Entwicklungsteam gelangt, unterscheidet man drei Arten:

- **Offene Anforderungen** hat der Kunde selbst brauchbar formuliert.
- **Latente Anforderungen** sind dem Kunden nicht bewusst oder so selbstverständlich, dass er sie nicht erwähnt.
- **Entwickler-Optionen** sind Punkte, die der Kunde offengelassen hat, weil sie ihm gleichgültig sind.

Auch latente Anforderungen und Entwickler-Optionen müssen dokumentiert werden. Gerade die latenten sind gefährlich: Dass ein Spiel nicht abstürzt und ohne spürbare Verzögerung auf Eingaben reagiert, sagt niemand, aber jeder erwartet es.

Das **Kano-Modell** (Abbildung 3.2) erklärt, warum das so ist. Es unterscheidet drei Arten von Merkmalen nach ihrer Wirkung auf die Zufriedenheit. **Basismerkmale** werden vorausgesetzt. Fehlen sie, ist der Kunde verärgert; sind sie vorhanden, bemerkt er es nicht. **Leistungsmerkmale** nennt der Kunde von sich aus, und je besser sie erfüllt sind, desto zufriedener ist er. **Begeisterungsmerkmale** erwartet niemand. Fehlen sie, vermisst sie keiner; sind sie da, begeistern sie. Mit der Zeit wird aus Begeisterung allerdings Gewohnheit. Als Spiele anfingen, Spielstände automatisch in der Cloud zu sichern, war das eine angenehme Überraschung. Heute ist man eher verärgert, wenn ein Spiel es nicht tut.

3.2.2 Arten von Anforderungen

Hart und weich. Anforderungen dienen später als Maßstab, an dem die fertige Software geprüft wird. Eine **harte Anforderung** ist entweder erfüllt oder nicht erfüllt, und das lässt sich objektiv feststellen. Ein Beispiel: „Das Programm besteht aus mindestens drei C++-Klassen.“ Eine **weiche Anforderung** kann mehr oder weniger gut erfüllt sein, oft ist sie vage formuliert: „Das Spiel soll Spaß machen.“ In Lehrbüchern sind Anforderungen meist hart, in der Praxis fast immer weich. Ziel der Anforderungsanalyse ist es, weiche Anforderungen so weit wie möglich prüfbar zu machen, etwa so: „Acht von zehn Testspielern wollen nach dem ersten Level weiterspielen.“

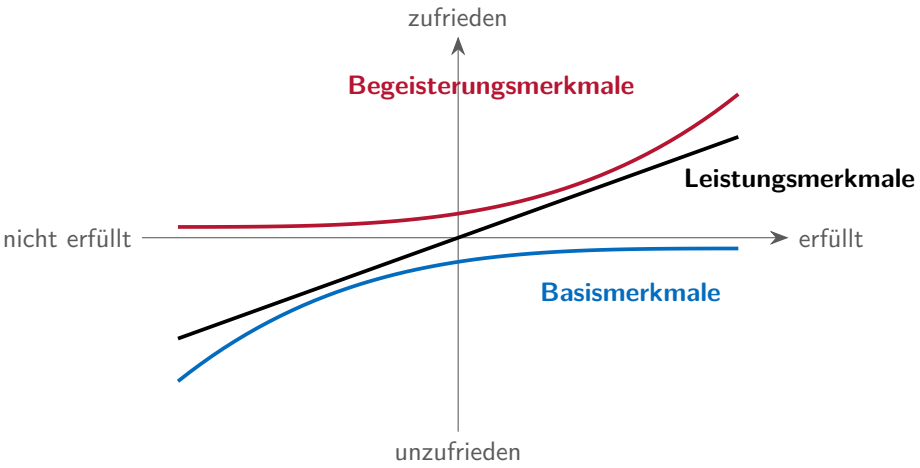


Abbildung 3.2: Das Kano-Modell: Zufriedenheit in Abhängigkeit davon, wie gut eine Anforderung erfüllt ist (nach Kano et al. 1984).

Funktional und nichtfunktional. Die Funktion einer Software ist die Beziehung zwischen ihren Eingaben und Ausgaben. **Funktionale Anforderungen** beschreiben, *was* das System tut. **Nichtfunktionale Anforderungen** beschreiben, *wie gut* es das tut und unter welchen Bedingungen. Heute teilt man die nichtfunktionalen Anforderungen meist noch einmal auf (Tabelle 3.2).

Art	Frage	Beispiel Pac-Man
Funktionale Anforderung	Was tut das System?	Münzen verschwinden bei Berührung und erhöhen den Punktestand.
Qualitätsanforderung	Wie gut tut es das?	Das Spiel reagiert ohne spürbare Verzögerung auf Tastendrücke.
Randbedingung	Was ist vorgegeben?	Unreal Engine und C++; Abgabe in drei Wochen.

Tabelle 3.2: Drei Arten von Anforderungen.

Die Grenze ist unscharf. Robustheit, Bedienbarkeit und Zeitverhalten betreffen eigentlich die Funktion, werden aber meist zu den nichtfunktionalen Anforderungen gezählt, weil sie sich schwer präzise fassen lassen. Aus demselben Grund sind nichtfunktionale Anforderungen oft weich.

3.2.3 Nichtfunktionale Anforderungen und Normen

Nichtfunktionale Anforderungen werden in der Praxis gern vernachlässigt. Sie werden weggelassen oder nur als Schlagwort genannt („benutzerfreundlich“, „performant“). Das wird ihrer Bedeutung nicht gerecht, denn sie entscheiden oft über den Erfolg eines Produkts.

Eine Hilfe sind Normen. ISO/IEC 25010 gliedert die Qualität von Software in Merkmale wie Zuverlässigkeit, Effizienz, Benutzbarkeit, Sicherheit und Wartbarkeit. ISO 9241 behandelt die Ergonomie interaktiver Systeme; Teil 110 nennt Interaktionsprinzipien wie die Aufgabenangemessenheit. Solche Normen sind nicht gesetzlich vorgeschrieben, werden aber von Kunden verlangt. Ihr Nachteil: Sie enthalten kaum harte Vorschriften, die sich einfach prüfen ließen.

Verbindlich geworden ist dagegen die **Barrierefreiheit**. Die *Web Content Accessibility Guidelines* (WCAG) verlangen, dass Inhalte wahrnehmbar, bedienbar, verständlich und robust sind. Für öffentliche Stellen gilt das in Deutschland seit Langem. Seit Juni 2025 verpflichtet das Barrierefreiheitsstärkungsgesetz auch private Anbieter vieler digitaler Produkte und Dienste. Für Spiele gibt es eigene, freiwillige Richtlinien.

Merke

Auch bei nichtfunktionalen Anforderungen gilt: nie auf den Versuch verzichten, präzise zu werden. Abschnitt 3.4.3 zeigt, wie das geht.

3.3 Wie findet man Anforderungen?

Die Anforderungsanalyse untersucht die Bedürfnisse der Benutzer, um daraus Anforderungen an das System abzuleiten. Praktisch bedeutet das, den **Ist-Zustand** zu analysieren und einen **Soll-Zustand** festzulegen. Tabelle 3.3 nennt die wichtigsten Techniken.

Technik	Geeignet für
Vorhandene Daten und Dokumente auswerten	Ist-Zustand
Beobachten	Ist-Zustand: wie Benutzer tatsächlich arbeiten oder spielen
Befragen (Fragebogen, Interview)	Ist- und Soll-Zustand
Szenarien durchspielen	Soll-Zustand: Abläufe, Ausnahmen, Lücken
Prototyp bauen und ausprobieren lassen	Soll-Zustand und Folgen von Neuerungen

Tabelle 3.3: Techniken zur Ermittlung von Anforderungen.

3.3.1 Fragen

Die naheliegendste Technik ist, die Beteiligten zu fragen. Dabei unterscheidet man zwei Arten von Fragen. **Offene Fragen** geben keine Antworten vor. Sie eignen sich, um Neues zu erfahren, die Antworten lassen sich aber schlecht vergleichen. **Geschlossene Fragen** geben die möglichen Antworten vor, häufig als Skala von Zustimmung bis Ablehnung (Likert-Skala). Sie lassen sich gut auszählen, aber die vorgegebenen Antworten können unvollständig sein.

Eine Befragung kann vollstandardisiert sein (Wortlaut und Reihenfolge der Fragen stehen fest), teilstandardisiert (ein Leitfaden mit überwiegend offenen Fragen) oder nicht standardisiert (nur die Themen sind vorgegeben). Für die Formulierung gilt:

- kurze, eindeutige Fragen, die nur einen Sachverhalt ansprechen,
- keine Fachbegriffe, die die Befragten nicht kennen,
- keine doppelten Verneinungen und keine Suggestivfragen,
- nicht zu viele Fragen.

Fragen hat zwei Grenzen. Erstens beschreiben Befragte den Normalfall; nach Ausnahmen muss man gezielt fragen („Und am Wochenende? Und wenn Sie krank sind?“). Zweitens

nennen Kunden vor allem das, was sie am Bestehenden stört. Was am Bestehenden gut ist, nehmen sie erst wahr, wenn es fehlt. Sie erwarten stillschweigend, dass alles Bewährte bleibt. Viele Anforderungen an ein neues System sind deshalb Wünsche nach Kontinuität, und gerade diese werden selten ausgesprochen.

3.3.2 Ausprobieren

Was Menschen wollen, lässt sich schlecht vorhersagen, auch von Fachleuten (siehe den Kasten unten). Deshalb ist es sinnvoll, nicht nur zu fragen, sondern auszuprobieren. Man baut einen **Prototyp** und beobachtet, wie Menschen damit umgehen. Bei Spielen heißt das **Playtest**. Ob ein Spiel Spaß macht, kann niemand im Voraus sagen, auch nicht das Team, das es seit Monaten jeden Tag spielt. Man muss zusehen, wie andere es spielen.

Bevor Sie weiterlesen: Erfahrene Teams bei Microsoft bauen eine neue Funktion, von der sie überzeugt sind, und erproben sie an echten Nutzern. Wie oft verbessert sie die Kennzahl, für die sie gedacht war: in neun von zehn Fällen, in zwei von drei oder in einem von drei?

Mythos und Befund: Wir wissen, was unsere Nutzer wollen

Bei Microsoft werden neue Funktionen von Websites und Diensten in kontrollierten Experimenten erprobt: Ein Teil der Nutzer sieht die neue Version, der Rest die alte, und gemessen wird, ob sich die Zielgröße verbessert. Kohavi, Crook und Longbotham berichteten 2009, dass nur etwa ein Drittel der so getesteten Ideen die Kennzahl verbesserte, für die sie gedacht war. Ein weiteres Drittel hatte keine messbare Wirkung, das letzte verschlechterte sie sogar (Abbildung 3.3). Dabei stammten die Ideen von erfahrenen Leuten, die von ihnen überzeugt waren, sonst wären sie gar nicht erst gebaut worden.

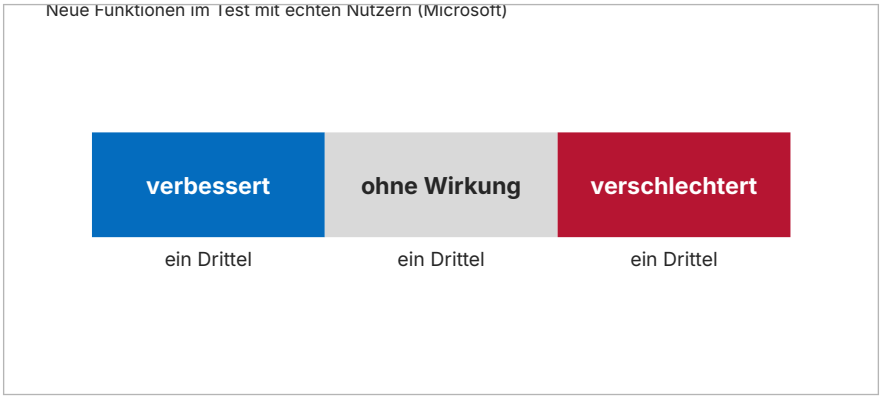


Abbildung 3.3: Wirkung neuer Funktionen im kontrollierten Experiment mit echten Nutzern (Daten: Kohavi, Crook und Longbotham 2009).

3.3.3 Szenarien

Definition 3.4: Szenario

Ein **Szenario** beschreibt ein konkretes Beispiel dafür, wie ein Ziel erfüllt oder nicht erfüllt wird. Es enthält typischerweise eine Folge von Interaktionsschritten (nach Pohl).

Szenarien nutzen aus, dass Menschen mit konkreten Beispielen besser zurechtkommen als mit abstrakten Beschreibungen. Zusammengehörige Szenarien fasst man zu **Anwendungsfällen** (Use Cases) zusammen.

Beispiel: Szenario „Kraftpille“

Der Spieler steuert Pac-Man durch das Labyrinth. Pac-Man frisst eine Kraftpille. Das System färbt alle Geister blau und kehrt ihre Laufrichtung um. Der Spieler steuert Pac-Man auf einen blauen Geist. Das System erkennt die Berührung, entfernt den Geist und zeigt 200 Punkte an. Die Augen des Geistes kehren ins Geisterhaus zurück. Nach einigen Sekunden beginnen die übrigen Geister zu blinken. Das System färbt sie wieder ein, und sie jagen Pac-Man erneut.

Das Szenario macht den Ablauf klar. Es zeigt aber auch, was noch offen ist: Wie lange hält die Wirkung genau? Was geschieht, wenn Pac-Man währenddessen eine zweite Pille frisst? Solche Fragen aufzudecken ist der Zweck eines Szenarios.

Man unterscheidet Szenarien unter anderem danach, was sie zeigen:

- Ein **Hauptszenario** beschreibt den normalen Ablauf, ein **Alternativszenario** eine Abwandlung davon, ein **Ausnahmeszenario** die Reaktion auf ein Ereignis, das den normalen Ablauf verhindert.
- Ein **positives Szenario** zeigt, wie ein Ziel erreicht wird, ein **negatives**, wie es verfehlt wird. Ein **Missbrauchsszenario** beschreibt eine unerwünschte Verwendung, bei einem Online-Spiel etwa das Schummeln.

Für die Dokumentation bringt man ein Szenario in eine feste Form: als nummerierte Folge von Schritten, in der jeder Schritt genau einen Akteur und eine Handlung nennt. In einer Tabelle mit einer Spalte je Akteur wird die Rollenverteilung besonders deutlich (Tabelle 3.4).

Aus Szenarien und Anwendungsfällen entstehen in der objektorientierten Analyse Use-Case- und Sequenzdiagramme.

3.3.4 Schwierigkeiten

Die Analyse scheitert selten an der Technik. Häufiger sind menschliche Ursachen:

- Kunde und Analytiker haben eigene Interessen, die sie nicht offenlegen.
- Der Kunde erwähnt nicht, was ihm selbstverständlich erscheint.
- Wünsche sind nicht machbar oder zu teuer. Dann müssen die Anforderungen gekürzt oder die Mittel aufgestockt werden.

	Spieler	System
1	wählt im Hauptmenü „Spiel starten“	
2		lädt das Labyrinth
3		setzt Pac-Man und die Geister auf ihre Startpositionen
4		zeigt „Ready!“ und spielt die Startmelodie
5	drückt eine Richtungstaste	
6		bewegt Pac-Man in die gewählte Richtung
7	steuert Pac-Man auf eine Münze	
8		entfernt die Münze und erhöht den Punktestand um 10
9		spielt das Fressgeräusch
10	steuert Pac-Man auf einen Geist	
11		zieht ein Leben ab
12		setzt alle Figuren auf ihre Startpositionen zurück

Tabelle 3.4: Ein Szenario in Tabellenform: Hauptszenario (Schritte 1–9) und Ausnahmeszenario (Schritte 10–12).

- Die Wünsche verschiedener Beteiligter widersprechen sich. Das muss offen angesprochen und entschieden werden.
- Entwickler glauben bereits zu wissen, was gebraucht wird, und halten die Analyse für unproduktiv.

Merke

Drei Regeln gelten immer: die Analyse als eigene Tätigkeit ernst nehmen, Zeit und Personal dafür einplanen und die Ergebnisse in eine Form bringen, die auch der Kunde versteht.

3.4 Wie hält man Anforderungen fest?

3.4.1 Lastenheft und Pflichtenheft

Definition 3.5: Lastenheft

Das **Lastenheft** ist die vom Auftraggeber festgelegte Gesamtheit der Forderungen an die Lieferungen und Leistungen eines Auftragnehmers innerhalb eines Auftrags (DIN 69901-5).

Definition 3.6: Pflichtenheft

Das **Pflichtenheft** enthält die vom Auftragnehmer erarbeiteten Realisierungsvorgaben auf der Grundlage des Lastenhefts (DIN 69901-5).

Das Lastenheft schreibt also der Kunde: Was will ich haben, und unter welchen Randbedingungen? Das Pflichtenheft schreibt der Auftragnehmer: Was werde ich liefern? Im Lastenheft sind Lücken und Widersprüche normal. Erst das Pflichtenheft soll vollständig und widerspruchsfrei sein. Der neutrale Oberbegriff für beide ist **Anforderungsspezifikation**: eine systematisch dargestellte Sammlung von Anforderungen.

Ein Pflichtenheft ist im Idealfall

- **zutreffend**: Es gibt die Vorstellungen des Kunden richtig wieder.
- **vollständig**: Es enthält jede Anforderung, die irgendwo existiert.
- **widerspruchsfrei**: Jede Anforderung ist mit allen anderen vereinbar.
- **neutral**: Es schränkt die Lösung nicht stärker ein als nötig.
- **nachvollziehbar**: Die Quelle jeder Anforderung ist dokumentiert, und jede Anforderung ist eindeutig gekennzeichnet.
- **objektivierbar**: Das fertige System lässt sich gegen die Anforderungen prüfen.

Für die Form kommen vier Wünsche hinzu: Das Dokument soll leicht verständlich, präzise, leicht zu erstellen und leicht zu verwalten sein. Diese Wünsche widersprechen sich teilweise. Was präzise ist, ist selten leicht verständlich. Gesucht ist ein Kompromiss.

Die Forderung nach Neutralität wird oft mit „What, not how!“ zusammengefasst: Das Pflichtenheft sagt, was das System leistet, nicht, wie es das tut. In der Praxis lässt sich das nicht vollständig durchhalten, weil vorhandene Komponenten und Vorgaben von außen die Lösung mitbestimmen. Als Ideal bleibt die Forderung trotzdem sinnvoll, denn wenn man anstelle einer Anforderung bereits einen Entwurf aufschreibt, schließt man womöglich die beste Lösung aus, bevor jemand über sie nachgedacht hat.

Tabelle 3.5 zeigt einen möglichen Aufbau. Diese Gliederung verwenden Sie in der Übung.

Abschnitt	**Inhalt**
0. Administrative Daten	Verfasser, Version, Datum, Freigabe
1. Zielbestimmung und Zielgruppen	Wofür und für wen ist das System gedacht? In welcher Umgebung wird es eingesetzt?
2. Funktionale Anforderungen	Funktionen (Szenarien, Use Cases) und Schnittstellen (Bedienoberfläche, andere Software)
3. Nichtfunktionale Anforderungen	Qualitätsanforderungen und Randbedingungen
4. Lieferumfang	Was wird übergeben?
5. Abnahmekriterien	Woran wird entschieden, ob das Ergebnis akzeptiert wird?
6. Anhänge	vor allem das Begriffslexikon

Tabelle 3.5: Möglicher Aufbau eines Lasten- oder Pflichtenhefts.

3.4.2 Game Design Document und User Stories

In der Spielebranche heißt das entsprechende Dokument **Game Design Document**. Es beschreibt Konzept, Spielwelt, Regeln, Steuerung, Zielplattformen, Werkzeuge, Team und

Zeitplan, also dieselben Dinge wie ein Pflichtenheft. Früher waren das umfangreiche Dokumente. Heute ist es meist ein Wiki, das laufend gepflegt wird, ergänzt um eine einzige Seite mit dem Kern des Spiels.

Noch kleiner ist die **User Story**.

Definition 3.7: User Story

Eine **User Story** beschreibt eine Anforderung in einem Satz aus Sicht eines Benutzers, in der Form: *Als* ⟨Rolle⟩ *möchte ich* ⟨Ziel⟩, *um* ⟨Nutzen⟩.

Beispiel: User Story mit Akzeptanzkriterien

Als Spieler *möchte ich* nach einer Kraftpille Geister fressen können, *um* mich für kurze Zeit wehren zu können.
Akzeptanzkriterien:

- Alle Geister werden blau und kehren um.
- Ein gefressener Geist bringt 200 Punkte.
- Nach sechs Sekunden jagen die Geister wieder.

Die Schablone erzwingt drei Angaben: für wen, was und wozu. Die **Akzeptanzkriterien** machen aus der weichen Story etwas Prüfbares. User Stories sind vor allem in agilen Projekten gebräuchlich (Kapitel 9).

3.4.3 Anforderungen formulieren

Anforderungen lassen sich formal (mit mathematischer Notation), grafisch (mit Diagrammen) oder in natürlicher Sprache darstellen. Formale Darstellungen sind präzise, aber für Laien nicht lesbar. Natürliche Sprache versteht jeder, sie ist aber mehrdeutig. In der Praxis überwiegt die natürliche Sprache, ergänzt durch Diagramme. Umso wichtiger sind Regeln für die Formulierung (Tabelle 3.6).

Die letzte Regel entspricht dem „What, not how“: Eine Anforderung nennt das Ziel, nicht den Lösungsweg.

Einfacher wird es, wenn schon der Satzbau die Regeln erzwingt. Dafür gibt es **Satzschablonen** (Abbildung 3.4). Jede Anforderung ist genau ein Satz. Er nennt das System als Handelnden, legt die Verbindlichkeit fest (*muss* für Pflicht, *sollte* für Wunsch, *wird* für eine Absicht) und endet mit einem Objekt und einem aussagekräftigen Verb, dem Prozesswort. Eine Bedingung kann vorangestellt werden.

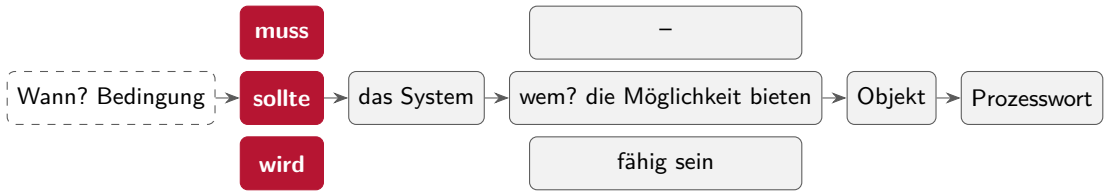


Abbildung 3.4: Satzschablone für Anforderungen (nach Rupp und den SOPHISTen).

Regel	Schlecht	Besser
Prägnant	Mehrere Sätze, die dasselbe sagen	Ein unerfahrener Benutzer soll das System ohne Schulung verwenden können.
Aktiv	Die Dauer der Verarbeitung soll halbiert werden.	Das System verarbeitet die Messdaten doppelt so schnell wie System XY.
Überprüfbar	Das System soll besser sein als das Vorgängersystem.	Das System bietet folgende Verbesserungen gegenüber System XY: ...
Verfeinert	Die Benutzung soll selbsterklärend sein.	Ein durchschnittlicher Nutzer kann nach zwei Minuten folgende Funktionen aufrufen: ...
Mit Mehrwert	Das System soll leicht benutzbar sein.	..., sodass sich die Nutzer auf andere Aufgaben konzentrieren können.
Begründet	Das System soll auch von Ungeschulten benutzbar sein.	..., weil es auch in Mietfahrzeugen eingesetzt wird.
Deklarativ	Durch komprimierte Datenübertragung soll die Antwortzeit um 10 % sinken.	Die Antwortzeit ist 10 % kürzer als bei System XY.

Tabelle 3.6: Sieben Regeln für die Formulierung von Zielen und Anforderungen (nach Pohl).

Beispiel: Anforderung nach Schablone

Nach dem Fressen einer Kraftpille *muss* das Spiel dem Spieler sechs Sekunden lang die Möglichkeit bieten, Geister zu fressen.

Hilfreich ist außerdem ein kritischer Blick auf bestimmte Wörter (Tabelle 3.7) und auf Schlagwörter, hinter denen sich weiche Anforderungen verstecken (Tabelle 3.8).

Regel	Worauf achten?
Im Aktiv formulieren	Wer tut etwas, das System oder der Benutzer? „Wird gelöscht“ lässt das offen.
Vollverben verwenden	„liest“, „erzeugt“ statt „ist“, „hat“; Vollverben verlangen ein Objekt.
Bedingungen vervollständigen	Zu jedem „wenn“ gehört ein „sonst“.
„Alle“, „immer“, „nie“ prüfen	Gilt das wirklich ohne Ausnahme?
Substantivierungen auflösen	„Anmeldung“: Wer meldet sich wo und wofür an?
Unbestimmte Substantive klären	„Der Bediener“: Gibt es nur einen? Welcher ist gemeint?
Implizite Annahmen erkennen	„Die Bestenliste“: Ist festgelegt, dass es eine gibt?

Tabelle 3.7: Sprachregeln für Anforderungen (Auswahl, nach Rupp).

3.4.4 Priorisieren

Am Ende der Analyse stehen meist mehr Anforderungen, als sich in der verfügbaren Zeit umsetzen lassen. Sie müssen nach Wichtigkeit geordnet werden. Ein einfaches Verfahren ist die **MoSCoW-Methode** mit vier Stufen (Tabelle 3.9).

Schlagwort	Bessere Anforderung
einfach bedienbar	Ein neuer Spieler schafft das erste Level ohne Anleitung in höchstens drei Versuchen.
flüssig	Auf den Rechnern im Pool fällt die Bildrate nie unter 60 Bilder pro Sekunde.
robust	Keine Eingabe über Tastatur oder Gamepad führt zu einem Abbruch des Programms.
übersichtlich kommentiert	Jede Klasse hat einen Kopfkommentar, der ihre Aufgabe in einem Satz beschreibt.
nicht zu große Module	Keine Funktion hat mehr als 50 Zeilen.

Tabelle 3.8: Vom Schlagwort zur prüfbaren Anforderung.

Stufe	Bedeutung	Beispiel Pac-Man
Must	Ohne das ist das Produkt wertlos.	Labyrinth, Steuerung, Münzen, ein Geist
Should	Wichtig, aber notfalls verzichtbar.	Vier Geister mit eigenem Verhalten, Kraftpillen
Could	Schön, wenn Zeit bleibt.	Früchte, Bestenliste
Won't	Diesmal ausdrücklich nicht.	Mehrspielermodus

Tabelle 3.9: Priorisierung nach der MoSCoW-Methode.

Die wichtigste Stufe ist die letzte. Indem man ausdrücklich aufschreibt, was man *nicht* baut, schützt man das Projekt davor, immer weiter zu wachsen. Wird die Liste auch nach außen klar kommuniziert, wie es bei *No Man's Sky* vom Kapitelanfang nicht geschehen ist, schützt sie zudem vor Erwartungen, die man nicht erfüllen kann. Das Beispiel *Star Citizen* aus Kapitel 1 zeigt, was ohne diese Stufe geschieht. Die Basismerkmale des Kano-Modells sind immer *Must*.

3.4.5 Das Begriffslexikon

Viele Missverständnisse entstehen, weil Beteiligte dasselbe Wort unterschiedlich verstehen. Deshalb legt man schon während der Analyse ein **Begriffslexikon** (Glossar) an und pflegt es über das ganze Projekt. Hinein gehören Begriffe, die wichtig sind und unterschiedlich ausgelegt werden könnten. Das sind oft gerade die Begriffe, die auf den ersten Blick völlig klar erscheinen.

Tabelle 3.10 zeigt, welche Angaben zu einem Begriff gehören. Die Abgrenzung und die offenen Unklarheiten sind dabei mindestens so wichtig wie die Definition selbst.

> Aus den Einträgen des Begriffslexikons werden in der objektorientierten Analyse Klassen, Attribute und Zustände.

3.4.6 Wozu der Aufwand?

Ein Pflichtenheft wird von vielen gelesen. Der Kunde prüft, ob er richtig verstanden wurde, und nimmt am Ende auf dieser Grundlage das Ergebnis ab. Das Management plant damit Aufwand und Termine. Die Entwickler erfahren, was sie bauen sollen. Die Tester leiten daraus

Begriff	**Geist**, synonym Gegner, Monster
Bedeutung	Eine vom Spiel gesteuerte Figur, die Pac-Man verfolgt und ihn bei Berührung ein Leben kostet.
Abgrenzung	Die Augen eines gefressenen Geistes auf dem Rückweg ins Geisterhaus sind kein Geist; sie können Pac-Man nicht fangen.
Gültigkeit	Ein Geist entsteht beim Start eines Levels im Geisterhaus und existiert bis zum Ende des Levels. Wird er gefressen, entsteht er nach der Rückkehr der Augen neu.
Bezeichnung	Ein Geist ist durch seine Farbe eindeutig bestimmt (rot, rosa, türkis, orange).
Unklarheiten	Noch offen: Dürfen sich zwei Geister gleichzeitig auf demselben Feld aufhalten?
Querverweise	Kraftpille, Geisterhaus, Leben

Tabelle 3.10: Beispiel für einen Eintrag im Begriffslexikon.

ihre Testfälle ab; ohne Anforderungen ist gar nicht festgelegt, welches Verhalten richtig ist. Und wer die Software später wartet oder ersetzt, erfährt, was sie leisten soll.

Fehlt das Dokument, bleiben Anforderungen ungeklärt und werden nicht erfüllt. Die Entwickler nehmen dann ihre eigenen Erwartungen als Maßstab. Bei der Abnahme lässt sich nicht entscheiden, ob das System richtig arbeitet.

Merke

Eine „Spezifikation im Kopf“ gibt es nicht.

Requirements Engineering ist dabei kein einmaliger Schritt. Anforderungen ändern sich während des Projekts, weil Kunden dazulernen, der Markt sich bewegt oder ein Prototyp etwas Unerwartetes zeigt. Sammeln, Ordnen, Priorisieren und Aufschreiben wiederholen sich deshalb. Dazu gehört das **Anforderungsmanagement**: Jede Anforderung erhält eine Kennung, eine Quelle und einen Status, und jede Änderung wird nachvollziehbar festgehalten.

Im Projektchen

Ihr Pflichtenheft ist das erste Dokument, das wir im Projekt von Ihnen sehen wollen. Prüfen Sie jede Anforderung darauf, ob jemand anderes am Ende eindeutig entscheiden könnte, ob sie erfüllt ist. Wo das nicht geht, hilft Tabelle 3.8. Schreiben Sie außerdem eine *Won't*-Liste. Ein Mehrspielermodus, ein Leveleditor und eine Online-Bestenliste sind schöne Ideen, und genau deshalb sollten sie dort stehen. Legen Sie schließlich ein kleines Begriffslexikon an. Spätestens wenn zwei Gruppenmitglieder unter einem „Level“ etwas Verschiedenes verstehen, werden Sie froh darüber sein.

Zusammenfassung

Das Wichtigste aus Kapitel 3

- Man kann ein Produkt richtig bauen und trotzdem das falsche Produkt bauen. Fehler in den Anforderungen sind die teuersten Fehler eines Projekts.
- Requirements Engineering umfasst das Ermitteln, Dokumentieren, Prüfen und Verwalten von Anforderungen; Anforderungen stammen von Stakeholdern.

- Funktionale Anforderungen beschreiben, was das System tut, nichtfunktionale, wie gut und unter welchen Bedingungen. Das Kano-Modell unterscheidet Basis-, Leistungs- und Begeisterungsmerkmale.
- Anforderungen ermittelt man durch Fragen, Beobachten und Ausprobieren sowie mit Szenarien; jede Methode hat Grenzen.
- Das Lastenheft enthält die Forderungen des Auftraggebers, das Pflichtenheft die Umsetzungsvorgaben des Auftragnehmers. In Spielen und agilen Projekten kommen Game Design Document und User Stories hinzu.
- Anforderungen formuliert man präzise und prüfbar, etwa mit Satzschablonen, priorisiert sie (z. B. MoSCoW) und hält die Begriffe in einem Begriffslexikon fest.

Kontrollfragen

1. Warum sind Fehler in den Anforderungen besonders teuer, und warum entstehen sie meist nicht durch Technik?

2. Was ist eine Anforderung? Warum verbindet die Definition die Sicht des Benutzers mit der des Systems?

3. Was ist ein Stakeholder? Nennen Sie vier Stakeholder eines Computerspiels.

4. Erklären Sie offene Anforderungen, latente Anforderungen und Entwickler-Optionen und ordnen Sie ihnen die Merkmale des Kano-Modells zu.

5. Nennen Sie je eine funktionale Anforderung, eine Qualitätsanforderung und eine Randbedingung für ein Spiel Ihrer Wahl. Welche davon sind hart, welche weich?

6. Welche zwei Grenzen hat die Befragung als Technik der Anforderungsermittlung?

7. Was ist ein Szenario? Worin unterscheiden sich Haupt-, Alternativ- und Ausnahmeszenario?

8. Wer schreibt das Lastenheft, wer das Pflichtenheft? Nennen Sie vier Eigenschaften eines guten Pflichtenhefts.

9. Formulieren Sie eine User Story mit zwei Akzeptanzkriterien.

10. Machen Sie aus der Anforderung „Das Spiel soll einfach zu bedienen sein“ eine prüfbare Anforderung.

11. Wozu dient die Stufe *Won't* in der MoSCoW-Methode?

12. Wozu dient ein Begriffslexikon, und was wird später im Projekt aus seinen Einträgen?

13. Wie hängen Anforderungen und Test zusammen?

Aufgaben

A1. Formulieren Sie die folgenden Anforderungen mit der Satzschablone und den Regeln aus Tabelle 3.6 so um, dass sie prüfbar werden: (a) „Das Spiel soll schnell laden.“ (b) „Die Steuerung soll intuitiv sein.“ (c) „Die Bestenliste wird angezeigt.“

A2. Ordnen Sie die folgenden Merkmale eines Pac-Man-Spiels nach dem Kano-Modell ein: (a) Das Spiel lässt sich jederzeit pausieren. (b) Es gibt mehr Level. (c) Die Geister reagieren mit kleinen Sprüchen, wenn man ihnen knapp entkommt. (d) Der Spielstand geht beim Beenden nicht verloren.

A3. Ihnen bleiben drei Wochen. Ordnen Sie nach der MoSCoW-Methode: Labyrinth und Steuerung, Münzen und Punkte, ein Geist, vier Geister mit eigenem Verhalten, Kraftpillen, Früchte, Bestenliste, Soundeffekte, Mehrspielermodus über das Netz, Leveleditor. Begründen Sie die *Won't*-Einträge.

A4. Schreiben Sie einen Eintrag im Begriffslexikon für den Begriff *Level* in der Form von Tabelle 3.10.

A5. Schreiben Sie für „Spiel pausieren" ein Szenario in Tabellenform wie in Tabelle 3.4: ein Hauptszenario und ein Ausnahmeszenario. Notieren Sie die Fragen, die beim Aufschreiben offen geblieben sind.

Rückblick

Die folgenden Fragen greifen frühere Kapitel wieder auf. Beantworten Sie sie, ohne zurückzublättern.

R1. Drei Wochen vor der Abgabe möchte jemand in Ihrer Gruppe zusätzlich einen Leveleditor einbauen. Termin und Gruppe stehen fest. Was folgt daraus nach dem magischen Dreieck, und mit welchem Mittel aus diesem Kapitel halten Sie die Entscheidung fest?

R2. Im Pflichtenheft steht die Qualitätsanforderung „Pac-Man bewegt sich auf jedem Rechner gleich schnell". Mit welcher Technik aus Kapitel 2 erfüllt man sie? Machen Sie aus dem Satz außerdem eine prüfbare Anforderung.

R3. Kapitel 1 begann mit vier Fällen: CrowdStrike, Citibank, *Concord* und *Star Citizen*. Bei welchen hätte besseres Requirements Engineering am ehesten geholfen, und womit genau?

Lernziele überprüfen

Können Sie …	**Kontrollfragen**	**Aufgaben**
begründen, warum Fehler in den Anforderungen besonders teuer sind?	1	–
die Begriffe Anforderung, Stakeholder und Requirements Engineering definieren?	2, 3	–
funktionale und nichtfunktionale, harte und weiche Anforderungen unterscheiden und Merkmale nach Kano einordnen?	4, 5	A2
Techniken zur Ermittlung nennen und Szenarien in Tabellenform aufschreiben?	6, 7	A5
Anforderungen prüfbar formulieren, nach MoSCoW priorisieren und ein Begriffslexikon anlegen?	8 bis 13	A1, A3, A4

Zum Weiterlesen

- Klaus Pohl, Chris Rupp: *Basiswissen Requirements Engineering.* dpunkt.verlag. Kompakt und auf die Zertifizierung zum CPRE ausgerichtet.
- Chris Rupp und die SOPHISTen: *Requirements-Engineering und -Management.* Hanser. Ausführlich und praxisnah, mit vielen Sprachregeln und Schablonen.
- Jesse Schell: *The Art of Game Design. A Book of Lenses.* CRC Press. Wie man herausfindet, was ein Spiel eigentlich sein soll.

A Formelsammlung

Die Formeln und Rechenregeln aus dem Skript auf einen Blick. In Klammern steht jeweils das Kapitel, in dem sie erklärt werden.

Game Engines (Kapitel 2)

Zeit pro Frame	$t_{\text{Frame}} = \dfrac{1000\,\text{ms}}{\text{Bilder pro Sekunde}}$, bei 60 Bildern pro Sekunde also $\approx 16{,}7\,\text{ms}$
Bewegung unabhängig von der Bildrate	$\Delta s = v \cdot \Delta t$ (Weg = Geschwindigkeit × Zeit seit dem letzten Frame)

GUI-Design (Kapitel 4)

Hicksches Gesetz	$T = a + b \cdot \log_2(n+1)$ (n gleich wahrscheinliche Möglichkeiten)
Fitts' Gesetz	$T = a + b \cdot \log_2\left(\dfrac{D}{W} + 1\right)$ (D Entfernung, W Größe des Ziels)
Gefundene Bedienprobleme (Nielsen & Landauer)	$P(n) = 1 - (1-L)^n$ mit $L \approx 0{,}31$; für $n = 5$ also rund 85 %
Antwortzeiten	0,1 s wirkt sofort, 1 s erhält den Gedankengang, ab 10 s wandert die Aufmerksamkeit ab

Testen (Kapitel 8)

Pfade bei n Verzweigungen hintereinander	2^n
Mittelwert	$\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$
Median	mittlerer Wert der sortierten Messwerte; bei gerader Anzahl der Mittelwert der beiden mittleren
Standardabweichung der Stichprobe	$s = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2}$
Standardfehler des Mittelwerts	$\mathrm{SE} = s/\sqrt{n}$
95-%-Konfidenzintervall	$\bar{x} \pm 1{,}96 \cdot \mathrm{SE}$ (Normalverteilung), bei kleinen Stichproben $\bar{x} \pm t_{n-1;\,0{,}975} \cdot \mathrm{SE}$
Breite des Intervalls	schrumpft mit $\sqrt{n}$: halbe Breite erfordert die vierfache Stichprobe

Prozessmodelle (Kapitel 9)

Littles Gesetz	mittlere Durchlaufzeit $= \frac{\text{Aufgaben in Bearbeitung}}{\text{Durchsatz}}$ (jeweils im Mittel)
Aufwand und Dauer beim Pair Programming	Aufwand = Personen × Dauer; zwei Personen mit je 60 % der Zeit verbrauchen 120 % der Personenstunden

Der Mensch im Projekt (Kapitel 10)

Kommunikationsbeziehungen	$C = \frac{N(N-1)}{2}$ (5 Personen: 10, 10 Personen: 45)

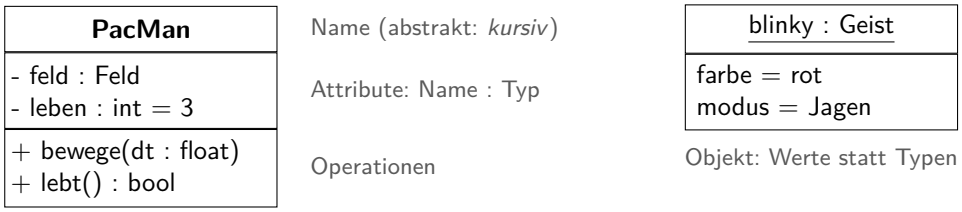
Projektmanagement (Kapitel 11)

Dauer	Dauer = Aufwand / Personaleinsatz (nur bei Arbeit ohne Abstimmung)
Dreipunktschätzung (PERT)	$E = \frac{O + 4M + P}{6}$
Vorwärtsrechnung	FAT = größter FET aller Vorgänger; FET = FAT + Dauer
Rückwärtsrechnung	SET = kleinster SAT aller Nachfolger; SAT = SET − Dauer
Gesamtpuffer	GP = SAT − FAT
Freier Puffer	FP = kleinster FAT aller Nachfolger − FET
Kritischer Pfad	alle Vorgänge mit GP = 0; seine Länge ist die Projektdauer

B UML-Notation auf einen Blick

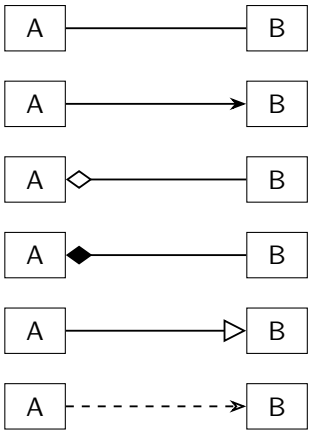
Die Notation der sieben Diagrammarten aus den Kapiteln 5 und 6, auf das Nötigste beschränkt.

Klassen und Objekte



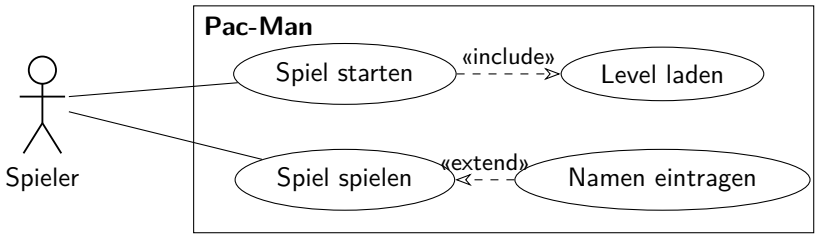
Sichtbarkeit: `+` public, `-` private, `#` protected, `~` package. Multiplizität: `1`, `0..1`, `*`, `1..*`, `4`.

Beziehungen zwischen Klassen



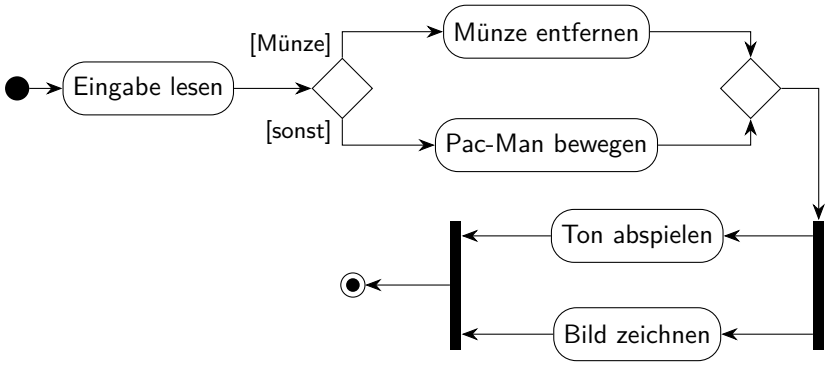
Assoziation	kennen einander
gerichtete Assoziation	A kennt B
Aggregation	B ist Teil von A und kann allein existieren
Komposition	B gehört zu genau einem A und stirbt mit ihm
Generalisierung	A ist ein B (B ist die Oberklasse)
Abhängigkeit	A benutzt B vorübergehend

Anwendungsfalldiagramm



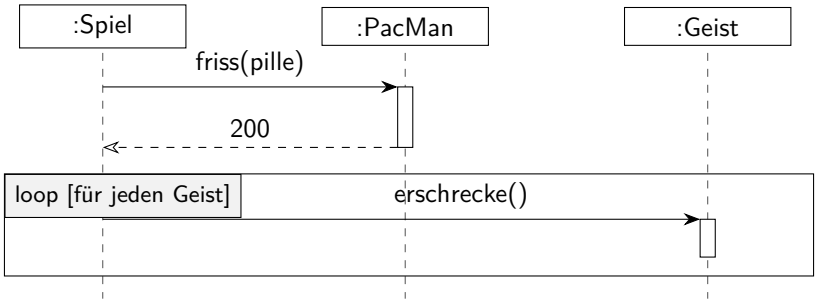
`«include»`: immer enthalten, Pfeil zum enthaltenen Fall. `«extend»`: unter einer Bedingung, Pfeil zum erweiterten Fall.

Aktivitätsdiagramm



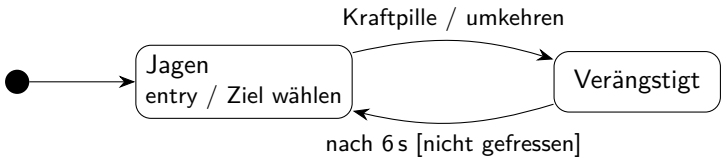
Punkt: Start, Punkt im Kreis: Ende. Raute mit einem Eingang: Entscheidung mit Bedingungen in eckigen Klammern; Raute mit mehreren Eingängen: Zusammenführung. Balken mit einem Eingang: Gabelung in parallele Zweige; Balken mit mehreren Eingängen: Vereinigung, die auf alle Zweige wartet.

Sequenzdiagramm



Lebenslinie gestrichelt, Aktivität als Balken, Nachricht als durchgezogener Pfeil, Antwort gestrichelt. Fragmente: `loop`, `alt`, `opt`. Kommunikationsdiagramm: dieselben Nachrichten, nummeriert (1, 2, 3.1, ...) an den Links zwischen Objekten.

Zustandsdiagramm



Übergang: `Ereignis [Bedingung] / Aktion`. Im Zustand: `entry /`, `do /`, `exit /`. Unterzustände in einem umschließenden Zustand; ein Übergang vom äußeren Zustand gilt für alle Unterzustände.

Literatur

Hier stehen die Studien, Bücher und Berichte, auf die sich das Skript stützt. Im Text werden sie mit Autor und Jahr genannt. Lesetipps zum Einstieg finden Sie am Ende jedes Kapitels.

ACM: *ACM Code of Ethics and Professional Conduct.* Association for Computing Machinery, 2018.

Adams, J. S.: Inequity in Social Exchange. In: Berkowitz, L. (Hrsg.): *Advances in Experimental Social Psychology* 2, Academic Press, 1965.

Alderfer, C. P.: An Empirical Test of a New Theory of Human Needs. *Organizational Behavior and Human Performance* 4(2), 1969.

Anderson, D. J.: *Kanban. Successful Evolutionary Change for Your Technology Business.* Blue Hole Press, 2010.

Aranda, J.; Easterbrook, S.: Anchoring and Adjustment in Software Estimation. *Proceedings of ESEC/FSE*, 2005.

Bacchelli, A.; Bird, C.: Expectations, Outcomes, and Challenges of Modern Code Review. *Proceedings of the International Conference on Software Engineering (ICSE)*, 2013.

Basili, V. R.; Turner, A. J.: Iterative Enhancement: A Practical Technique for Software Development. *IEEE Transactions on Software Engineering* SE-1(4), 1975.

Bass, B. M.: *Leadership and Performance Beyond Expectations.* Free Press, 1985.

Bastani, H.; Bastani, O.; Sungu, A.; Ge, H.; Kabakcı, Ö.; Mariman, R.: Generative AI without Guardrails Can Harm Learning: Evidence from High School Mathematics. *Proceedings of the National Academy of Sciences* 122(26), 2025.

Beck, K.: *Extreme Programming Explained.* Addison-Wesley, 1999.

Beck, K.: *Test-Driven Development: By Example.* Addison-Wesley, 2002.

Becker, J.; Rush, N.; Barnes, E.; Rein, D.: Measuring the Impact of Early-2025 AI on Experienced Open-Source Developer Productivity. METR, arXiv:2507.09089, 2025.

Belbin, R. M.: *Management Teams. Why They Succeed or Fail.* Heinemann, 1981.

Bell, T. E.; Thayer, T. A.: Software Requirements: Are They Really a Problem? *Proceedings of ICSE*, 1976.

Bentley, J.: Programming Pearls: Bumper-Sticker Computer Science. *Communications of the ACM* 28(9), 1985.

Berkowitz, L.: Group Standards, Cohesiveness, and Productivity. *Human Relations* 7(4), 1954.

Bettenburg, N.; Just, S.; Schröter, A.; Weiss, C.; Premraj, R.; Zimmermann, T.: What Makes a Good Bug Report? *Proceedings of FSE*, 2008.

Blake, R. R.; Mouton, J. S.: *The Managerial Grid.* Gulf Publishing, 1964.

Boehm, B. W.: Software Engineering. *IEEE Transactions on Computers* C-25(12), 1976.

Boehm, B. W.: Guidelines for Verifying and Validating Software Requirements and Design Specifications. *Proceedings of EURO IFIP*, 1979.

Boehm, B. W.: A Spiral Model of Software Development and Enhancement. *IEEE Computer* 21(5), 1988.

Boehm, B. W.: Software Risk Management: Principles and Practices. *IEEE Software* 8(1), 1991.

Borges, J. L.: Del rigor en la ciencia. 1946; auch in: *El hacedor.* Emecé, 1960.

Bossavit, L.: *The Leprechauns of Software Engineering.* Leanpub, 2012.

Bridgman, T.; Cummings, S.; Ballard, J.: Who Built Maslow's Pyramid? *Academy of Management Learning & Education* 18(1), 2019.

Brooks, F. P.: *The Mythical Man-Month. Essays on Software Engineering.* Addison-Wesley, 1975; Jubiläumsausgabe 1995.

Brooks, F. P.: No Silver Bullet. Essence and Accidents of Software Engineering. *IEEE Computer* 20(4), 1987.

Buehler, R.; Griffin, D.; Ross, M.: Exploring the „Planning Fallacy": Why People Underestimate Their Task Completion Times. *Journal of Personality and Social Psychology* 67(3), 1994.

Burns, J. M.: *Leadership.* Harper & Row, 1978.

Cámara, J.; Troya, J.; Burgueño, L.; Vallecillo, A.: On the Assessment of Generative AI in Modeling Tasks: An Experience Report with ChatGPT and UML. *Software and Systems Modeling* 22, 2023.

Conway, M. E.: How Do Committees Invent? *Datamation*, April 1968.

Cowan, N.: The Magical Number 4 in Short-Term Memory. *Behavioral and Brain Sciences* 24(1), 2001.

CrowdStrike: *External Technical Root Cause Analysis – Channel File 291.* 2024.

Cusumano, M.; MacCormack, A.; Kemerer, C. F.; Crandall, B.: Software Development Worldwide: The State of the Practice. *IEEE Software* 20(6), 2003.

Deci, E. L.: Effects of Externally Mediated Rewards on Intrinsic Motivation. *Journal of Personality and Social Psychology* 18(1), 1971.

DeMarco, T.; Lister, T.: *Peopleware. Productive Projects and Teams.* Dorset House, 1987; 3. Auflage Addison-Wesley, 2013.

DIN 69901-5:2009: *Projektmanagement – Projektmanagementsysteme – Teil 5: Begriffe.* Beuth, 2009.

Dijkstra, E. W.: Notes on Structured Programming. EWD 249, 1970.

Dijkstra, E. W.: The Humble Programmer. *Communications of the ACM* 15(10), 1972.

Dijkstra, E. W.: On the Cruelty of Really Teaching Computing Science. EWD 1036, 1988.

Doran, G. T.: There's a S.M.A.R.T. Way to Write Management's Goals and Objectives. *Management Review* 70(11), 1981.

Dzidek, W. J.; Arisholm, E.; Briand, L. C.: A Realistic Empirical Evaluation of the Costs and Benefits of UML in Software Maintenance. *IEEE Transactions on Software Engineering* 34(3), 2008.

Edmondson, A. C.: Psychological Safety and Learning Behavior in Work Teams. *Administrative Science Quarterly* 44(2), 1999.

Eveleens, J. L.; Verhoef, C.: The Rise and Fall of the Chaos Report Figures. *IEEE Software* 27(1), 2010.

Fagerholt, E.; Lorentzon, M.: *Beyond the HUD. User Interfaces for Increased Player Immersion in FPS Games.* Masterarbeit, Chalmers University of Technology, 2009.

Fairley, R. E.: *Software Engineering Concepts.* McGraw-Hill, 1985.

Fisher, R.; Ury, W.: *Getting to Yes. Negotiating Agreement Without Giving In.* Houghton Mifflin, 1981. Deutsch: *Das Harvard-Konzept.*

Fitts, P. M.: The Information Capacity of the Human Motor System in Controlling the Amplitude of Movement. *Journal of Experimental Psychology* 47(6), 1954.

Flyvbjerg, B.; Gardner, D.: *How Big Things Get Done.* Currency, 2023.

Flyvbjerg, B. et al.: The Empirical Reality of IT Project Cost Overruns: Discovering a Power-Law Distribution. *Journal of Management Information Systems* 39(3), 2022.

Fowler, M.: Mocks Aren't Stubs. martinfowler.com, 2007.

Fowler, M.: *Refactoring. Improving the Design of Existing Code.* Addison-Wesley, 1999.

Fucci, D.; Erdogmus, H.; Turhan, B.; Oivo, M.; Juristo, N.: A Dissection of the Test-Driven Development Process: Does It Really Matter to Test-First or to Test-Last? *IEEE Transactions on Software Engineering* 43(7), 2017.

Furnham, A.; Steele, H.; Pendleton, D.: A Psychometric Assessment of the Belbin Team-Role Self-Perception Inventory. *Journal of Occupational and Organizational Psychology* 66(3), 1993.

Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J.: *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1994.

Gerard, H. B.; Mathewson, G. C.: The Effects of Severity of Initiation on Liking for a Group: A Replication. *Journal of Experimental Social Psychology* 2(3), 1966.

Gesellschaft für Informatik: *Ethische Leitlinien.* Beschlossen am 29. Juni 2018.

Gilb, T.: *Principles of Software Engineering Management.* Addison-Wesley, 1988.

Gladwin, T.: Culture and Logical Process. In: Goodenough, W. H. (Hrsg.): *Explorations in Cultural Anthropology.* McGraw-Hill, 1964.

Glasl, F.: *Konfliktmanagement. Ein Handbuch für Führungskräfte, Beraterinnen und Berater.* Haupt und Freies Geistesleben, zahlreiche Auflagen.

Gold, J.: *Object-Oriented Game Development.* Addison-Wesley, 2004.

Grenning, J.: Planning Poker or How to Avoid Analysis Paralysis while Release Planning. Renaissance Software Consulting, 2002.

Hannay, J. E.; Dybå, T.; Arisholm, E.; Sjøberg, D. I. K.: The Effectiveness of Pair Programming: A Meta-Analysis. *Information and Software Technology* 51(7), 2009.

Harel, D.: Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming* 8(3), 1987.

Hersey, P.; Blanchard, K. H.: *Management of Organizational Behavior.* Prentice Hall, 1969.

Hertzfeld, A.: -2000 Lines of Code. In: *Revolution in the Valley.* O'Reilly, 2004; auch online auf folklore.org.

Hick, W. E.: On the Rate of Gain of Information. *Quarterly Journal of Experimental Psychology* 4(1), 1952.

Huseman, R. C.; Hatfield, J. D.; Miles, E. W.: A New Perspective on Equity Theory: The Equity Sensitivity Construct. *Academy of Management Review* 12(2), 1987.

IEEE: *IEEE Standard Glossary of Software Engineering Terminology.* IEEE Std 610.12-1990.

Ingham, A. G.; Levinger, G.; Graves, J.; Peckham, V.: The Ringelmann Effect: Studies of Group Size and Group Performance. *Journal of Experimental Social Psychology* 10(4), 1974.

INSAG (International Nuclear Safety Advisory Group): *The Chernobyl Accident: Updating of INSAG-1.* Safety Series 75-INSAG-7, IAEA, 1992.

Isla, D.: Handling Complexity in the Halo 2 AI. Vortrag, Game Developers Conference, 2005.

ISO/IEC/IEEE 24765: *Systems and Software Engineering – Vocabulary.*

ISTQB: *Standardglossar der Testbegriffe*, deutsche Fassung, German Testing Board.

Judge, T. A.; Piccolo, R. F.: Transformational and Transactional Leadership: A Meta-Analytic Test of Their Relative Validity. *Journal of Applied Psychology* 89(5), 2004.

Kano, N.; Seraku, N.; Takahashi, F.; Tsuji, S.: Attractive Quality and Must-Be Quality. *Journal of the Japanese Society for Quality Control* 14(2), 1984.

Karau, S. J.; Williams, K. D.: Social Loafing: A Meta-Analytic Review and Theoretical Integration. *Journal of Personality and Social Psychology* 65(4), 1993.

Klein, G.: Performing a Project Premortem. *Harvard Business Review* 85(9), S. 18–19, 2007.

Klimek, P.; Hanel, R.; Thurner, S.: Parkinson's Law Quantified: Three Investigations on Bureaucratic Inefficiency. *Journal of Statistical Mechanics*, P03008, 2009.

Knuth, D. E.: Structured Programming with go to Statements. *Computing Surveys* 6(4), 1974.

Kohavi, R.; Crook, T.; Longbotham, R.: Online Experimentation at Microsoft. *Data Mining Case Studies Workshop, KDD*, 2009.

Kravitz, D. A.; Martin, B.: Ringelmann Rediscovered: The Original Article. *Journal of Personality and Social Psychology* 50(5), 1986.

Latané, B.; Williams, K.; Harkins, S.: Many Hands Make Light the Work: The Causes and Consequences of Social Loafing. *Journal of Personality and Social Psychology* 37(6), 1979.

Laudon, K. C.; Laudon, J. P.; Schoder, D.: *Wirtschaftsinformatik. Eine Einführung.* Pearson.

Law, E. L.-C.; Brühlmann, F.; Mekler, E. D.: Systematic Review and Validation of the Game Experience Questionnaire (GEQ). *Proceedings of CHI PLAY*, 2018.

Leveson, N. G.; Turner, C. S.: An Investigation of the Therac-25 Accidents. *IEEE Computer* 26(7), 1993.

Leventhal, G. S.: What Should Be Done with Equity Theory? In: Gergen, K. J.; Greenberg, M. S.; Willis, R. H. (Hrsg.): *Social Exchange.* Plenum, 1980.

Lewin, K.; Lippitt, R.; White, R. K.: Patterns of Aggressive Behavior in Experimentally Created „Social Climates". *Journal of Social Psychology* 10(2), 1939.

Lions, J.-L. et al.: *Ariane 5. Flight 501 Failure. Report by the Inquiry Board.* ESA/CNES, 1996.

Little, J. D. C.: A Proof for the Queuing Formula $L = \lambda W$. *Operations Research* 9(3), 1961.

Lopez et al.: Visual Cues and Food Intake: A Preregistered Replication of Wansink et al. (2005). *Journal of Experimental Psychology: General*, 2024.

Ludewig, J.; Lichter, H.: *Software Engineering. Grundlagen, Menschen, Prozesse, Techniken.* dpunkt, 3. Auflage 2013.

Mars Climate Orbiter Mishap Investigation Board: *Phase I Report.* NASA, 1999.

Maslow, A. H.: A Theory of Human Motivation. *Psychological Review* 50(4), 1943.

Mayer, R. E.: *Multimedia Learning.* Cambridge University Press, 2001.

McConnell, S.: *Code Complete.* 2. Auflage, Microsoft Press, 2004.

Mehrabian, A.; Ferris, S. R.: Inference of Attitudes from Nonverbal Communication in Two Channels. *Journal of Consulting Psychology* 31(3), 1967.

Mehrabian, A.; Wiener, M.: Decoding of Inconsistent Communications. *Journal of Personality and Social Psychology* 6(1), 1967.

Menabrea, L. F.: Sketch of the Analytical Engine Invented by Charles Babbage. Übersetzt und mit Anmerkungen von A. A. Lovelace. *Scientific Memoirs* 3, 1843.

Méndez Fernández, D.; Wagner, S.; Kalinowski, M. et al.: Naming the Pain in Requirements Engineering: Contemporary Problems, Causes, and Effects in Practice. *Empirical Software Engineering* 22(5), 2017.

METR: We Are Changing Our Developer Productivity Experiment Design. Bericht vom 24. Februar 2026, metr.org.

Meyer, B.: *Object-Oriented Software Construction.* Prentice Hall, 1988.

Miller, G. A.: The Magical Number Seven, Plus or Minus Two. *Psychological Review* 63(2), 1956.

Miller, R. B.: Response Time in Man-Computer Conversational Transactions. *Proceedings of the AFIPS Fall Joint Computer Conference*, 1968.

Mitchell, D. J.; Russo, J. E.; Pennington, N.: Back to the Future: Temporal Perspective in the Explanation of Events. *Journal of Behavioral Decision Making* 2(1), S. 25–38, 1989.

Myers, G. J.: *The Art of Software Testing.* Wiley, 1979.

Nagappan, N.; Maximilien, E. M.; Bhat, T.; Williams, L.: Realizing Quality Improvement through Test Driven Development: Results and Experiences of Four Industrial Teams. *Empirical Software Engineering* 13(3), 2008.

Naur, P.; Randell, B. (Hrsg.): *Software Engineering. Report on a Conference Sponsored by the NATO Science Committee, Garmisch 1968.* NATO, 1969.

Nielsen, J.: *Usability Engineering.* Academic Press, 1993.

Nielsen, J.; Landauer, T. K.: A Mathematical Model of the Finding of Usability Problems. *Proceedings of INTERCHI*, 1993.

Norman, D. A.: *The Design of Everyday Things.* Basic Books, 1988; überarbeitete Ausgabe 2013.

Nunes, J. C.; Drèze, X.: The Endowed Progress Effect: How Artificial Advancement Increases Effort. *Journal of Consumer Research* 32(4), 2006.

Opdyke, W. F.: *Refactoring Object-Oriented Frameworks.* Dissertation, University of Illinois at Urbana-Champaign, 1992.

Parnas, D. L.: On the Criteria to Be Used in Decomposing Systems into Modules. *Communications of the ACM* 15(12), 1972.

Peng, S.; Kalliamvakou, E.; Cihon, P.; Demirer, M.: The Impact of AI on Developer Productivity: Evidence from GitHub Copilot. arXiv:2302.06590, 2023.

Pencavel, J.: The Productivity of Working Hours. *The Economic Journal* 125(589), 2015.

Petre, M.: UML in Practice. *Proceedings of ICSE*, 2013.

Petri, C. A.: *Kommunikation mit Automaten.* Dissertation, 1962.

Pittman, J.: *The Pac-Man Dossier.* 2009, online.

Politowski, C.; Petrillo, F.; Montandon, J. E.; Valente, M. T.; Guéhéneuc, Y.-G.: Are Game Engines Software Frameworks? A Three-perspective Study. *Journal of Systems and Software* 171, 2021.

Politowski, C.; Petrillo, F.; Ullmann, G. C.; Guéhéneuc, Y.-G.: Game Industry Problems: An Extensive Analysis of the Gray Literature. *Information and Software Technology* 134, 2021.

Pohl, K.; Rupp, C.: *Basiswissen Requirements Engineering.* dpunkt.verlag.

Roediger, H. L.; Karpicke, J. D.: Test-Enhanced Learning: Taking Memory Tests Improves Long-Term Retention. *Psychological Science* 17(3), 2006.

Royce, W. W.: Managing the Development of Large Software Systems. *Proceedings of IEEE WESCON*, 1970.

Rupp, C. und die SOPHISTen: *Requirements-Engineering und -Management.* Hanser.

Ryan, R. M.; Rigby, C. S.; Przybylski, A.: The Motivational Pull of Video Games: A Self-Determination Theory Approach. *Motivation and Emotion* 30(4), 2006.

Sana, F.; Forrin, N. D.; Sharma, M.; Dubljevic, T.; Ho, P.; Jalil, E.; Kim, J. A.: Optimizing the Efficacy of Learning Objectives through Pretests. *CBE – Life Sciences Education* 19(3), 2020.

Schreier, J.: How BioWare's *Anthem* Went Wrong. *Kotaku*, April 2019.

Schulz von Thun, F.: *Miteinander reden 1. Störungen und Klärungen.* Rowohlt, 1981.

Schwaber, K.; Sutherland, J.: *The Scrum Guide.* 2020.

Shen, J. H.; Tamkin, A.: How AI Impacts Skill Formation. Anthropic, arXiv:2601.20245, 2026.

Sommerville, I.: *Software Engineering.* 10. Auflage, Pearson.

Stevens, W. P.; Myers, G. J.; Constantine, L. L.: Structured Design. *IBM Systems Journal* 13(2), 1974.

Stogdill, R. M.: Personal Factors Associated with Leadership: A Survey of the Literature. *Journal of Psychology* 25, 1948.

Suchman, L. A.: *Plans and Situated Actions. The Problem of Human-Machine Communication.* Cambridge University Press, 1987.

Takeuchi, H.; Nonaka, I.: The New New Product Development Game. *Harvard Business Review* 64(1), 1986.

Thomas, K. W.; Kilmann, R. H.: *Thomas-Kilmann Conflict Mode Instrument.* Xicom, 1974.

Thompson, G.; Vecchio, R. P.: Situational Leadership Theory: A Test of Three Versions. *The Leadership Quarterly* 20(5), 2009.

Tokey, S. S.; Boudaoud, B.; Kim, J.; Spjut, J.; Claypool, M.: Pushing the Limits? Frame Rate Benefits to Players for up to 500 Hz in First Person Shooter Games. *Proceedings of NOSSDAV*, 2025.

Tuckman, B. W.: Developmental Sequence in Small Groups. *Psychological Bulletin* 63(6), 1965.

Tuckman, B. W.; Jensen, M. A. C.: Stages of Small-Group Development Revisited. *Group & Organization Studies* 2(4), 1977.

van Knippenberg, D.; Sitkin, S. B.: A Critical Assessment of Charismatic-Transformational Leadership Research: Back to the Drawing Board? *The Academy of Management Annals* 7(1), 2013.

Wansink, B.; Painter, J. E.; North, J.: Bottomless Bowls: Why Visual Cues of Portion Size May Influence Intake. *Obesity Research* 13(1), 2005.

Washburn, M.; Sathiyanarayanan, P.; Nagappan, M.; Zimmermann, T.; Bird, C.: What Went Right and What Went Wrong: An Analysis of 155 Postmortems from Game Development. *Proceedings of ICSE*, 2016.

Watzlawick, P.; Beavin, J. H.; Jackson, D. D.: *Menschliche Kommunikation.* Huber, 1969 (Original 1967).

WHO: *Global Report on Health Equity for Persons with Disabilities.* World Health Organization, 2022.

Williams, L.; Kessler, R. R.; Cunningham, W.; Jeffries, R.: Strengthening the Case for Pair Programming. *IEEE Software* 17(4), 2000.

Xia, X.; Bao, L.; Lo, D.; Xing, Z.; Hassan, A. E.; Li, S.: Measuring Program Comprehension: A Large-Scale Field Study with Professionals. *IEEE Transactions on Software Engineering* 44(10), 2018.

Stichwortverzeichnis

Akzeptanzkriterium, 48
Alternativszenario, 45
Anforderung, 40
Anforderungsmanagement, 51
Anforderungsspezifikation, 47
Assets, 31
Ausnahmeszenario, 45

Barrierefreiheit, 43
Basismerkmal, 41
Begeisterungsmerkmal, 41
Begriffslexikon, 50
Behavior Tree, 29
Blueprints, 29

Core System, 31
Culling, 30

Digitales Gut, 13
Disintermediation, 14
Dynamische Preisgestaltung, 14

Entwickler-Option, 41

Feature Creep, 8
Frame, 27
Funktionale Anforderung, 42

Game Design Document, 47
Game Engine, 27
Game Loop, 27
Geschlossene Frage, 43
Git, 18

Harte Anforderung, 41
Hauptszenario, 45
Head-up-Display, 30

Ist-Zustand, 43

Kano-Modell, 41
Kollisionserkennung, 29

Lastenheft, 46
Latente Anforderung, 41
Leistungsmerkmal, 41

Magisches Dreieck, 9
Make or buy, 34
Media Engineering, 19
Meilenstein, 17
Missbrauchsszenario, 45
MoSCoW-Methode, 49

Netzwerkeffekt, 13
Netzwerkmodul, 31
Nichtfunktionale Anforderung, 42

Objektorientierte Programmierung, 18
Offene Anforderung, 41
Offene Frage, 43

Pflichtenheft, 46
Planungsfehlschluss, 12
Playtest, 44
Postmortem, 14
Preisdifferenzierung, 13
Produkterfolg, 10
Projekterfolg, 9
Prototyp, 44

Renderer, 30
Requirements Engineering, 8, 40
Ressourcen-Manager, 31

Satzschablone, 48
Scripting, 29
Skelettanimation, 29
Software, 12
Software Development Lifecycle, 20
Software Engineering, 16
Softwarekrise, 15
Soll-Zustand, 43
Stakeholder, 41
Strukturierte Programmierung, 18
Szenario, 45
Szenengraph, 30

User Story, 48

Virtual Production, 26

Weiche Anforderung, 41