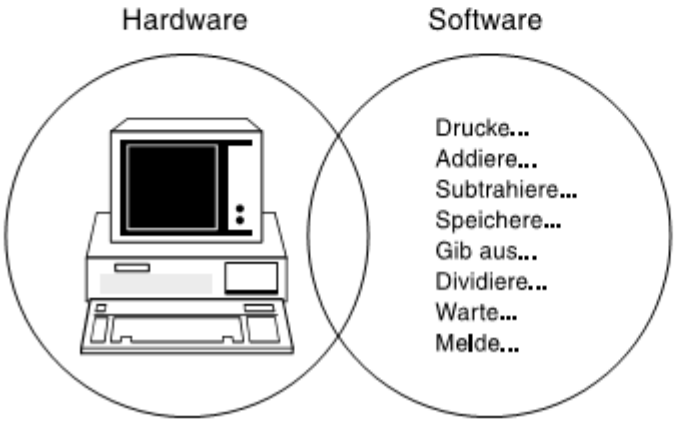
Moderne Computeranlagen bestehen immer aus zwei Komponenten:

1. der **Hardware**[3] und
2. der **Software**[4].

Die **Hardware** umfasst sämtliche Geräte der elektronischen Datenverarbeitung.

Die **Software** liefert vor allem die Anweisungen, die die Geräte ausführen sollen. Die Anweisungen werden dabei zu **Programmen** zusammengefasst. Anders als die Hardware ist die Software nicht greifbar.

Hard- und Software sind direkt aufeinander angewiesen: Ohne Hardware können die Anweisungen nicht ausgeführt werden, ohne Software weiß die Hardware nicht, was sie erledigen sollen. Diese Abhängigkeit ist ähnlich wie bei einem CD-Player in der Unterhaltungselektronik: Mit dem Gerät alleine können Sie nicht viel anfangen, da die Musik auf den CDs gespeichert ist. Haben Sie dagegen nur die CDs, aber keinen CD-Player, können Sie die CDs nicht abspielen.



Zusammenspiel von Hard- und Software

Grundsätzlich lässt sich Software in zwei große Gruppen unterteilen:

1. die **Systemsoftware** und
2. die **Anwendungssoftware**.

Die **Systemsoftware** ist immer konkret für eine spezielle Hardware beziehungsweise Hardware-Familie entwickelt – zum Beispiel für Personal Computer oder einen bestimmten Großrechner-Typ.

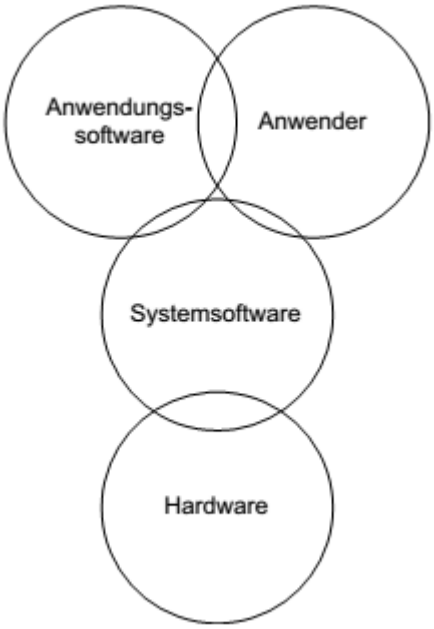
Der wichtigste Teil der Systemsoftware sind die **Betriebssysteme**. Sie steuern interne Prozesse wie die Verwaltung der Datenträger und die Koordination der Ein- und Ausgabe. Dazu gehört auch die Behandlung von eventuell auftretenden Fehlern. Außerdem stellen sie Anwendungsprogrammen wichtige Funktionen wie Laden, Speichern oder Drucken zur Verfügung und bieten dem Anwender eine Schnittstelle zur Bedienung des Computers. Bei vielen modernen Betriebssystemen erfolgt die Bedienung über eine grafische Benutzeroberfläche. Weit verbreitete Betriebssysteme sind zum Beispiel die verschiedenen Mitglieder der Windows-Familie, Linux, Unix oder MacOS.

Zur Systemsoftware werden außerdem Programme gezählt, die Hilfe bei der Nutzung und Wartung eines Computers anbieten – zum Beispiel Anwendungen zur Pflege und Verwaltung von Datenträgern wie Festplatten, Anwendungen zum Sichern von Daten oder Anwendungen zum Anzeigen von Systeminformationen. Diese **Dienstprogramme** gehören bei vielen modernen Betriebssystemen mit zum Lieferumfang.

Ein weiterer Bestandteil der Systemsoftware sind Anwendungen zur Entwicklung von Computerprogrammen. Dazu gehören zum Beispiel die Programmiersprachen und die integrierten Entwicklungsumgebungen. Diese integrierten Entwicklungsumgebungen fassen mehrere Programme zum Erstellen und Testen von Anwendungen unter einer Oberfläche zusammen.

So viel zur Systemsoftware.

Anwendungssoftware wird für die Lösung bestimmter Aufgaben eingesetzt – zum Beispiel für das Erfassen und Bearbeiten von Texten oder für Berechnungen in Tabellen. Anwendungssoftware setzt dabei in der Regel auf der Systemsoftware auf. Das heißt, sie benutzt Funktionen, die zum Beispiel das Betriebssystem zur Verfügung stellt.



Zusammenspiel von System- und Anwendungssoftware

Anwendungssoftware wird immer für ein bestimmtes Betriebssystem entwickelt. Eine Textverarbeitung, die zum Beispiel unter einem Windows-Betriebssystem läuft, kann nicht mit einem Linux-Betriebssystem eingesetzt werden. Dazu müsste eine spezielle Version dieser Textverarbeitung erstellt werden.

Der Markt für Anwendungssoftware ist kaum noch zu überblicken. Je weiter sich Computer verbreitet haben, desto größer wurde auch die Anzahl der verfügbaren Softwarelösungen. Neben universell einsetzbaren Programmen wie Textverarbeitung oder Tabellenkalkulation gibt es eine ganze Reihe spezialisierter Lösungen, die sich mit einem ganz bestimmten Problembereich beschäftigen – beispielsweise der Steuerung einer Maschine oder der Erstellung von Leiterplatten für elektronische Bauteile.

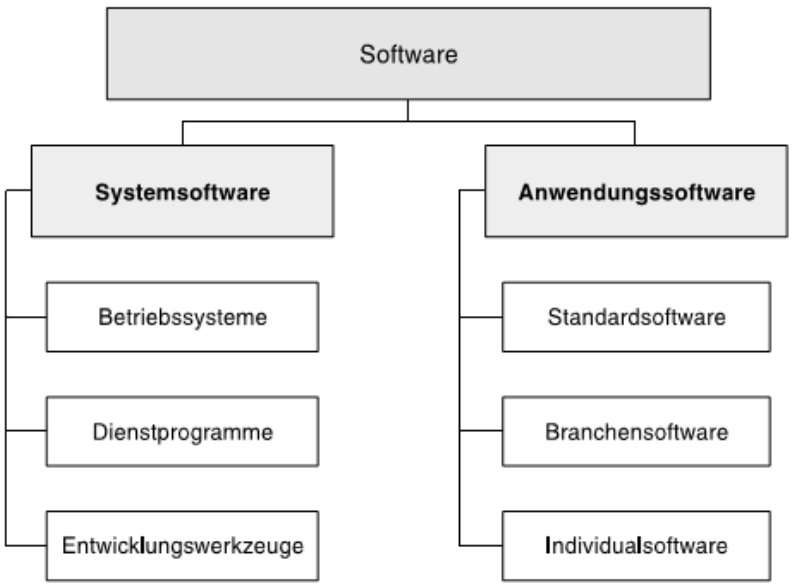
Am häufigsten werden so genannte **Standardprogramme** eingesetzt, die für einen großen Markt entwickelt werden. Solche Programme geben Funktionen für ein Anwendungsgebiet vor, konkrete Anpassungen an den Einzelfall sind in der Regel nur sehr eingeschränkt oder gar nicht möglich. Standardprogramme werden daher vor allem in Bereichen eingesetzt, die weder branchenspezifisch noch an eine spezielle betriebliche Funktion gebunden sind.

So können Sie mit einer Textverarbeitung sowohl eine kurze Notiz als auch ein Buch mit mehreren Hundert Seiten erstellen. Ein Tabellenkalkulationsprogramm berechnet sowohl die Bilanz eines mittelständischen Unternehmens als auch das Angebot eines Kleinunternehmers.

Weitere häufig eingesetzte Standardprogramme sind Datenbank-Management-Systeme zur Verwaltung von Informationen und Präsentationsprogramme zur anschaulichen Darstellung von Informationen.

Bei spezifischen Problemen – zum Beispiel der Steuerung einer ganz bestimmten Maschine – wird die nötige Software häufig speziell für einen Auftraggeber angefertigt. Solche **Individualprogramme** sind in der Regel sehr teuer und können häufig nur in einem Unternehmen eingesetzt werden, da sie ganz konkret auf die besonderen Erfordernisse des Auftraggebers zugeschnitten werden.

Eine Mischform zwischen Standardprogrammen und Individualprogrammen stellen die **Branchenprogramme** dar. Sie werden speziell auf die Bedürfnisse einer bestimmten Branche – zum Beispiel den Betrieb eines Krankenhauses oder einer Druckerei – zugeschnitten, berücksichtigen aber nicht die konkreten Erfordernisse eines einzelnen Unternehmens. Zwar wird Branchensoftware häufig in größeren Stückzahlen verkauft, trotzdem liegen die Preise zum Teil noch weit über denen von Standardsoftware.



Gliederung von Software

Anforderungen an die Entwicklung

Bei der Entwicklung von Software müssen also neben der eigentlichen Programmierarbeit noch zahlreiche weitere Bereiche bewältigt werden, die zum Teil noch nicht einmal direkt mit der Informatik im Zusammenhang stehen:

- Die Kommunikation zwischen Anwendern und Entwicklern muss eindeutig und ohne Missverständnisse erfolgen.
- Die Anforderungen an das System müssen so exakt wie eben möglich definiert und dokumentiert werden.
- Ein komplexes System muss in ein handhabbares Modell abgebildet werden.
- Ein komplexes System muss in mehrere Teilsysteme zerlegt werden.
- Die Zusammenarbeit dieser Teilsysteme muss exakt definiert werden.
- Es sollte von vornherein darauf geachtet werden, dass möglichst viele kleine Teile des Systems in anderen Software-Produkten wiederverwendet werden können.
- Das System muss möglichst einfach zu pflegen sein. Das setzt wiederum eine möglichst genaue und vor allem verständliche Dokumentation voraus.
- Das System sollte sich möglichst einfach ändern und erweitern lassen.
- Das System muss in möglichst vielen Umgebungen arbeiten können.
- Die Entwicklung und auch die Wartung müssen durch feste Vorgehensweisen und klar definierte Zuständigkeiten koordiniert werden. Das erfordert eine detaillierte Projektorganisation und ein umfangreiches Projektmanagement.

Hier wird noch einmal ganz deutlich, dass zur Entwicklung von Software-Systemen weitaus mehr gehört als nur gute beziehungsweise sehr gute Programmierkenntnisse. Ein „guter“ Software-Ingenieur beziehungsweise Software-Techniker muss zusätzlich über folgende Fähigkeiten verfügen:

- ausgeprägte kommunikative Kompetenzen,
- analytische Denkweise,
- sehr gute Abstraktionsfähigkeiten,
- hohe Flexibilität,
- Kreativität,
- Kenntnisse einschlägiger Methoden zur Dokumentation komplexer Vorgänge,

Was macht überhaupt die Qualität einer Software aus?

Die Frage lässt sich aus zwei Blickwinkeln beantworten:

- einer **externen Sicht** durch den Anwender und
- einer **internen Sicht** durch die Entwickler.

Externe Qualitätsfaktoren

Aus Sicht des Anwenders sind folgende Faktoren wichtig:

- Die Software muss korrekt, zuverlässig, robust und vertrauenswürdig arbeiten.
- Der Umgang mit der Software muss möglichst einfach sein. Sie muss bedienerfreundlich sein.
- Sie muss die Aufgaben möglichst schnell lösen. Sie muss effizient arbeiten.

Korrektheit bedeutet zunächst einmal, dass die Software die Aufgaben richtig löst – also zum Beispiel keine Berechnungsfehler enthält oder gespeicherte Daten genau so wieder zur Verfügung stellt wie sie eingegeben wurden.

Einige Beispiele:

Ein Programm, das durch eine falsche Formel statt 16 Prozent Mehrwertsteuer nur 0,16 Prozent Mehrwertsteuer berechnet, ist nicht korrekt.

Ein Programm, das beim Speichern von Adressen Umlaute wie ä, ö oder ü durch ein a, ein o oder ein u ersetzt, ist ebenfalls nicht korrekt.

Korrektheit bedeutet aber auch, dass sich das Software-System genau so verhält, wie es in der **Anforderungsdefinition** – also in der Beschreibung der geforderten Funktionalität – festgelegt wurde. Das setzt allerdings voraus, dass die Anforderungsdefinition die Problemstellung auch exakt und vollständig beschreibt.

Und genau das ist ein großes Problem im Software-Engineering: Die Anforderungsdefinition wird oft formlos erstellt und ist nur in seltenen Fällen absolut exakt und auch vollständig. Wenn aber bereits in der Anforderungsdefinition eine wichtige Funktion fehlt, oder nicht korrekt und eindeutig beschrieben ist, kann diese Funktion natürlich auch in dem fertigen Produkt nicht richtig umgesetzt werden.

Vom Problem zum Software-System

Die **Zuverlässigkeit** gibt an, dass sich ein Software-System in einem bestimmten Zeitraum korrekt verhält – also richtig funktioniert. Allgemein gesprochen gilt Software dann als zuverlässig, wenn sie **meistens funktioniert**.

Ein Beispiel:

Ein Textverarbeitungsprogramm, das einen Text 99 Mal speichert, aber beim 100. Speichern die Funktion schlicht und einfach nicht ausführt, kann durchaus noch als zuverlässig gelten. Ein Textverarbeitungssystem dagegen, das die Funktion **Speichern** bei jedem zweiten Mal nicht ausführt, ist nicht zuverlässig.

Kommen wir nun zur **Robustheit**.

Ein Software-System muss auch unter **unvorhergesehenen Umständen** weiter funktionstüchtig bleiben.

Ein Beispiel:

In einem Programm zur Adressverwaltung gibt ein Mitarbeiter versehentlich bei der Postleitzahl einen Buchstaben mit ein. Das Programm darf jetzt auf keinen Fall abstürzen[16] und seine Arbeit abrupt beenden. Genauso wenig darf es aber auch die falschen Daten verarbeiten. Es muss den Anwender auf seinen Fehler hinweisen und eine erneute Eingabe der Daten verlangen.

Die Reaktion auf unvorhergesehene Umstände wird auch **Ausnahmebehandlung** oder – im Fachjargon – **Exception Handling**[a)] genannt.

Als **vertrauenswürdig** gilt eine Software, wenn sie keine Schäden verursacht – auch dann nicht, wenn Fehler auftreten.

Bitte beachten Sie, dass vertrauenswürdige Software nicht unbedingt korrekt sein muss. Ein Computerspiel, das ständig stehen bleibt, ist nicht korrekt, aber immer noch vertrauenswürdig, da kein wirklicher Schaden entsteht – außer Ärger beim Spieler.

Kommen wir nun zur **Benutzerfreundlichkeit**.

Die Software muss vom Anwender schnell erlernt und einfach benutzt werden können. Dazu gehören unter anderem folgende Aspekte:

- Die Bedienung sollte möglichst selbsterklärend sein.

Der Anwender sollte jederzeit wissen, was die Software von ihm erwartet und welche Schritte er ausführen muss. Dazu gehört zum Beispiel, dass eindeutig klar ist, welche Eingaben in welcher Reihenfolge durchgeführt werden müssen.

Interne Qualitäsfaktoren:

Aus Sicht des Entwicklers sind vor allem folgende Faktoren wichtig:

- Die Software muss einfach zu warten sein. Sie muss sich schnell ändern lassen – zum Beispiel, um Fehler zu korrigieren oder um geänderten Anforderungen gerecht zu werden.
- Möglichst viele Teile der Software sollten wiederverwendbar sein beziehungsweise aus wiederverwendbaren Teilen bestehen.
- Die Software muss mit möglichst wenig Änderungen auf unterschiedlichen Plattformen arbeiten können. Sie muss portabel sein.
- Die Software muss möglichst effizient arbeiten und möglichst effizient erstellt werden.

Schauen wir uns auch die internen Qualitätsfaktoren der Reihe nach an.

Eine gute **Wartbarkeit** liegt dann vor, wenn sich die Software ohne größere Schwierigkeiten auch durch Dritte ändern beziehungsweise korrigieren lässt. Das heißt: Nicht nur der Programmierer, der die Software beziehungsweise einen Teil der Software erstellt hat, sollte wissen, was die Software macht und wie sie es macht.

Die Wartbarkeit wird unter anderem unterstützt durch

- eine Aufteilung der Software in mehrere kleinere Teile,
- eine detaillierte und nachvollziehbare Dokumentation,
- verbindliche Programmierrichtlinien, die ein schnelles Verstehen des Quellcodes ermöglichen und
- ein klar definiertes Änderungs-Management („Wer hat wann welche Änderungen in dem Programm vorgenommen?“)

Der „Produktionsprozess“ von Software

Wie Sie bereits erfahren haben, erfordert die Entwicklung qualitativ hochwertiger Software eine systematische und strukturierte Vorgehensweise. Ein erster Schritt besteht dabei darin, den Prozess zur Lösung des Problems zu beschreiben – also festzulegen: „Wie kommen wir vom Problem zum fertigen Produkt?“. Sie definieren ein **Vorgehensmodell** und gliedern dabei den Prozess in überschaubare Abschnitte, die dann einzeln geplant, durchgeführt und überprüft werden.

Sie haben auch bereits einige grundlegende Prinzipien Methoden für die Auflösung komplexer Systeme kennen gelernt, die sich auch auf Software-Systeme anwenden lassen. Für sich allein gesehen, nützen diese Prinzipien und Methoden aber herzlich wenig. Sie müssen auch wissen, wie Sie sie Zielgerichtet einsetzen. Nehmen wir einmal ein Beispiel aus dem Alltag:

Sie wollen in Ihrer Wohnung ein großes schweres Bild an eine Wand hängen. Damit das Bild nicht sofort wieder herunterfällt, beschließen Sie, es mit einem Haken und einem Dübel sicher zu befestigen. Um dieses „Problem“ zu lösen, reicht es nicht aus, die erforderlichen Werkzeuge – eine Bohrmaschine, einen Haken und einen Dübel – zu beschaffen und einzusetzen. Sie müssen vor dem Einsatz der Werkzeuge zunächst einmal weitere Fragen klären:

- ✓ Sie müssen wissen, an welcher Stelle Sie das Loch in die Wand bohren wollen oder müssen.
- ✓ Das Loch darf nicht zu groß oder zu klein für den Dübel sein.
- ✓ Die Größe des Dübels hängt wiederum von dem verwendeten Haken ab.
- ✓ Die Stärke des Hakens wird durch das Gewicht des Bildes bestimmt.

Sie sehen selbst; bei einer banalen Tätigkeit wie Bilder aufhängen sind eine Reihe von Entscheidungen nötig, die alle die Qualität des Ergebnisses unmittelbar beeinflussen.

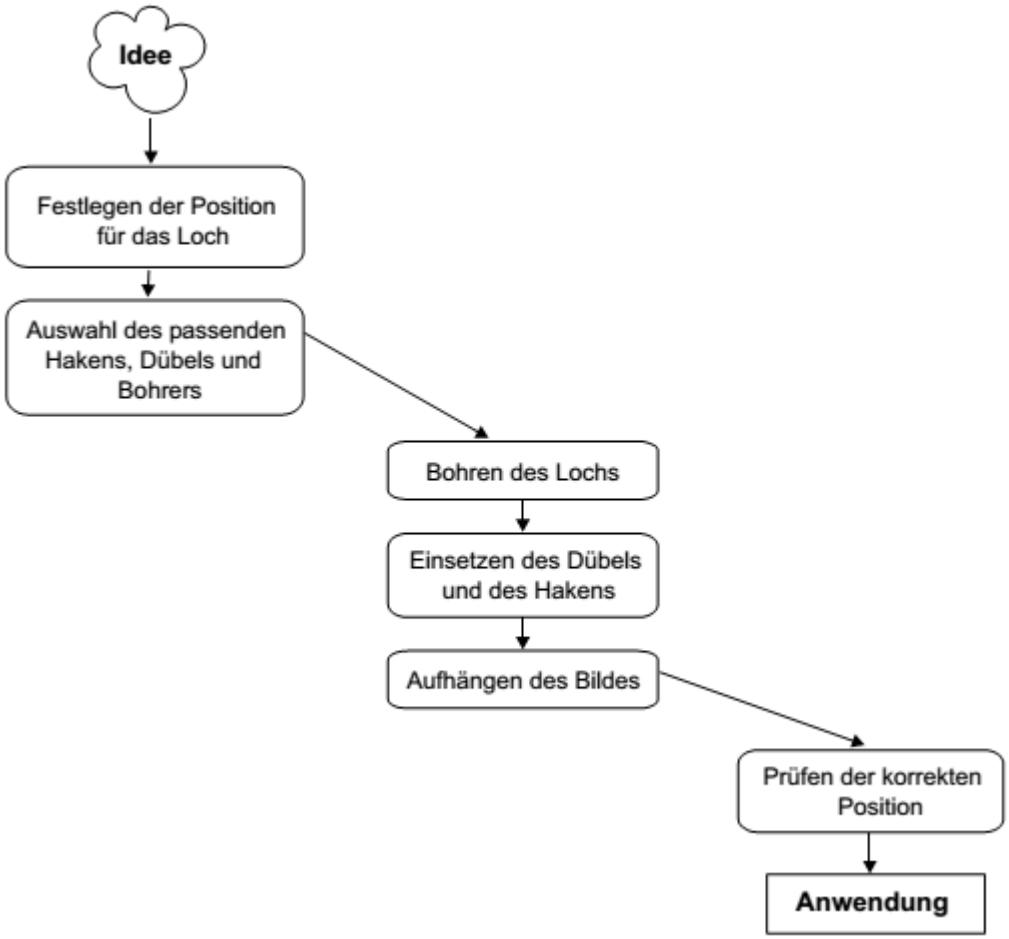
Das Vorgehensmodell

Obwohl der Weg von Ihrer Idee(„An dieser Stelle wäre ein Bild ansehnlich“) bis zur Anwendung(Betrachten des aufgehängten Bildes) kurz und überschaubar ist, werden Sie trotzdem Ihre Arbeit strukturieren, vorbereiten und planen. Sie definieren für den gesamten Prozess ein Vorgehensmodell, das den Vorgang in kleinere Schritte gliedert. Jeder dieser Schritte ist klar definiert und liefert ein überprüfbares Ergebnis.

Für den Prozess „Bild aufhängen“ könnte das Vorgehensmodell zum Beispiel so aussehen:

1. Festlegen der Position für das Loch,
2. Auswahl des passenden Hakens, Dübels und Bohrers,
3. Bohren des Lochs,
4. Einsetzen des Dübels und des Hakens,
5. Aufhängen des Bildes und
6. Prüfen der korrekten Position.

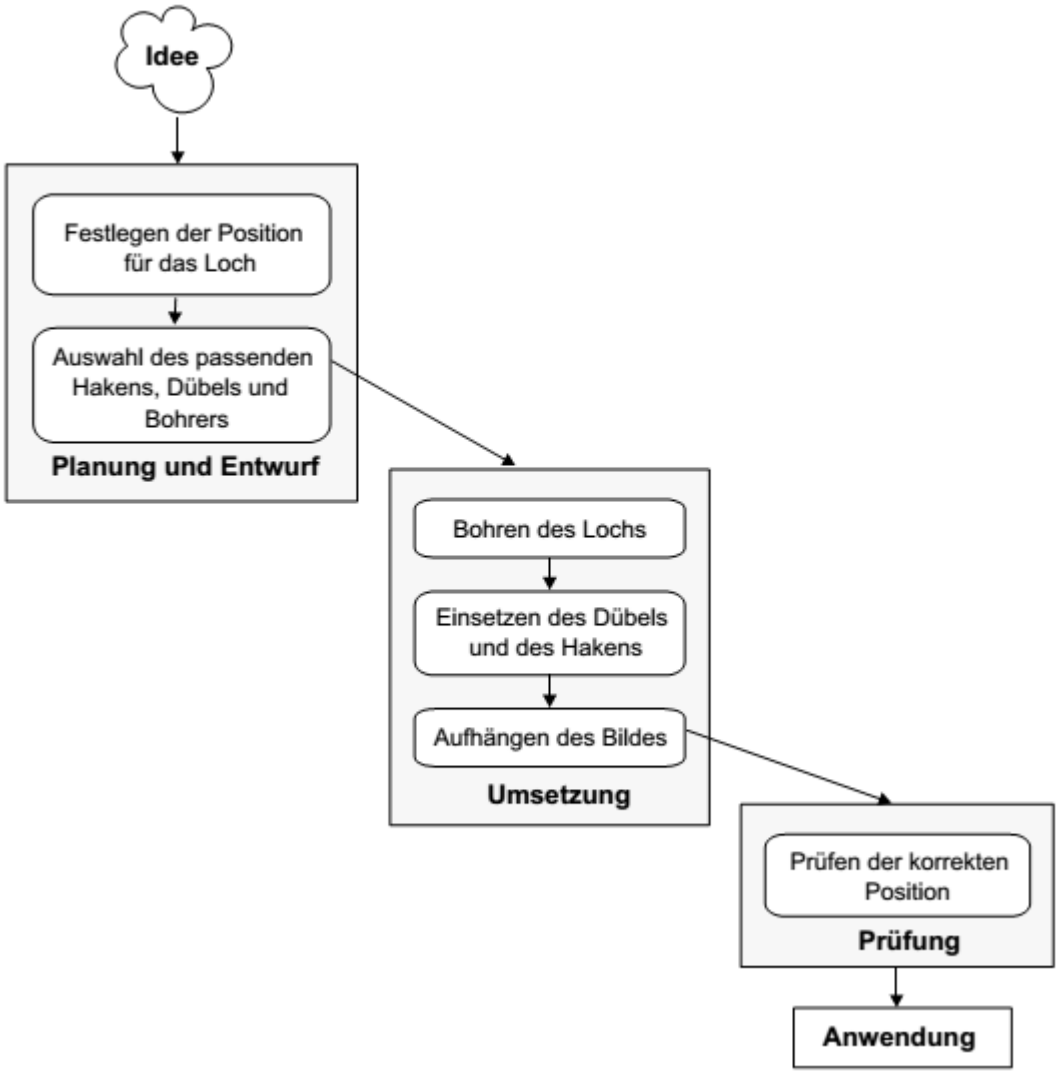
Grafische Darstellung:



<u>Schritte beim Prozess „Bild aufhängen“</u>

Die einzelnen Schritte lassen sich dabei zu mehreren Phasen zusammenfassen, die hintereinander durchlaufen werden müssen. In den Schritten 1 und 2 wird zum Beispiel **geplant und entworfen**, in den Schritten 3 bis 5 wird der Entwurf **umgesetzt** und im letzten Schritt wird die Umsetzung **überprüft**. Die Umsetzungsphase baut dabei auf den Ergebnissen der Planungs- und Entwurfsphase auf und die Prüfungsphase auf den Ergebnissen der Umsetzungsphase.

Erweiterung der grafischen Darstellung:

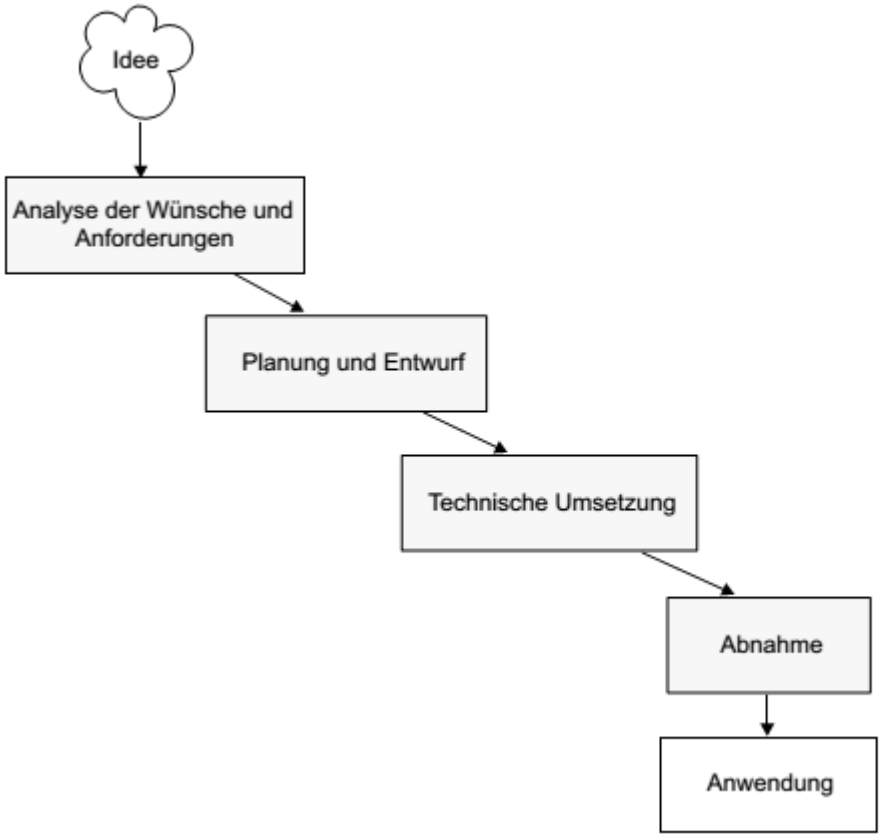


<u>Phasen beim Prozess „Bild aufhängen“</u>

Je umfangreicher und komplexer ein Prozess ist, desto mehr Phasen lassen sich unterscheiden. Einen Hausbau könnte man zum Beispiel sehr grob in folgende Phasen untergliedern:

- Analyse der Wünsche und Anforderungen des Bauherrn,
- Planung und Entwurf des Architekten,
- technische Umsetzung und
- Abnahme.

Auch hier liefert jede Phase ein Ergebnis, das dann in der darauffolgenden Phase eingesetzt wird. So beeinflussen die Wünsche und Anforderungen der Bauherren den Entwurf des Architekten, der Entwurf des Architekten wird technisch umgesetzt, die technische Umsetzung wird abgenommen und schließlich gibt die Abnahme das Haus zum Einsatz frei.



Phasen beim Hausbau

Die Phasen der Software-Entwicklung

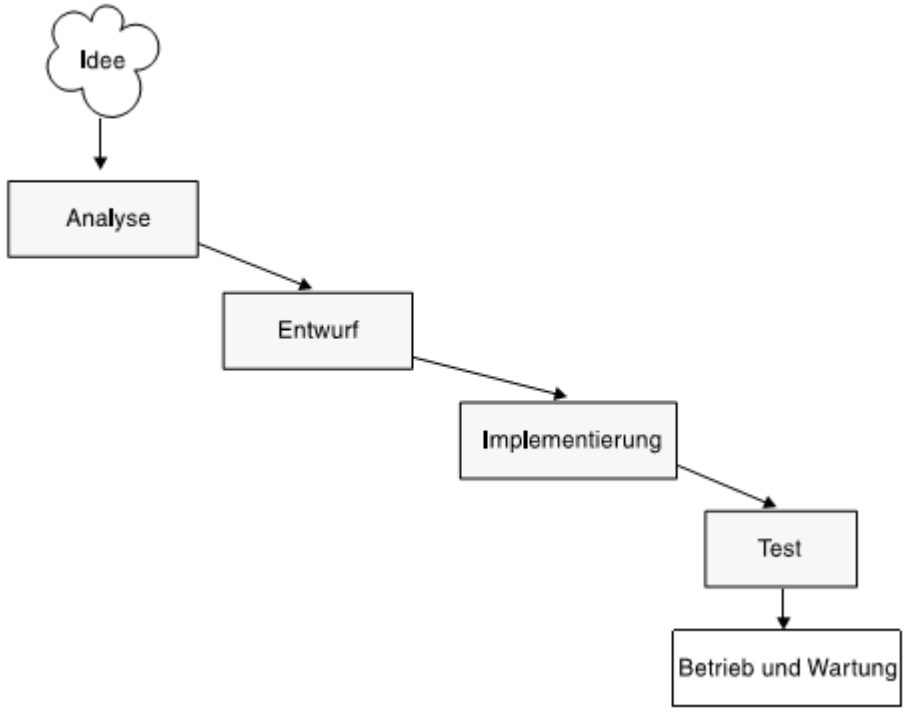
Die Unterteilung in Phasen findet sich auch bei den meisten Vorgehensmodellen für die Software Entwicklung wieder. Dabei können grundsätzlich folgende Phasen unterschieden werden:

- die Analyse („Was soll die Software machen?“),
- der Entwurf („Wie soll die Software die Aufgaben lösen?“),
- die Implementierung– die eigentliche Programmierung,
- der Test des Systems sowie
- Betrieb und Wartung.

Die ersten vier Phasen umfassen dabei die Entstehung der Software und die letzte Phase den Einsatz. Hier ist das Produkt also bereits komplett fertig gestellt.

Hinweis:
Was sich genau hinter diesen einzelnen Phasen verbirgt, erfahren Sie im weiteren Verlauf dieses Kurses.

Strukturierte Darstellung der Phasen bei der Software-Entwicklung:



Die Phasen bei der Software-Entwicklung

Alle Phasen zusammen in ihrer zeitlichen Abfolge werden auch als **Software-Lebenszyklus** oder **Software Life Cycle** bezeichnet. Der Software Lebenszyklus ist also die Zeitspanne, in der ein Softwareprodukt entwickelt und eingesetzt wird – bis zum Ende seiner Benutzung.

Aus diesem grundlegenden Phasenmodell wurden verschiedene Vorgehensmodelle entwickelt. Ein Vorgehensmodell untergliedert einen Prozess in verschiedene aufeinanderfolgende Phasen. Jede Phase kann einzeln geplant, durchgeführt und überprüft werden. Die Ergebnisse einer Phase werden in der nachfolgenden Phase genutzt.

Sequenzielle Vorgehensmodelle

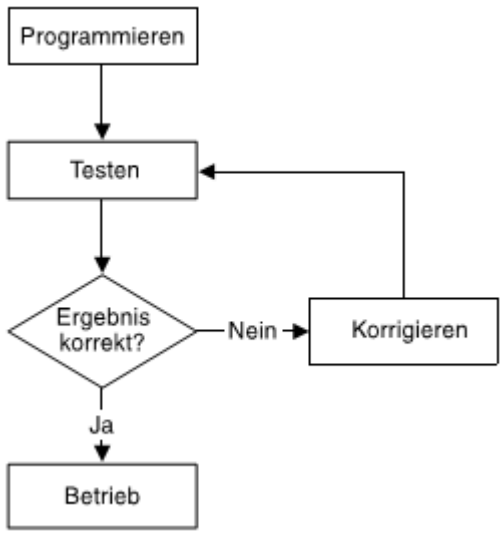
Nun stelle ich Ihnen einige sequenzielle Vorgehensmodelle für die Software-Entwicklung vor. Diese Modelle durchlaufen die einzelnen Phasen mehr oder weniger strikt hintereinander – daher auch die Bezeichnung sequenziell.

Merke: Eine **Sequenz** ist – allgemein ausgedrückt – eine Folge oder Reihe.

Beginnen wir mit dem einfachsten „Modell“ – dem Code-and-Fix-Zyklus.

➢ Der Code-and-Fix-Zyklus

Der **Code-and-Fix-Zyklus** – auch **Build-and-Fix-Zyklus** genannt – besteht eigentlich nur aus zwei Schritten:
1. Der Programmierer erstellt die Anweisungen und lässt sie übersetzen.
2. Er testet, ob die Anweisungen das gewünschte Ergebnis liefern. Wenn das der Fall ist, kann die Software in Betrieb genommen werden. Treten beim Test Fehler auf, wird das Programm korrigiert und erneut getestet. Das gesamte Verfahren wird so lange wiederholt, bis die Software fertig ist. Grafisch lässt sich dieses „Modell“ so darstellen:



Code and Fix bedeutet übersetzt so viel wie „Programmieren und Reparieren“.

Erläuterung:
Der Code-and-Fix-Zyklus ermöglicht eine extrem schnelle Herstellung von Software, da die Ideen des Programmierers sofort in ein ausführbares Programm umgesetzt werden. Deshalb hat wahrscheinlich schon jeder, der einmal ein Computerprogramm erstellt hat, mit diesem Modell gearbeitet – häufig aber ohne zu wissen, wie seine Entwicklungstechnik nun genannt wird. Grundsätzlich muss der Code-and-Fix-Zyklus nicht zwangsläufig auch „schlechte“ Software erzeugen. Die Qualität hängt aber sehr stark von der Erfahrung des Entwicklers und auch vom Umfang der Software ab. Erfahrene Programmierer können also kleine überschaubare Programme durchaus auch mit dem Code-andFix-Zyklus erfolgreich umsetzen. Für komplexe Software-Systeme eignet sich der Code-and-Fix-Zyklus in Reinform aber nicht. Denn:

• Es existiert in der Regel keine Dokumentation.

Nur der Programmierer selbst weiß, wie sein Programm arbeitet und worauf besonders geachtet werden muss. Ein anderer Entwickler müsste sich erst mühselig in das Programm eindenken – falls es ihm überhaupt gelingt. Die fehlende Dokumentation kann auch dann zum Problem werden, wenn Entwickler und Anwender nicht identisch sind. Da die Anforderungen des Anwenders nicht festgehalten werden, erstellt der Programmierer unter Umständen eine Software, die etwas ganz anderes macht als der Anwender eigentlich benötigt – zum Beispiel, weil Missverständnisse aufgetreten sind. Auch nachträgliche Erweiterungen oder Änderungen des Programms können zu einem ernsthaften Problem werden. Oft weiß spätestens nach einigen Monaten auch der Programmierer nicht mehr genau, wie sein Programm eigentlich funktioniert.

• Das Programm wächst in der Regel „wild“.

Bei einem Fehler wird in der Regel nur dieser eine Fehler betrachtet und auch korrigiert. Dabei zählt vor allem, dass das Programm „funktioniert“ – egal wie. Korrekturen erfolgen also nicht systematisch, sondern oft durch einfaches Ausprobieren. Das führt nicht selten dazu, dass zwar der eine Fehler behoben ist, durch die Korrektur aber ein neuer Fehler an einer anderen Stelle auftritt. Diese andere Stelle ist aber gar nicht getestet worden, da das Programm ja dort zunächst keine Probleme gemacht hat. Mit jeder Korrektur nimmt also quasi automatisch die Qualität und auch Wartbarkeit des Programms ab.

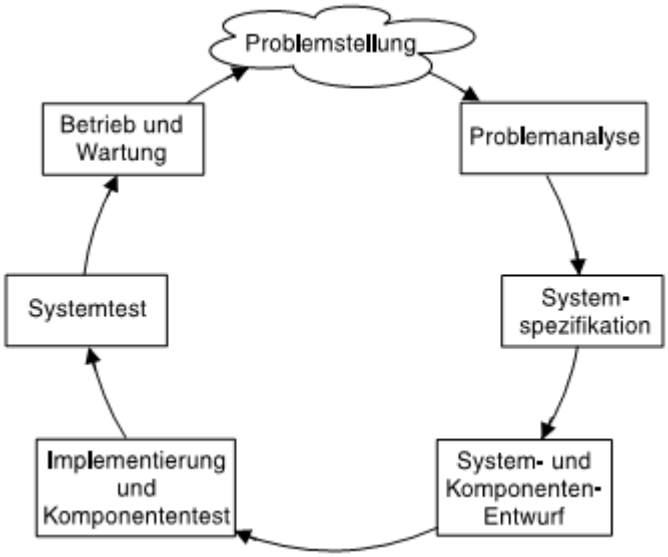
• Die gesamte Entwicklung hängt an einer einzigen Person – dem Programmierer.

Damit ist der maximale Umfang der Software stark beschränkt – es sei denn, die Entwicklung kann auch mehrere Jahre oder noch länger dauern. Wenn der Programmierer die Arbeit an seinem Werk einstellt, bedeutet das in der Regel auch immer das Ende der Entwicklung. Zusammengefasst lässt sich sagen: Für kleine überschaubare Programme mit klar definiertem Aufgabengebiet ist der Code-and-Fix-Zyklus mit einigen Einschränkungen durchaus brauchbar. Für komplexe Software-Systeme ist er in Reinform absolut ungeeignet.

➢ Das Software Life Cycle-Modell

Das Software Life Cycle-Modell war eines der ersten Vorgehensmodelle für die Software-Entwicklung überhaupt. Es unterteilt die Erstellung in insgesamt sechs Phasen, die strikt hintereinander durchlaufen werden. Die nächste Phase darf erst dann begonnen werden, wenn die vorlaufende Phase abgeschlossen ist. Die Wiederholung einer Phase ist im ursprünglichen Modell nur dann gestattet, wenn diese Phase im Lebenszyklus der Software auch wieder an der Reihe ist.

Grafisch lässt sich das Modell so darstellen:



Schauen wir uns die einzelnen Phasen des Modells etwas genauer an:
Die **Problemstellung** gibt den Anstoß für die Entwicklung. Hier wird beschlossen, dass ein Prozess durch Software unterstützt beziehungsweise abgelöst werden soll. Übertragen auf den Hausbau fällt hier also zunächst einmal die grundsätzliche Entscheidung: „Wir bauen“. Ob der Hausbau sinnvoll ist und wie das Haus gebaut werden soll, spielt hier keine Rolle.

Bei der **Problemanalyse** wird festgehalten, welche Aufgaben die Software grundsätzlich erledigen soll. Ausgangspunkt sind dabei die Wünsche der Anwender. Außerdem erfolgt hier eine erste grobe Prüfung der Wirtschaftlichkeit – also „Lohnt sich die Entwicklung überhaupt?“

In der **Systemspezifikation** werden die noch groben Angaben zu den Aufgaben der Software konkretisiert. Hier wird also detailliert der genaue Leistungsumfang festgelegt.

Beim **System- und Komponentenentwurf** geht es dann darum: „Wie soll die Funktionalität erreicht werden?“ Dazu wird zunächst das System insgesamt betrachtet und dann in kleinere Teile – die Komponenten – untergliedert. Das System wird also strukturiert, hierarchisiert und modularisiert.
In der Phase **Implementierung und Komponententest** werden die einzelnen Komponenten technisch umgesetzt und dann zunächst getrennt getestet. Hier erfolgt also die eigentliche Programmierung.

Im **Systemtest** wird dann das System unter möglichst realistischen Bedingungen insgesamt getestet. Hier wird also geprüft, ob die einzelnen Komponenten korrekt zusammenarbeiten. In der letzten Phase – **Betrieb und Wartung** – wird die Software schließlich eingesetzt. Falls dabei noch Fehler auftreten, werden sie durch die Wartung behoben. Das Software Life Cycle-Modell lässt sich eigentlich nur dann erfolgreich anwenden, wenn der Entwicklungsprozess optimal abläuft – also jede Phase tatsächlich ein vollständiges und korrektes Ergebnis liefert. Die Praxis hat aber gezeigt, dass das in der Regel bei der Software-Entwicklung nie der Fall ist. Oft stellt sich nämlich erst in einer nachfolgenden Phase heraus, ob die vorlaufenden Phasen tatsächlich erfolgreich abgeschlossen wurden.

Ein Beispiel:

Bei der Problemanalyse für eine Leihbücherei ist keine Funktion zum Ändern der Kundendaten erfasst worden. Sie wurde schlicht und einfach vergessen. Der Fehler macht sich aber unter Umständen erst sehr spät bemerkbar – im Extremfall erst im Betrieb. Als Konsequenz müsste jetzt im Software Life Cycle-Modell ein neuer Zyklus begonnen werden, da sich ja die Problemanalyse geändert hat. Der Lebenszyklus der Software wäre damit bereits wieder beendet, bevor er überhaupt richtig begonnen hat.

Ein weiteres Problem des Software Life Cycle-Modells ist die sehr geringe Beteiligung der Anwender. Das fertige Produkt bekommen sie erst in der letzten Phase zu sehen. Und nur hier können die Anwender dann prüfen, ob ihre Anforderungen richtig verstanden und umgesetzt worden sind. Wenn das nicht der Fall ist, muss unter Umständen ebenfalls wieder ein komplett neuer Zyklus durchgeführt werden. Das bedeutet aber hohen Aufwand und damit auch hohe Kosten. In der Konsequenz werden Änderungswünsche nicht selten gar nicht mehr umgesetzt und der Anwender muss sich mit dem zufrieden geben, was er bekommen hat – gleichgültig, ob brauchbar, eingeschränkt brauchbar oder unbrauchbar.

Das Software Life Cycle-Modell wurde daher weiterentwickelt – zum Beispiel zum Wasserfall-Modell.

- **Das Wasserfall-Modell**

Das Wasserfall-Modell geht ebenfalls von mehreren Phasen aus, die nacheinander durchlaufen werden. Es handelt sich also – wie beim Software Life Cycle-Modell – ebenfalls um ein sequenzielles Vorgehen.

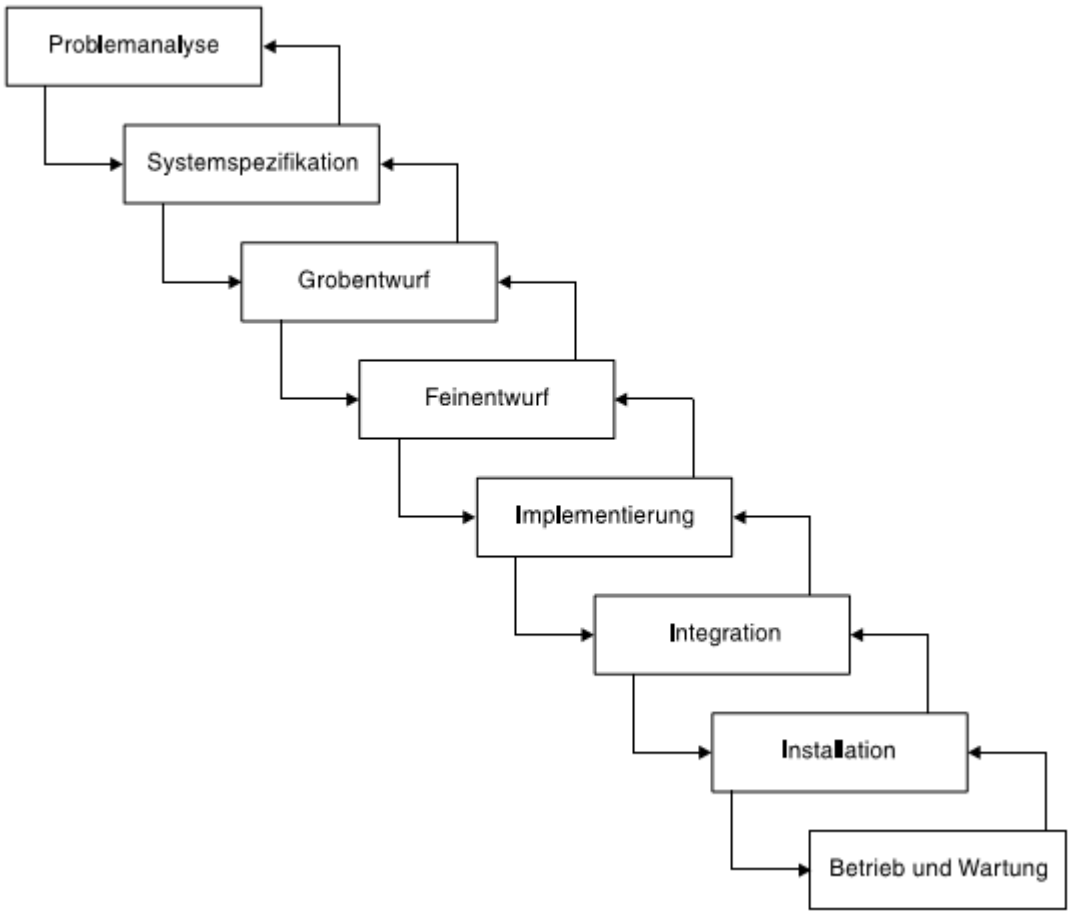
Anders als beim Software Life Cycle-Modell ist aber eine Rückkoppelung zwischen zwei Phasen vorgesehen. Stellt sich in einer Phase heraus, dass die vorlaufende Phase nicht vollständig war, kann sie wiederholt werden. Ein Rücksprung über mehrere Phasen ist dabei allerdings nicht möglich. Das heißt, es kann immer nur die unmittelbar vorangegangene Phase wiederholt werden.

Merke: Eine Wiederholung wird im Fachjargon auch **Iteration** genannt.

Jede Phase im Wasserfall-Modell erzeugt eigene Dokumente. Das ist ein weiterer wesentlicher Unterschied zum Software Life Cycle-Modell, bei dem die Dokumente für die einzelnen Phasen nicht zwingend erforderlich sind. Außerdem finden sich im Wasserfall-Modell noch weitere Phasen, die die Phasen aus dem Software Life Cycle-Modell verfeinern. So wird zum Beispiel der System und Komponentenentwurf auf die beiden Phasen Grobentwurf und Feinentwurf verteilt. Im Grobentwurf wird zunächst das System als Ganzes betrachtet, im Feinentwurf die einzelnen Bestandteile des Systems. Zusätzlich finden sich eigene Phasen für die Integration und die Installation.
Bei der Integration werden die einzelnen Teile des Systems zusammengebaut. In der Phase der Installation kann das System unter anderem einem so genannten Beta-Test unterzogen werden.

Bei diesem Beta-Test erfolgen die Prüfungen nicht mehr durch die Entwickler, sondern durch eine ausgewählte Gruppe von Anwendern. Wenn Sie so wollen, wird hier die „Praxistauglichkeit“ der Software getestet.

Grafisch lässt sich das Wasserfall-Modell so darstellen:

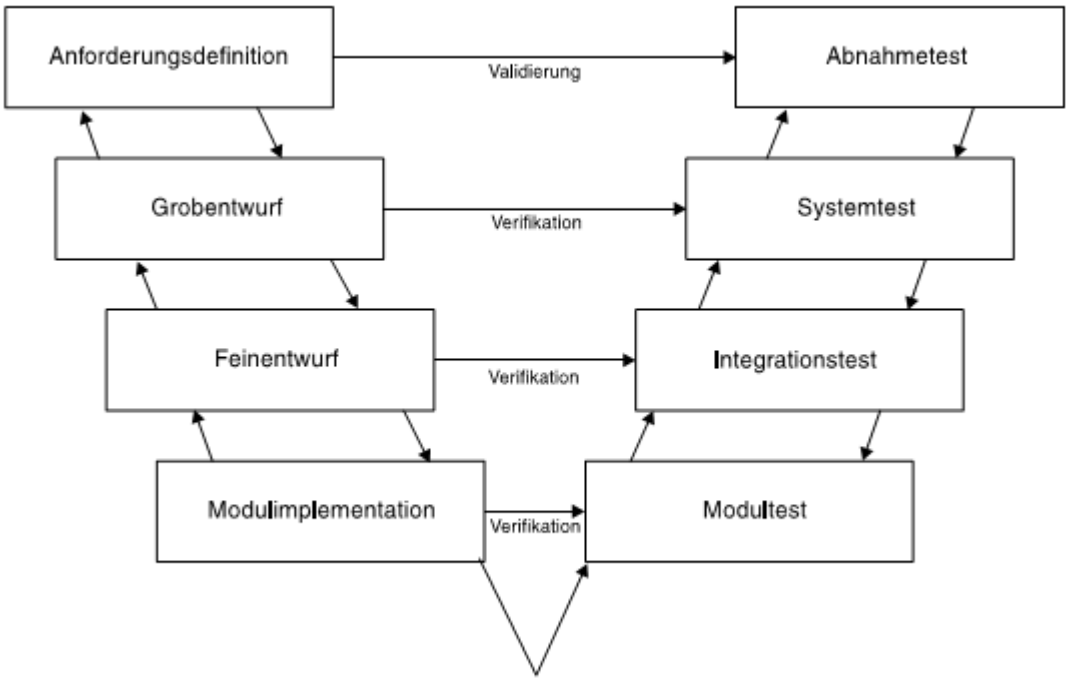


Das Wasserfall-Modell hebt zwar mit der Wiederholung von vorlaufenden Phasen einen gravierenden Nachteil des Software Life Cycle Modells auf, ist aber durch den sequenziellen Ablauf ebenfalls nicht sonderlich flexibel. Stellt sich zum Beispiel während der Implementierung heraus, dass die Problemanalyse unvollständig oder fehlerhaft ist, kann auch beim Wasserfall-Modell keine nachträgliche Korrektur oder Anpassung erfolgen. Ein Rückgriff über mehrere Phasen ist ja auch bei diesem Modell nicht vorgesehen. Trotzdem ist das Wasserfall-Modell durchaus noch verbreitet. Es eignet sich zum Beispiel recht gut für die Erstellung von Software, bei der die einzelnen Schritte tatsächlich sequenziell ausgeführt werden können und bei der mögliche Risiken vor der Entwicklung weitgehend ausgeschlossen werden können. Dazu gehört auch, dass die Anforderungsanalyse möglichst eindeutig beschrieben werden kann – zum Beispiel durch verbindliche Normen für den Einsatzbereich.

- **Das V-Modell**

Das V-Modell ist ein weiteres sequenzielles Modell. Es erweitert das Wasserfall Modell und macht die Qualitätssicherung zum ausdrücklichen Bestandteil des Prozesses.

Grafisch lässt sich das Modell so darstellen:



Erläuterung:

Auf der linken Seite des Vs finden Sie die vier Phasen **Anforderungsdefinition**, **Grobentwurf**, **Feinentwurf** und **Modulimplementation**. Diese Phasen werden von oben nach unten durchlaufen. Die Phase **Anforderungsdefinition** entspricht im Wesentlichen der Phase **Problemanalyse** aus den beiden anderen Modellen. In der Phase **Modulimplementation** erfolgt die Programmierung. Auf der rechten Seite des Vs finden Sie verschiedene Tests für die Phasen auf der linken Seite. Diese Tests werden von unten nach oben durchlaufen. Auf der obersten Ebene erfolgt dabei die Validierung des Systems, die drei unteren Ebenen dienen der Verifikation.

Bei der **Validierung** wird geprüft, ob ein Ergebnis auch mit einem Sachverhalt übereinstimmt. Bei der **Verifikation** wird die Richtigkeit eines Ergebnisses geprüft. Auf Software übertragen bedeutet das: Die Validierung prüft, ob das richtige Produkt entwickelt wird – also das Produkt, das die Anwender benötigen. Die Verifikation überprüft, ob das Produkt selbst richtig – also korrekt – ist.

Genau wie die beiden anderen sequenziellen Modelle ist auch das V-Modell relativ anfällig für Fehler in den frühen Phasen. Auch hier bedeutet ein Fehler häufig sehr hohen Aufwand und damit sehr hohe Kosten.

Hinweis:
Eine spezielle Variante des V-Modells wird als Standard-Verfahren bei den Bundesbehörden eingesetzt.
Es besteht aus insgesamt vier einzelnen Modellen: **Systemerstellung**, **Qualitätssicherung**, **Konfigurationsmanagement** und **Projektmanagement**. Was sich genau hinter diesen Modellen verbirgt, wollen wir uns hier nicht weiter ansehen. Detaillierte Informationen finden Sie zum Beispiel bei *Hansen* oder bei *Balzert* im Band 2.

Das Lastenheft

Noch einmal zur Erläuterung:
Das Lastenheft legt fest, **was** die Software grundsätzlich machen soll. Das „Wie“ – also die technische Umsetzung spielt im Lastenheft keine Rolle.

Das Lastenheft beschreibt die Anforderungen aus Kundensicht, nicht aus Entwicklersicht. Das bedeutet: Der Kunde muss genau verstehen, was im Lastenheft beschrieben ist. Kunde und Entwickler erarbeiten das Lastenheft in der Regel gemeinsam.

Der Aufbau eines Lastenheftes

Damit das Lastenheft möglichst einfach nachzuvollziehen und auch zu prüfen ist, sollten Sie es inhaltlich strukturieren. Wie Sie die Struktur aufbauen, ist dabei beliebig. Feste Vorgaben gibt es nicht.
Balzert empfiehlt aber zum Beispiel folgenden grundsätzlichen Aufbau:

1. Zielbestimmung
2. Produkteinsatz
3. Produktübersicht
4. Produktfunktionen
5. Produktdaten
6. Produktleistungen
7. Qualitätsanforderungen
8. Ergänzung

Bei der **Zielbestimmung** werden die Ziele dargestellt, die mit dem Einsatz des Produkts erreicht werden sollen. Die Angabe kann dabei durchaus recht allgemein erfolgen – zum Beispiel „Die Leihbücherei soll mit der Software ihre Kunden, Medien und die Ausleihe verwalten können.“ Beim **Produkteinsatz** werden die **Anwendungsbereiche** und die **Zielgruppen** beschrieben – also, wer arbeitet wo mit dem Produkt. Die **Produktübersicht** beschreibt das Umfeld des Produktes – zum Beispiel, welche Akteure mit dem System arbeiten. Die Beschreibung kann dabei in einfacher grafischer Form erfolgen oder als Text.

Bei den **Produktfunktionen** werden die wichtigsten Funktionen aus Anwendersicht beschrieben. Dabei spielt es zunächst einmal keine Rolle, ob diese Funktionen überhaupt durch die Software umgesetzt werden sollen. Um in späteren Phasen eindeutig auf eine Beschreibung zurückgreifen zu können, sollten die Produktfunktionen durchnummeriert werden – zum Beispiel als LF10, LF20 und so weiter.

Bei den **Produktdaten** werden die wichtigsten Daten, die das System verarbeiten können soll, aufgeführt. Dabei sollte nach Möglichkeit auch angegeben werden, wie oft diese Daten vorkommen können. Die Produktdaten werden ebenfalls nummeriert – zum Beispiel mit LD10 und so weiter. Bei den **Produktleistungen** werden – falls erforderlich – besondere Anforderungen an die Produktfunktionen und die Produktdaten beschrieben – zum Beispiel, welchen Umfang eine Ausgabeliste haben soll. Die Nummerierung erfolgt nach dem Schema LL10 und so weiter.

Die **Qualitätsanforderungen** beschreiben besondere Qualitätsmerkmale des Produkts – zum Beispiel, dass es besonders anwenderfreundlich oder leicht zu ändern sein muss. Die Beschreibung kann entweder sprachlich oder als Tabelle zum Ankreuzen erfolgen.

Qualitätsfaktor	sehr gut	gut	normal	unwichtig
Zuverlässigkeit				
Robustheit				
Benutzbarkeit				
Wiederverwendbarkeit				
...				

In die Spalte Qualitätsfaktor können Sie zum Beispiel die internen und externen Qualitätsfaktoren eintragen, die Sie bereits kennengelernt haben. Anschließend kreuzen Sie in den Spalten daneben an, welche Bedeutung der jeweilige Qualitätsfaktor hat. Bei den Ergänzungen schließlich findet alles Platz, was wichtig ist, aber nicht bei den anderen Punkten dargestellt werden kann – zum Beispiel besondere Anforderungen an die Bedienung. Wie ein Lastenheft konkret aussehen kann, erfahren wir beim Lastenheft für die Leihbücherei. Jetzt wollen wir uns zunächst einmal ansehen, wie Sie die nötigen Informationen für das Lastenheft beschaffen können.

Informationssammlung für das Lastenheft

Wenn wir noch einmal die möglichen Initialzündungen für das Erstellen von Software betrachten, lassen sich vier verschiedene Ausgangssituationen für das Erstellen des Lastenheftes unterscheiden:

1. Der Kunde liefert Ihnen bereits ein Lastenheft.
2. Sie wollen eine Standard-Software erstellen.
3. Sie wollen eine bestehende Software überarbeiten.
4. Sie wollen eine komplette neue Software entwickeln.

Der erste Fall – der Kunde liefert ein Lastenheft – findet sich zum Beispiel häufiger bei der Auftragsvergabe über Ausschreibungen. Der Kunde legt detailliert seine Anforderungen fest. Sie müssen dann „nur noch" ein entsprechendes Angebot erstellen. Im zweiten Fall – bei der Entwicklung von Standard-Software – können Sie für das Lastenheft zum Beispiel Marktanalysen durchführen. Befragen Sie mögliche Kunden, welche Funktionen die Software haben sollte, oder sehen Sie sich entsprechende Produkte von Mitbewerbern an. Aus diesen Daten können Sie dann das Lastenheft entwickeln – allerdings weitestgehend im „Alleingang".
Eine konkrete Abstimmung mit dem Kunden ist allein schon deshalb schwierig, weil gar nicht genau klar ist, wer denn nun genau Ihr Kunde ist. Im dritten Fall – bei der Überarbeitung einer bestehenden Software – sollten Sie zunächst einmal eine Ist-Analyse durchführen – also möglichst detailliert festhalten,

• welche Funktionen die bisherige Software unterstützt,
• welche dieser Funktionen überarbeitet werden müssen,
• welche dieser Funktionen übernommen werden können und
• auch welche Funktionen fehlen.

Mit den Daten der Ist-Analyse können Sie dann gezielt mit dem Kunden festlegen, was genau gemacht werden soll. Der vierte Fall – das Entwickeln einer komplett neuen Software – ist in der Regel mit dem meisten Aufwand verbunden. Hier müssen Sie versuchen, durch möglichst viele Informationen die oft vagen Beschreibungen des Kunden zu präzisieren. Eine Möglichkeit dazu bieten Fragebögen, die Sie aus der Kundenbeschreibung entwickeln können – zum Beispiel:

Musterfragebogen

Welche Daten müssen genau für einen Kunden vom System verarbeitet werden können?		
Name und Vorname	ja	nein
Anschrift	ja	nein
eindeutige Kennzeichnung – zum Beispiel eine Kundennummer	ja	nein
....		
Erweitern Sie die Liste bitte um fehlende Daten.....		

Achten Sie dabei darauf, dass die Fragen möglichst präzise sind und nach Möglichkeit mit **Ja** oder **Nein** beantwortet werden können. Fragen, die eine freie Antwort ermöglichen, liefern oft keine eindeutigen Informationen.

Fragen, auf die vor allem mit Ja oder Nein geantwortet werden kann, wer- den auch **geschlossene Fragen** genannt. Fragen, die auf eine ausführliche Antwort zielen, heißen **offene Fragen**.

Eine andere Möglichkeit ist das Erstellen von Stellenbeschreibungen und Stellenprofilen. Bitten Sie die Mitarbeiter des Unternehmens möglichst detailliert zu beschreiben, welche Tätigkeiten sie an ihrem Arbeitsplatz ausführen und welche Randbedingungen dabei beachtet werden müssen. Um den Mitarbeitern das Erstellen der Beschreibungen möglichst einfach zu machen, können Sie ein Formblatt entwickeln, das zum Beispiel folgende Fragen enthält:

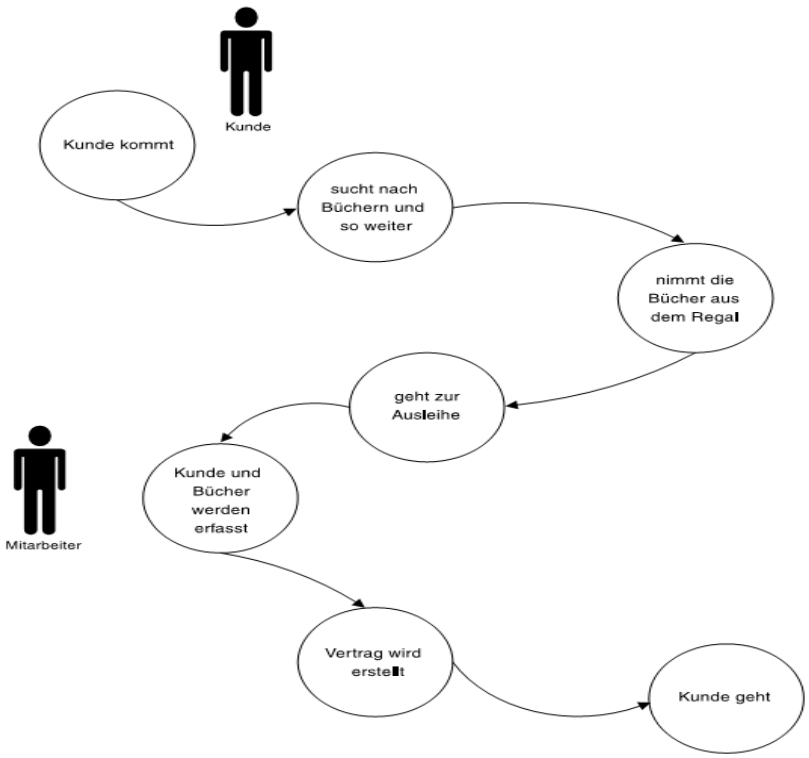
- Welche Tätigkeit führen Sie aus? (Bitte vergeben Sie nach Möglichkeit einen kurzen beschreibenden Namen)
- Welche Daten benötigen Sie für die Ausführung?
- Woher stammen die Daten?
- Welches Ergebnis erzeugen Sie?
- Wohin wird das Ergebnis weitergeleitet?
- Sehr hilfreich ist auch das Abfragen periodischer Tätigkeiten, wie
- Schreiben Sie bitte Ihre täglichen Tätigkeiten auf.
- Schreiben Sie bitte Ihre wöchentlichen Tätigkeiten auf.

Tipp: In vielen Unternehmen gibt es bereits fertige Stellenbeschreibungen.

Fragen Sie gegebenenfalls in der Personalabteilung nach. Oft fällt es allerdings Mitarbeitern schwer, besonders Prozesse eindeutig zu beschreiben. Lassen Sie dann einfache Skizzen erstellen, die die einzelnen Schritte in der richtigen Reihenfolge und die Beteiligten darstellen. So entsteht eine Art Drehbuch – im Fachjargon auch Storyboard genannt.

Storyboard ist die englische Übersetzung für „Drehbuch“

Für den Prozess „Ausleihe“ in der Bücherei könnte diese Skizze zum Beispiel so aussehen:



Weitere Möglichkeiten, Informationen zu beschaffen, sind:

• Interviews mit Mitarbeitern,
• Brainstorming-Workshops,
• teilnehmende Beobachtung oder
• die direkte Mitarbeit als eine Art „Praktikant“.

Brainstorming ist eine Technik, bei der spontane Einfälle zu einem Thema gesammelt werden. Die Strukturierung erfolgt in der Regel erst nach dem Sammeln der Einfälle.

Bei der teilnehmenden Beobachtung nimmt ein Dritter als „Zuschauer“ an einem Arbeitsschritt teil. Er wirkt nicht aktiv mit, sondern versucht, den Arbeitsschritt möglichst exakt festzuhalten – zum Beispiel durch Notizen.

Wörtlich übersetzt bedeutet ***brainstorm*** so viel wie „Geistesblitz“.

Das Lastenheft für die Leihbücherei

Nehmen wir jetzt einmal an, die Informationssammlung für die Leihbücherei ist abgeschlossen und hat folgende Ergebnisse gebracht:

Die Bücherei verleiht Medien. Dazu gehören Bücher, Zeitschriften, Musik-CDs und Filme auf Video und DVD. Jedes Medium hat eine eindeutige Kennzeichnung. Neben den ausleihbaren Medien gibt es Zeitschriften, die nur in den Räumen der Bücherei selbst gelesen werden dürfen. Eine Ausleihe ist nicht möglich. Diese Zeitschriften sind der Präsenzbestand.

Die Medien können nur von Personen ausgeliehen werden, die einen Leihausweis besitzen. Im Leihausweis werden der Name und die Anschrift und eine eindeutige laufende Nummer festgehalten. Personen mit einem Leihausweis sind die Kunden der Bücherei.

Bei der Ausleihe sucht sich der Kunde die gewünschten Medien selbst in der Bücherei aus. An einer zentralen Stelle werden abschließend die Daten des Kunden, die ausgeliehenen Medien und das späteste Rückgabedatum festgehalten. Diese Daten werden im Leihvertrag erfasst.

Die Rückgabe erfolgt an einer anderen Stelle der Bücherei. Hier werden die Daten des Kunden und die zurückgegebenen Medien erfasst. Damit endet der Leihvertrag.

Die zurückgegebenen Medien werden zentral gesammelt und mehrmals täglich durch Aushilfen wieder in die Regale einsortiert. Medien, die ausgeliehen sind, kann ein anderer Kunde für sich reservieren. An mehreren Stellen der Bücherei sollen Kunden Listen abrufen können, welche Medien grundsätzlich im Bestand der Bücherei sind und welche Medien zurzeit ausgeliehen werden können. Zusätzlich sollen die Kunden an dieser Stelle auch Zugriff auf die Daten der Universitäts-Bibliothek nehmen können. Damit sind die wichtigsten Prozesse, Leistungen, Akteure und Daten beschrieben.

Das Lastenheft könnte jetzt so aussehen:

1. **Zielbestimmung**

   Die Software Buch 2010 soll die Leihbücherei bei der Verwaltung der Kunden und Medien sowie bei der Ausleihe und der Rückgabe der Medien unterstützen. Außerdem soll sie den Kunden einen Überblick über die vorhandenen und ausleihbaren Medien verschaffen können.

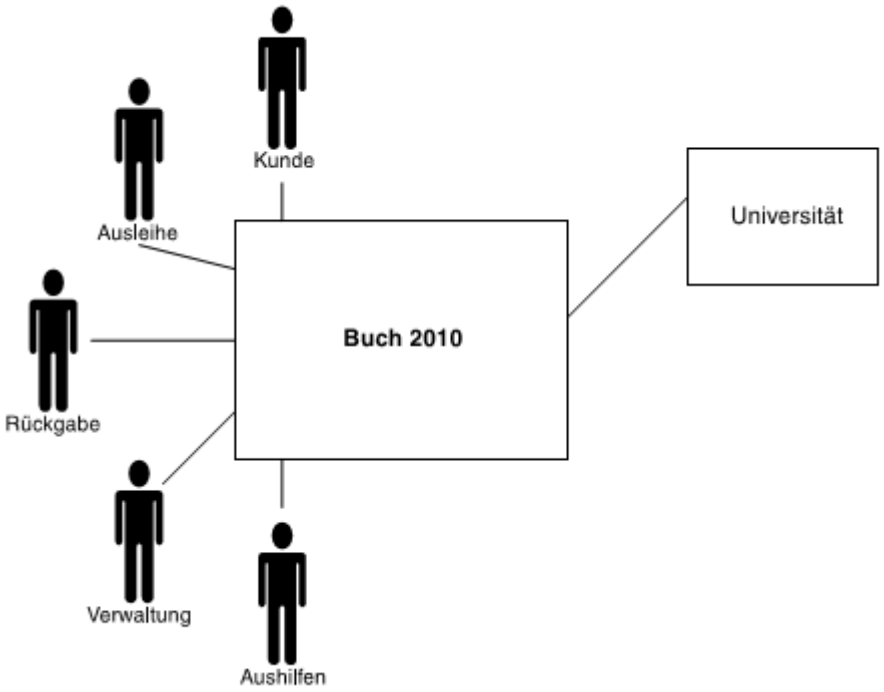
2. **Produkteinsatz**

   Das Produkt wird in den Räumen der Leihbücherei eingesetzt. Es wird von Mitarbeitern und Aushilfen in der Ausleihe, der Rückgabe, der Verwaltung und von Kunden bedient.

3. **Produktübersicht**

   Wesentliche Akteure sind die Mitarbeiter und Aushilfen sowie die Kunden. Außerdem soll die Software mit dem System der Universitäts-Bücherei verbunden werden.

Die Umwelt lässt sich grafisch so darstellen:



4. Produktfunktionen

Die wesentlichen Produktfunktionen werden im Folgenden immer mit einer eindeutigen Nummer, dem Namen, den Akteuren und einer Kurzbeschreibung der Funktion dargestellt:

LF10 **Pflege der Kundendaten**

Akteure Kunde, Mitarbeiter in der Verwaltung

Beschreibung Daten zum Kunden werden neu erfasst, geändert oder gelöscht.

LF20 **Pflege der Mediendaten**

Akteure Mitarbeiter in der Verwaltung

Beschreibung Daten zu den Medien werden erfasst, geändert oder gelöscht.

LF30 **Ausleihe**

Akteure Kunde, Mitarbeiter in der Ausleihe

Beschreibung Der Kunde sucht die Medien aus. Die Daten zum Kunden und zu den Medien werden erfasst und zusammen mit der maximalen Ausleihdauer im Leihvertrag festgehalten.

LF40 **Rückgabe**

Akteure Kunde, Mitarbeiter in der Rückgabe

Beschreibung Der Kunde gibt die Medien zurück. Die Daten zum Kunden und zu den zurückgegebenen Medien werden erfasst.

LF50 **Einsortieren der zurückgegebenen Medien**

Akteure Aushilfen

Beschreibung Die zurückgegebenen Medien aus LF40 werden wieder in die Regale einsortiert.

LF60 **Reservierung**

Akteure Kunde, Mitarbeiter in der Ausleihe

Beschreibung Der Kunde reserviert ein aktuell ausgeliehenes Medium für sich selbst.

LF70 **Listen**

Akteure Kunde

Beschreibung Der Kunde kann Listen zu den verfügbaren Medien erzeugen. Dabei können alle Medien oder nur die aktuell ausleihbaren Medien angezeigt werden. Zusätzlich soll ein Zugriff auf entsprechende Listen der Universitätsbibliothek möglich sein.

5. Produktdaten

LD10 Kundendaten (maximal 100 000)

LD20 Mediendaten (maximal 1 000 000)

LD30 Ausleihdaten (maximal 100 000)

LD40 Reservierungsdaten (maximal 10 000)

6. Produktleistungen

LL10 Die Erzeugung der Listen aus LF70 soll nicht länger als 20 Sekunden dauern.

LL20 Die Liste aus LF70 soll jeweils 20 Einträge auf einer Bildschirmseite anzeigen.

7. Qualitätsanforderungen

Qualitätsfaktor	sehr gut	gut	normal	unwichtig
Zuverlässigkeit			x	
Robustheit			x	
Benutzbarkeit	x			
Wiederverwendbarkeit			x	
Änderbarkeit			x	
Portierbarkeit				x

Die Benutzbarkeit muss sehr gut sein, da die Aushilfen ständig wechseln. Umfangreiche Schulungen können daher nicht erfolgen.

8. Ergänzungen

Das Einlesen der Daten in LF30 und LF40 soll automatisiert erfolgen. Dazu werden Strichcodes und entsprechende Lesegeräte verwendet.

Die Anbindung an das System der Universität erfolgt in der ersten Version lediglich durch eine Verknüpfung zur Internet-Seite.

Weitere Vorgehensmodelle

Prototyping

Schauen wir uns zunächst einmal an, was sich hinter den Begriffen „**Prototyp**“und „**Prototyping**“verbirgt:

Prototypen sind – allgemein betrachtet – erste Ausprägungen eines Produktes, die wesentliche Eigenschaften des späteren Endprodukts darstellen. So werden zum Beispiel in der Automobilbranche Prototypen gebaut, um das Fahrverhalten eines Autos unter realen Bedingungen zu testen oder einfach nur, um das Auto auf einer Messe präsentieren zu können.

Merke: Das Erstellen eines Prototyps wird Prototyping genannt.

Prototypen lassen sich auch in der Software-Entwicklung sehr gut einsetzen zum Beispiel, um dem Anwender Teile des Systems oder auch das vollständige System zu demonstrieren. Damit lässt sich ein großes Dilemma lösen: Die oft sehr abstrakte Beschreibung des Systems die Systemspezifikation wird für den Anwender sichtbar und damit auch sehr viel leichter zu verstehen. So lassen sich auch Fehler in der Systemspezifikation, die zum Beispiel durch Missverständnisse aufgetreten sind, sehr viel schneller aufspüren.

Prototypen in der Software-Entwicklung sind in der Regel ausführbare Programme beziehungsweise simulieren ausführbare Programme.

Über einem Prototyp lassen sich zum Beispiel sehr gut einzelne Verarbeitungsschritte abbilden. Der Prototyp kann beispielsweise eine Eingabemaske mit verschiedenen Eingabefeldern simulieren und so dem Anwender einen Eindruck verschaffen, wie die Datenerfassung im fertigen Produkt erfolgen wird. Der Anwender kann dann bereits an den Prototypen kontrollieren, ob alle wesentlichen Informationen eingegeben werden können und ob er mit dem Ablauf an sich zurechtkommt. Eine andere Möglichkeit für den Einsatz eines Prototyps wäre die Darstellung von Listen für die Ausgabe.

Prototypen eignen sich damit also sehr gut für eine **Validierung** durch die Anwender.

Grundsätzlich lassen sich bei der Software-Entwicklung verschiedene Arten von Prototypen unterscheiden:

- der vollständige Prototyp,
- der unvollständige Prototyp,
- der „Wegwerf“-Prototyp und
- der wiederverwendbare Prototyp.

Beschreibung einzelnen Arten:

Der vollständige Prototyp bildet alle wesentlichen Funktionen des Systems vollständig ab. Er wird damit quasi zur Grundlage für die Systemspezifikation. Da der Aufwand und damit die Kosten für die Erstellung allerdings sehr hoch sind, werden vollständige Prototypen nur sehr selten eingesetzt.

Vollständige Prototypen werden auch **Pilotsystem** genannt.

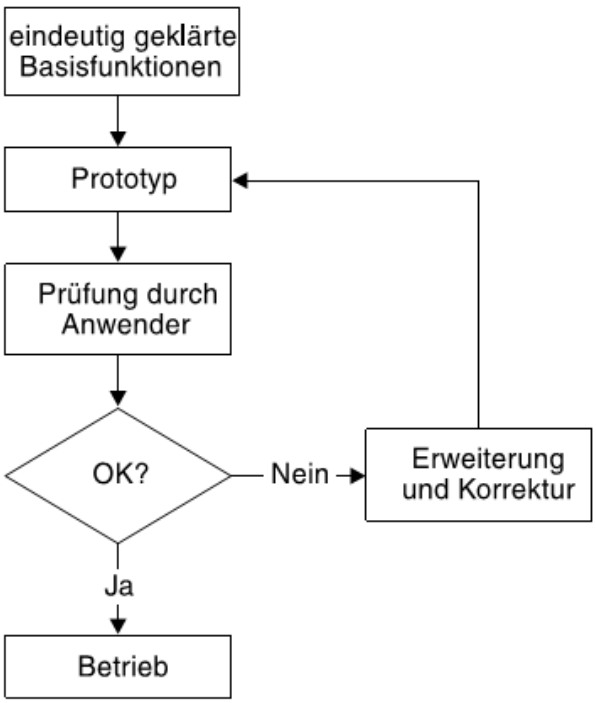
Der unvollständige Prototyp setzt nur bestimmte Teile des Systems um – zum Beispiel lediglich einen bestimmten Prozess oder die grafische Benutzeroberfläche. Unvollständige Prototypen lassen sich noch weiter unterteilen in vertikale Prototypen und horizontale Prototypen. Ein vertikaler Prototyp bildet dabei einen bestimmten Prozess durch sämtliche Schichten des Systems ab – zum Beispiel die Ausleihe eines Buchs in unserer Leihbücherei. Ein horizontaler Prototyp dagegen bildet eine Schicht des Systems möglichst vollständig nach – zum Beispiel die grafische Oberfläche. Hier wird dann nur demonstriert, wie die Oberfläche aussehen könnte. Die Funktionalität – also zum Beispiel die Reaktion auf einen Mausklick auf eine Schaltfläche dagegen fehlt.

Ein „Wegwerf"-Prototyp wird nur zur Demonstration eingesetzt – zum Beispiel, um dem Anwender einen ersten kurzen Eindruck zur vermitteln oder um einem möglichen Kunden eine Vorstellung von der Software zu geben. Nach der Demonstration wird der Prototyp nicht mehr verwendet – er wird „weggeworfen". Beim „Wegwerf"-Prototyp zählt vor allem die Geschwindigkeit der Entwicklung. Alle anderen Aspekte spielen keine Rolle.

Wegwerf"-Prototypen werden auch **Throw-away-Prototypen** genannt. Häufig findet sich auch die Bezeichnung **Rapid Prototyping**.

Ein wiederverwendbarer Prototyp schließlich dient zur schrittweisen Entwicklung des vollständigen Systems. Dazu wird in der Regel zunächst ein erster Prototyp erstellt, der einige wenige eindeutig geklärte Basisfunktionen enthält. Dieser Prototyp wird dann von den Anwendern geprüft und beurteilt. Anhand der Beurteilung wird der erste Prototyp verbessert und weiter ausgebaut. Der so entstandene neue Prototyp wird wieder durch die Anwender beurteilt und weiter ausgebaut. So entsteht ein evolutionäres Modell.

Eine **Evolution** ist eine ständig fortschreitende Entwicklung



Ein evolutionäres Modell mit Prototypen