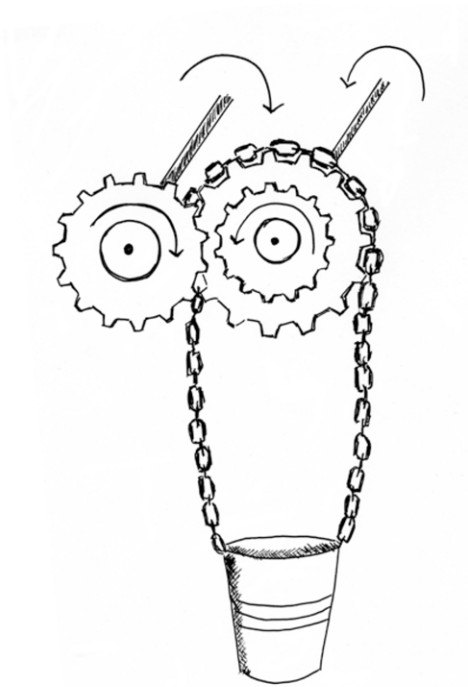


# Kapitel 2

## Verständlichkeit durch geteilte Abstraktionen



- 2.1 Divergenz im Problembereich und Konvergenz im Lösungsbereich
- 2.2 Spezifikation als Abstraktion einer Anwendungsfunktion
- 2.3 Abläufe spezifikationsorientiert
- 2.4 Funktionalität und Architektur mit Freiheitsgraden
- 2.5 Sichten auf Funktionen in Analyse und Entwurf
- 2.6 Anwendungsfälle versus User Stories
- 2.7 Ausprägungen der Entwicklungsschritte

## 2 Verständlichkeit durch geteilte Abstraktionen

Dieses Kapitel ist ein rein technisches Kapitel. Es befasst sich mit dem Know-how der Entwickler für das Erstellen erfolgreicher Abstraktionen und betrachtet hierbei Erkenntnisse des klassischen Software Engineering, die im agilen Umfeld von Nutzen sein können. Häufig werden Elemente aus der spezifikationsorientierten und der agilen Entwicklung individuell kombiniert. Somit kann es vorkommen, dass ein agiles Team beispielsweise Anwendungsfälle statt User Stories<sup>12</sup> verwendet.

Programmcode ist fein und versteckt die "wahren" Absichten eines Programmierers. Um eine klare Sprache zu sprechen, muss man abstrakt Begriffe einordnen und in Modellvorstellungen reden können. Selbst wenn man den Quellcode über alles liebt, benötigt man Abstraktionen – also Begriffe und vereinfachende Modelle – um sich den Kollegen und den Kunden gegenüber verständlich zu machen.

Wichtige **Abstraktionen** bzw. deren Voraussetzungen werden im Folgenden diskutiert:

- **Problembereich und Lösungsbereich**

In Kapitel 2.1 wird die Dualität der Begriffe Problembereich/Lösungsbereich betrachtet, da sich ein Entwickler nicht nur im Lösungsbereich, sondern auch im Problembereich auskennen muss, um den Kunden zu verstehen. Hierbei sind beide Bereiche klar voneinander abzugrenzen, um Fehler im Denken zu vermeiden. Ein Entwickler sollte stets sagen, ob er sich mit seinen Überlegungen und Modellen gerade im Problembereich oder im Lösungsbereich befindet, damit seine Aussagen korrekt gedeutet werden können.

Direkt in den Lösungsbereich zu wechseln, ist ein limitierendes Denken. Es soll zuerst im Problembereich divergierend analysiert werden, was von der Anwendung her alles gut sein könnte. Hierbei sind diverse Alternativen möglich. Dann erst darf das konvergierende Denken im Lösungsbereich erfolgen, das die beste technische Lösung herausarbeitet.

- **Spezifikation als Abstraktion einer Anwendungsfunktion**

Die Dualität der Leistungen Spezifikation/benutzbare Anwendungsfunktion wird in Kapitel 2.2 behandelt. Zu jeder Anwendungsfunktion gehört irgendeine Art von **Spezifikation als Abstraktion**. Man sollte nie ins Blaue hinein programmieren, ohne vorher seine Absicht im Projekt **high-level** vorzustellen und für seine Ideen ein **shared understanding** im Projekt zu gewinnen. "Vom Hirn ins Terminal" hat man um 1960 noch praktiziert. Diese Zeiten sollten längst vorbei sein. Was die Ausprägung einer Spezifikation ist, kann anwendungsorientiert und agil grundlegend verschieden sein. Das sei jetzt dahingestellt. Aber ohne **geteilte Abstraktionen** verliert ein Projekt seine Ziele.

---

<sup>12</sup> User Stories wurden von Kent Beck bei Extreme Programming vorgeschlagen, sind aber bei den sonstigen agilen Ansätzen nicht verbindlich.

- **Abläufe spezifikationsorientiert**

Die Dualität der Leistungen Anwendungsfall/Szenario der spezifikationsorientierten Welt wird in Kapitel 2.3 betrachtet, da sie den Unterschied zwischen vollständigem Wissen und Teilwissen beim Ablauf eines Anwendungsfalls widerspiegelt. Auch ein agiler Entwickler muss wissen, dass er sich einer vollständigen Lösung nur annähern kann. Wie schon erwähnt, werden Anwendungsfälle wegen ihrer Klarheit auch in der agilen Welt verwendet. Das Verwenden von User Stories ist nur bei Extreme Programming Pflicht.

- **Funktionalität und Architektur mit Freiheitsgraden**

Auf die Dualität von Funktionalität und Architektur wird in Kapitel 2.4 eingegangen. Diese Dualität stellt für jedes zu bauende System den eigentlichen Knackpunkt dar. Die gewählten Funktionen sollen dergestalt im Rahmen der gewählten Architektur ablaufen, dass auch weitere "vernünftige" Funktionen in den Rahmen dieser Architektur passen und gegebenenfalls Funktionsänderungen möglich sind. Die **Architektur** ist ein **Modell für die Struktur und den Ablauf von Funktionen**. Ändert man die Architektur, so kann dies Einfluss auf die Funktionen haben. Ändert man die Funktionen, so kann die seitherige Architektur obsolet werden.

Bei **agilen Verfahren** ist eine **Änderbarkeit** von Funktionen und Architektur **Voraussetzung für den Erfolg**.

- **Sichten auf Funktionen in Analyse und Entwurf**

Die jeweils bei Analyse bzw. Entwurf im Zentrum des Interesses stehenden Kategorien von Funktionsklassen werden in Kapitel 2.5 analysiert.

Es sollte stets klar sein, ob man gerade analysiert oder entwirft, da je nachdem die Behandlung anderer Funktionsklassen erwartet wird. Analyse und Entwurf dürfen auch bei einer agilen Vorgehensweise nicht "durcheinandergewürfelt" werden. Dies hilft, unnötige Verwirrungen zu vermeiden.

- **Anwendungsfälle versus User Stories**

Kapitel 2.6 befasst sich mit der spezifikationsorientierten Beschreibung der Abläufe von Anwendungsfällen, mit der agilen Beschreibung von Abläufen einer Funktion mit Hilfe von User Stories und dem Unterschied zwischen Anwendungsfällen und User Stories. Auch wenn man agil entwickelt, sollte die spezifikationsorientierte Welt nicht unbekannt sein.

- **Ausprägungen der Entwicklungsschritte**

Die Entwicklungsschritte für ein spezifikationsorientiertes bzw. agiles Vorgehen werden in Kapitel 2.7 analysiert, da es hierbei durchaus Unterschiede gibt, obwohl in beiden Fällen Anforderungen erstellt, analysiert, entworfen, programmiert und getestet werden. Der Unterschiede muss man sich jedoch bewusst sein!

## **2.1 Divergenz im Problembereich und Konvergenz im Lösungsbereich**

Wie schon gesagt, ist es ein limitierendes Denken, direkt in den Lösungsbereich zu wechseln. Es soll zuerst im Problembereich divergierend analysiert werden, was von

der Anwendung her alles gut sein könnte. Hierbei sind diverse Alternativen möglich. Dann erst darf das konvergierende Denken im Lösungsbereich erfolgen, das die beste technische Lösung herausarbeitet.

## Problembereich

Software schwebt nicht im freien Raum. Software wird geschrieben, um die Probleme eines Anwenders zu lösen. Ehe man loslegt zu programmieren, sollte man also wissen, was programmiert werden soll.<sup>13,14</sup> Entweder gibt es schon mehr oder weniger genaue Forderungen, die noch zu schärfen sind, oder aber diese Forderungen sind erst zu ermitteln.<sup>15</sup> Forderungen sind im Wesentlichen funktional und betreffen dann die **Verarbeitung** der Geschäftsprozesse des Problembereichs. Dabei ist der **Problembereich (Problem Domain)** derjenige Ausschnitt der realen Welt, der durch Software zu unterstützen ist.

Um die Vorstellungen des Kunden zu erfüllen, ist es bei nicht trivialen Aufgabenstellungen notwendig, den Problembereich zu betrachten und (in Teilen) zu modellieren bzw. durch eine einfache Beschreibung zu erfassen. Die Verarbeitung der Geschäftsprozesse des Problembereichs wird im Rahmen der **Analyse** aus logischer und nicht aus Implementierungssicht analysiert und modelliert. Dies geschieht, damit das Wissen über den Problembereich in abstrakter Form erfasst wird. Ein Programmierer darf also nicht nur in der Welt der Konstruktion eines Programms, dem sogenannten Lösungsbereich<sup>16</sup>, denken, sondern er muss stets auch den Problembereich kennen, damit das Ziel des Projekts nicht verfehlt wird. Er muss die Dualität Problembereich/Lösungsbereich virtuos beherrschen.

Der sogenannte **Problembereich** oder **Problem Domain** entspricht dem Ausschnitt der realen Welt, der später durch die zu realisierende Software abgedeckt werden soll. Der Problembereich wird aber nur aus einer logischen, idealen Sicht betrachtet. Man führt hierbei eine **Analyse des Problembereichs** durch und modelliert die **Verarbeitungsschritte der Geschäftsprozesse** ohne die Betrachtung technischer Funktionen wie beispielsweise einer persistenten Datenhaltungsfunktion oder einer grafischen Ausgabe.



Ein technisches System gibt es im Problembereich noch nicht. Technische Funktionen wie eine persistente Datenhaltungsfunktion oder eine grafische Ausgabe gibt es **normalerweise** erst ab dem Entwurf, also im sogenannten Lösungsbereich.

Die Betrachtung des Problembereichs ist absolut erforderlich, um den Kunden zu verstehen. Dies ist vollkommen unabhängig davon, welches Vorgehensmodell für die Software-Entwicklung gewählt wird. Der Problembereich ist sowohl bei spezifikations-

<sup>13</sup> Dies wurde in der agilen Entwicklung in der Praxis zunächst unterschätzt. Autoren wie Eric Evans [Eva03] oder Jeff Patton [Pat14] betonen die Bedeutung des Problembereichs für agile Projekte.

<sup>14</sup> Bei einem spezifikationsorientierten Vorgehen ist das Studium des Problembereichs unumgänglich. Aber auch bei einem agilen Vorgehen braucht man eine Vision des zukünftigen Systems, insbesondere, welche Zielgruppe und welche Probleme unterstützt werden sollen, damit sich ein jeder zurecht findet und nicht am Ziel des Projekts vorbeiprogrammiert.

<sup>15</sup> Besonders wichtig ist bei agilen Verfahren eine mit dem Kunden abgestimmte Version des zukünftigen Systems.

<sup>16</sup> Der Lösungsbereich umfasst entwerfen, implementieren, testen und integrieren.

orientierten<sup>17</sup>, als auch bei agilen<sup>18</sup> Methoden zu betrachten. So orientiert sich das Domain-Driven Design von Eric Evans [Eva03] an der agilen Softwareentwicklung und betont dennoch die Notwendigkeit des Studiums des Problembereichs, ehe programmiert wird. Auch wenn man bereits programmiert, muss man stets bereit sein, iterativ zu arbeiten und sich bei Fehlern oder Änderungen erneut mit dem Problembereich zu befassen. Erwartet wird, dass Entwickler und Fachexperten stets eng zusammenarbeiten.

Der größte Unterschied bei der agilen Entwicklung ist, dass man den Problembereich nur an derjenigen Stelle genauer anschaut, die vor ihrer Programmierung steht.

### Lösungsbereich

Das Wissen des **Lösungsbereichs** (Entwurf, Implementierung, Test & Integration) baut auf dem Wissen des Problembereichs über die Verarbeitungslogik auf. Beim Betreten des Lösungsbereichs tritt aber beim Entwurf neues, technisches Wissen zu dem Wissen über die Verarbeitungsfunktionen der Geschäftsprozesse hinzu.

Die Modellierung der Verarbeitungslogik des Problembereichs und die technische Lösung des Lösungsbereichs auf einen Schlag bewerkstelligen zu wollen, würde den Menschen überfordern und wäre fatal. Man muss hier sequentiell vorgehen.



Im **Lösungsbereich** befasst man sich mit der **Programmkonstruktion**<sup>19</sup>. Dabei stellt man Programme auf, welche die Daten der betrachteten Anwendung bearbeiten. Im Lösungsbereich geht es um die reale Welt mit allen ihren physischen Randbedingungen und Einschränkungen. Im Lösungsbereich existiert ein technisches System mit seinen Rechnern. Betritt man den Lösungsbereich, so legt man beim Entwurf die Architektur des Systems fest, d. h. die Statik und die Dynamik der Programme.<sup>20</sup>

## 2.2 Spezifikation als Abstraktion einer Anwendungsfunktion

Ein Kunde will durch die Anwendungsfunktionen eines Programms unterstützt werden. Anwendungsfunktionen beruhen auf Ideen, sei es, dass diese Ideen als schriftliche Spezifikation auf Papier oder im Kopf der Entwickler vorliegen. Ob eine Spezifikation durch User Stories aus dem agilen Umfeld erfolgt oder durch **Anwendungsfälle**, die aus der spezifikationsorientierten Welt bekannt sind, sei dahingestellt.

Der Begriff eines **Anwendungsfalls** (engl. **Use Case**) wurde von Jacobson [Jac92] bekannt gemacht und hat sich weltweit durchgesetzt. In diesem Buch wird dieser

<sup>17</sup> Arbeitet man spezifikationsorientiert, so erstellt man vor der Programmierung eines Systems die Spezifikation des Systems.

<sup>18</sup> Arbeitet man agil, so produziert man zyklisch lauffähige Fragmente eines Systems in der Hoffnung, dass die Summe der Fragmente das gewünschte Ganze ergibt.

<sup>19</sup> Die Machbarkeit einer technischen Lösung sollte parallel zur Analyse des betrachteten Ausschnitts des Problembereichs untersucht werden.

<sup>20</sup> Das ist eine einfache Vorstellung. In Wirklichkeit wird eine Architektur iterativ entwickelt.

Begriff in seiner deutschen Übersetzung verwendet. Entscheidend ist, dass beim Start des Programmierens eines Anwendungsfalls ein ausreichend stabiles, mit dem Kunden abgestimmtes Verständnis des Gewünschten vorliegt, das iterative Verbesserungen erlaubt.

Anwendungsfälle der spezifikationsorientierten Welt sind meist Spezifikationen für **Anwendungsfunktionen**<sup>21</sup>, die durch den Nutzer aufgerufen werden. Eine Anwendungsfunktion realisiert dabei den Vertrag eines (spezifizierten) Anwendungsfalls, in anderen Worten, was der Anwendungsfall vorschreibt. Ein Anwendungsfall kann sich jedoch im Problembereich oder dem Lösungsbereich befinden. Eine benutzbare Funktion befindet sich stets im Lösungsbereich. Sogenannte **Core level concerns** umfassen die Verarbeitung der Anwendungsfälle der Systemanalyse. Sogenannte **Cross-Cutting-concerns** erfassen die querschnittlichen, technischen Funktionen (siehe Kapitel 2.5.3).

Ein Anwendungsfall ist eine Leistung oder ein Service eines Systems für einen Nutzer. Er muss nicht von einem Nutzer ausgelöst werden, aber kommt ihm zugute.



Anwendungsfälle der spezifikationsorientierten Welt und ihr Vergleich mit User Stories der agilen Welt werden in Kapitel 2.6 behandelt.

## 2.3 Abläufe spezifikationsorientiert

Anwendungsfälle und Szenarien der spezifikationsorientierten Welt sind beides Spezifikationen. Sie erlauben es dem Entwickler, in der Erwartungshaltung eines Nutzers des zukünftigen Systems zu denken.

Ein **Anwendungsfall** spezifiziert eine Leistung eines Systems für einen Nutzer, ist Teil eines Geschäftsprozesses und muss ein Ergebnis haben. Ein Anwendungsfall ist sozusagen ein Typ und weist in seinem Ablauf Variablen auf, die verschiedene Werte annehmen können. Damit sind Alternativen im Ablauf, d. h. verschiedene Zweige des Ablaufs, möglich.

Ein **Szenario in der spezifikationsorientierten Welt** ist eine spezielle Ausprägung eines Anwendungsfalls. Ein **Szenario eines Anwendungsfalls** hat scharf definierte Werte für die Variablen eines Anwendungsfalls und ist damit in seinem Ablauf alternativlos.

Schon das Vorliegen eines Szenarios für einen Basisablauf eines Anwendungsfalls kann aber für einen Kunden bereits sehr hilfreich sein. Der Basisablauf eines Anwendungsfalls ist der sogenannte "Gut"-Fall. Im "Gut"-Fall gibt es keine Fehler in einer Anwendung.

---

<sup>21</sup> Aber nicht jeder Anwendungsfall führt zu einer Anwendungsfunktion. Es gibt Anwendungsfälle, die zeitgesteuert ausgelöst werden und keinem Aufruf einer Anwendungsfunktion eines Nutzers entsprechen.

Ein Anwendungsfall ist auf Grund seiner Alternativen deutlich mächtiger als ein Szenario. Startet man nur mit Szenarien im Sinne von scharfen Werten für Anwendungsfälle, so muss man dieses Wissen iterativ ergänzen, um zu breiter verwendbaren Lösungen zu kommen.

Zwischen einem Anwendungsfall und einem Szenario eines Anwendungsfalls gibt es in der **spezifikationsorientierten Welt** die folgenden Unterschiede:

- Ein Anwendungsfall arbeitet mit Variablen und Alternativen.
- Ein Szenario arbeitet mit scharf definierten Variablen. Deshalb kann es bei einem Szenario keine verschiedenen Zweige des Ablaufs (Alternativen) geben.



Zu einem Ablauf eines Anwendungsfalls gibt es viele Szenarien. Ein Anwendungsfall ist sozusagen der Typ und ein Szenario ist die Ausprägung.

## 2.4 Funktionalität und Architektur mit Freiheitsgraden

Anwendungsfunktionen stellen für einen Anwender oder Nutzer eines Systems einen Mehrwert dar. Anwendungsfunktionen laufen im Bauplan der gewählten Architektur. Eine Architektur beschreibt die Zerlegung eines Systems in Teile, das Zusammenwirken dieser Teile und die Beschreibung der Strategie für diese Architektur, um die für ein System geforderten Leistungen, beispielsweise die Leistung eines Anwendungsfalls, zu erbringen.

Die Architektur eines Systems wird so konstruiert, dass die gewünschten Anwendungsfunktionen nach dem Bauplan der Architektur ablaufen können. Die Architektur eines Systems sollte jedoch auch stabil sein und bei einer Erweiterung des Systems um eine weitere gewünschte "vernünftige" Anwendungsfunktion nicht zerbrechen. Zusätzlich zu den Anwendungsfunktionen kann eine Architektur aber noch weitere Funktionen<sup>22</sup> enthalten, beispielsweise zum Start-up (Betriebssicherheit), zur Informationssicherheit oder zur Parallelität und Interprozesskommunikation.

Eine Architektur stellt zwar Anwendungsfunktionen zur Verfügung, gilt aber in der Regel auch für weitere Anwendungsfunktionen. Sie wird für eine Gruppe ähnlicher Funktionen geschrieben und ist nicht spezifisch für eine spezielle Funktion. Architektur und Funktionen ergänzen einander und sind komplementär.

Auch wenn eine Architektur die Ablauffähigkeit von Funktionen ermöglicht, so muss sie auch **nicht-funktionalen Anforderungen** genügen, insbesondere Forderungen an die

- Performance,
- Bedienbarkeit,
- Wiederverwendbarkeit der Komponenten,
- Änderbarkeit,

<sup>22</sup> Auf die funktionale Sicherheit wird in diesem Buch nicht eingegangen.

- Ausbaufähigkeit und Wiederverwendbarkeit des Systems (vor allem bei einem Stufenplan<sup>23</sup>) oder
- Verständlichkeit.

Bei den **agilen Ansätzen** ist die **Änderbarkeit** extrem wichtig, da man sich in einem zunächst unbekannten Gebiet bewegt, das erst mit der Zeit erkundet wird. Zur Stabilisierung der Qualität bei häufigen Änderungen benötigt man **automatische Tests** und die Möglichkeit zu schnellem **Refactoring** (Gewinnung einer besseren Architektur durch kontinuierliche Überarbeitung von problematischen Stellen im Code).



Bei Agilität ist eine Änderbarkeit der Architektur und Funktionen eine zwingende Voraussetzung für den Erfolg eines Software-Systems.



Eine rein funktionale Betrachtung eines Systems reicht nicht aus. Eine **Architektur** muss die **funktionalen Anforderungen** und die **nicht-funktionalen Anforderungen** erfüllen. Eine Architektur wird für eine Gruppe ähnlicher Funktionen geschrieben und ist nicht spezifisch für eine spezielle Funktion.



Funktionalität und Architektur sind komplementär zu sehen. Zwar ermöglicht eine Architektur das Ablaufen einer bestimmten Funktionalität, die Architektur eines Systems selbst stellt aber den Rahmen für die Statik und Dynamik des Systems dar, der nicht von einer speziellen Anwendungsfunktion abhängig sein soll.

## 2.5 Sichten auf Funktionen in Analyse und Entwurf

In diesem Kapitel wird ein Satz von Funktionsklassen<sup>24</sup> genannt, die üblicherweise beim Bau von Softwaresystemen auftreten. Dabei wird erörtert, dass diese Funktionsklassen mit Ausnahme der Verarbeitungsfunktionen der Geschäftsprozesse in der Regel<sup>25</sup> erst beim Entwurf betrachtet werden und nicht bereits in der Analyse, in der analysiert und aus logischer Sicht modelliert wird. Die verschiedenen Funktionsklassen sind zueinander komplementär.

Anwendungsfunktionen sind Funktionen eines Systems aus Anwendersicht. Anwendungsfunktionen und damit auch Anwendungsfälle können Teilstücke der noch zu erklärenden Grundfunktionen der Informationstechnik enthalten, aber auch Teilstücke von Funktionen, die gewährleisten, dass ein System bestimmte Eigenschaften oder Qualitäten hat, die auf einzelne Funktionen abgebildet werden können. Zu solchen Funktionen gehören beispielsweise Funktionen der Betriebssicherheit oder für die erforderliche Parallelität. Sowohl die Grundfunktionen der Informationstechnik als auch diese Qualitäten werden in den folgenden Kapiteln 2.5.1 und 2.5.2 detailliert betrachtet.

<sup>23</sup> Wird ein System in mehreren Bauabschnitten (Stufen) realisiert, so kann das in einem Stufenplan beschrieben werden.

<sup>24</sup> Auf die funktionale Sicherheit (Safety) wird in diesem Buch nicht eingegangen.

<sup>25</sup> Es gibt Ausnahmen.



## 2.5.1 Grundfunktionen der Informationstechnik

Die Grundfunktionen der Informationstechnik sind:

- verarbeiten,
- speichern,
- ein-/ausgeben und
- übertragen.



Also benötigt ein System die folgenden Funktionen:

- Verarbeitungsfunktionen,
- Datenhaltungsfunktionen,
- MMI<sup>26</sup>-Funktionen und
- Übertragungsfunktionen (Funktionen der **Rechner-Rechner-Kommunikation**).

**Verarbeitungsfunktionen** werden objektorientiert in der Regel bereits im Rahmen der **Analyse** modelliert.<sup>27</sup>

**Datenhaltung**, **MMI-Funktionen** und die **Übertragungsfunktionen** (Funktionen der Rechner-Rechner-Kommunikation) werden objektorientiert erst beim **Entwurf** relevant.

Ein stand-alone Rechner braucht keine Rechner-Rechner-Kommunikation. Besteht ein verteiltes System aber aus mehreren Rechnern, so braucht man Funktionen der Rechner-Rechner-Kommunikation. Ferner muss man sich beim Entwurf auch Gedanken darüber machen, wie die Software auf die verschiedenen Rechner verteilt werden soll. Statt von **Verteilung** spricht man auch von **Deployment**.

## 2.5.2 Funktionen zur Gewährleistung von Qualitäten<sup>28</sup>

Softwaresysteme müssen außer ihrer Kernfunktionalität der Grundfunktionen der Informationstechnik auch noch bestimmte **Systemeigenschaften** oder **Qualitäten** aufweisen, die durch das Vorhandensein spezieller Funktionen zu gewährleisten sind. Diese Qualitäten sind<sup>29</sup>:

- Betriebssicherheit,
- Informationssicherheit und
- Parallelität<sup>30</sup>.

<sup>26</sup> MMI bedeutet Man-Machine Interface bzw. Mensch-Maschine-Schnittstelle. Hierfür ist auch der Begriff HMI (Human Machine Interface) gebräuchlich.

<sup>27</sup> Eine detaillierte Analyse findet bei Scrum erst in einem der Sprints statt.

<sup>28</sup> Auf funktionale Sicherheit wird in diesem Buch nicht eingegangen.

<sup>29</sup> Qualitäten können bei der Verwendung von User Stories bei agilen Ansätzen als Akzeptanzkriterien (engl. confirmations) berücksichtigt werden. Das Entwicklungsteam kann daraus direkt geeignete Tests ableiten.

<sup>30</sup> Wenn eine Parallelität erwünscht ist.

Das Vorhandensein dieser Qualitäten beruht auf dem Vorhandensein spezieller Funktionen! Diese Funktionen sollten nicht später in ein System "hineingetestet" werden. Sie können die Architektur entscheiden! Es muss durch frühzeitige und kontinuierliche Tests sichergestellt werden, dass das System bestimmte Funktionen bereits während der Entwicklung erfüllt.<sup>31</sup>

Die Betriebssicherheit enthält die Fähigkeiten, ein System zu starten und zu stoppen, Fehler zu erkennen und Fehler auszugeben. Informationssicherheit (engl. security) umfasst im Wesentlichen den Schutz vor unerwünschten Zugriffen. Eine Parallelität ergibt sich in einem System durch das Vorhandensein von nebenläufigen Prozessen wie Betriebssystemprozessen oder Threads.

Aus der Eigenschaft der Parallelität von Handlungssträngen ergibt sich die Notwendigkeit der Interprozesskommunikation dieser parallelen Prozesse.



Im Folgenden sollen die Betriebssicherheit, Informationssicherheit und Parallelität diskutiert werden:

### • **Betriebssicherheit**

An die Betriebssicherheit werden Anforderungen der folgenden Art gestellt:

- Das System muss beim Start-up in einfacher Weise gestartet werden können und beim Shut-down definiert beendet werden können.
- Das System soll Fehler erkennen und behandeln können (Fehlererkennung und Fehlerbehandlung).
- Das System soll über aufgetretene Fehler informieren (Fehlerausgabe).

Die Funktionen der **Betriebssicherheit** umfassen:

- den Start-up des Systems,
- den Shut-down des Systems,
- eine Fehlererkennung und Fehlerbehandlung wie z. B. eine Rekonfiguration bei verteilten Systemen und
- die Fehlerausgabe zur Laufzeit.



Befasst man sich mit dem Hochfahren (engl. start-up) und Herunterfahren (engl. shut-down) des Systems, so muss man sich zwangsläufig auch mit der **Persistenz der Daten** befassen.

Daten, die beim erneuten Starten des Systems zur Verfügung stehen sollen, müssen spätestens beim shut-down des Systems persistent in einer Datenbank oder in Dateien gespeichert werden.



Bei **verteilten Systemen** wird bei der Fehlerbehandlung oft verlangt, dass das System im Fehlerfall neu konfiguriert (rekonfiguriert) werden kann. Beispiele dafür

<sup>31</sup> Spezifikationsorientierte Ansätze haben hier Schwierigkeiten, da ggf. wiederholt iteriert werden muss, was zum ständigen Neuschreiben der Spezifikation führen kann.

sind ein Umschalten auf andere Rechner oder ein definierter Neustart "abgestürzter" Programme. Ferner muss die Fehlerausgabe zur Laufzeit aufgetretene Fehler an zentraler Stelle dem Nutzer melden (**Single System View**) und eine einfache Zuordnung der Fehlermeldungen zu ihren Ursachen ermöglichen (Fehlerausgabe).

### Aufgetretene Fehler

Da der Entwurf sich mit einem technischen System befasst, muss man beim Entwurf auch mit den technischen Fehlern des Systems fertig werden. Dahingegen befasst man sich bei der Analyse nur mit logischen Fehlern in dem erwarteten Ablauf einer Anwendung.



Ist beispielsweise ein gewünschtes Buch gerade in der Bibliothek ausgeliehen und darum nicht verfügbar, so stellt dies einen Fehlerfall in dem erwarteten Ablauf einer Anwendung "Bücherverwaltung" dar. Ein technischer Fehler ist z. B. eine fehlerhafte Übertragung über eine Kommunikationsverbindung oder ein Festplattencrash.

- **Informationssicherheit**

Des Weiteren werden meist Anforderungen an die **Informationssicherheit** (engl. **security**) gestellt.

Informationssicherheit bedeutet beispielsweise, dass ein bestimmter Nutzer oder ein bestimmtes Fremdsystem nur im Rahmen seiner Berechtigung die Funktionen und Daten nutzen darf.



Dies bedeutet, dass die zu schützenden Objekte eines Systems vor unbefugtem Zugriff bewahrt werden müssen. Hierfür müssen spezielle Informationssicherheitsfunktionen entworfen werden wie z. B. bestimmte Mechanismen für die Authentisierung<sup>32</sup>.

- **Parallelität**

Große Systeme strukturiert man in der Regel in **parallele (nebenläufige) Einheiten**, oftmals verteilt auf mehrere Rechner. Dabei stellen Betriebssystemprozesse im Gegensatz zu Threads, die in einem Betriebssystemprozess ablaufen, verteilbare Einheiten dar.

Eine **Interprozesskommunikation** (engl. **interprocess communication**, IPC) erlaubt einen Informationsaustausch zwischen den jeweiligen parallelen Einheiten (Betriebssystemprozesse oder Threads) auf demselben oder auf verschiedenen Rechnern.



Im Rahmen der objektorientierten Modellierung der Analyse werden grundsätzlich alle Funktionen bzw. Objekte als parallele Einheiten betrachtet. Dagegen muss man sich beim Entwurf mit den in der entsprechenden Programmiersprache bzw.

<sup>32</sup> Das Wort Authentifizierung wird gleichbedeutend verwendet.

dem entsprechenden Betriebssystem oder der entsprechenden **Middleware**<sup>33</sup> zur Verfügung stehenden Mitteln für die **Parallelität** befassen und diese konkret in ein System einbauen. Beim Entwurf ist scharf zwischen einer sequentiellen Programmierung und nebenläufigen Code-Sequenzen zu unterscheiden.

### 2.5.3 Funktionsklassen in Analyse und Entwurf

Im Folgenden werden alle<sup>34</sup> Funktionsklassen für Anwendungssoftware zusammengestellt. Diese sind

- zum einen die in Kapitel 2.5.1 genannten **Grundfunktionen der Informationstechnik** und
- zum anderen die in Kapitel 2.5.2 aufgeführten Funktionsklassen, die aus der Betrachtung von **Qualitäten** resultieren.

Die Grundfunktionen der Informationstechnik sind:

- Verarbeitung,
- Datenhaltung,
- Ein- und Ausgabe sowie
- Übertragung (Rechner-Rechner-Kommunikation).

Qualitäten, die das Vorhandensein spezieller Funktionen voraussetzen, sind:

- Betriebssicherheit,
- Informationssicherheit und
- Parallelität<sup>35</sup>.

Die Summe der Funktionen aus den Grundfunktionen der Informationstechnik und den Funktionen für die Betriebssicherheit, Informationssicherheit und Parallelität minus der Verarbeitung wird in diesem Buch als **technische Funktionen** bezeichnet.



Bis auf die Verarbeitung, die in der Analyse studiert wird, werden die anderen Funktionen – **technische Funktionen** genannt – in der Regel erst beim Entwurf betrachtet.

Im Fall des objektorientierten Ansatzes betrachtet man in der Analyse die **Verarbeitungsfunktionen** aus Sicht des Problembereichs. Ferner wird aufgeschrieben, was ein- und ausgegeben, gespeichert oder übertragen wird. Wie eine Ein- und Ausgabe, Speicherung oder Übertragung erfolgt, ist in der Regel Sache des Lö-

<sup>33</sup> Eine Middleware ist eine Programmschicht, welche sich über mehrere Rechner erstreckt und vor allem eine Interprozesskommunikation für verteilte Anwendungen zur Verfügung stellt. Weitere Funktionen einer Middleware sind beispielsweise Persistenzdienste oder die Verwaltung von Namen.

<sup>34</sup> Die funktionale Sicherheit wird in diesem Buch nicht betrachtet.

<sup>35</sup> Eine Parallelität erfordert natürlich Funktionen der Interprozesskommunikation, um die parallelen Prozesse zu verbinden.

sungsbereichs. Ebenso werden in der Regel die Betriebssicherheit, Informationssicherheit und Parallelität erst im Lösungsbereich relevant.

Die folgende Tabelle zeigt für häufig vorkommende Fälle die Sichten eines Entwicklers im Rahmen des Problembereichs und des Lösungsbereichs für den objektorientierten Ansatz<sup>36</sup>:

Funktionen	Analyse/ Problembereich	Entwurf/ Lösungsbereich
Verarbeitung	detailliert betrachtet	detailliert betrachtet
Datenhaltung	was wird gespeichert	detailliert betrachtet
Ein- und Ausgabe	was wird ein- bzw. ausgegeben	detailliert betrachtet
Rechner-Rechner-Kommunikation	was wird übertragen	detailliert betrachtet
Betriebssicherheit	nicht betrachtet	detailliert betrachtet
Informationssicherheit	nicht betrachtet	detailliert betrachtet
Parallelität/IPC	nicht betrachtet	detailliert betrachtet

Tabelle 2-1 Funktionsklassen im Fall einer objektorientierten Lösung

## 2.6 Anwendungsfälle versus User Stories

Anwendungsfälle und User Stories<sup>37</sup> sind gebräuchliche Mittel, um die Anforderungen an die Abläufe von Funktionen eines Systems aus der Sicht eines Nutzers zu erfassen. Man darf sich aber nicht nur auf die funktionalen Eigenschaften eines Produkts konzentrieren. Auch die nicht funktionalen Anforderungen an ein Produkt müssen erfüllt sein.

Kapitel 2.6.1 behandelt Anwendungsfälle, Kapitel 2.6.2 User Stories, Kapitel 2.6.3 zieht einen Vergleich zwischen Anwendungsfällen und User Stories.

### 2.6.1 Anwendungsfälle

Wird ein **Geschäftsprozess** oder werden Teile eines Geschäftsprozesses **automatisiert**, d. h. auf die Maschine gebracht, so werden den Nutzern dieses Systems Anwendungsfälle in programmierter Form zur Verfügung gestellt.



Ein **Anwendungsfall** ist kein lauffähiges Programm, sondern stellt nur eine **Spezifikation einer Funktion** dar, die oft durch den Nutzer selbst abgerufen werden kann.

<sup>36</sup> Je nach Projekt kann es erforderlich sein, technische Funktionen bereits im Rahmen der Analyse zu behandeln.

<sup>37</sup> User Stories stammen von Kent Beck.

Die Spezifikation einer funktionalen Leistung, die ein **System**<sup>38</sup> im Rahmen eines Geschäftsprozesses einem Anwender zur Verfügung stellt, wird als **Anwendungsfall des Systems** bezeichnet. Ein System kann viele Anwendungsfälle haben.



Ein **Anwendungsfall** kommt einem Nutzer zugute und führt für diesen Nutzer zu einem bestimmten **Ergebnis**. Ein Anwendungsfall muss stets ein Ergebnis haben, sonst liegt kein Anwendungsfall vor.



Nach Jacobson [Jac92] wird jeder Anwendungsfall durch einen sogenannten **Aktor** ausgelöst. Ein Aktor ist eine Rolle, ein Gerät oder Fremdsystem am Rande – also außerhalb – des betrachteten Systems.

Ein Anwendungsfall lässt sich wie folgt charakterisieren:

1. Ein Anwendungsfall gehört zu einem Geschäftsprozess oder Teil eines Geschäftsprozesses und soll automatisiert werden, also vom zu realisierenden System zur Verfügung gestellt werden.
2. Ein Anwendungsfall ist eine funktionale Leistung oder ein Service eines Systems für einen Nutzer.
3. Ein Anwendungsfall kann Alternativen haben.
4. Ein Anwendungsfall ist eine Spezifikation, die zu implementieren ist.
5. Ein Anwendungsfall muss ein Ergebnis haben.

### 2.6.1.1 Anwendungsfälle in der Praxis

Lauffähige **Anwendungsfälle** können in der Praxis entweder

- durch einen **Aktor** wie eine Rolle, ein Gerät oder ein Fremdsystem ausgelöst werden (**ereignisorientierte Anwendungsfälle**),
- zu bestimmten Zeitpunkten oder zyklisch vom System selbst gestartet werden (**zeitgesteuerte Anwendungsfälle**)<sup>39</sup> oder
- durchgehend ablaufen (**fortlaufend aktive Anwendungsfälle**).

Eine ereignisorientierte Auslösung eines Anwendungsfalles bedeutet, dass der Anwendungsfall asynchron ausgelöst wird, z. B. weil der Nutzer eine Taste anschlägt.

### 2.6.1.2 Realisierung von Anwendungsfällen

Im Rahmen der Objektorientierung werden die Anwendungsfunktionen meist nicht durch die Methode eines einzelnen Objekts, sondern durch die Zusammenarbeit von

<sup>38</sup> Es gibt Systeme n-ter Stufe, auf Deutsch: Es gibt auch Anwendungsfälle für Teilsysteme bis hin zu den Klassen. Ein Anwendungsfall ist ein Service, den ein System oder ein Zerlegungsprodukt eines Systems zur Verfügung stellt.

<sup>39</sup> Für die Auslösung zeitgesteuerter Ereignisse führte Jacobson [Jac92] die system clock als Aktor ein. Die Realisierung zeitgesteuerter Ereignisse wird zwar als Anwendungsfall spezifiziert, stellt aber keine Anwendungsfunktion dar, da sie vom Anwender nicht aufgerufen wird.

mehreren Objekten zur Verfügung gestellt. Man sagt in UML dazu, dass ein Anwendungsfall durch eine **Kollaboration von Objekten** realisiert wird. Eine Kollaboration nach UML ist im folgenden Bild gezeigt:

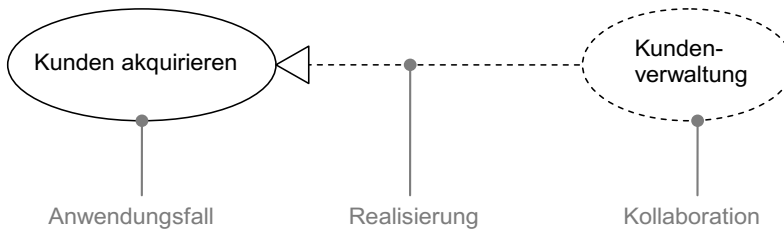


Bild 2-1 Eine Kollaboration

Die Wechselwirkungen zwischen den Objekten werden auf einen Nachrichtenaustausch zwischen den Objekten abgebildet. Bildlich gesprochen heißt dies, dass die Objekte miteinander "reden". Für jeden Anwendungsfall wird dabei ein sogenanntes Kommunikations- oder Sequenzdiagramm erstellt, um den Ablauf des entsprechenden Anwendungsfalls zu studieren. Kommunikationsdiagramme und Sequenzdiagramme können sowohl bei der Analyse als auch beim Entwurf erfolgreich für das Studium der Abläufe von Anwendungsfällen verwendet werden.

### 2.6.1.3 Teilschritte von Anwendungsfällen

Es sollte vermieden werden, einen einzelnen Teilschritt eines Anwendungsfalls als eigenen Anwendungsfall zu bezeichnen. Ein **Teilschritt eines Anwendungsfalls** sollte nur als **Sub-Anwendungsfall** bezeichnet werden.

Die einzelnen **Teilschritte eines Anwendungsfalls** können zeitlich asynchron zueinander ausgeführt werden. Dennoch sind sie miteinander **zeitlich verkettet**, dergestalt, dass ein Teilschritt nach dem anderen ausgeführt wird und zwar jeder Teilschritt nur ein einziges Mal und in einer definierten Reihenfolge. Die einzelnen Teilschritte eines Anwendungsfalls können zwar asynchron zueinander ausgeführt werden, aber sie sind voneinander abhängig. Sie sind logisch verkettet. Sie werden nicht unabhängig voneinander ausgeführt.

Erst, wenn der letzte Teilschritt abgearbeitet ist, liegt das durch den ersten Teilschritt initiierte Ergebnis vor.



Sonst würden Sie beispielsweise 5 Versicherungsverträge erhalten, ohne je einen Antrag gestellt zu haben, wenn jeder Teilschritt ein eigener Use Case wäre.

### Das System Bibliotheksverwaltung als Beispiel

Genauso ist es beim Ausleihen von Büchern in einer Bibliothek. Die Teilschritte "Buch ausleihen" und "Buch zurückgeben" gehören in eine Kette von Verarbeitungsschritten des Anwendungsfalls "Buch ausleihen und zurückgeben". Wenn Sie ein Buch ausleihen, geben Sie genau dieses Buch, das sie mitgenommen haben, zurück und nicht ein anderes! Oder geben Sie ein Buch zurück, ohne dass Sie vorher genau

dieses Buch ausgeliehen haben? Das wäre möglich, wenn "Buch ausleihen" und "Buch zurückgeben" zwei unabhängige Anwendungsfälle wären, die vollkommen unabhängig voneinander aufgerufen werden könnten. Dann könnten Sie auch 5x denselben Titel zurückgeben, ohne ihn je ausgeliehen zu haben.

#### 2.6.1.4 Anwendungsfälle in Analyse und Entwurf

Ganz zu Beginn der Analyse eines Systems betrachtet man zunächst die Geschäftsprozesse und fällt dann die Entscheidung, welche Geschäftsprozesse bzw. welche Anteile davon durch das System unterstützt werden sollen und welche nicht. Die Geschäftsprozesse oder Teile von Geschäftsprozessen, die auf dem Rechner laufen, beinhalten die realisierten **Anwendungsfälle**.

Geforderte Anwendungsfälle kann man in der **Analyse** studieren, wobei im Rahmen der Objektorientierung in der Analyse in der Regel nur die Funktion der **Verarbeitung** betrachtet wird. Beim **Entwurf** kommen dann **technische Funktionen** wie die Ein-/Ausgabe oder die persistente Datenspeicherung hinzu, die einen Anwendungsfall letztendlich auf einem Rechner zum Laufen bringen.

#### 2.6.1.5 Anwendungsfälle und Alternativen

Durch einen Anwendungsfall wird nicht nur ein bestimmter Ablauf beschrieben, sondern eine ganze Menge von möglichen Abläufen.

Der **Basisablauf**<sup>40</sup> eines Anwendungsfalls beschreibt den "Gut-Fall", bei dem das gewünschte Ergebnis, also das eigentliche Ziel des Anwendungsfalls, erzielt wird. Hierbei kann es innerhalb des Basisablaufs durchaus Alternativen geben.



Beispielsweise kann im Basisablauf eine Ausgabe wahlweise auf den Bildschirm oder auf den Drucker erfolgen. Formal gesehen ist das Ergebnis eines Anwendungsfalls ein Wert, der an einen Akteur gegeben wird wie z. B. die Ausgabe eines Dokumentes auf einem Drucker.

Außer dem Basisablauf gibt es ferner noch sogenannte **Alternativabläufe**. Diese **Alternativabläufe** stellen sozusagen **Fehlerfälle** dar, da bei Alternativabläufen das eigentliche Ziel eines Anwendungsfalls nicht erreicht wird.



Ein Alternativablauf kann kein Ergebnis haben, aber auch ein Ergebnis, welches allerdings nicht dem eigentlich gewünschten Ergebnis des Basisablaufs entspricht.

Im Rahmen der **Analyse** bewegt man sich im Problembereich. Alternativabläufe sind daher stets **Fehler in der Anwendung**.



<sup>40</sup> Der Basisablauf ist der "normale Ablauf" eines Anwendungsfalls, wie er im positiven Fall erwartet wird.



In der Analyse gibt es noch kein technisches System. So hat der Anwendungsfall "Buch ausleihen und zurückgeben" beispielsweise den Alternativablauf (Fehler in der Anwendung), dass das gesuchte Buch in der Bibliothek gar nicht geführt wird, oder den Alternativablauf, dass das gesuchte Buch gerade ausgeliehen ist.

Ein **technischer Fehler** wird erst beim Studium des **Lösungsbeereichs** gefunden, bei dem es ein lauffähiges System und technische Funktionen gibt.



Es kommen deshalb beim Entwurf weitere Fehlerfälle als Alternativen hinzu, nämlich dass das Rechnersystem oder technische Funktionen des Systems gerade ausgefallen sind oder nicht wunschgemäß funktionieren. Ein Beispiel für einen technischen Fehler ist beispielsweise, dass eine Kommunikationsverbindung unterbrochen ist.

### 2.6.1.6 Szenarien spezifikationsorientiert und agil

Anwendungsfälle der spezifikationsorientierten Welt stellen nichts anderes dar als eine **Verallgemeinerung von Szenarien** der spezifikationsorientierten Welt.

Ein **Szenario** der spezifikationsorientierten Welt repräsentiert einen speziellen Ablauf mit definierten Parametern. Ein Anwendungsfall kann als ein Typ mit Variablen angesehen werden und ein Szenario als eine Instanz eines Anwendungsfalls.



Im Folgenden ein Beispiel:



Bild 2-2 Beispiel für Anwendungsfall versus Szenario (spezifikationsorientiert)

Ein Anwendungsfall wird nicht – wie bei Szenarien üblich – mit ganz konkreten Datenwerten durchgespielt, sondern mit "Variablen". Alle Informationen, die fließen, tragen einen Namen. Während bei einem Anwendungsfall nach **Basisablauf** und **alternativer Ablauf** je nach dem Bereich, in dem sich der für eine Fallunterscheidung relevante Parameter befindet, unterschieden wird, läuft ein Szenario in der spezifikationsorientierten Welt einfach komplett entsprechend seinen Parametern ab.

Ein Szenario hat keine Alternativen. Ein Anwendungsfall enthält viele Szenarien.

Auch wenn in der **agilen Welt** ein **Szenario** oft eine Instanz eines Anwendungsfalls ist, so gibt es dennoch in der agilen Welt keine klare Spezifikation von "Szenario". Man orientiert sich vielfach einfach nur an der sprachlichen Bedeutung von "Szenario". Ein Szenario ist in diesem Falle einfach ein unscharfes Hilfsmittel, die Interaktion eines Systems mit seiner Umwelt in einem gewissen Anwendungsfall zu beschreiben.



In der agilen Entwicklung hat sich eine Praktik etabliert, die es erlaubt, Akzeptanztests in Form von Szenarien zu beschreiben, die das angestrebte Verhalten des zu erstellenden Systems definieren. Sie nennt sich **Behaviour Driven Development (BDD)** und wird in der Regel dazu verwendet, bereits vor der Implementierung von Funktionalität besagte Akzeptanztests direkt aus Kundenforderungen abzuleiten, insbesondere den Akzeptanzkriterien eines Anwendungsfalls. Die dazu verwendete **Domain Specific Language (DSL)** nennt sich Gherkin und wird mittlerweile von einer Vielzahl an Werkzeugen unterstützt. Gherkin<sup>41</sup> besteht vor allem aus Schlüsselworten wie Funktionalität, Szenario, Angenommen, Wenn, Dann und Und. Betrachtet man den im Folgenden abgedruckten Gherkin-Testfall, dann fällt sofort die Ähnlichkeit zu einer Beschreibung eines Anwendungsfalls oder einer größeren User Story auf. Hier der bereits erwähnte Testfall [cucuhp]:

Funktionalität: Division

Um dumme Fehler zu vermeiden, müssen die Kassierer in der Lage sein, Divisionen auszurechnen

Szenario: Reguläre Zahlen

Angenommen ich habe die Zahl 3 im Taschenrechner eingegeben

Und ich habe die Zahl 2 im Taschenrechner eingegeben

Wenn ich auf "Division" drücke

Dann sollte als Resultat 1,5 am Bildschirm ausgegeben werden

Um den Testfall ausführbar zu machen, muss in der Entwicklungsphase natürlich noch ein wenig Code geschrieben werden. Es wird aber tatsächlich der in Gherkin geschriebene Testfall geparkt und mittels regulärer Ausdrücke auf Testfunktionen gemapped. Durch die regulären Ausdrücke lässt sich eine Testfallsprache realisieren, bei der die beispielsweise in einem Satz eingebetteten Parameter als Variablen gelesen werden und die Testfunktion im Code somit wiederverwendet wird, sobald man in einem anderen Szenario oder Testfall denselben Satzaufbau mit beliebigen anderen Parametern verwendet.

BDD unterstützt somit tatsächlich die Entwicklung des vom Kunden angestrebten Systems, da dabei die Testfälle, die zur fachlichen Verifikation verwendet werden, vom Kunden bzw. Product Owner gelesen oder sogar geschrieben werden. Erwähnenswert ist auch, dass BDD somit ähnlich, wie es bei den User Stories ist, die Kollaboration<sup>42</sup> zwischen allen Parteien in den Vordergrund stellt.

<sup>41</sup> Kommt ursprünglich vom Werkzeug Cucumber [cucuhp], das ein BDD-Tool für Ruby darstellt.

<sup>42</sup> Die Kollaboration wird deshalb gefördert, weil für die Testfälle eine Prosa-artige Sprache verwendet wird und der Product Owner oder der Endkunde auch ohne Programmierkenntnisse diese lesen und sogar daran mitarbeiten kann.

### 2.6.1.7 Basisablauf ohne Alternativen als Sequenz von Einzelschritten

Das folgende Bild zeigt den Basisablauf eines Anwendungsfalls ohne Alternativen:



*Bild 2-3 Basisablauf eines Anwendungsfalls als eine Sequenz von Einzelschritten*

Ein Basisablauf kann eine synchrone oder asynchrone Folge von Einzelschritten sein. Erfahrungsgemäß stellt sich das Finden von Anwendungsfällen als eine sehr anspruchsvolle Aufgabe heraus. Ein Anfänger findet beispielsweise anstelle des Anwendungsfalls "Buch ausleihen und zurückgeben" die beiden Anwendungsfälle "Buch ausleihen" und "Buch zurückgeben".

### 2.6.1.8 Verfeinerung von Anwendungsfällen in der Systemanalyse

Um die Anwendungsfälle eines Systems zu strukturieren, kann ein Anwendungsfall modular aus Teilen aufgebaut werden. Zwischen diesen Teilen bestehen die folgenden Beziehungsarten:

- **Erweiterungsbeziehung**

Eine Erweiterungsbeziehung drückt die potenzielle **Erweiterung eines lauffähigen Anwendungsfalls** durch einen Sub-Anwendungsfall aus. Das Verhalten eines lauffähigen Anwendungsfalls kann, muss aber nicht erweitert werden.



Bei Erfüllen von Bedingungen an sogenannten Erweiterungspunkten wird der erweiternde Sub-Anwendungsfall ausgeführt. Fehlt die Bedingung, wird der Anwendungsfall am Erweiterungspunkt erweitert. Der erweiternde Sub-Anwendungsfall ist ein Inkrement, das in den zu erweiternden Anwendungsfall eingeschoben wird. Der zu erweiternde Anwendungsfall ist auch ohne die Erweiterung lauffähig.

Das Schlüsselwort «**extend**» dient in UML zur Trennung des optionalen erweiternden Verhaltens von dem erweiterten Anwendungsfall. Erweiterungsbeziehungen werden in den folgenden Fällen verwendet:

- Modellierung alternativer Abläufe, die selten ausgeführt werden.
- Modellierung optionaler Teile des Anwendungsfalls.
- Modellierung getrennter Unterabläufe, die nur in speziellen Fällen auftreten.
- Um mehrere verschiedene Sub-Anwendungsfälle in einen bestimmten Anwendungsfall einfügen zu können.

### • Inklusionsbeziehung

Eine Inklusionsbeziehung drückt aus, dass ein Anwendungsfall einen Sub-Anwendungsfall braucht und benutzt, **um überhaupt lauffähig zu werden**.



Dabei hat der Sub-Anwendungsfall, der benutzt wird, oftmals die Eigenschaft eines **Bibliothekbausteins**, das heißt, dass er mehrfach verwendet wird. Aber auch ein "freies" Strukturieren ist möglich.

Eine Inklusionsbeziehung wird nach UML durch eine Abhängigkeitsbeziehung mit dem Schlüsselwort **«include»** modelliert.

### • Vererbungsbeziehung

Eine Vererbungsbeziehung drückt aus, dass ein Anwendungsfall die Spezialisierung eines anderen Anwendungsfalls ist.



Diese drei Beziehungen sind im folgenden Bild zu sehen:

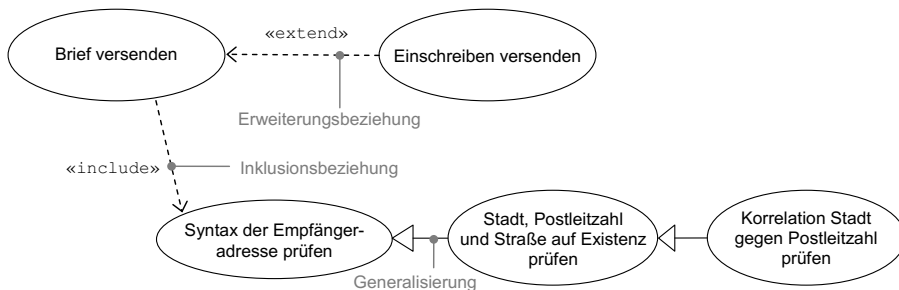


Bild 2-4 Beispiel für Beziehungen

#### 2.6.1.9 Erweiterungsbeziehung

Ein Standardbeispiel für Erweiterungen sind Fehlerfälle in der Anwendung. Diese werden zunächst nicht betrachtet, da zuallererst der Normalfall (Basisablauf) modelliert wird. Fehlerfälle kommen als Erweiterungen zum Basisablauf hinzu.

Die Notation einer Erweiterungsbeziehung ist – wie im folgenden Bild gezeigt – nach UML ein Abhängigkeitspfeil mit dem Schlüsselwort **«extend»**:

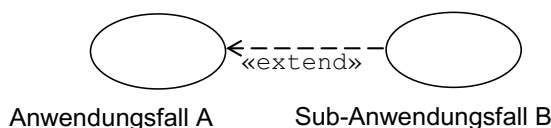


Bild 2-5 Form der extend-Beziehung

Der `extend`-Pfeil geht vom **erweiternden Sub-Anwendungsfall B** zum **erweiterten Anwendungsfall A**. Eine durch das Schlüsselwort `«extend»` gekennzeichnete Abhängigkeitsbeziehung stellt die Erweiterungsbeziehung dar.

Die obige Grafik bedeutet:

- Der Anwendungsfall A kann durch den Sub-Anwendungsfall B erweitert werden.
- Der Anwendungsfall A wird völlig unabhängig vom Sub-Anwendungsfall B beschrieben und ist ohne die Erweiterung lauffähig.
- In der Anwendungsfallbeschreibung von A ist die Einfügestelle anzugeben.

### 2.6.1.10 Inklusionsbeziehung

Eine Inklusionsbeziehung kann eine mehrfache Wiederverwendbarkeit (Bibliotheksbaustein) unterstützen, aber auch nur eine frei gewählte Verfeinerung. Eine mehrfache Wiederverwendung ist jedoch das eigentliche Ziel, da sie willkürliche Zerlegungen vermeidet. Auf jeden Fall muss der inkludierte Sub-Anwendungsfall aufgerufen werden, damit der aufrufende Anwendungsfall überhaupt erst lauffähig wird.

Die Notation einer `include`-Beziehung ist im folgenden Bild zu sehen:

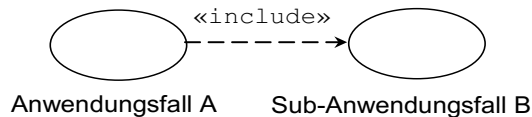


Bild 2-6 Form der `include`-Beziehung

Der `include`-Pfeil geht vom benutzenden Anwendungsfall A zum benutzten Sub-Anwendungsfall B. Die obige Grafik bedeutet:

Der Anwendungsfall A benutzt den Sub-Anwendungsfall B, d. h., er führt den Anwendungsfall B während seiner Abarbeitung aus. Der **Sub-Anwendungsfall B** ist für die Bereitstellung der erforderlichen Funktionalität des Anwendungsfalls A **absolut notwendig**.

### 2.6.1.11 Vererbungsbeziehung

Eine Generalisierung zwischen Anwendungsfällen funktioniert genauso wie eine Generalisierung zwischen Klassen. Der untergeordnete Anwendungsfall übernimmt die Eigenschaften des übergeordneten Anwendungsfalls und kann diesen spezialisieren. Ein Anwendungsfall kann also vom Typ eines anderen Anwendungsfalls sein. Es wird die bei Klassen übliche Notation eines Vererbungspfeils angewandt.

## 2.6.2 User Stories

Kent Beck [Bec99] führte den Begriff einer User Story ein, um von schwergewichtigen Spezifikationen wegzukommen und um die gegenseitige Kommunikation von Entwicklern und Kunden in den Mittelpunkt zu stellen.

Eine **User Story** ist gut geeignet, um **Funktionalitäten** für ein zu realisierendes System aufzunehmen und zwar in derjenigen Weise, wie ein Nutzer das zukünftige System benutzen will.



Daher sollen **User Stories** aus **Nutzersicht** erfasst werden. User Stories werden am besten durch Gespräche mit Kunden und Nutzern gewonnen. Use Cases hingegen werden in der Regel durch Berater oder System- bzw. Softwareingenieure entwickelt. Statt des Begriffs User Story hört man auch den Begriff **Feature**. In agilen Kreisen wird der Begriff "story" oder "user story" bevorzugt.

Eine Story soll auf einer einzigen Notizkarte notiert werden. User Stories sind vermutlich die populärste agile Technik, um die Funktionalität eines Produkts zu erfassen.

Eine **User Story** beschreibt eine kurzgefasste Darstellung einer Anforderung des Kunden, ohne technisch zu tief ins Detail zu gehen.



Eine **User Story** dient

- als Kurzfassung und
- als Auslöser für Gespräche.



Dadurch sollen Stakeholder des Kunden auch ohne weitreichende Fachkenntnisse die Grundgedanken ihrer Anforderungen an ein System für ihre Auftragnehmer, die Entwickler, formulieren können. User-Stories sollen für alle Beteiligten des Projekts (Kunden und Entwickler) verständlich in Alltagssprache geschrieben werden, damit es bei der Kommunikation keine Verständnisschwierigkeiten gibt. Auf technische Details wird größtenteils oder gänzlich verzichtet. Eine User Story sollte im optimalen Fall mit wenigen Worten und Sätzen eine Anforderung eines Kunden auf den Punkt bringen. Sie sollte idealerweise innerhalb von wenigen Wochen<sup>43</sup> realisiert werden können.

### 2.6.2.1 Story Cards

User Storys werden üblicherweise auf Karteikarten, sogenannte **Story Cards**, geschrieben. Diese Karteikarten können neben den User Stories auch weitere Informationen wie z. B. das Risiko der User Story beinhalten. Für Story Cards gibt es keine vorgeschriebene Norm. Sie können im Vorfeld frei gestaltet werden.

Eine **Story Card** ist eine Karteikarte, die die Beschreibung genau einer User Story, sowie eventuell weitere Informationen enthält.



<sup>43</sup> Eine User Story kann anfänglich sehr groß sein. Wenn sie "näher" kommt, sollte sie so zerteilt werden, dass sie in wenigen Tagen umsetzbar ist. Schließlich sollen beispielsweise in einem Sprint von Scrum zur Produktion eines Systemfragments einige User Stories realisiert werden.

Eine Story Card repräsentiert somit eine User Story entweder auf

- **physische Weise** in Form von handgeschriebenen Karteikarten an einer Wand (Story-Wall) oder auf
- **virtuelle Weise** in Form von Requirement Management Tools oder Project Management Tools wie z. B. JetBrains YouTrack.

Klassischerweise werden Story Cards physisch auf dem Papier verfasst, aber auch Tools für virtuelle Story Cards können eine Alternative sein.

### Physische Story Cards

Physische Story Cards sind handgeschriebene Karteikarten, auf denen User Stories und eventuell zusätzliche Informationen stehen. Initiator dieser Story Cards war Kent Beck im Jahr 1999 beim Projekt "C3 Payroll", das als Geburtsstunde von Extreme Programming gilt.

Eine mögliche Story Card konnte in Anlehnung an Kent Beck ursprünglich folgendermaßen aufgebaut sein:

Storynummer	Datum	Name der Story-Card	Priorität	
<Beschreibung der User Story>				
				Abhängigkeit zu
				Aktivitätstyp
				<input type="radio"/> Neu <input type="radio"/> Behebung <input type="radio"/> Erweiterung
				Risiko

*Bild 2-7 Aufbau einer anfänglichen Story Card in Anlehnung an Kent Beck*

Beck empfahl damals, neben der User Story auf einer Story Card einige optionale Parameter hinzuzufügen.

Folgende Parameter waren laut **Kent Beck** empfehlenswert [Bec99]:

- **Aufgabenbeschreibung** – Möglichst präzise, leicht verständliche und kurze Beschreibung der User Story
- **Datum** – Erstellungsdatum der Story Card

- **Storynummer (Story Card ID)** – Fortlaufende Nummer zur Identifikation der Story Card
- **Aktivitätstyp** – Beschreibt die Art der Aufgabe, ob sie komplett neu ist, ein Fehler zu beheben ist oder eine Funktion zu erweitern ist.
- **Priorität** – Die Wichtigkeit, die der Stakeholder der Aufgabe zuordnet. Beispiel: Intervall von 1 – 10. 1: weniger wichtig, 10: sehr wichtig. Auch die Prioritätsvergabe der Entwicklung ist von Interesse.
- **Abhängigkeit** – Zeigt die Abhängigkeit zu einer anderen Story Card (über die Storynummer).
- **Funktionstest** – Funktioniert die Funktion?
- **Risiko** – Wie hoch sieht die Entwicklung das Risiko bei der Implementierung?
- **Aufwandschätzung** (der Entwicklung) – Die Entwickler schätzen ab, wie viel Zeit für die Aufgabe in Anspruch genommen wird.
- **Notizen** – Weitere Notizen der Stakeholder und/oder der Entwickler
- **Aufgaben-Nachverfolgung** – Die Entwickler können in einer Tabelle den Fortschritt der Aufgabe eintragen, damit jeder Projektbeteiligte den aktuellen Stand verfolgen kann.

Mittlerweile wird jedoch von so sehr aufwendigen Story Cards abgeraten. Ron Jeffries, einer der Mitbegründer von Extreme Programming, analysierte, dass derart komplexe Story Cards nicht den erhofften Nutzen haben und dass eine Story Card so einfach wie möglich gehalten werden sollte.

Es kann aber trotzdem projektspezifisch sinnvoll sein, bestimmte Informationen zusätzlich zur User Story auf einer Karte festzuhalten. Welche Parameter letztendlich auf eine Karte genommen werden, hängt von den Organisatoren des Planungsprozesses ab. Diese entscheiden, welche Parameter für das Projekt relevant sind.

Das folgende Bild gibt ein Beispiel für eine physische Story Card, wie sie idealerweise aufgebaut sein sollte:

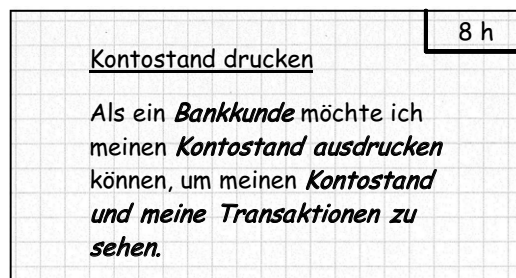


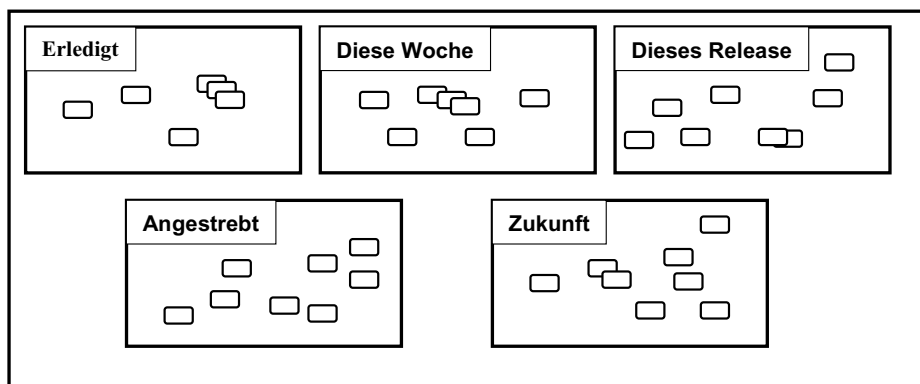
Bild 2-8 Beispiel für eine Story Card



Das ist eine minimale Form einer Story Card. Hier ist nur der Titel, die User Story und die geschätzte Arbeitszeit angegeben. Der Aufbau dieser minimalen Form wird in Kapitel 2.6.2.6 beschrieben.

Nachdem die Story Cards erstellt wurden, werden sie an einer Wand (sog. Story-Wall oder Story-Wand) bzw. auf Plakaten an einer Wand befestigt, um kategorisiert zu werden und um für alle Beteiligten sichtbar zu sein.

Das folgende Bild zeigt beispielhaft eine Story-Wall:



*Bild 2-9 Beispiel für eine Story-Wall*

### **Virtuelle Story Cards**

Eine andere Möglichkeit, Story Cards zu verfassen, ist der digitale Weg über ein Software-Tool. Seit dem Projekt C3 Payroll im Jahre 1999 wurde eine Menge Projekt-Management-Tools (PM-Tools) entwickelt. Ziel solcher Tools ist es, die Arbeit der Softwareentwickler zu erleichtern und eine bessere Übersicht über ein Projekt zu verschaffen.

Mit Hilfe vieler solcher PM-Tools lassen sich Story Cards erstellen. Story Cards werden oftmals auch als Tickets bezeichnet. Jedes Ticket hat eine ID und kann auf ein anderes Ticket verweisen. Ein Ticket kann in verschiedene Kategorien verschoben werden (z. B. von dem Zustand "in Bearbeitung" in den Zustand "Behoben"). Da die Story Cards digital sind, werden die User Stories oftmals von allen Beteiligten diktiert und von einer Person in das PM-Tool übertragen. Im Idealfall können alle Beteiligten an einem großen Bildschirm/Projektor sehen, was bisher verfasst wurde. Ein großer Vorteil von digitalen Story-Walls ist, dass von jedem Beteiligten mit den nötigen Zugangsrechten die Stories (Tickets) am Bildschirm eingesehen und bearbeitet werden können.

Ein Beispiel-PM-Tool ist die browser-basierte Software YouTrack von JetBrains. Mit dem nötigen Zugriff kann jeder Projektbeteiligte Story Cards erstellen, bearbeiten, verschieben oder löschen. Das folgende Bild zeigt einen Ausschnitt einer Story-Wall des Tools YouTrack:

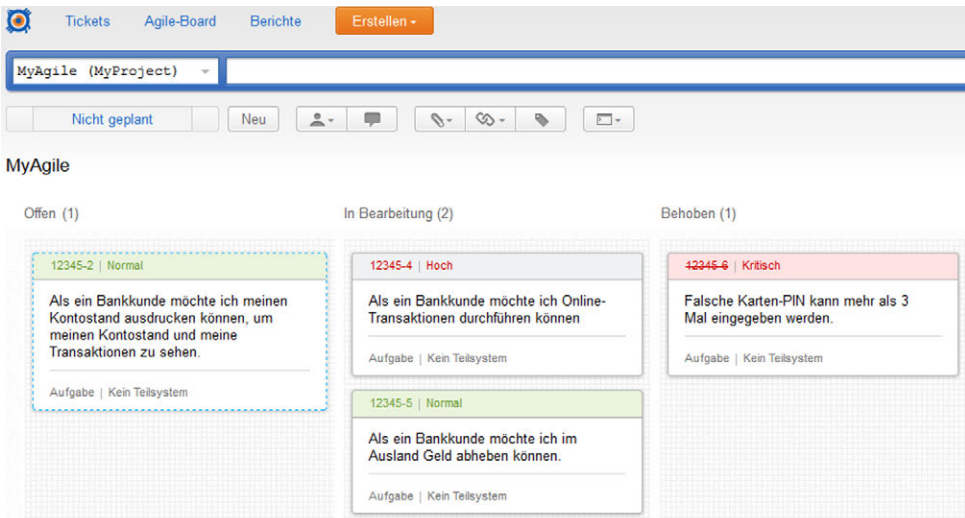


Bild 2-10 Ausschnitt einer Story-Wall im Tool YouTrack von JetBrains

### 2.6.2.2 Ermittlung von User Stories

Vor dem Aufstellen von User Stories muss als Voraussetzung der Kontext des zu betrachtenden Systems analysiert werden.

User Stories werden ermittelt:

- durch Gespräche mit den Nutzern,
- durch die Aufnahme der Tätigkeiten von Nutzern in der Praxis,
- mit Hilfe von Fragebögen,
- durch die Aufnahme von Altsystemen und
- in Story-Workshops.

### 2.6.2.3 Bestandteile einer User Story

Eine User Story hat drei Teile:

- eine knappe schriftliche **Beschreibung als Erinnerung**, Gespräche über die User Stories zu führen sowie diese zur Planung zu verwenden,
- **Gespräche** über die Stories zur Ermittlung von Details,
- Festlegung von **Akzeptanzkriterien**, die durch Akzeptanztests verifiziert werden, um die vollständige Umsetzung einer Story zu verifizieren.



Es gibt eine Merkregel von Ron Jeffries zu User Stories [ronref], und zwar die 3 C's:

- Card,
- Conversation und
- Confirmation.

Das bedeutet:

- Die Kurzbeschreibung einer User Story sollte auf eine **einzige Karte** passen. Sie sollte also kurz sein. Eine Karte kann oftmals nicht alle Informationen, die zu der entsprechenden Forderung gehören, enthalten. Der Text muss aber ausreichen, um die Forderung zu identifizieren.
- Der Fokus wird auf die Konversation gelegt – eine Story dient im Prinzip nur als Erinnerung zu kommunizieren. Eine Forderung wird durch Kommunikation vom Kunden zum Programmierer transportiert. Diese Kommunikation darf nicht abbrechen, insbesondere nicht bei der Aufwandsabschätzung. Die verbale Dokumentation **kann durch Dokumente ergänzt werden**.
- Eine Story sollte genau formulierte Bedingungen enthalten, wann sie als umgesetzt gilt (Akzeptanzkriterien). Dieses dritte C betrifft also den **acceptance test**, der im erfolgreichen Fall besagt, dass die Story korrekt implementiert und das Ergebnis akzeptiert wurde. **Akzeptanzkriterien** beantworten die Frage, was getestet werden soll, damit die Realisierung einer Story vom Kunden abgenommen wird.

#### 2.6.2.4 Eigenschaften von User Stories

Positive Eigenschaften einer User Story sind nach Bill Wake [billwa] die sogenannten INVEST-Eigenschaften:

- Independent (unabhängig, die User Storys können in beliebiger Reihenfolge geliefert werden),
- Negotiable (verhandelbar, die Details einer Story werden durch Programmierer und Kunden während der Entwicklung verhandelt),
- Valuable (werthaltig für User oder Kunden),
- Estimatable (abschätzbar durch die Programmierer),
- Small (klein, so dass eine Story innerhalb weniger Tage realisiert werden kann) und
- Testable (testbar).

User Stories sollen unabhängig in demjenigen Sinne sein, dass sie **unabhängig entwickelt** werden können.

So wie es bei Anwendungsfällen gemeinsame Sub-Anwendungsfälle gibt, kann es auch bei User Stories gemeinsame Sub-User Stories als Teile von User Stories geben. Werden solche Sub-User Stories inkludiert, so können die **inkludierten Sub-User Stories nicht getrennt entwickelt** werden, da sie für die **Lauffähigkeit** der entsprechenden User Story essentiell sind.

Das folgende Bild zeigt am Beispiel einer Bankanwendung, dass Anwendungsfälle voneinander abhängig sein können. In diesem **Beispiel** sind 4 verschiedene **Anwendungsfälle gekoppelt**, da sie alle den gemeinsamen Sub-Anwendungsfall "Kunden authentifizieren" verwenden:

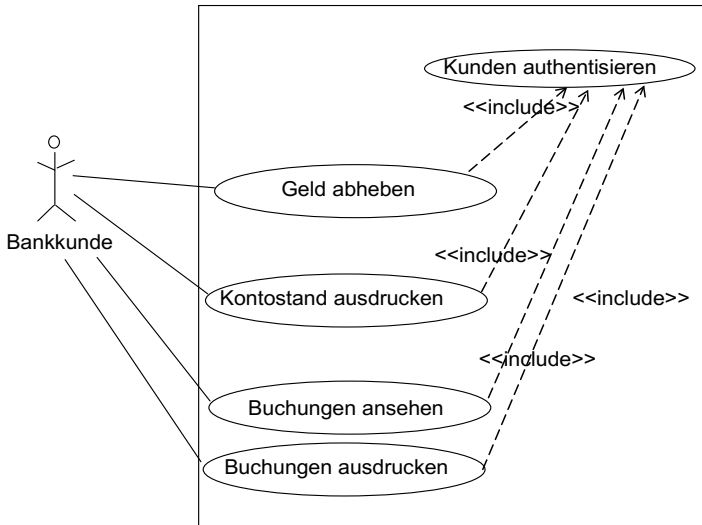


Bild 2-11 Gemeinsamer Sub-Anwendungsfall

Wird der Sub-Anwendungsfall "Kunden authentifizieren" verändert, so sind 4 Anwendungsfälle betroffen.

### 2.6.2.5 User Stories als Planungsinstrument

Eine User Story soll im festgelegten Zeitraum einer Iteration umgesetzt werden. Damit kann eine **User Story** als **Planungsinstrument** verwendet werden.



User Stories spielen eine große Rolle bei der **Planung (Aufwandsschätzung, Arbeitsgeschwindigkeit)**. Anwendungsfälle werden nicht zu Planungszwecken verwendet. User Stories werden gerne in Story Points abgeschätzt. Welche Arbeitszeit ein Story Point tatsächlich bedeutet, kann man an Fallbeispielen durch das Studium der Arbeitsgeschwindigkeit (engl. velocity) des Entwicklungsteams messen.

User Stories werden bewusst "klein geschnitten", damit sie in das Zeitraster einer Iteration passen.



Gegenüber einem Anwendungsfall mangelt es einer User Story oft an Kontextinformationen. Große Stories – beispielsweise einen kompletten Anwendungsfall mit allen Alternativen – nennt man auch **Epos**<sup>44</sup> (englisch **Epic**). Die Idee ist hier tatsächlich, dass ein Epos eben aus mehreren kleineren Geschichten besteht.

<sup>44</sup> Die Verwendung eines Epos erlaubt es, die Funktionalität auf hoher Ebene zu betrachten.

### 2.6.2.6 Formulierung von User Stories

Der Hauptteil einer Story folgt oft der folgenden Schablone<sup>45,46</sup>:

Als <Rolle/Persona>  
möchte ich <die folgende Funktion/das folgende Feature> haben,  
um <den Mehrwert/das Ziel> XYZ zu erreichen.

Als Beispiel hierzu siehe Bild 2-8.

Personas können – wie auch Rollen – dazu dienen, die Zielgruppe eines Systems zu modellieren. **Personas** repräsentieren konkrete Nutzer einer Rolle. Dabei sollten die verschiedenen Personas durch ihre verschiedenartigen Ziele voneinander abgegrenzt werden.



Es werden einige fiktive Personen geschaffen, die stellvertretend für die tatsächlichen Anwender stehen sollen. Hierbei wird versucht, einem "Repräsentanten" eines Teils der Zielgruppe einen Namen, ein Gesicht, ein Alter, einen persönlichen und evtl. sogar kulturellen Hintergrund zu geben. Schreibt man User Stories aus Sicht einer Persona, dann gibt man dem Entwickler die Chance, sich besser in die Anwenderseite hineinzuversetzen, als wenn er nur abstrakte Rollen verwendet. Zusammen mit der stattfindenden Konversation kann das oft zu besseren Lösungen führen, als man vorher hätte spezifizieren können.

Sogenannte **Proto-personas** sind eine Variante der typischen Personas mit dem wichtigen Unterschied, dass sie zunächst nicht das Ergebnis der Suche nach Usern sind. Stattdessen resultieren sie aus Brainstorming Workshops, bei denen die Firmenteilnehmer die Überzeugungen der Organisation aufgrund ihrer Geschäftskennntnisse und ihrem Verständnis formulieren im Hinblick darauf, wer das Produkt benutzen oder bedienen wird und aus welchem Grund. Proto-personas sind für eine Organisation der Startpunkt, um zu beginnen, ihre Produkte zu bewerten und einige frühe Design-Hypothesen zu erzeugen. Sie helfen, dass die Sicht des Kunden ins Bewusstsein der Firma rückt und in die strategische Planung eingeht. Dies gilt besonders dann, wenn die Schöpfer der Proto-personas in der Position sind, die strategische Ausrichtung der Firma zu beeinflussen. [umupea]

Insgesamt geht es also um die Frage, welche Funktionalität das Produkt zur Verfügung stellt, um die Ziele bestimmter Personas, Proto-personas oder Rollen zu erfüllen.

**User Stories sind keine niedergeschriebene Form von Anforderungen.**



User Stories sind Diskussionen zur Lösung von Problemen des zukünftigen Systems, die zu einer Übereinstimmung führen, was man bauen soll [Pat14].

<sup>45</sup> Diesem Muster darf nicht sklavisch gefolgt werden. Es gibt es auch nicht-funktionale Forderungen.

<sup>46</sup> Das Pattern "As a ... I want ... to ..." wurde bei der Firma Connextra entwickelt und wird vor allem Rachel Davies zugeschrieben

Bei einer User Story kommen zum identifizierenden Kurztext ferner noch

- Hinweise,
- Randbedingungen (Einschränkungen) sowie
- Akzeptanzkriterien

hinzu.

Akzeptanztests sollen die Akzeptanzkriterien erfüllen. Diese Kriterien müssen vom Kunden getragen werden, da schließlich der Kunde das entstandene Produkt abnehmen soll. Deshalb sollten die Akzeptanzkriterien in Gesprächen mit dem Kunden ermittelt werden.

### 2.6.3 User Stories im Vergleich zu Anwendungsfällen

Im Folgenden wird der Zusammenhang zwischen User Stories und Anwendungsfällen betrachtet.

Die minimale Schablone für eine User Story ist nur ein erster Einstieg in eine User Story oder in einen Anwendungsfall, letztlich in die Spezifikation einer Leistungserbringung. Es ist ein einfacher Satz, dass die entsprechende Funktion benötigt wird. Das ist aber natürlich nur ein Anfang. Hinter jeder Funktion steckt ein Ablauf mit einem bestimmten Ziel. Das Ziel ist bei einer User Story vom Prinzip her dasselbe wie bei einem Anwendungsfall, nämlich das erwartete Ergebnis.

Eine vollständige Spezifikation kostet Zeit und veraltet, wenn der Kunde im Laufe der Zeit andere Ziele setzt. Daher ist es grundsätzlich gut, wenn Kunde und Entwickler die User Stories gemeinsam entwickeln und ggf. gemeinsam auf das Wesentliche reduzieren.

Eine User Story wird so zurechtgeschnitten, dass sie aus planerischen Gründen in die Time-Box einer Iteration passt. Man kann die erste User Story als einen Schnellschuss betrachten, der aber iterativ in Zusammenarbeit der Entwickler mit den Kunden auf Basis der demonstrierten Ergebnisse ergänzt und verbessert wird, bis die User Story den Anforderungen genügt.



Man muss damit rechnen, dass bei einer agilen Vorgehensweise zunächst nur Teilstücke der gewünschten Abläufe gefunden werden.

Bei übereinstimmendem Ziel für einen Anwendungsfall und einer User Story bzw. einer Folge von User Stories können bei der Verwendung von User Stories nur alternative Teile entfallen, nicht aber sequentielle Teile einer Verarbeitungskette, die für die Erbringung des Ergebnisses des Basisablaufs eines Anwendungsfalls erforderlich sind. Eine User Story ist damit ein Subset eines Anwendungsfalls **ohne nicht erforderliche Alternativen**.

### 2.6.3.1 Leichtgewichtigkeit versus Schwergewichtigkeit

Eine User Story ist ein **leichtgewichtiges Dokument**. Es sollte auf eine einzige Karte geschrieben werden können. Die Kurzbeschreibung einer User Story umfasst nicht alle Details. Ein Anwendungsfall ist ein **schwergewichtiges Dokument**. Er umfasst den Basisablauf und alternative Abläufe. Ein Anwendungsfall stellt eine formale Spezifikation dar.

### 2.6.3.2 Archivierung

Als Spezifikation lebt ein **Anwendungsfall permanent**, solange das Produkt entwickelt und gewartet wird. **Stories** leben **transient** während der Dauer des Iterations-schritts, in welchem sie zur Software hinzugefügt werden. User Stories werden nicht archiviert.

Anwendungsfälle stellen Anforderungsspezifikationen dar. Sie werden daher möglichst vollständig beschrieben. Sie werden archiviert. Eine User Story ist keine Spezifikation, sondern dient nur als Aufhänger, um **Gespräche** über Anforderungen zu führen, die **umgesetzten Systemfragmente abzunehmen** und um die Realisierung von **neuen Systemfragmenten in Zeitscheiben einzuplanen**.

Die Stories dienen nicht als Dokumentation oder gar als Vertrag, sondern vielmehr als Erinnerung an einen Gesprächsbedarf. Eine User Story ist nur ein Mittel zum Zweck.

### 2.6.3.3 Konvergenz einer User Story zum Kern eines Anwendungsfalls

Da mit den Kunden diskutiert wird, welche Aufgaben die Nutzer des Kunden durchführen wollen, ist die Annahme vernünftig, dass zumindest die richtigen Arbeitsschritte gefunden werden. Die richtigen Sequenzen von Teilschritten müssen formuliert werden, sonst würde das Ziel, das für einen Anwendungsfall und für die entsprechenden User Stories identisch sein sollte, nicht erreicht.

Gegenüber einem Anwendungsfall können bei einer User Story gewisse Alternativen im Basisablauf und gewisse Alternativabläufe als Fehlerfälle der Anwendung oder verursacht durch technische Probleme fehlen. Wenn der Kunde solche Alternativen nicht anfordert, scheinen diese Fälle für ihn nicht relevant zu sein.

Fehlerfälle aufgrund technischer Probleme

- können ärgerlich werden,
- können einen großen Sachschaden verursachen oder
- sogar eine Gefahr für Leib und Leben bedeuten.

Das technisch geschulte Entwicklungsteam muss den Kunden unbedingt auf technische Fehlerfälle hinweisen. Kunde und Entwickler sollten sich in einem fortlaufenden Dialog befinden. Die Kompetenz des Kunden im Problembereich und die Kompetenz der Entwickler im technischen Bereich sind zu verschmelzen, um ein gutes System zu bauen.

Letztendlich kann man mit einer (ggf. wegen ihrer Größe zerlegten) User Story als einem Mittel der Planung für eine einzige Iteration beginnen. Im Projektverlauf muss man durch weitere Iterationen die Vollständigkeit der tatsächlich benötigten Ablaufsequenz erreichen. Da man nicht spezifikationsorientiert arbeitet, entspricht der Umfang der entsprechenden Dokumentation<sup>47</sup> aber nur dem wirklich Erforderlichen.

Die Zahl der betrachteten Alternativen innerhalb eines Basisablaufs und der zu berücksichtigenden Fehler in der Anwendung selbst muss der Kunde entscheiden, da er seine Domäne am besten kennt. Die durch technische Fehler verursachten Alternativen müssen vom Entwicklungsteam aufgezeigt werden. Die Lösung dieser Probleme ist im Dialog mit dem Kunden zu finden.

Letztendlich kann eine User Story als Planungseinheit für eine Zeitscheibe verwendet werden. Es ist vorstellbar, dass eine Kette von Teilschritten eines Anwendungsfalls in mehreren Zeitscheiben mit jeweils einer eigenen User Story realisiert wird. Alternativen der Anwendung ohne signifikanten Nutzen werden beim Einsatz von User Stories durch die Mitwirkung des Kunden vermieden. Für Alternativen, die durch technische Fehler verursacht werden, trägt das Entwicklungsteam die Verantwortung.

## 2.7 Ausprägungen der Entwicklungsschritte

In jedem Projekt durchläuft das zu entwickelnde Stück Software die folgenden Entwicklungsschritte:

- Anforderungen aufstellen bzw. Steuerung über den Kundennutzen<sup>48</sup>
- Problembereich analysieren und modellieren,
- entwerfen,
- programmieren,
- testen<sup>49</sup> und
- integrieren.

Diese Entwicklungsschritte überlappen sich zeitlich<sup>50</sup> und werden in der Regel iterativ durchgeführt. Im Fehlerfall muss man zu früheren Entwicklungsschritten zurückkehren.

Wenn man die **Anforderungen** des Kunden nicht beachtet, verfehlt man das Ziel des Projektes. Ohne die Kundenanforderungen zu analysieren und sich ein logisches

---

<sup>47</sup> Generell soll dokumentiert werden, was hilfreich ist. Dokumentieren sollte jedoch nicht die Hauptaufgabe sein oder anfangen zu behindern.

<sup>48</sup> Formale Anforderungen werden in der spezifikationsorientierten Welt verwendet. Formale Anforderungen treten in der agilen Welt in den Hintergrund. Dort gibt meist der Kundennutzen das Ziel vor.

<sup>49</sup> TDD zieht Komponententests nach vorne und verringert somit den Feedback-Zyklus extrem. Funktionstests können jedoch erst erfolgen, wenn eine Funktion vorliegt.

<sup>50</sup> Eine Ausnahme ist das sequenzielle Wasserfallmodell (Baselinemanagement-Modell). Siehe hierzu Kapitel 3.1.1.1.



Modell der Verarbeitung im Entwicklungsschritt **Analyse**<sup>51</sup> auszudenken, fehlt die Grundlage für den **Entwurf**, der unter Einschluss aller Funktionskategorien die statische und dynamische Programmstruktur festlegt und mit seiner Architektur den Bauplan für das zu schreibende Programm darstellt. Das **implementierte** Programm wird **getestet**, um dessen Qualität festzustellen und ggf. noch zu verbessern.

In den Kapiteln 2.7.1 bis 2.7.6 werden die einzelnen Schritte der Entwicklung für ein einfaches System, das nicht über mehrere Ebenen hierarchisch in Teilsysteme zerlegt werden muss, im Überblick vorgestellt.

### 2.7.1 Anforderungen aufstellen bzw. Steuerung über den Kundennutzen

In Abstimmung mit dem Kunden werden in der **spezifikationsorientierten Welt** die Anforderungen aufgestellt. Diese Anforderungen sollen nach Möglichkeit technologieunabhängig sein, damit der Auftraggeber nicht die Verantwortung für die zu liefernde Einheit übernimmt und damit der Auftragnehmer die beste Technologie ermitteln kann. Die Anforderungen sind in der Regel abstrakt. Es steht dann grundsätzlich noch nicht fest, ob die Forderungen durch Hardware oder Software umgesetzt werden.<sup>52</sup>

Zu Beginn eines Projektes können oft nicht alle Anforderungen gefunden werden, die Einfluss auf die Architektur haben. Dann droht eine Überarbeitung der Architektur.



Beim Erstellen der Anforderungsspezifikation befasst man sich sowohl mit funktionalen als auch mit nicht-funktionalen Anforderungen.

In der **agilen Welt** versucht man, durch Gespräche mit dem Kunden herauszufinden, welche Features für den Kunden den größten Nutzen versprechen, um diese zu realisieren.<sup>53</sup> Diese Features werden dann nacheinander realisiert. Die Anforderungen an die noch zu realisierenden Produkte können problemlos solange noch verändert werden, bis das entsprechende Produkt in Produktion geht.

<sup>51</sup> Eine grobe Einteilung der Analyse ist: Überprüfung der Anforderungen auf Konsistenz, die Definition der Systemgrenzen und der Leistungen des Systems sowie die logische Modellierung des Systems.

<sup>52</sup> Solche Vorgaben können in Form von Anforderungen als Einschränkungen des Lösungsraums erhoben werden.

<sup>53</sup> Wenn man den Nutzen durch Schätzen quantifiziert, so kann man ihn durch den geschätzten Aufwand teilen. Damit erhält man den geschätzten Nutzen pro verbrauchte Geldeinheit. Damit hat man die Möglichkeit, solche Funktionen zu priorisieren, die wirtschaftlich sind und eine return on investment versprechen.

## 2.7.2 Analyse

Eine grobe Einteilung der Analyse ist:

1. Überprüfung der Anforderungen an das betrachtete Produkt<sup>54</sup> auf Konsistenz (Widerspruchsfreiheit) bzw. der Kundenwünsche auf Konsistenz,
2. Festlegung der Grenzen des Produkts,
3. Modellierung des betrachteten Produkts.

Diese Schritte werden im Folgenden detailliert vorgestellt:

- **Überprüfung der Anforderungen an das betrachtete System bzw. der Kundenwünsche an ein Produkt auf Konsistenz**

In diesem Schritt werden zunächst die Kundenvorgaben auf Konsistenz überprüft. Im Rahmen der Konsistenzprüfung muss durchleuchtet werden, wo Änderungen erforderlich sind.

- **Definition der Grenzen des Produkts**

Die Grenzen des zu realisierenden Produkts müssen abgesteckt werden. Es muss entschieden werden, was im Rahmen des Projekts zu realisieren ist und was nicht zum Produkt gehört. Die einzelnen Leistungen, die das Produkt erbringen soll, müssen festgehalten werden.

- **Modellierung des betrachteten Produkts**

Das betrachtete Produkt ist technologieunabhängig und damit auch unabhängig von den physischen Randbedingungen der technischen Lösung wie der Verwendung eines Betriebssystems, eines Datenbankmanagementsystems oder nebenläufiger Betriebssystem-Prozesse. Man spricht auch von der Modellierung der Essenz, vom Erstellen des Fachkonzeptes oder von der Modellierung der Logik. Modelliert werden u. a. die Wechselwirkungen der verschiedenen Objekte der betrachteten Einheit.

## 2.7.3 Entwurf

Die Essenz der betrachteten Einheit in einer idealen Welt muss in den Bauplan für ablauffähige Programme der realen Welt umgesetzt werden.

Beim Entwurf kommt beispielsweise auch die Abbildung der Essenz auf verteilte Rechner und nebenläufige Betriebssystem-Prozesse oder Threads ins Spiel. Darüber hinaus ist der Einsatz von Standard-Software zum Beispiel für die Kommunikation über ein Netz oder zur Speicherung der Daten in Datenbanken mit Hilfe eines Datenbankmanagementsystems zu betrachten.

Da der Entwurf von den Möglichkeiten der eingesetzten Standard-Software wie Betriebssystem, Datenbankmanagementsystem, Netzwerksoftware oder dem Werkzeug für die Generierung der Dialoge der Mensch-Maschine-Schnittstelle abhängt,

---

<sup>54</sup> Das betrachtete Produkt kann ein Gesamtsystem oder eine Komponente eines Gesamtsystems sein.

muss die Entscheidung über die einzusetzende Technologie gefällt werden, ehe man zu programmieren beginnt.

Der Entwurf befasst sich mit der Umsetzung der Ergebnisse der Analyse in eine ablauffähige Programmstruktur, d. h. in eine Architektur.<sup>55</sup>



### 2.7.4 Programmierung

Das entworfene System wird realisiert. In der Softwareentwicklung wird Programmierung auch als Implementierung bezeichnet.

### 2.7.5 Test & Integration

Die realisierten Programme müssen getestet und integriert werden. Die Testfälle sind auf Grund der Spezifikation für die betrachtete Einheit zu erstellen. Sie sollten in der spezifikationsorientierten Welt bereits beim Aufstellen der Anforderungen formuliert werden. Die richtige Auswahl an Testfällen – sowohl qualitativ als auch quantitativ – ist entscheidend für das Finden von Fehlern.

Bei den **agilen Methoden** ist es auf Grund der zahlreichen Iterationen besonders empfehlenswert, von Anfang an die Schnittstellen testgetrieben zu entwickeln und entstandene Tests automatisiert und kontinuierlich auszuführen. Ein Funktionstest bei Vorliegen des entsprechenden Moduls reicht nicht aus. Es müssen auch die Funktionen des Systems getestet werden, da sich diese meist über mehrere Module erstrecken.

### 2.7.6 Abnahme

Bei der Abnahme eines Produkts wird – in der Regel beim Kunden – getestet, ob das betrachtete Produkt mit den Wünschen des Kunden übereinstimmt und die Erwartungen des Kunden erfüllt. Nach der Abnahme gehen die Programme in den Betrieb und damit auch in die Wartung.

---

<sup>55</sup> Mit der Architektur wird aber schon vor dem Entwurf begonnen.

Mit Scrum zum gewünschten System

Goll, J.; Hommel, D.

2015, VIII, 185 S. 46 Abb., Softcover

ISBN: 978-3-658-10720-8