1 Einführung

»Debugging is twice as hard as writing the code in the first place. Therefore, if you write code as cleverly as possible, you are, by definition, not smart enough to debug it. «

Brian Wilson Kernighan (kanadischer Informatiker und Koautor der Programmiersprache C)

Das Zitat von Brian Kernighan klingt zwar witzig, doch es war durchaus ernst gemeint. Brian brachte in einem Satz auf den Punkt, dass an einen Tester hohe intellektuelle Anforderungen gestellt werden.

Ein weiteres Zitat aus der Praxis stammt von einem Softwareentwickler in einer Lebensversicherung: »Ich habe noch anderes zu tun, als meinen Code zu testen.« Eines seiner Module hatte gerade eine größere Produktionsstörung verursacht, nachdem er es ohne Abnahmeprozess und ungetestet im Produktionssystem implementiert hatte. Die notwendige Einsicht fehlte bei diesem Entwickler und führte zu seiner recht ungehaltenen Äußerung.

Eine weitere Situation spielte sich einmal in einer Großbank ab. Es sollte ein Workshop mit einem erfahrenen Business-Analysten zur Definition von geeigneten Testfällen geplant werden. Ein großer Releasewechsel stand in den nächsten Monaten an und es waren noch keine Test- und Abnahmeverfahren definiert. Der erfahrene und ansonsten sehr kompetente Analyst lehnte das Meeting telefonisch ab mit den Worten: »Wieso soll ich mir die ganze Mühe für das Testen machen. Das Eröffnen eines Fehlers (engl. Defect) ist billiger.« Gemeint hatte er, dass er bedeutend weniger Zeit für die Eröffnung eines Störungstickets benötigt als für die Definition und die Durchführung von Testfällen (engl. Test Cases). Ob die Behebung eines Fehlers in der Produktion wirklich billiger ist, hat er sich in dieser Situation sicher nicht überlegt.

Diese drei Aussagen zeigen einige Eigenheiten des Testens auf:

1. Testen benötigt Intelligenz und Erfahrung.
2. Testen ist mühsam und unbeliebt.
3. Testen muss wirtschaftlich sein und bleiben.
4. Der Nutzen bzw. die Notwendigkeit ist den meisten nicht bewusst oder bekannt.

1.1 Ungenügendes Testen ist leider Praxis

Ungenügendes oder fehlendes Testen ist leider weit verbreitet. Nicht von ungefähr wird bei Softwarelösungen gerne der Begriff des »Management by banana« verwendet. Gemeint ist »das Produkt beim Kunden reifen lassen«. Die Ursache liegt nicht in der bösen Absicht der Entwickler, sondern häufiger in deren Unwissenheit und Unerfahrenheit. Projektteams sind meistens bestens ausgebildet in den einzusetzenden Technologien, haben ein fachliches Grundwissen und sind geschult in den verschiedensten Projektmodellen. Selten verfügt aber nur ein einziges Mitglied über eine entsprechende Ausbildung im Testen.

In vielen Fachbüchern zum Thema Data Warehousing und Business Intelligence ist zwar beschrieben, wie Systeme gebaut und später betrieben werden müssen. Aber selten enthält eines dieser Bücher ein Kapitel zum Testen. Sofern in einem Abschnitt das Testen erwähnt ist, wird es nur auf ein oder zwei Seiten behandelt. Im Verhältnis zu den mehreren Hundert Seiten einzelner Bücher ist dies auch eine klare Aussage zum Testverständnis der Autoren.

Aufgrund des fehlenden Wissens wird mit dem Testen viel zu spät begonnen. Irgendwo am Ende einer Entwicklungsphase folgt in den meisten Projektplänen die Testphase, die nur als Vorgang definiert ist, der nicht weiter gegliedert ist. Dabei wird übersehen, dass ohne Testplanung leider keine effektive Testdurchführung stattfinden kann. Dies bedeutet, dass Testen schon viel früher in den Projekten zu beginnen hat. Idealerweise schon kurz nach dem Projektstart.

Eine objektbezogene Planung ist heute in den meisten Projektvorgehensmodellen üblich für Analyse, Spezifikation und Realisierung. Das heißt, es wird für jeden Vorgang ein messbares Lieferergebnis als Output definiert. Oder, mit anderen Worten, jede Aktivität liefert am Ende ein bewertbares Resultat. Testen wird nur als Vorgang geplant, was meistens der Testdurchführung entspricht. Wenn die Testplanung fehlt, kann jedoch nichts Vernünftiges durchgeführt werden.

Die Verantwortung für ungenügende Tests darf nicht allein dem Projektteam zugeschoben werden. Der Auftraggeber steht genauso in der Verantwortung, im Projektauftrag messbare Akzeptanzkriterien für die einzelnen Lieferobjekte und für das gesamte System zu definieren.

Durch die fehlende Definition von Testfällen dient die Testphase in vielen Projektplänen nur noch als Puffer, um Zeit- oder Kostenüberschreitungen aus anderen Vorgängen aufzufangen. Durch die Verwendung der Testphase für andere Aufgaben werden der Projektplan, das Budget und der Einführungstermin eingehalten. Echte Tests werden nur wenig durchgeführt.

Ein weiteres Problem besteht darin, dass Tests nur durch den Entwickler durchgeführt werden. Er prüft einzig, ob seine Module durchlaufen, ohne abzustürzen. Es existieren keine klar formulierten Testfälle und seine Prüfungen definiert er selbst. Das bedeutet, es werden nur minimale funktionale Modultests durchgeführt.

Wenn Personen der Fachabteilungen in den Testprozess einbezogen werden, sind diese meistens ungenügend vorbereitet. Manchmal kennen sie nicht mal den Zweck des Systems, sondern erhalten einfach den Auftrag: »Das System steht auf dem Server XY bereit, macht mal in den nächsten zwei Wochen ein paar Tests.« Die Testpersonen sind damit überfordert und es finden im definierten Zeitraum gar keine Testaktivitäten statt. Um erfolgreich testen zu können, ist zuvor eine minimale Benutzerschulung notwendig und es müssen formulierte Testfälle vorhanden sein. Zusätzlich müssen die Tester frühzeitig informiert werden, damit sie die benötigte Zeit auch reservieren können.

Eine Ursache für ungenügende Tests ist der Zeitdruck in Projekten. Das Reduzieren der Entwicklungsbudgets lässt eine Umsetzung manchmal nicht mehr zu oder führt zu mangelhafter Qualität. Daher wird jeder Projektleiter eine Testphase einplanen und das Budget dann anderweitig verwenden. Somit kann er zumindest die Funktionen zur Verfügung stellen. Wenn ungenügend getestet wird, gibt es auch keine zu korrigierenden Fehler. Somit erreicht er sein Ziel: die termingerechte Einführung des Systems. Die Folgen muss dann der Betriebsverantwortliche tragen. Er hat jede Menge Produktionsstörungen und muss sehen, wie er diese im Rahmen des im Service Level Agreement (SLA) vereinbarten Budgets behandeln kann. Der Projektleiter kümmert sich nicht mehr darum. Er hat eine unterzeichnete Projektabnahme und ist bereits an der Planung und Umsetzung seines nächsten Projekts.

Der Umgang mit Fehlern (engl. Defects) ist in der Praxis ebenfalls oft ungenügend. Nicht in allen Projekten existieren Vereinbarungen zu Klassifikation von Fehlern und zur Projektabnahme. Die Fehler werden nicht zentral erfasst und die Lösung wird nicht protokolliert. Somit ist am Ende des Projekts nicht bekannt, welche Fehler mit welcher Gewichtung offen sind. Der Einsatz einer Softwarelösung zur Fehlernachverfolgung (engl. Defect-Tracking) würde hier die Arbeit erleichtern, beispielsweise das Open-Source-Tool Bugzilla. Des Weiteren enthalten einige Projektpläne kein Zeitfenster für das Korrigieren der Fehler und das erneute Testen, als ob Systeme nach den ersten Tests vollständig und perfekt wären.

Eine weitere Unsitte ist das Korrigieren der Fehlerprioritäten. Fehler werden in tiefere Klassen eingeordnet, damit das System abgenommen werden kann. Dieses Herunterstufen (engl. Downgraden) von Fehlern nützt längerfristig niemandem. Man erhält dadurch ein System mit ungenügender Qualität.

Die Liste von möglichen Ursachen für ungenügende Tests könnte beliebig fortgesetzt werden. Zusammenfassend kann gesagt werden, dass es sicher keine böse Absicht ist, ein ungenügendes System abliefern zu wollen. Meistens handelt es sich nur um fehlendes oder ungenügendes Testwissen. Dieses Buch möchte diese Wissenslücke schließen und insbesondere noch auf die Eigenheiten des Testens von analytischen Systemen, wie Business-Intelligence-Anwendungen und Data Warehouses, eingehen.

Ein offener Testfall in einer Großbank

Als Tester war ich verantwortlich für die Verifikation eines geänderten Sourcing-Prozesses einer Shared Dimension. Im Testsystem blieb der geplante Ladeprozess längere Zeit unausgeführt und ein Testen war nicht möglich. An einem Abend, zwei Tage vor Ende der Testphase, rief mich der Chefentwickler an und fragte nach, wieso ich den Test nicht abnehme. Ohne meine Abnahme könne das Release nicht eingeführt werden. Das Release bestand aus mehreren Hundert Änderungen am gesamten Warehouse.

Mein Einwand, dass der Ladeprozess noch nie ausgeführt wurde und ich somit keine Tests durchführen kann, interessierte ihn wenig. Seine Antwort war lapidar: »Dann setz deine Testfälle einfache auf ›passed‹. Wir schauen uns das Ganze dann in der Produktion an. Alle anderen machen dies genauso.« Hier musste ich zuerst einmal leer schlucken und tausend Gedanken gingen mir durch den Kopf. Einerseits wollte ich nicht der Verhinderer eines kommenden Release sein. Dies klingt nach Ärger und der späteren Suche nach Schuldigen. Andererseits wollte ich nicht Testfälle auf ›passed‹ setzen, obwohl es nicht stimmte. Dies käme für mich einer vorsätzlichen Lüge gleich. Außerdem glaubte ich nicht daran, dass mir der Chefentwickler später in der Produktion die notwendige Unterstützung zur Behebung des Fehlers geben würde. Auch war ich über die Aussage entsetzt, dass es üblich sei, einfach Tests auf ›passed‹ zu setzen.

Ich entschied mich, bei meinen Prinzipien zu bleiben, und antwortete: »Ich bin nur verantwortlich für die Testfälle und nicht für das gesamte Release. Solange ich die Tests nicht durchführen kann, werde ich sie offen lassen.« Nun spürte der Chefentwickler den Druck, den er auf mich ausüben wollte. Er kümmerte sich um das Problem der nicht durchgeführten Ladeprozesse und er analysierte die Situation. Schlussendlich wurde festgestellt, dass sämtliche Einträge für das Scheduling der Jobs verloren gegangen waren. Die Ladeprozesse wurden deshalb nie ausgeführt. Noch in derselben Nacht liefen die Jobs erstmalig und fehlerfrei durch. Ein Tag vor Releasefreigabe konnte ich die Testfälle nun mit gutem Gewissen auf ‚passed' setzen.

Zusammenfassend bedeutet dies, dass es nicht um ein erfolgreiches Testen ging, sondern nur um die Behauptung, dass getestet wurde. Ob es wirklich Praxis war, die Testfälle einfach auf passed zu setzen, habe ich nie herausgefunden. In diesem Fall hat es sich gelohnt, dem Druck nicht nachzugeben. Dieser Dimensionsladeprozess wäre wahrscheinlich nie korrekt gelaufen.

1.2 Wirtschaftlichkeit des Testens

Testen muss wirtschaftlich sein. Abbildung 1–1 stellt schematisch die Folgekosten von Fehlern in der degressiven Kurve und die Kosten durch das Testen in der progressiven Kurve dar. Dies bedeutet, wenn nicht getestet wird, entstehen hohe Kosten durch Fehler. Einerseits sind dies direkte Kosten, die der Schaden des Fehlers verursacht. Andererseits sind es Kosten, die für die Suche und das Beseitigung des Fehlers notwendig sind. Über ein effektives Testen werden die Fehler und somit die Folgekosten reduziert. Doch auch das Testen verursacht Kosten. Wie die Kurven zeigen, stehen ab einem gewissen Punkt die Kosten des Testens in keinem Verhältnis mehr zum Nutzen.

Effizientes Testen ist somit bis zum Schnittpunkt der beiden Kurven sinnvoll. Das Problem ist, dass nur bekannte Fehler quantifiziert werden können. Es bleibt somit immer ein Restrisiko, dass der eine nicht gefundene Fehler den Millionenschaden verursachen kann.

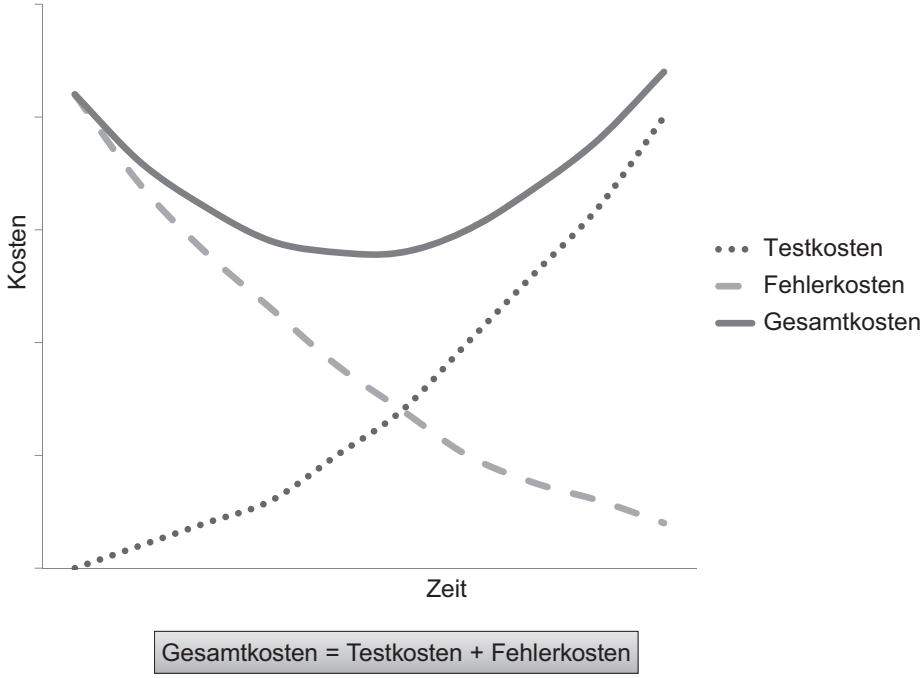

Abb. 1–1 *Wirtschaftlichkeit des Testens*

Ariane 5 – ein kleiner Fehler mit einer großen Auswirkung

Ein gutes Beispiel für die Auswirkungen eines einzelnen Fehlers ist die Ariane-5-Rakete, die am 5. Juni 1996 explodierte. Ursache war ein Fehler in einem ADA-Modul, das für die Regulierung der seitlichen Steuerraketen zuständig war. Eine 64-bit-Floatingpoint-Variable wurde dabei mit einer 16-bit-Integer-Variablen verglichen. Dies führte zu einem Overflow-Fehler und die Steuerung der Raketenflugbahn war nicht mehr möglich. Da Ariane 5 die Flugbahn verließ und unkontrolliert abzustürzen drohte, wurde sie automatisch, nach erst 40 Sekunden Flug, über dem Meer gesprengt, inklusive aller Forschungssatelliten. Die Bilder gingen damals um die Welt. Die Herstellungskosten nur für die Rakete und die Satelliten wurden mit 500 Mio. USD beziffert. Die spätere Untersuchung zeigte, dass die Entwicklung dieses ADA-Moduls auf einer fehlerhaften Spezifikation beruhte, die nie verifiziert wurde, und dass das Softwaremodul ungenügend getestet wurde.

1.3 Vollständiges Testen ist nicht möglich

Die Komplexität von heutigen Informatiksystemen lässt kein vollständiges Testen zu. Jede einzelne Funktion bietet mehrere Möglichkeiten. Werden Kombinationsmöglichkeiten aller Funktionen berechnet, indem alle Varianten der einzelnen Funktionen miteinander multipliziert werden, entsteht eine unwahrscheinlich hohe Zahl. Jede dieser Kombinationen zu testen, würde Jahre dauern.

Bereits der zweite der sieben Grundsätze des Testens beschreibt diese Situation.[1]

Bei der Testplanung ist eine Risikoabwägung notwendig zwischen dem Testaufwand auf der einen Seite und der Wahrscheinlichkeit und Auswirkung von unentdeckten Fehlern auf der anderen Seite.

1.4 Gegenüberstellung von BI- mit klassischen Projekten

Bill Inmon, einer der Urväter des Data Warehouse, prägte vor Jahren folgenden Satz [Inmon 2005]:

»Traditional projects start with requirements and end with data. Data Warehousing projects start with data and end with requirements.«

Damit wollte er sagen, dass in klassischen Projekten transaktionsbasierte Systeme entstehen, mit denen Informationen erfasst und in Datenbanken gespeichert werden. Ein typisches Beispiel ist ein System zur Pflege von Kundenstammdaten.

Business-Intelligence-Projekte verwenden die existierenden Datenbanken, um Analysen in den verschiedensten Formen zu erstellen, sei es als Listen in Papierform, elektronisch als Führungscockpits, Risikoanalysen mittels Data Mining oder in weiteren Varianten.

Wenn sich die Projekte schon stark unterscheiden, muss dies auch einen Einfluss auf das Testvorgehen haben. In Tabelle 1–1 haben wir die Projekte einander gegenübergestellt und Unterschiede herausgearbeitet. Business-Intelligence-Projekte haben wir noch folgendermaßen unterteilt:

- **ETL (Extract, Transform and Load)**
  Dies sind alle Prozeduren, die Daten aus den Quellsystemen lesen und in ein Data Warehouse oder eine andere Form von analytischem Datenspeicher schreiben. ETL-Prozesse sind auch für alle Transformationen und Übertragungen von Daten innerhalb des Data Warehouse zuständig, beispielsweise von der Staging Area ins definitive Modell.

1. Alle sieben Prinzipien des Testens sind in Abschnitt 1.6 beschrieben.

- **Speicherung**
  Dies können ein Core Data Warehouse, mehrere separate multidimensionale Datenbanken oder eine sonstige proprietäre Datenspeicherformen sein. Diese multidimensionale Datenbanken sind die Quelle für das Frontend oder die Ausgangslage für die Power User, die selbst Ad-hoc-Abfragen erstellen.
- **Frontend**
  Darunter verstehen wir alle Komponenten, die für die Visualisierung der Informationen zuständig sind. Das können Listen, Cockpits etc. sein.

In der Gegenüberstellung haben wir rein technische Projekte nicht berücksichtigt, beispielsweise für den Aufbau der Infrastruktur, die die Installation und Konfiguration eines BI-Servers und das Einrichten des Berechtigungssystems beinhaltet. In der Darstellung haben wir uns nur auf Projekte mit fachlichen Inhalten entlang des Datenflusses orientiert, von der Beschaffung bis zur Präsentation. Die Gliederung ist stark vereinfacht und viele Projekte beinhalten mehrere oder sogar alle Ausprägungen dieser Projekttypen. Unser Weglassen der technischen Infrastrukturprojekte bedeutet nicht, dass diese Projekte von untergeordneter Bedeutung sind.

	Klassische IT- Projekte	**Business-Intelligence-Projekte**		
		ETL	**Speicherung**	**Frontend**
Projektmethodik	Mehrheitlich traditionelle Modelle (Wasserfall, V-Modell, …) Agile Methoden gewinnen zunehmend an Bedeutung	Mehrheitlich iterativ, Prototyping oder agil, beispielsweise Scrum		
Anforderungen	Klar definiert			Unscharf definiert
Code	Statisches SQL in Code Ausprogrammierte Funktionen	Dynamisches SQL oder MDX Interaktive Codegenerierung		Dynamisches SQL oder MDX
Datenmodelle	Konsistent mit referenzieller Integrität, relational, 3NF	Heterogene Quellen	Multidimensional, temporär inkonsistent	
Datenzugriff	Update einzelner Datensätze Zeilenweise lesend (alle Attribute einer einzelnen Zeile)	Massenupdates (Bulkload)		Spaltenweise lesend (einzelne Attribute aus allen Zeilen einer Tabelle)

→

	Klassische IT- Projekte	Business-Intelligence-Projekte		
		ETL	Speicherung	Frontend
Daten und Testdaten	Nicht vorhanden, diese entstehen durch Erfassung im Testsystem	In den operativen Quellsystemen vorhanden. Wie können diese genutzt werden? (Sicherheit muss geregelt werden.)		
Testschwerpunkte	Anwendungsfallgetrieben Funktionsbasiert	Datengetrieben		
		Datenqualität Funktionale Tests Integrations- und Regressionstests bei Beschaffung oder Releasewechsel Performance	Datenqualität Performance	Integrations- und Regressionstests bei Beschaffung oder Releasewechsel Performance

Tab. 1–1 *Gegenüberstellung klassischer IT-Projekte mit Business-Intelligence-Projekten*

In den klassischen IT-Projekten wird häufig noch mit Phasenmodellen gearbeitet. Am Anfang werden klar verbindliche Spezifikationen geschrieben, ausgehend von den formulierten Anforderungen (engl. Requirements). Anschließend folgen die Realisierung, die Tests und die Einführung.

In Business-Intelligence-Projekten ist das Definieren von klaren Anforderungen meistens sehr schwierig. Nicht unüblich sind Sätze wie: »Wir möchten ein Set von Auswertungen über unsere Kundendaten haben, um den Verkaufsprozess besser steuern zu können« oder: »Wir vermuten, dass wir unser Potenzial nicht ausnutzen, weil uns die Informationen fehlen«. Daher wurde in Business-Intelligence-Projekten schon früh mit nicht linearen Projektvorgehensmodellen begonnen, wie Prototyping oder iterativen Modellen. Speziell das Rapid Prototyping eignet sich besonders, um schnell ein erstes Set von Ergebnissen zu liefern, um daran die Anforderungen diskutieren und präzisieren zu können.

In beiden Projektwelten hat sich in den letzten Jahren die Methode Scrum durchgesetzt. Dabei wird manchmal vergessen, dass es noch weitere agile Methoden gibt, wie beispielsweise Extreme Programming, die bezüglich Agilität teilweise noch weiter gehen.

Auch bezüglich des Codes gibt es Unterschiede. Klassische Projekte verwenden statisches SQL zum Lesen und Schreiben von Daten. Diese SQL-Anweisungen sind eingebettet im Programmcode, beispielsweise in C++, Visual Basic oder Java. Dieser Code ist kompiliert und ebenso statisch. In BI-Werkzeugen, speziell

in den Abfragewerkzeugen im Frontend-Bereich, wird die Abfrage dynamisch generiert. Nur ganz selten schreibt ein Entwickler SQL- oder MDX-Code.

In Business-Intelligence-Systemen sind die Datenspeicher temporär inkonsistent, und zwar aus mehreren Gründen:

- Lose Kopplung bei ROLAP-Speicherung. Das heißt, es existiert keine referenzielle Integrität innerhalb eines sternförmigen Modells.
- Unterschiedliche Ladezeitpunkte aus den verschiedenen Quellsystemen, was eine Synchronisation erst zu einem späteren Zeitpunkt möglich macht.

Da bereits Quelldaten vorhanden sind, werden diese gerne für Tests genommen. Das bedeutet, dass Tests mit produktiven Daten in Business-Intelligence-Projekten nicht unüblich sind. Wichtig ist hier das Beachten der Sicherheitsanforderungen. Meistens geht es um die Frage der Anonymisierung und um das Verändern von Summen. Trotz des Anonymisierens und Veränderns von Werten bleibt das Testen mit produktiven Daten ein Problem. Wahrscheinlich wird keine Bank einem externen Berater die Kundendaten für Tests zur Verfügung stellen.

Anwendungsfälle (engl. Use Cases) bilden häufig die Grundlage zur Definition von funktionalen Tests. In analytischen Systemen gibt es nur wenige oder nur bedeutungslose Anwendungsfälle. Viel wichtiger sind datengetriebene Testfälle, die noch ausführlich in Abschnitt 5.1 behandelt werden.

1.5 Einfluss von Daten aufs Testen

Wie schon erwähnt, sind bei den meisten analytischen Systemen die Daten aus den Quellsystemen bereits vorhanden und werden daher gerne fürs Testen verwendet. Dagegen ist nichts einzuwenden, solange die Daten, soweit notwendig, entsprechend geschützt werden. Dies geschieht mit Zugriffsbeschränkungen und durch das Anonymisieren der Daten.

Die Datenqualitätsprobleme der Quellsysteme werden dabei gleich mit importiert. Verschiedene Fehlerlogs aus mehreren Projekten wurden ausgewertet, um die Bedeutung der Daten und deren Qualität zu analysieren. Die Auswertung in Abbildung 1–2 ist nicht absolut zu verstehen, da es sich um keine empirisch relevante Menge von ausgewerteten Projekten handelt.

Die Auswertung zeigt, dass zu Beginn der Testphase noch Implementierungsfehler die größte Gruppe von Fehlern sind. Implementierungsfehler sind falsch gesetzte Filter, unvollständige Umsetzung der Spezifikation oder ungenau umgesetzte Designvorgaben, um einige Beispiele zu nennen. Viele dieser funktionalen Fehler werden mit den ersten Tests erkannt und können schnell behoben werden. Gegen Ende der Testphase spielen diese funktionalen Fehler nur noch eine untergeordnete Rolle. Meistens haben die dann noch existierenden Fehler nur noch geringe Auswirkungen und sind daher von untergeordneter Bedeutung.

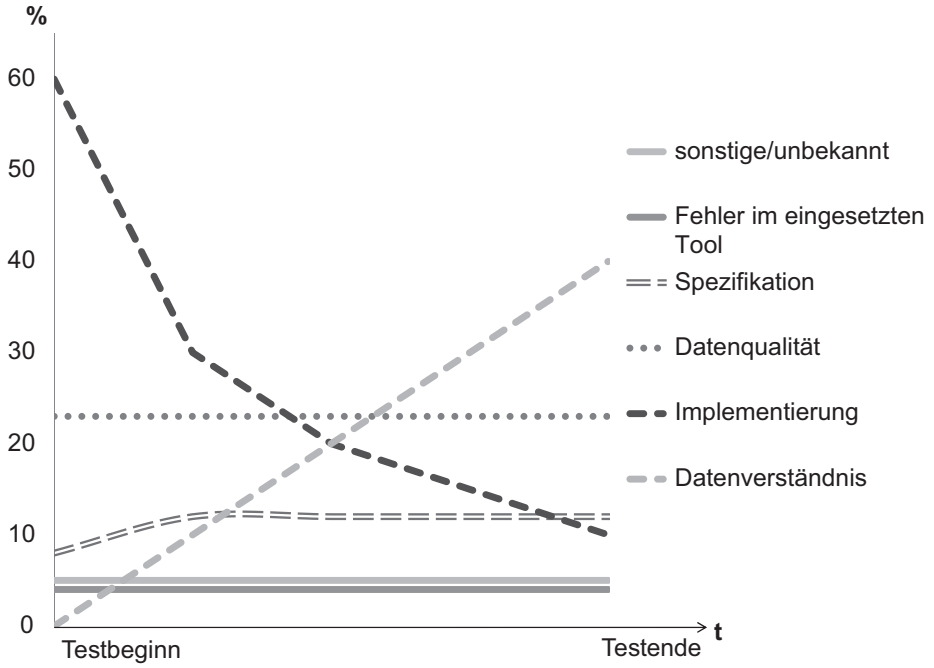

Abb. 1–2 *Prozentualer Anteil von Fehlern nach Kategorien im Verlauf über die Testphase*

Eine größere Gruppe von festgestellten Fehlern hat nicht direkt mit dem neuen oder zu ändernden analytischen System zu tun. Sie werden verursacht durch Qualitätsprobleme in den Quelldaten. Bei analytischen Systemen werden besondere Anforderungen an die Robustheit von Importschnittstellen gestellt. Durch das Testen mit echten Daten werden echte Fehler sofort erkannt und das System kann bereits darauf ausgerichtet werden[2].

Eine weitere Gruppe von datenbezogenen Fehlern hat die Ursache im Datenverständnis. Diese Gruppe war die Überraschung bei den Auswertungen, da die Existenz dieser Fehler zu Beginn nicht einmal vermutet wurde. Da die ersten Testfälle noch sehr einfache funktionale Überprüfungen sind, treten datenbezogene Fehler erst mit der Zeit auf. Mit dem Fortschritt des Testens werden immer komplexere Testfälle durchgeführt. Dabei kommt es zu Interpretationsfehlern. Beispielsweise ist eine Kennzahl für den Tester missverständlich und er interpretiert das Ergebnis als fehlerhaft. Funktionale Anpassungen am System sind nicht notwendig, sondern es braucht nur eine Erklärung. Es wäre jetzt einfach, die Fehlermeldung zu schließen mit der Begründung, dass es kein Fehler ist und der Tester ungenügende Kenntnisse hatte. Genau diese Fehlergruppe weist auf Mängel in

2. Weitere Hinweise zum Qualitätsmerkmal »Fehlertoleranz« und »Reife« enthält Abschnitt 4.2.3. Informationen zum Testen mit produktiven Daten sind in Abschnitt 6.2 beschrieben.

der Benutzbarkeit (engl. Usability) hin[3]. Eine Verbesserung ist hier nicht einfach. Möglich wird sie durch den Einsatz einer guten und umfangreichen Benutzeranleitung, eines ausführlichen Hilfesystems oder eines Data Dictionary. Idealerweise werden gleich mehrere dieser Instrumente eingesetzt.

Über die ganze Testphase hinweg machen datenbezogene Fehler die größte Gruppe von Fehlern aus. Die beiden Gruppen Datenqualität und Datenverständnis werden später in diesem Buch noch ausführlich behandelt.

1.6 Die 7 Prinzipien des Testens

Seit vielen Jahren haben sich sieben Grundsätze des Testens etabliert. Die genaue Herkunft und wann diese Prinzipien aufgestellt wurden, lässt sich nicht genau ermitteln. Im Internet existieren dazu verschiedene Versionen. Diese Prinzipien sind in die Lehrpläne zum Certified Tester nach ISTQB[4]-Standard [URL: ISTQB] eingeflossen.

Die sieben Prinzipien sind die Essenz des Testens. Oder, mit anderen Worten, es sind die sieben Grundwahrheiten des Testens, die nachfolgend beschrieben sind.

1. Prinzip: Testen zeigt die Anwesenheit von Fehlern

Mit Testen wird das Vorhandensein von Fehlern nachgewiesen. Nicht zulässig ist die Schlussfolgerung, dass eine Softwarelösung keine Fehler mehr enthält, nur weil keine Fehler mehr gefunden werden. Fehlerfreiheit lässt sich nie beweisen. Testen reduziert einzig die Wahrscheinlichkeit von unentdeckten gravierenden Fehlern.

2. Prinzip: Vollständiges Testen ist nicht möglich

Die Komplexität von Softwarelösungen ergibt unzählige Testmöglichkeiten. Durch die Vielzahl von Kombinationsmöglichkeiten der unterschiedlichen Funktionen und Eingabewerten, ist es unmöglich, sämtliche Varianten zu testen. Tests sind daher immer nur Stichproben. Der Umfang der Tests wird geprägt von Risiko- und Wirtschaftlichkeitsüberlegungen. Ein Restrisiko bleibt.

3. Prinzip: Mit dem Testen sollte so früh wie möglich begonnen werden

Je später Fehler erkannt werden, desto teurer ist deren Korrektur. Daher sollte mit der Testplanung gleich am Anfang des Projekts und mit dem Testen bereits zu

3. Verschiedene Aspekte zum Qualitätsmerkmal »Bedienbarkeit« und »Datenverständnis« sind in Abschnitt 4.2.2 beschrieben.
4. International Software Testing Qualifications Board

Beginn der Entwicklung begonnen werden. Reviews der Konzepte und Spezifikationen können schon vor dem Schreiben der ersten Codezeile durchgeführt werden.

4. Prinzip: Beachten von Fehlerhäufung

Fehler sind nie gleichmäßig verteilt. Einzelne Testobjekte werden immer eine größere Fehlerdichte aufweisen. Dies hat verschiedene mögliche Ursachen. Die Komplexität der Lösung ist höher als in anderen Modulen. Oder der Entwickler hatte ungenügend Erfahrung.

Fehlerhäufungen (engl. Defect Cluster) sind ein Hinweis, dass es an dieser Stelle noch weitere unentdeckte Fehler gibt. Es sollten unbedingt weitere Tests durchgeführt werden. Wer sich nur auf die ursprünglich vordefinierten Testfälle verlässt und sich mit der Behebung der gefundenen Fehler zufriedengibt, zeigt einen eher fatalistischen Umgang mit dem Testen.

5. Prinzip: Varianz bei den Testfällen

Das Anwenden der immer gleichen Testfälle führt dazu, dass die Software bereits auf diese Tests vorbereitet ist. Das heißt, es werden zwar immer weniger Fehler gefunden. Es wird aber nicht der Nachweis erbracht, dass auch weniger Fehler existieren.

Die vorhandenen Testfälle müssen daher regelmäßig überarbeitet und durch neue Testfälle und andere Methoden ergänzt werden. Ansonsten sinkt die Effektivität des Testens.

Für dieses Verhalten von Software wird der Begriff des »Pesticide paradox« verwendet. Ähnlich wie Schädlinge mit der Zeit eine gewisse Resistenz gegen Gifte entwickeln können, kann auch die Software eine Testresistenz entwickeln.

6. Prinzip: Testen ist abhängig von verschiedenen Einflussfaktoren

Es gibt nicht das universelle Set von Testfällen und -methoden. Tests sind immer abhängig von verschiedenen externen und internen Einflussfaktoren. Interne Faktoren sind die Kritikalität und Komplexität des Systems. Externe Faktoren sind die Projektmethode, die zur Verfügung stehende Zeit und die vorhandenen Testressourcen, wie Software, Personen etc.

7. Prinzip: Keine Fehler bedeutet ein brauchbares System, ist eine falsche Schlussfolgerung

Ein fehlerfreies System bedeutet noch lange nicht, dass das System auch von den Benutzern akzeptiert wird. Das frühzeitige Einbeziehen der Benutzer und Rapid Prototyping im Entwicklungsprozess sind vorbeugende Maßnahmen gegen unzufriedene Kunden.

1.7 Validieren und Verifizieren

Die Begriffe Validieren und Verifizieren umschreiben unterschiedliche Arten von Prüfungen. Grundlage ist eine andere Sichtweise des Systems. Die Unterschiede von Validierung und Verifikation sind in Tabelle 1–2 dargestellt.

Validierung	Verifikation
Ziel: Bestätigen der Praxistauglichkeit des Systems in Bezug auf Robustheit der Module	Ziel: Bestätigen der Praxistauglichkeit des Systems in Bezug auf Erfüllung des Einsatzzwecks
Überprüft, ob ein Modul oder System die spezifizierten Anforderungen erfüllen kann.	Überprüft die Korrektheit, Vollständigkeit und formale Richtigkeit eines Systems oder Phasenergebnisses.
Tut das System die richtigen Dinge?	Ist es das richtige System?
Vollständigkeit	Korrektheit
Externe Sicht auf einzelne Funktionen oder Module	Interne und externe Sicht auf die Systemanforderungen
Erfüllung von funktionalen und nicht funktionalen Anforderungen	Erfüllung von technischen Anforderungen, Codierungsstandards, rechtlichen und regulatorischen Vorgaben

Tab. 1–2 *Gegenüberstellung Validierung und Verifikation*

1.8 Was Testen nicht kann

Testen dient ausschließlich der qualitativen Überprüfung eines Systems und dem Aufzeigen von Fehlern. Wunder sind nicht zu erwarten.

Edsger Wybe Dijkstra (1930-2002) formulierte die Grenzen des Testens in einem Satz: *»Testing can only show the presence of bugs, but not their absence.«* Auf Deutsch übersetzt bedeutet der Satz, dass Testen nur die Anwesenheit von Fehlern, jedoch nie die Abwesenheit von Fehlern zeigen kann. Edsger Dijkstra hatte Mathematik studiert und sich schon früh für den Weg zum Informatiker entschieden. Er entwickelte verschiedene Algorithmen und war an der Entwicklung eines der ersten multitaskingfähigen Betriebssysteme beteiligt. Später war er Professor für Mathematik. Des Weiteren wurde er bekannt durch seine prägnanten Formulierungen, die gerne zitiert werden.

Sein Satz zum Testen beschreibt in wenigen Worten genau den Inhalt des ersten der sieben Grundprinzipien[5]. Egal wie aufwendig und intensiv getestet wird, es können nur Fehler entdeckt werden. Es bleibt weiterhin die Vermutung, dass die gefundenen Fehler nur eine Teilmenge aller Fehler sind. Wie groß die Menge der unentdeckten Fehler ist und wie groß die Auswirkungen der unentdeckten Fehler sind, kann nicht beantwortet werden. Das bedeutet, dass das Risiko von

5. Die sieben Grundprinzipien sind im vorherigen Abschnitt 1.6 beschrieben.

unentdeckten Fehlern nicht berechnet werden kann. Es bleibt im Bereich von Vermutungen und Spekulationen.

Ein anderes Zitat, das die Grenzen des Testens aufzeigt, stammt von Dr. Boris Beizer: *»Good testing works best on good code and good design. And no testing technique can ever change garbage into gold.«* Sinngemäß übersetzt bedeutet dies, dass gute Tests erfolgversprechender für ein System mit einem ausgereiftem Design und einer guten Implementierung (sprich Programmierung) angewandt werden können. Es ist nicht möglich, mit irgendwelchen Tests aus Müll eine perfekte Lösung zu erreichen.

Boris Beizer ist amerikanischer Informatiker und Autor. Seine Publikationen drehen sich alle um das Thema Qualität von Informatiksystemen und speziell um das Testen. Tests können auch konzeptionelle Fehler oder eine qualitativ schlechte Umsetzung aufzeigen. Insbesondere dann, wenn nicht nur funktionale Prüfungen geplant sind, sondern die Qualitätsmerkmale nach ISO 9126 für Testfälle angewandt werden. Es ist üblicherweise nicht mehr möglich, aus einem völlig verkorksten System durch das Korrigieren von Fehlern eine Lösung mit genügender Qualität zu erreichen. Manchmal bleibt nichts anderes übrig, als das System in ungenügender Qualität einzuführen und in einem Folgeprojekt mit Release 2.0 die konzeptionellen Fehler zu korrigieren. Dieses neue System muss als zusätzliche Anforderung noch die Migration des alten Systems erfüllen können. Dies ist nicht selten eines der größten Risiken des Nachfolgerelease. Die logische Schlussfolgerung ist, dass bereits zu Beginn eines Projekts entsprechende qualitative Prüfungen einzubauen sind, um konzeptionelle und Systemdesignfehler zu vermeiden.

In den meisten Business-Intelligence-Projekten wird nicht programmiert. ETL-Prozeduren, Abfragen und Auswertungen werden über grafische Oberflächen parametrisiert. Übertragen auf Business-Intelligence-Systeme bedeutet dies nicht, dass Testen eine untergeordnete Bedeutung hat, weil kein Code geschrieben wird. Genauso sind qualitative Prüfungen von Anfang an in ein Projekt einzubauen. Dabei haben die Datenflüsse und die zu erwartende Datenqualität eine wichtige Bedeutung. Gerade die Datenqualität der Quellsysteme bedeutet häufig eine Limitierung in Business-Intelligence-Projekten. Da die Datenqualität nicht direkt verbessert werden kann, muss zuerst das Bewusstsein dafür geschaffen werden. In den Auswertungen bleibt immer eine gewisse Unschärfe. Einzelne Analysen werden sogar unmöglich sein. Fehler aufgrund der ungenügenden Datenqualität werden jedoch gerne dem Analysewerkzeug zugeordnet. Der Übermittler der schlechten Nachricht wird somit auch hier zum Schuldigen gemacht. Vereinzelt ist es schon vorgekommen, dass Business-Intelligence-Systeme nicht verwendet und später abgestellt wurden, weil man den generierten Informationen nicht traute. Es ist immer wieder interessant zu beobachten, wie eine neue Initiative mit einem anderen Auswertungswerkzeug gestartet wird und wie groß das Erstaunen ist, dass die Datenqualität immer noch nicht besser ist und die Aussagekraft von Auswertungen und Analysen ungenügend bleibt.