# Software Engineering

10., aktualisierte Auflage

Ian Sommerville

Pearson

EXTRAS ONLINE

# Software Engineering

10., aktualisierte Auflage

Ian Sommerville

Pearson

- *Schnittstellenmissverständnis:* Eine aufrufende Komponente versteht die Schnittstelle der aufgerufenen Schnittstelle falsch und setzt ein bestimmtes Verhalten der aufgerufenen Komponente voraus. Die aufgerufene Komponente verhält sich anders als erwartet, was wiederum zu unerwartetem Verhalten der aufrufenden Komponente führt. Zum Beispiel könnte eine binäre Suchmethode mit einem Parameter aufgerufen werden, der ein ungeordnetes Feld ist. Dann wird diese Suche fehlschlagen.
- *Zeitabstimmungsfehler:* Diese treten in Echtzeitsystemen auf, die einen gemeinsamen Speicher oder eine Nachrichtenübergabeschnittstelle verwenden. Produzent und Konsument der Daten arbeiten möglicherweise mit unterschiedlicher Geschwindigkeit. Wenn nicht besondere Sorgfalt auf den Schnittstellenentwurf verwendet wird, kann der Konsument auf veraltete Informationen zugreifen, weil der Produzent der Informationen die gemeinsamen Schnittstelleninformationen nicht aktualisiert hat.

Das Testen auf Schnittstellenfehler ist schwierig, weil einige dieser Fehler eventuell nur unter ungewöhnlichen Umständen auftreten. Nehmen wir an, ein Objekt implementiert eine Warteschlange als eine Datenstruktur mit fester Länge. Ein aufrufendes Objekt könnte nun annehmen, die Warteschlange würde als Datenstruktur beliebiger Länge implementiert, und prüft daher bei der Einstellung eines Artikels nicht auf einen Warteschlangenüberlauf.
Dieser Umstand kann während der Tests nur entdeckt werden, indem man eine Folge von Testfällen entwirft, die einen Überlauf der Warteschlange hervorrufen. Die Tests sollten überprüfen, wie aufrufende Objekte diesen Überlauf behandeln. Da dies selten vorkommt, könnten die Tester denken, dass sich hier das Prüfen nicht lohnt, wenn sie die Testmenge für das Warteschlangenobjekt entwerfen.
Ein weiteres Problem kann aufgrund von Wechselwirkungen zwischen verschiedenen Fehlern in den einzelnen Modulen oder Objekten auftreten. Es kann passieren, dass Fehler in dem einen Objekt nur entdeckt werden, wenn sich ein anderes Objekt auf eine unerwartete Weise verhält. Angenommen ein Objekt ruft ein anderes Objekt auf, um einen Dienst zu erhalten, und das aufrufende Objekt nimmt an, dass die erhaltene Antwort richtig ist. Falls der aufgerufene Dienst auf irgendeine Weise fehlerhaft ist, könnte der zurückgegebene Wert zwar gültig, aber falsch sein. Das Problem ist daher nicht sofort auffindbar, sondern wird erst offensichtlich, wenn eine spätere Weiterverarbeitung, die den zurückgegebenen Wert benutzt, fehlschlägt.
Einige allgemeine Richtlinien für Schnittstellentests sind:

1. Untersuchen Sie den zu testenden Code und identifizieren Sie alle Aufrufe an externe Komponenten. Entwerfen Sie eine Testreihe, bei der die Werte der an die externen Komponenten übergebenen Parameter an den äußersten Enden ihrer Gültigkeitsbereiche liegen. Diese extremen Parameterwerte werden am ehesten Inkonsistenzen der Schnittstellen aufdecken.

2. Wenn über eine Schnittstelle Zeiger übergeben werden, testen Sie die Schnittstelle stets auch mit Null-Zeigern als Parameter.

3. Für eine durch eine Prozedurschnittstelle aufzurufende Komponente sollten Sie Tests entwerfen, die absichtlich zum Versagen der Komponente führen. Unterschiedliche Annahmen über Ausfälle gehören zu den häufigsten Missverständnissen der Spezifikation.

4 Verwenden Sie bei Nachrichtenaustauschsystemen Lasttests. Dies bedeutet, dass Sie Tests entwerfen sollten, die ein Vielfaches der in der Praxis wahrscheinlich auftretenden Nachrichten hervorrufen. Dies ist eine effiziente Möglichkeit, Synchronisationsprobleme aufzudecken.

5 Wenn verschiedene Komponenten über gemeinsamen Speicher zusammenarbeiten, dann entwerfen Sie Tests, die die Reihenfolge bei der Aktivierung der Komponenten variieren. Diese Tests können irrige implizite Annahmen des Programmierers über die Reihenfolge der Datenerzeugung und -verarbeitung aufdecken.

Manchmal ist es besser, Inspektionen und Reviews statt Tests für die Suche nach Schnittstellenfehlern zu nutzen. Inspektionen können sich gezielt den Komponentenschnittstellen widmen und Fragen über das angenommene Verhalten von Schnittstellen aufwerfen.

### 8.1.4 Testen von Systemen

Beim Testen des Systems während der Entwicklung werden die Komponenten integriert, die eine Version des Systems erzeugen, anschließend wird das integrierte System getestet. Systemtests überprüfen, ob Komponenten miteinander kompatibel sind, korrekt miteinander interagieren und die richtigen Daten zur richtigen Zeit über ihre Schnittstellen übermittelt. Systemtests überlappen sich offensichtlich mit Komponententests, doch es gibt zwei wichtige Unterschiede:

1 Während eines Systemtestens können wiederverwendbare Komponenten, die separat entwickelt wurden, sowie handelsübliche Systeme mit den neu entwickelten Komponenten integriert werden. Das vollständige System ist dann getestet.

2 In dieser Phase können Komponenten, die von unterschiedlichen Teammitgliedern oder Teilteams entwickelt wurden, integriert werden. Das Testen von Systemen ist eher ein kollektiver als ein individueller Prozess. In einigen Unternehmen werden Systemtests von einem separaten Testteam durchgeführt, ohne Entwickler und Programmierer einzubeziehen.

Alle Systeme besitzen **emergentes Verhalten**. Dies bedeutet, einige Systemfunktionen und Eigenschaften werden nur sichtbar, wenn die Komponenten zusammengefügt werden. Dabei kann es sich um geplantes emergentes Verhalten handeln, das getestet werden soll. Zum Beispiel könnten Sie eine Authentifizierungskomponente mit einer Komponente integrieren wollen, die die Systemdatenbank aktualisiert. Dadurch erhalten Sie eine Systemfunktion, die Informationenaktualisierung auf autorisierte Benutzer beschränkt. Manchmal jedoch ist emergentes Verhalten ungeplant und unerwünscht. Sie müssen Tests entwickeln, die überprüfen, dass das System nur das tut, was es tun soll.
Systemtests sollten sich darauf konzentrieren, das Zusammenspiel zwischen Komponenten und Objekten zu testen, aus denen das System aufgebaut ist. Auch wiederverwendete Komponenten oder Systeme könnten hier getestet werden, um zu überprüfen, ob sie bei Integration mit neuen Komponenten wie erwartet funktionieren. Das Testen dieser Interaktionen sollte solche Komponentenfehler aufdecken, die nur auftreten, wenn eine Komponente von einer anderen Komponente im System benutzt wird.

Dadurch können außerdem solche Missverständnisse gefunden werden, denen Komponentenentwickler in Bezug auf andere Komponenten im System erliegen.
Ein effektiver Ansatz für Systemtests ist anwendungsfallbasiertes Testen, da hier der Fokus auf Interaktionen liegt. Jeder Anwendungsfall wird typischerweise von mehreren Komponenten oder Objekten im System implementiert. Beim Testen dieses Anwendungsfalls werden diese Interaktionen notwendigerweise auftreten. Falls Sie ein Sequenzdiagramm erstellt haben, um die Implementierung des Anwendungsfalls zu modellieren, können Sie daran die Objekte oder Komponenten ablesen, die an der Interaktion beteiligt sind.
Im Beispiel der Wildnis-Wetterstation sendet die Systemsoftware zusammengefasste Wetterdaten an einen entfernten Computer, wie in *Abbildung 7.3* zu sehen. ▶ Abbildung 8.8 zeigt die Abfolge der Operationen der Wetterstation bei der Ausführung einer Anfrage zum Sammeln von Daten für das Wetterberichtssystem. Sie können dieses Diagramm verwenden, um die zu testenden Operationen festzustellen und die zur Durchführung der Tests benötigten Testfälle zu entwerfen. Das Abschicken einer Berichtsanfrage wird daher die Ausführung der folgenden Abfolge von Methoden nach sich ziehen:

```
SatKomm:anfordern(Bericht) → Wetterstation:meldeWetter → KommVerbin-
dung:hole(Zusammenfassung) → Wetterdaten:zusammenfassen
```

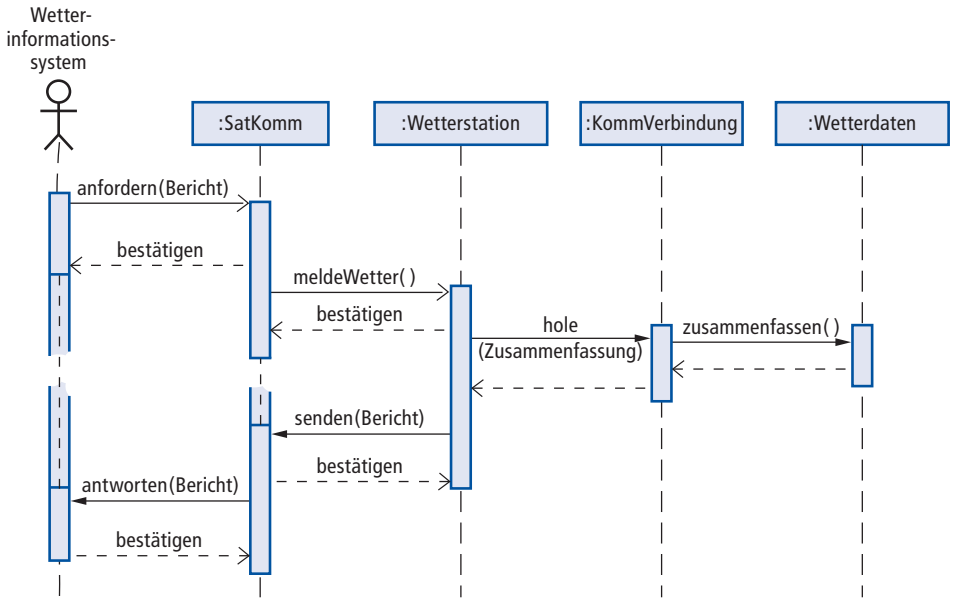


Abbildung 8.8: Sequenzdiagramm zur Sammlung von Wetterdaten.

Das Sequenzdiagramm hilft Ihnen, die spezifischen Testfälle zu entwerfen, die Sie benötigen, sobald sich herausstellt, welche Eingaben benötigt werden und welche Ausgaben erzeugt werden:

- Die Eingabe einer Berichtsanfrage sollte eine entsprechende Bestätigung auslösen. Am Ende sollte ein Bericht zurückgegeben werden. Während des Tests sollten Sie

zusammengefasste Daten erzeugen, die verwendet werden können, um zu prüfen, ob der Bericht richtig aufgebaut ist.

- Der Eingang einer Berichtsanfrage beim Objekt `Wetterstation` zieht die Erstellung eines zusammenfassenden Berichts nach sich. Dies können Sie isoliert testen, indem Sie Rohdaten nehmen, die der Gesamtübersicht entsprechen, welche Sie für den Test von `SatKomm` vorbereitet haben, und prüfen, ob das Objekt `Wetterstation` diese Zusammenfassung richtig erstellt. Diese Rohdaten werden ebenfalls zum Testen des Objekts `Wetterdaten` benutzt.

Natürlich habe ich das Sequenzdiagramm in Abbildung 8.8 so vereinfacht, dass es keine Ausnahmen zeigt. Ein vollständiger Test eines Anwendungsfalls bzw. Szenarios muss diese Ausnahmen berücksichtigen und sicherstellen, dass diese richtig behandelt werden.

Bei den meisten Systemen ist es schwierig festzulegen, wie viele Systemtests nötig sind und wann man mit dem Testen aufhören sollte. Wirklich vollständige Tests, bei denen jeder denkbare Ablauf der Ausführung eines Programms getestet wird, sind praktisch undurchführbar. Das Testen muss daher auf einer Untermenge der möglichen Testfälle beruhen. Softwarefirmen sollten im Idealfall Richtlinien zur Auswahl dieser Untermenge entwickeln. Die Richtlinien können auf allgemeinen Testrichtlinien beruhen, wie zum Beispiel, dass alle Programmanweisungen mindestens einmal ausgeführt werden müssen. Andererseits können sie auch auf Erfahrungen beim Einsatz des Systems basieren und sich auf das Testen der Funktionalität des betriebenen Systems konzentrieren. Zum Beispiel:

- Alle über Menüs erreichbaren Systemfunktionen sollten getestet werden.
- Kombinationen von Funktionen (z. B. zur Textformatierung), auf die über dasselbe Menü zugegriffen wird, müssen getestet werden.
- Wenn Benutzereingaben eine Rolle spielen, müssen alle Funktionen sowohl mit richtigen als auch mit falschen Eingaben getestet werden.

Erfahrungen mit weitverbreiteten Softwareprodukten wie Textverarbeitungen und Tabellenkalkulationen zeigen, dass bei ihnen vergleichbare Testrichtlinien angewendet wurden. Wenn Softwarefunktionen isoliert voneinander benutzt werden, arbeiten sie normalerweise korrekt. Probleme tauchen nach Whittaker (2002) dann auf, wenn Kombinationen von weniger häufig benutzten Funktionen nicht zusammen getestet wurden. Er nennt das Beispiel von Fußnoten, die, wenn sie in einem verbreitet eingesetzten Textverarbeitungsprogramm mit einem mehrspaltigen Layout kombiniert werden, zu einem fehlerhaften Textlayout führen.

Das automatisierte Testen von Systemen ist in der Regel schwieriger als die Automatisierung von Modul- oder Komponententests. Automatisierte Modultests beruhen darauf, die Ausgaben vorherzusagen und dann diese Vorhersagen in einem Programm zu codieren. Die Vorhersage wird dann mit dem Ergebnis verglichen. Es kann jedoch auch sein, dass ein System Ausgaben erzeugt, die umfangreich sind oder die nicht leicht vorhergesagt werden können. Man kann dann diese Ausgaben untersuchen und ihre Plausibilität überprüfen, auch wenn man sie im Voraus nicht unbedingt erzeugen konnte.

## Inkrementelle Integrationstests

Zu einem Systemtest gehört das Zusammenführen der unterschiedlichen Komponenten und das anschließende Testen des so erzeugten integrierten Systems. Sie sollten stets einen inkrementellen Ansatz der Systemintegration und der Integrationstests verfolgen, bei dem Sie eine Komponente integrieren, das System testen, eine weitere Komponente integrieren, wieder testen und so weiter. Falls Probleme auftreten, dann liegt dies wahrscheinlich an dem Zusammenspiel mit der zuletzt hinzugefügten Komponente.

Inkrementelle Integrationstests sind grundlegend für agile Methoden, wo Regressionstests jedes Mal ausgeführt werden, wenn ein neues Inkrement hinzukommt.

*http://software-engineering-book.com/web/integration/*

## 8.2 Testgetriebene Entwicklung

Testgetriebene Entwicklung (TDD, *Test-Driven Development*) ist ein Ansatz der Programmentwicklung, bei dem Testen und Codeentwicklung ineinander greifen (Beck, 2002; Jeffries und Melnik, 2007). Hier greifen die inkrementelle Codeentwicklung zusammen mit einer Reihe von Tests für dieses Inkrement. Die Arbeit am nächsten Inkrement wird nicht eher begonnen, bis der vorher entwickelte Code alle seine Tests bestanden hat. Testgetriebene Entwicklung wurde als Teil der agilen XP-Entwicklungsmethoden eingeführt. Heute hat sie jedoch breite Akzeptanz erreicht und kann ebenso für agile als auch für planbasierte Prozesse eingesetzt werden.
Die grundlegenden TDD-Prozesse sind in ▶ Abbildung 8.9 dargestellt. Folgende Schritte werden dabei ausgeführt:

1. Sie beginnen mit der Identifizierung der geforderten Funktionalität. Diese sollte normalerweise klein und in ein paar Codezeilen implementierbar sein.
2. Sie schreiben einen Test für diese Funktionalität und implementieren diese als einen automatisierten Test. Dies bedeutet, dass der Test ausgeführt werden kann und protokolliert wird, ob er bestanden ist oder nicht.
3. Dann führen Sie den Test zusammen mit allen anderen implementierten Tests aus. Da anfangs die Funktionalität noch nicht implementiert ist, wird der neue Test zunächst scheitern. Dies ist Absicht, denn es zeigt, dass der Test etwas zu der Testmenge hinzufügt.
4. Dann implementieren Sie die Funktionalität und führen den Test noch einmal aus. Dazu gehört eventuell die Überarbeitung von vorhandenem Code, um ihn zu verbessern, sowie die Ergänzung durch neuen Code.
5. Wenn alle Tests erfolgreich laufen, gehen Sie dazu über, den nächsten Funktionalitätsbaustein zu implementieren.

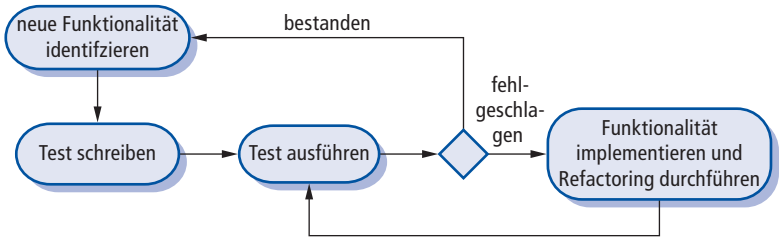


Abbildung 8.9: Testgetriebene Entwicklung.

Eine automatisierte Testumgebung wie die JUnit-Umgebung, die das Testen von Java-Programmen unterstützt (Tahchiev et al., 2010), ist für TDD sehr wichtig. Da der Code in sehr kleinen Inkrementen entwickelt wird, müssen Sie in der Lage sein, alle Tests jedes Mal auszuführen, wenn Sie Funktionalität hinzufügen oder das Programm überarbeiten. Die Tests sind daher in einem separaten Programm eingebettet, das die Tests ausführt und das zu testende System aufruft. Dieser Ansatz ermöglicht es, Hunderte von getrennten Tests in ein paar Sekunden auszuführen.

Testgetriebene Entwicklung unterstützt Programmierer darin, sich zu verdeutlichen, was ein Codesegment tatsächlich tun soll. Wenn Sie einen Test schreiben wollen, dann müssen Sie verstehen, was mit dem Code beabsichtigt ist. Dieses Verständnis vereinfacht das Schreiben des geforderten Codes. Es ist einleuchtend, dass bei unvollständigem Wissen oder Verständnis TDD nicht das richtige Verfahren ist.

Wenn Ihr Wissen nicht ausreicht, um die Tests zu entwickeln, so werden Sie auch nicht den erforderlichen Code schreiben können. Enthält Ihre Berechnung beispielsweise eine Division, dann sollten Sie sicherstellen, dass keine Zahl durch 0 dividiert wird. Wenn Sie vergessen, einen Test dafür zu schreiben, dann wird auch der Prüfcode niemals in das Programm aufgenommen werden.

Neben dem besseren Problemverständnis hat testgetriebene Entwicklung noch andere Vorteile:

- *Codeabdeckung:* Im Prinzip sollte jedes Codesegment, das Sie schreiben, mindestens einen zugehörigen Test haben. Dann können Sie sicher sein, dass der gesamte Code im System tatsächlich ausgeführt wurde. Der Code wird getestet, sobald er geschrieben wurde, daher werden Fehler früh im Entwicklungsprozess entdeckt.
- *Regressionstests:* Eine Testsuite wird wie ein Programm inkrementell entwickelt. Sie können Regressionstests immer ausführen, um zu überprüfen, dass Änderungen am Programm keine neuen Fehler eingeführt haben.
- *Vereinfachtes Fehlerbeheben:* Wenn ein Test erfolglos ist, sollte es offensichtlich sein, wo das Problem liegt. Der neu geschriebene Code muss überprüft und modifiziert werden. Sie müssen keine Debugger benutzen, um das Problem zu lokalisieren. Erfahrungsberichte zur testgetriebenen Entwicklung lassen den Schluss zu, dass es hier so gut wie nie nötig ist, einen automatischen Debugger zu verwenden (Martin, 2007).
- *Systemdokumentation:* Die Tests selbst stellen eine Art von Dokumentation dar, in der beschrieben wird, was der Code machen sollte. Das Nachvollziehen der Tests kann es vereinfachen, den Code zu verstehen.

Einer der wichtigsten Vorteile von TDD ist die Kostenreduktion von Regressionstests. Bei einem Regressionstest werden Testmengen, die bereits erfolgreich ausgeführt wur-

den, erneut ausgeführt, nachdem Änderungen am System vorgenommen wurden. Damit wird geprüft, dass diese Änderungen keine neuen Fehler in das System eingeführt haben und dass der neue Code wie erwartet mit dem vorhandenen Code interagiert. Regressionstests sind sehr teuer und bei manuellem Testen manchmal undurchführbar, da sie viel Zeit- und Arbeitsaufwand in Anspruch nehmen. Sie müssen versuchen, die wesentlichsten Tests zur erneuten Ausführung auszuwählen, und es kann leicht passieren, dass wichtige Tests ausgelassen werden.

Automatisiertes Testen reduziert die Kosten des Regressionstestens dramatisch. Vorhandene Tests können schnell und billig neu ausgeführt werden. Bei der testgetriebenen Entwicklung müssen nach einer Änderung an einem System alle vorhandenen Tests erfolgreich durchgeführt werden, bevor irgendeine weitere Funktionalität hinzugefügt wird. Als Programmierer können Sie sicher sein, dass die neue Funktionalität, die Sie hinzugefügt haben, keine Probleme mit dem existierenden Code verursacht oder aufgedeckt hat.

Den größten Nutzen bietet testgetriebene Entwicklung bei der Entwicklung von neuer Software, bei der die Funktionalität entweder in neuem Code implementiert wird oder bei der Komponenten aus Standardbibliotheken verwendet werden. Falls Sie jedoch große Codekomponenten oder Altsysteme wiederverwenden, dann müssen Sie Tests für diese Systeme als Ganzes schreiben. Sie können diese Teile nicht einfach in separate testbare Elemente zerlegen. Inkrementelle testgetriebene Entwicklung ist unpraktisch. Testgetriebene Entwicklung kann außerdem bei Systemen, die Multithreading verwenden, ineffektiv sein. Die verschiedenen Threads könnten zu verschiedenen Zeiten in unterschiedlichen Testläufen verschachtelt sein und so eventuell unterschiedliche Ergebnisse produzieren.

Wenn sie TDD einsetzen, so benötigen Sie dennoch einen Systemtestprozess, um das System zu validieren; das heißt, Sie müssen prüfen, dass das System die Anforderungen aller Systembeteiligten erfüllt. Systemtests testen außerdem Leistungsfähigkeit und Zuverlässigkeit und stellen fest, dass das System nichts tut, was es nicht tun sollte, wie zum Beispiel unerwünschte Ausgaben produzieren. Andrea (2007) schlägt mögliche Erweiterungen von Testwerkzeugen vor, um einige Aspekte des Systemtestens mit TDD zu integrieren.

Testgetriebene Entwicklung ist heute ein weitverbreiteter und allgemeiner Ansatz zum Testen von Software. Die meisten Programmierer, die diesen Ansatz verfolgen, sind damit zufrieden und halten es für einen produktiveren Weg der Softwareentwicklung. Es wird auch behauptet, dass der Einsatz von TDD die bessere Strukturierung eines Programms fördert und die Codequalität steigert. Experimente, die diese Behauptung belegen, waren jedoch uneindeutig.

## 8.3 Freigabetests

Freigabetest (*release testing*) wird der Testprozess einer speziellen Systemversion genannt, die für die Benutzung außerhalb des Entwicklerteams bestimmt ist. Normalerweise ist das Systemrelease für Kunden und Benutzer gedacht. In einem komplexen Projekt könnte das Release jedoch auch für andere Teams sein, die damit verbundene Systeme entwickeln. Bei Softwareprodukten könnte das Release für das Produktmanagement sein, um die Software für den Verkauf vorzubereiten.

Es gibt zwei wichtige Unterscheidungen zwischen Freigabetests und Systemtests während des Entwicklungsprozesses:

1. Das Team, das die Systementwicklung vornimmt, sollte nicht für Freigabetests verantwortlich sein.

2. Freigabetests sind ein Mittel zur Gültigkeitsprüfung, mit denen sichergestellt wird, dass ein System seine Anforderungen erfüllt und gut genug für die Anwendung durch die Kunden des Systems ist. Systemtests, die vom Entwicklerteam durchgeführt werden, sollten sich auf die Entdeckung von Fehlern im System konzentrieren (Fehlertests).

Das Hauptziel des Freigabetestens ist, den Anbieter des Systems davon zu überzeugen, dass das System gut genug zur Benutzung ist. Gelingt dies, kann es als Produkt veröffentlicht oder an den Kunden ausgeliefert werden. Freigabetests müssen daher zeigen, dass das System die spezifizierte Funktionalität, Leistung und Verlässlichkeit liefert und dass es im Normalbetrieb nicht versagt.
Bei Freigabetests handelt es sich normalerweise um einen Blackbox-Testvorgang, dessen Tests aus der Systemspezifikation abgeleitet werden. Das System wird als eine Blackbox behandelt, deren Verhalten nur durch die Untersuchung ihrer Eingaben und der dazugehörigen Ausgaben festgestellt werden kann. Eine andere Bezeichnung dafür ist *funktionales Testen*, so genannt, weil der Tester sich nur mit der Funktionalität und nicht mit der Implementierung der Software beschäftigt.

### 8.3.1 Anforderungsbasiertes Testen

Ein allgemeiner Grundsatz der Ausgestaltung von guten Anforderungen ist, dass Anforderungen testfähig sein sollten. Das heißt, eine Anforderung sollte so verfasst werden, dass ein Beobachter mithilfe eines dafür entworfenen Tests überprüfen kann, ob sie erfüllt wurde. Bei anforderungsbasiertem Testen handelt es sich deshalb um eine systematische Herangehensweise an den Testentwurf, in der Sie jede Anforderung in Betracht ziehen und eine Testreihe für sie gestalten. Anforderungsbasiertes Testen ist also eher ein Validierungs- als ein Fehlertest – Sie versuchen ja zu zeigen, dass das System seine Anforderungen ordnungsgemäß implementiert hat.
Betrachten Sie z. B. die folgenden Systemanforderungen für das Mentcare-System, die sich mit dem Überprüfen von Arzneimittelallergien befassen:

> *Falls ein Patient nicht weiß, dass er allergisch auf bestimmte Arzneimittel ist, dann sollte eine Verschreibung dieses Medikaments eine Warnmeldung an den Systembenutzer auslösen.*
>
> *Falls die Person, die das Rezept ausstellt, sich entschließt, eine Allergiewarnung zu ignorieren, dann sollte sie einen Grund dafür angeben.*

Um festzustellen, ob diese Anforderungen erfüllt wurden, müssen unter Umständen mehrere miteinander verknüpfte Tests entwickelt werden:

1. Richte einen Patientendatensatz ohne bekannte Allergien ein. Verschreibe Medikamente für bekannte Allergien. Überprüfe, dass keine Warnmeldung vom System ausgegeben wird.

2. Richte einen Patientendatensatz mit einer bekannten Allergie ein. Verschreibe dem Patienten die Arzneimittel, auf die der Patient allergisch reagiert und überprüfe, dass die Warnung vom System ausgegeben wird.

3. Richte einen Patientendatensatz ein, bei dem Allergien auf zwei oder mehr Arzneimittel erfasst sind. Verschreibe beide Medikamente getrennt voneinander und überprüfe, dass die richtige Warnung für jedes Medikament ausgegeben wird.

4. Verschreibe zwei Medikamente, auf die der Patient allergisch reagiert. Überprüfe, dass zwei Warnungen korrekt ausgegeben werden.

5. Verschreibe ein Medikament, dass eine Warnung ausgibt, und ignoriere diese Warnung. Überprüfe, dass das System vom Benutzer eine Begründung verlangt, warum die Warnung nicht beachtet wurde.

Aus dieser Aufzählung können Sie ersehen, dass zum Testen einer Anforderung nicht nur ein einziger Test gehört. Normalerweise müssen Sie mehrere Tests schreiben, um sicherzustellen, dass Sie die Anforderung komplett abdecken. Außerdem sollten Sie Aufzeichnungen Ihrer anforderungsbasierten Tests führen, wodurch Sie die Tests mit den konkreten Anforderungen in Verbindung setzen, die Sie getestet haben.

### 8.3.2 Szenariobasiertes Testen

Szenariobasiertes Testen ist eine Methode des Freigabetestens, bei der Sie typische Benutzerszenarios erfinden und diese verwenden, um Testfälle für das System zu entwickeln. Ein Szenario ist eine Geschichte, die eine Möglichkeit beschreibt, das System zu benutzen. Szenarios sollten realistisch sein und die tatsächlichen Systemnutzer sollten sich darin wiederfinden können. Falls Sie Szenarios oder User-Storys als Teil des Requirements-Engineering-Prozesses verwendet haben (*Kapitel 4*), dann können Sie diese möglicherweise als Testszenarios wiederverwenden.

In einem kurzen Aufsatz über szenariobasiertes Testen schlägt Kaner (2003) vor, dass ein Testszenario wie eine Geschichte aufgebaut sein sollte, die glaubwürdig und einigermaßen komplex ist. Die Projektbeteiligten sollen dadurch motiviert werden; das heißt, sie sollten eine Beziehung zu diesem Szenario haben und es für wichtig erachten, dass das System den Test besteht. Kaner fordert darüber hinaus, dass der Test einfach auszuwerten sein soll. Falls es Probleme mit dem System gibt, dann sollte das Team, das die Freigabetests durchführt, diese erkennen.

Als Beispiel eines möglichen Szenarios für das Mentcare-System beschreibt ▶ Abbildung 8.10 eine Möglichkeit, wie das System für Planung und Durchführung von Hausbesuchen verwendet werden kann.

Katja Müller ist eine Krankenschwester, die sich im Bereich psychiatrische Ambulanz spezialisiert hat. Eine ihrer Aufgabengebiete ist, Patienten zu Hause zu besuchen, um zu kontrollieren, ob deren Behandlung effektiv ist und dass sie nicht an Nebenwirkungen der Medikamente leiden.
An einem Tag für Hausbesuche meldet sich Frau Müller beim Mentcare-System an und druckt ihren Terminplan für die Hausbesuche dieses Tages zusammen mit zusammengefassten Informationen über die zu besuchenden Patienten aus. Sie gibt ein, dass sie die Datensätze dieser Patienten auf ihren Laptop laden möchte. Sie wird aufgefordert, ihr Passphrase einzugeben, um die Datensätze auf dem Laptop zu verschlüsseln.
Einer der Patienten, den sie besucht, ist Herr Schmidt, der mit Medikamenten gegen Depression behandelt wird. Herr Schmidt hat den Eindruck, dass die Medizin ihm hilft, aber er glaubt, dass sie den Nebeneffekt hat, ihn nachts wach zu halten. Frau Müller ruft den Datensatz von Herrn Schmidt auf und wird aufgefordert, ihre Passphrase zum Entschlüsseln des Datensatzes einzugeben. Sie überprüft das verschriebene Medikament und fragt die Nebenwirkungen ab. Schlaflosigkeit ist eine bekannte Nebenwirkung, daher notiert sie das Problem in dem Datensatz und schlägt vor, dass Herr Schmidt die Sprechstunde besucht, um seine Medikamente ändern zu lassen. Er ist einverstanden, deshalb gibt Frau Müller eine Notiz ein, die sie daran erinnert ihn anzurufen, wenn sie zurück in der Ambulanz ist, um einen Termin mit einem Arzt zu machen. Sie beendet die Beratung und das System verschlüsselt den Datensatz von Herrn Schmidt wieder.
Nachdem alle Beratungen abgeschlossen sind, kehrt Katja Müller in die Ambulanz zurück und lädt die Datensätze der besuchten Patienten in die Datenbank. Das System erzeugt eine Anrufliste für Frau Müller von denjenigen Patienten, die sie für Folgeinformationen kontaktieren und Sprechstundentermine machen muss.

Abbildung 8.10: Eine User-Story für das Mentcare-System.

Dieses Szenario testet eine Anzahl von Funktionen des Mentcare-Systems:

1. Authentifizierung bei der Anmeldung im System;
2. ausgewählte Patientendatensätze auf bzw. von einem Laptop laden;
3. Terminplan für Hausbesuche;
4. Patientendatensätzen auf einem mobilen Gerät ver- und entschlüsseln;
5. Datensätze abrufen und verändern;
6. Verbindung zur Medikamentendatenbank, die Informationen über Nebenwirkungen enthält;
7. das System zur Anrufsaufforderung.

Wenn Sie für Freigabetests zuständig sind, dann spielen Sie dieses Szenario in der Rolle von Katja Müller durch und beobachten, wie sich das System in Reaktion auf unterschiedliche Eingaben verhält. Als „Katja Müller“ machen Sie absichtlich Fehler, wie beispielsweise die falsche Passphrase zur Decodierung von Datensätzen eingeben. Dadurch wird die Reaktion des Systems auf Fehler überprüft. Sie sollten jedes auftretende Problem sorgfältig festhalten, einschließlich der Performanzprobleme. Falls ein System zu langsam ist, dann ändert sich dadurch die Art, wie es benutzt wird. Wenn es zum Beispiel zu lange dauert, einen Datensatz zu verschlüsseln, dann werden Benutzer, die es eilig haben, möglicherweise diesen Schritt überspringen. Wenn sie dann ihren Laptop verlieren, könnte eine nicht autorisierte Person die Patientendatensätze einsehen.
Wenn Sie einen szenariobasierten Ansatz benutzen, testen Sie in der Regel mehrere Anforderungen innerhalb eines Szenarios. Daher überprüfen Sie – außer der Überprü-

# Copyright