Nils Röttger
(nils.roettger@imbus.de)
hat an der Universität in Göttingen Informatik mit dem Schwerpunktthema Softwaretest studiert. Seit 2008 arbeitet er am Hauptsitz der imbus AG in Möhrendorf und hat seitdem als Projektleiter und Testmanager verschiedene Projekte erfolgreich zu Ende geführt. Seit 2013 ist er für den Bereich Mobile Testing verantwortlich.

Michael Heller
(michael.heller@imbus.de)
studierte Informatik an der Friedrich-Alexander-Universität Erlangen-Nürnberg. Er verantwortet als Product Owner die Beratungsmodule von imbus zum agilen Testen, unter anderem für das imbus T5 Assessment Modell zum Testen in Scrum-Projekten. Michael Heller hat 14 Jahre Erfahrung als Kundenberater und Trainer in der IT. Er war 7 Jahre als Product Owner eines Scrum-Projekts mit drei verteilten Entwicklungsstandorten aktiv. Als Test Consultant liegen seine Schwerpunkte auf der Prozessberatung für Testmanagement und agile Entwicklungsformen.

Methoden und Testentwurfsverfahren im agilen App-Test

Seit dem Siegeszug der Smart Devices dreht sich die Uhr weiter. Mobile Apps übernehmen inzwischen Aufgaben, die früher Desktopsystemen vorbehalten waren. Anhand von Beispielen zeigt der Artikel auf, vor welchen Herausforderungen ein App-Test in einem typischen agilen Projekt steht. Mit dem Pairwise Testing von Systemkonfigurationen der Apple Watch und ihren unterschiedlich ausgestatteten Varianten wird eine klassische Methode vorgestellt, die hilft, mit der Gerätevielfalt umzugehen. Von der Shopping App bis hin zur Smartwatch gilt: Der Test sollte die Nase vorne haben, frühzeitig Schwachstellen aufdecken und so das Feedback nicht den Bewertungen im App Store überlassen. Systematische, methodische funktionale Tests und Toolketten für automatisierte Tests bilden die Grundlage. Die Ergänzung des Testvorgehens um weitere Testentwurfsverfahren, wie erfahrungsbasiertes Testen im Takt agiler Entwicklungssprints, ist dann das Salz in der Suppe.

Motivation

„Mobile Geräte laufen stationären PCs den Rang ab." So titelt das Statistische Bundesamt in einer Erhebung 2014 [destatis]. Ebenso aus dem letzten Jahr stammt eine andere Erkenntnis: Erstmals wurde mehr als die Hälfte aller Online-Einkäufe von mobilen Endgeräten aus abgewickelt [hp]. Noch in den Kinderschuhen steckt dagegen die wirtschaftliche Bedeutung von Apps auf Wearables. Aber in den letzten zwei Monaten erlebten wir mehr als ein Flurgespräch mit dem gleichen Tenor: „Ach, Du hast auch eine". Gemeint war die Apple Watch. Jetzt ist das experimentierfreudige Völkchen, das bei Software-QS-Anbietern zu finden ist, sicher keine signifikante Stichprobe der deutschen Gesellschaft. Und vielleicht schaffen es Apps auf Uhren in den nächsten Jahren noch nicht bis zum Massenmarkt. Aber diesem Feld steht sicher ein bedeutsames Wachstum bevor.

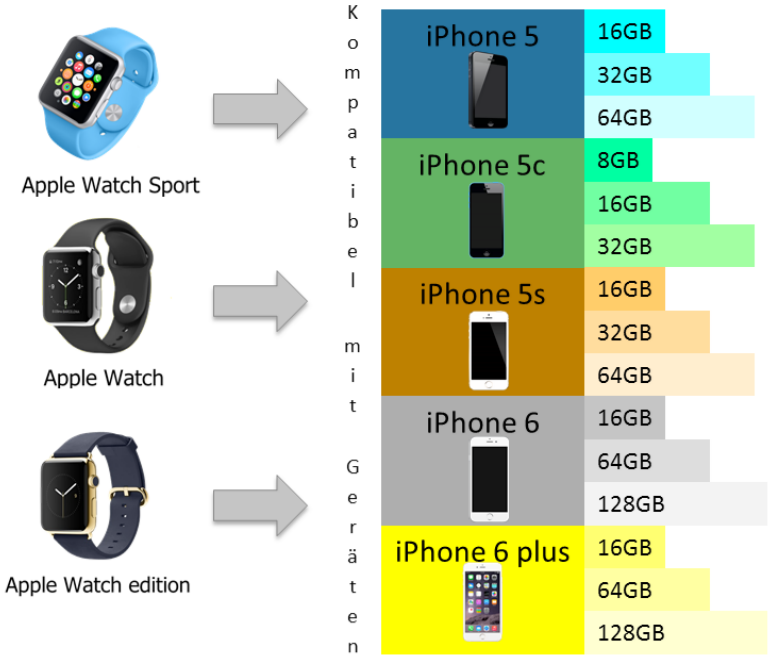

Abb. 1: iPhone-Varianten, die mit einer Apple Watch verbunden werden können

Trotz beschränkter Ressourcen eines Smartphones oder gar Wearables sollen deren Apps zunehmend die Ansprüche erfüllen, die bisher an Desktopsysteme gestellt wurden. Deswegen stehen Testverantwortliche vor zahlreichen Herausforderungen:

- Die Bedienung von Smartwatches unterscheidet sich in wesentlichen Punkten von der eines Smartphones und die Bedienung eines Smartphones unterscheidet sich stark von der eines Desktopsystems.
- Zusätzliche Geräte, die mit dem Smartphone verbunden sind, erweitern die Komplexität eines Tests einer mobilen App („App-Test“) gegenüber dem gleichen Programm auf dem Desktop.
- Mobile Apps werden oft an Orten und in einer Weise verwendet, die für traditionelle Desktopsysteme nicht vorkommen, wie Wandern.
- Es werden immer mehr Apps in den Stores bereitgestellt, die Desktopanwendungen ergänzen oder ersetzen. Solche Apps haben einen umfangreicheren Code als viele der Apps, die in den ersten Jahren in den Stores zu finden waren. Es entstehen zusätzliche Anforderungen an den App-Test, die noch vor Kurzem keine Rolle spielten. Themen wie Datenschutz-, Sicherheits- oder auch Usability-Tests haben in so manchem App-Testkonzept gefehlt, mittlerweile sind sie Standard.
- Einige Stores überprüfen eine App vor der Veröffentlichung auf ein paar harte Regeln. Unter Umständen lässt sich eine App nicht rechtzeitig freigeben, weil einfache Rechtschreibfehler auf dem Startscreen übersehen wurden. Die frühzeitige Überprüfung dieser „Store Guidelines“ sollten Testarchitekten beachten. Sie ist eine Variante des Konformitätstests und stellt eine typische Art des App-Tests dar.
- Hinzu kommt eine Vielzahl von Herstellern mit verschiedenen Geräten und vielen unterschiedlichen Betriebssystemen. Mit jedem weiteren Gerät steigt die Anzahl der Testfälle exponentiell.

Neue Techniken treten häufig in Kombination mit neuen Entwicklungsmethoden auf. So wird eine App oft nach den Vorgaben agiler Softwareentwicklungsprozesse entstehen, versprechen diese doch schnellere „time to market“. Was bedeutet das für die Tests, was ist am agilen Testen anders?

Die Antwort sei schon einmal vorweggenommen: Agiles Testen ist gar nicht so anders. Aber es betont den Wert von häufigem und frühzeitigem Feedback. Und damit passt es gut zur schnelllebigen Welt der App Stores. Der Fokus auf frühes Feedback verschiebt den Wert mancher Testaktivitäten und die Art und Weise, wie man sie aufsetzt.

Im Kern des Artikels steht die Frage: Wie lassen sich „agil“ systematische Tests, automatisierte Tests, Testtools oder auch explorative Tests – die später noch erklärt werden – zu einer Qualitätssicherungslandschaft vereinen? Am Leichtesten lassen sich Zusammenhänge anhand konkreter Beispiele verstehen. Im Folgenden haben wir zu jedem Themengebiet ein Beispiel herausgegriffen.

Systematische Tests

Beginnen wir mit den systematischen Tests. Leider finden sich immer noch Teams und Projekte, die manche der agilen Leitsätze zu wörtlich verstehen. Im agilen Manifest steht, dass „funktionierende Software wichtiger ist als umfassende Dokumentation“ [AM]. Es finden sich schon mal Projekte mit Millionen „Lines of Code“, die ohne Dokumentation und ohne einen einzigen niedergeschriebenen Testfall auskommen. Dabei sind niedergeschriebene Testfälle eine wichtige Grundlage für eine sinnvolle Planung von systematischen Tests. Regressionstests beispielsweise stellen sicher, dass in neuen Versionen alte Features noch funktionieren. Es sollen funktionierende neue Softwarefunktionalitäten im Monats- oder gar Wochentakt geliefert werden. Es muss auch in dieser hohen Frequenz beurteilt werden, welche Tests angepasst, welche wiederholt, welche in den Regressionstestpool neu aufgenommen werden müssen und welche Tests entfallen können.

Hier läuft nicht immer alles glatt. Jeder, der schon mal in App-Store-Bewertungen gestöbert und dann bei einer App über Hunderte von Meldungen der Geschmacksrichtung „Nach dem Update unbrauchbar“ gestolpert ist, kann sich sein eigenes Bild von der Qualität mancher App machen. Vielleicht stammen all diese Meldungen nur von Nutzern einer bestimmten alten Android-Version. Das nutzt dem Hersteller aber wenig, der Ruf ist ruiniert.

Die grundlegenden Methoden, die zu systematischen Tests führen, sind natürlich plattformunabhängig und übertragbar. Im Folgenden zeigen wir ein anschauliches Beispiel.

Methodenbeispiel: Pairwise Testing von Systemkonfigurationen der Apple Watch

Wollte man einen bestimmten Testfall für die beiden verfügbaren Typen der Apple Watch kombiniert mit allen verfügbaren iPhone-Varianten auf jeder Version von iOS, die sich beim Endanwender finden kann, durchführen, stellt sich als erste Frage: Wie viele Varianten gibt es denn überhaupt?

Für den Test können wir davon ausgehen, dass Gold als Material des Gehäuses keine Rolle spielt. Die Größe der Watch dagegen schon, mit 390px x 312px (42 mm) und 340px x 272px (38 mm) bieten die Displays verschiedene Auflösungen. iPhones vor der Versionsnummer 5 lassen sich nicht mit der Watch koppeln, sie fallen also weg. Die reine Zählung von Hardware-Varianten ab iPhone 5 würde 15 Geräte ergeben (siehe Abbildung 1).

Multipliziert mit zwei Display-Varianten der Uhr und vier verschiedenen Ständen von iOS sind wir bei 120 Systemkonfigurationen, die man testen könnte. Systematisch zu testen bedeutet aber nicht, alles Denkbare zu testen. Wie wahrscheinlich ist es, dass ein Testfall auf dem iPhone 5 mit 16 GB Ausstattung und mit 64 GB Speicher funktioniert, mit 32 GB aber ein Fehler auftritt?

Hier entscheiden wir uns in Anlehnung an die Grenzwertmethode dafür, lieber nur die minimalen und die maximalen Speicherausstattungen in den Test einzubeziehen. Die Annahme über das wahrscheinliche Auftreten eines Fehlers kann richtig

Apple Watch	iPhone-Version	iPhone-Kapazität	iOS
42mm	iPhone 5	minimal	8.2
38mm	iPhone 5s	maximal	8.3
	iPhone 5c		8.4
	iPhone 6		9.0 Beta
	iPhone 6 Plus		

Tabelle 1: Systemkonfigurationen mit Apple Watch und iPhone

oder falsch sein. Sie ist trotzdem typisch für professionelles und systematisches Testen. Hinter der Entscheidung, ob ein Test durchgeführt werden sollte, muss immer eine Risikoeinschätzung stehen. Auf je mehr Fachwissen ein Tester zurückgreifen kann, umso eher findet er wertvolle Tests. Hier liegt einer der Vorteile im agilen Entwickeln: Durch die enge Zusammenarbeit im Team zwischen Fachbereich, Entwicklern und Testern steht auch die Risikoanalyse oft auf einer breiteren Basis.

Auch mit den ersten Einschränkungen bei den Varianten getesteter iPhone-Kapazität in Tabelle 1 ergibt sich immer noch eine große Kombinatorik von 80 Testfällen.

Für die zu testenden iOS-Varianten hätten wir natürlich wie bei der Kapazität auch auf Zwischenversionen verzichten können, um weniger testen zu müssen. Aber wir gehen hier davon aus, dass Betriebssystemabhängigkeiten häufigere Fehlerursachen darstellen. Um trotzdem nicht mit 80 Testfällen konfrontiert zu sein, kombinieren wir die gefundenen Parameter jetzt in Tabelle 2 paarweise zu nur noch 20 Testfällen.

Paarweise [PW] bedeutet, dass jedes beliebige Paar an Attributen mindestens einmal vorkommt. Der grundlegende Gedanke ist dabei, dass Fehler, die durch die Kombination dreier unterschiedlicher Bedingungen entstehen, sehr viel seltener sind als solche, die sich auch schon bei zweien zeigen. So findet sich zum Beispiel das Paar aus iPhone 5s und 42mm Watch zwei Mal in den Testnummern 4 und 14. Die Paarung iPhone 6 und iOS 8.4 kommt nur genau einmal im Test Nummer 7 vor. Mathematisch gesehen ist das hier gezeigte Beispiel weder minimal (es könnte auch mit weniger Testfällen gehen) noch sind die Paarungen bestmöglich verteilt. Wer das weiter optimieren möchte, kann auf orthogonale Arrays zurückgreifen [BaMcK11].

Bei den Testfällen mit Tilde (~) ist der in der Tabelle 2 vorgeschlagene Wert nicht mehr notwendig, um eine paarweise Abdeckung zu erzielen. Wir nutzen diesen Freiheitsgrad in der Spalte „Kapazität angepasst“, um das iPhone mit der geringsten Speicherausstattung (iPhone5C, 8 GB) mehr zu testen, und nutzen außerdem die Möglichkeit, doch noch ein paar der ursprünglich ausgeschlossenen Speicherausstattungsvarianten mit zu testen, ohne mehr Testfälle zu erzeugen. Dabei bewerten wir wieder die Wahrscheinlichkeit, dass eine solche Variante später im Feld

Testfall	Apple Watch	iPhone-Version	iPhone-Kapazität	iOS	Kapazität angepasst
1	42mm	iPhone 5	minimal	8.2	minimal (16 GB)
2	38mm	iPhone 5	maximal	8.3	maximal (64 GB)
3	38mm	iPhone 5s	minimal	8.2	minimal (16 GB)
4	42mm	iPhone 5s	maximal	8.2	maximal (64 GB)
5	42mm	iPhone 5c	minimal	8.4	minimal (8 GB)
6	38mm	iPhone 5c	maximal	9.0 Beta	maximal (32 GB)
7	38mm	iPhone 6	maximal	8.4	maximal (128 GB)
8	42mm	iPhone 6	minimal	9.0 Beta	minimal (16 GB)
9	42mm	iPhone 6 Plus	maximal	8.2	maximal (128 GB)
10	38mm	iPhone 6 Plus	minimal	8.3	minimal (16 GB)
11	~42mm	iPhone 5	~maximal	8.4	32 GB
12	~38mm	iPhone 5	~minimal	9.0 Beta	16 GB
13	~38mm	iPhone 5s	~minimal	8.4	32 GB
14	~42mm	iPhone 5s	~maximal	9.0 Beta	16 GB
15	~38mm	iPhone 5c	~maximal	8.2	8 GB
16	~42mm	iPhone 5c	~minimal	8.3	8 GB
17	~42mm	iPhone 6	~minimal	8.2	64 GB
18	~38mm	iPhone 6	~maximal	8.3	128 GB
19	~42mm	iPhone 6 Plus	~maximal	8.4	64 GB
20	~38mm	iPhone 6 Plus	~minimal	9.0 Beta	128 GB

Tabelle 2: Skriptgenerierte Testfallmenge und überarbeitete iPhone-Kapazität

auch vorkommt. Tabelle 2 zeigt somit einen Weg, die riesige Vielfalt von zu testenden Systemkonfigurationen zu beherrschen.

Automatisierung

Will man im nächsten Schritt 2000 Testfälle für jede der durch die Pairwise-Testing-Methode ausgewählten 20 Systemvarianten alle zwei Wochen als Regressionstest durchführen, landet man schnell bei Testautomatisierung. Der Automatisierungsgrad beim Testen in agilen Softwareentwicklungsprojekten ist oft sehr hoch. Neben der vielleicht eingesparten Zeit für manuelle Tests oder der Möglichkeit für erhöhte Testabdeckung liegt der größte Vorteil automatisierter Tests im frühzeitigen maschinellen Feedback. Dadurch ist ein hohes Maß an fest eingebauter Qualitätssicherung für jedes der häufigen Releases garantiert.

Wie steht es um die Automatisierbarkeit, wenn es um Apps geht, welche Rolle spielen Tools und Toolketten? Wie wählt man für das Projekt das am besten passende Tool aus?

Für die Evaluierung eines Testautomatisierungstools gibt es etliche Kriterienkataloge im Web oder auch in der Fachliteratur (z. B. [Kno15], Kapitel: „Selection Criteria for a Test Automation Tool“). Für den App-Test sind neue Kriterien hinzugekommen. Die Betriebssysteme im mobilen Umfeld unterscheiden sich fundamental. Gesten auf dem Touchscreen vom Testautomat zu erzeugen, kann für eine Uhr ganz andere Tools benötigen als auf dem Tablet. Die Unterscheidung zwischen einem festen oder leichten Fingerdruck kennt das Tablet nicht, die Apple Watch schon. Häufig werden Apps auch auf andere Betriebssysteme portiert. Wenn das Automatisierungstool dafür nicht geeignet ist, lauern spätere Mehraufwände.

Für den vollen Funktionsumfang von manchen Tools muss der Quellcode des SUT (System Under Test) um einige Zeilen Code ergänzt werden. Sollte der Quellcode nicht zur Verfügung stehen, können manche Funktionalitäten nicht automatisiert werden.

Ein weiteres wichtiges Kriterium für die Auswahl im App-Umfeld ist die Testumgebung. Wie viel muss auf echten Geräten getestet werden? Für welchen Test eignen sich auch Emulatoren oder Simulatoren? Können wir bestimmte Tests auch auf Geräten in der Cloud durchführen?

Echte Geräte muss man natürlich erst beschaffen. Für die mit der oben gezeigten Pairwise-Testing-Methode ermittelten Testfälle müssten 20 Geräte beschafft

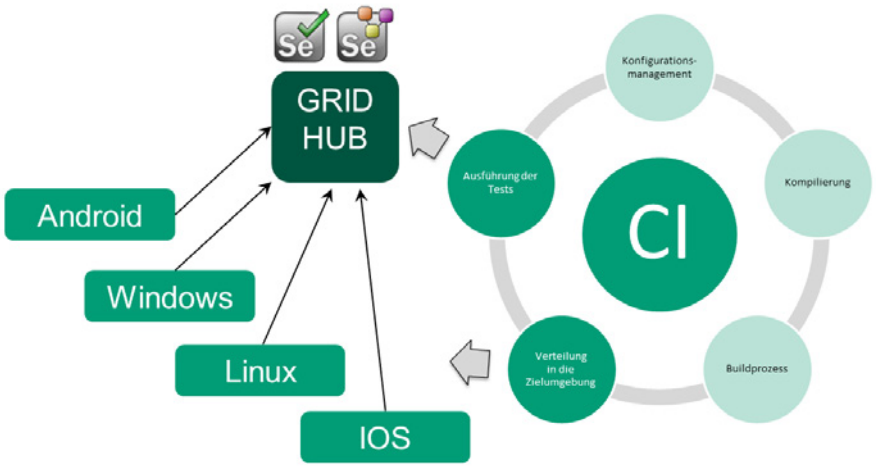

Abb. 2: Verteilte Testautomatisation

werden. Weiter ist wichtig, auf welcher Plattform das Automatisierungstool installiert wird und wie die Automatisierungsagenten auf den Mobilgeräten angesprochen werden. Hier kommen verschiedene Kopplungen wie WLAN, Bluetooth oder auch USB zum Einsatz. Bei plattformübergreifenden Apps können auch verschiedene, zu den Zielplattformen passende Automatisierungstools gleichzeitig benutzt werden. Hierfür bietet sich verteiltes Testen an.

Toolbeispiel: Verteiltes automatisiertes Testen mit Selenium-Grid

Schnelles Feedback kann auch bedeuten, dass automatische Tests parallelisiert werden, um dadurch schneller zu sein. Ein an der Benutzeroberfläche automatisierter Test kann schon mal fünf Minuten laufen. Wenn man alle Tests nacheinander startet, wird schnell die Nacht oder gar ein Wochenende zu kurz, um alle gewünschten Testfälle unterzubringen. Auch dafür gibt es Toollösungen, zum Beispiel den sogenannten Grid Hub vom Open-Source-Tool Selenium [Selenium]. Ein prominenter Anwender dieses Tools ist das Web-Kaufhaus eBay.

Abbildung 2 zeigt die Einbettung des Grid Hub in eine „Continuous Integration“ (CI). Die Entwicklertools zum Kompilieren und Bauen der Software (Buildprozess) werden dabei so ausgebaut, dass nach dem Build auch gleich automatische Installationen in verschiedene Umgebungen stattfinden. All die verfügbaren Testumgebungen werden an einem zentralen Punkt, dem „Grid Hub“ angemeldet. Der Hub weiß, welche Umgebungen gerade verfügbar sind, und kann dadurch die Aufforderungen für den Test entgegennehmen und in den verteilten Testumgebungen die Tests starten. Er kann damit skalierbar die Ausführung von Tests parallelisieren, was zur Reduktion der Ausführungszeiten führt. Der Grid Hub leistet einen wesentlichen Beitrag zum Umgebungsmanagement. Er hilft auch dabei, Crossover-Tests zu managen, also die Verbindung von Web und nativen Apps für Tests herzustellen.

Erfahrungsbasiertes Testen

Ein hoher Grad an parallelisierter Automatisierung mit Tools wie Ranorex, Silk Mobile oder auch Selenium-Grid kann viel bewirken. Methodische Testfallauswahl, Tools und Automatisierungsframeworks können somit entscheidend bei funktionalen Tests helfen.

Was ist mit Lücken, an die automatisierte Tests schwer herankommen? Ein Beispiel dafür sind kleine Memory Leaks, die sich im Laufe der Zeit aufschaukeln. Automatisierte Tests starten die Umgebung häufig neu, um definierte Anfangszustände und damit Reproduzierbarkeit zu gewährleisten, und entdecken so Memory Leaks meist nicht. Memory Leaks könnten wir auch mit einer statischen Analyse finden, für die in agilen Projekten allerdings oft keine Zeit bleibt.

Was ist mit so genannten nicht-funktionalen Themen wie Installationstests, Usability-Tests, Sprachtests oder Performance-Tests? Für letztere bieten schon vorhandene automatisierte Systemtestfälle durchaus eine Grundlage, wenn sie der Tester mit geeigneten Lasttools ergänzt. Beim Thema Usability andererseits nützt der Testroboter wenig. Und nicht jeder kann sich ein Usability-Labor leisten, das Entwicklungsprojekte von Beginn bis Ende begleitet.

Die Antwort in den Teststrategien vieler agiler Projekte lautet: Einige der Tests sind explorativ. Explorative Tests sind schnell durchführbar, da sie auf vorher spezifizierte Testfälle verzichten. Sie können damit schnell Feedback erzeugen. Sie sind trotzdem gezielt, weil jeder explorative Test eine Mission erfüllt. Und da sie auf die Erfahrung menschlicher Tester setzen, können sie neben der Vertiefung von funktionalen Tests (auch genannt „deep coverage“) sehr gut nicht-funktionale Kriterien adressieren. Auf explorative Tests sollte man also nicht verzichten [QSTag].

Die Test-Charta mit der darin enthaltenen Testmission ist der zentrale Bestandteil im explorativen Test. Sie wird im (agilen) Team abgestimmt, bevor der Test startet. Beim „session based testing“ [SBTM], einer Variante explorativen Testens, wird die Länge der Testsitzung vorab festgelegt (oft 90 Minuten) und es gibt eine Nachbesprechung („debriefing“), bei der die durch den Test gewonnene Erfahrung geteilt wird.

Explorativer Test mit Missionsbeispiel: Testcharta für Multi-Timer-App

Auch hier hilft wieder ein Beispiel für explorative Missionen beim App-Test, um den Stellenwert explorativen Testens zu verstehen. Bleiben wir bei Uhren: Eine der schon länger verfügbaren ist die als Kickstarter-Projekt entstandene Smartwatch Pebble. Der Hersteller bietet recht gute Hinweise auf Usability-Themen in Form eines „Pebble User Experience Design Guide“ [PeppleUXDesign]. Dieser lässt sich als Checkliste in einem explorativen Test nutzen. Eine Mission könnte somit folgendermaßen beschrieben sein:

- *Testobjekt*: Multi Timer App [PebbleAppStore] auf der Pebble Watch (The Original)
- *Testmission:* Es soll die praktische Verwendbarkeit der App bei 5 gleichzeitig aktiven Timern mit zufällig variierenden Start- und Stoppzeiten bewertet werden. Insbesondere, ob es Abweichungen von den Richtlinien des „User Experience Guides“ gibt, die sich negativ auf die Bedienung auswirken.

An dieser Mission lassen sich typische Eigenheiten explorativen Testens festmachen. Exploratives Testen bedient sich gerne erprobter Checklisten, um die Qualität des Testens zu erhöhen. Für den App-Test bietet das Internet einen reichhaltigen Fundus, der als Ausgangsbasis für die Checklisten verwendet werden kann. Bei Bedarf werden diese Checklisten an die eigenen Bedürfnis-

se angepasst. Mind-Maps sind in Projekten einfach zu handhaben – zum Beispiel Mind-Maps zum Android Testing [AndroidTesting]/iOS Testing [iOSTesting] und zum App-Test allgemein „getting startet with mobile testing“ [MobileTesting] der Website ministryoftesting.com. Explorative Tests sind erfahrungsbasiert, Checklisten sammeln Erfahrungen. Es geht im Missionsbeispiel nicht darum, den „User Experience Guide“ im Sinne einer formalen Spezifikation zu prüfen (Konformitätstest). Sondern eben um die Verwendung als Checkliste.

Wenn der Tester während seiner explorativen Testsitzung feststellt, dass es zwar mit fünf Timern gut funktioniert, aber mit noch mehr Timern nicht mehr, kann das in der Nachbesprechung thematisiert werden und es kann Anlass für weitere Tests sein. Exploratives Testen setzt einen klaren Fokus, aber erlaubt gleichzeitig den Blick über den Tellerrand.

Je nach Ausrichtung der Mission bringen explorative Tests oft Ergebnisse, die nicht gleich in einer Fehlerdatenbank landen sollten. Oft sind im Nachgang von explorativen Tests über die Abschlussbesprechung hinaus Gespräche notwendig, um mit Produktmanagern, Anwendern, Entwicklern oder anderen Stakeholdern die Frage „is it a bug or a feature“ zu entscheiden.

Beispiel Testentwurfsverfahren: A/B-Test für Multi-Timer-App

Ein denkbarer nächster Schritt nach nicht eindeutigem Feedback zur Multi-Timer-App könnte A/B-Testing sein. Diese Testmethode findet zunehmend Verbreitung, insbesondere im Web- und App-Bereich [GrCr14]. Mit dem A/B-Testing werden schon einsatzfähige Anwendungen verbessert. Ein Teil der Endanwender wird dazu (oft ungefragt) mit einer zweiten Variante einer Website oder App konfrontiert. Mit analytischen Werkzeugen wird das Bedienverhalten der Anwender an den Hersteller zurückgemeldet. Bei einer Multi-Timer-App könnten das eine Variante A mit großer Schriftart und weniger Timern und eine Variante B mit kleinerer Schriftart und mehr Timern sein. Folgende Metriken helfen, die Endkundenakzeptanz für die Varianten zu messen: benötigte Bedienzeit, Häufigkeit der Verwendung der Mehrfach-Timer-Option, Anzahl von Korrekturen durch die Anwender oder ähnliche Maße.

Fazit

Irgendwie sind Wearables, Apps und die mobile Welt anders. Sie sind neu, schnelllebig und agil. Aber wie wir in den Beispielen gesehen haben, sollten trotzdem alte und bewährte Methoden beziehungsweise Testentwurfsverfahren bei der Qualitätssicherung eine Rolle spielen. Qualität braucht systematische und methodische Ansätze, damit wenig Fehler durchrutschen. Sie braucht Tools, die Tests schneller machen. Weil vieles neu ist, müssen die Tester gerade bei den Tools am Ball bleiben. Es gibt mit jedem Device und mit jedem Jahr neue und wertvolle Hilfen durch die Tools, die es für Tester zu entdecken gilt. Tools können durch Testautomatisierung helfen, Update-Desaster zu vermeiden. Aber erst wenn die menschliche Expertise und Erfahrung breit und mit Erkundungsblick (explorativ) dazu kommt, läuft die Uhr rund.

Klar ist: Eine gute Teststrategie zu etablieren, und damit eine App professionell zu testen, wird nicht für lau zu haben sein. Und trotzdem gilt mehr denn je: „quality first is free“. Qualität nachzureichen ist immer teurer, als sich gleich darum zu kümmern. Egal, ob sich die Uhr mechanisch oder in einem Display dreht: Wenn die Qualität einer App nicht passt, wird die App schnell das Zeitliche segnen. ■

Literatur & Links

[AM] Agiles Manifest, http://agilemanifesto.org/iso/de/

[AndroidTesting] Ministry of Testing, http://www.ministryoftesting.com/2012/09/android-testing-mindmap/

[Apple15] Apple-Pressemitteilungen, http://www.apple.com/pr/library/

[BaMcK11] G. Bath, J. McKay, Praxiswissen Softwaretest – Test Analyst und Technical Test Analyst, dpunkt.verlag, 2011

[destatis] Statistisches Bundesamt, https://www.destatis.de/DE/ZahlenFakten/GesellschaftStaat/EinkommenKonsumLebensbedingungen/AusstattungGebrauchsguetern/Aktuell.html

[GrCr14] J. Gregory, L. Crispin, More Agile Testing, Addison-Wesley, 2014

[hp] Hewlett-Packard Development Company, http://h41112.www4.hp.com/events/agenda-technical/HP-Invent15_TD_BS34_14.00_Mobile-Apps-Von-der-Entwicklung-bis-zum-Test.pdf

[iOSTesting] Ministry of Testing, http://www.ministryoftesting.com/2012/09/ios-testing-mindmap-checklist/

[Kno15] D. Knott, Hands-On Mobile App Testing, Addison-Wesley, 2015

[MobileTesting] Ministry of Testing, http://www.ministryoftesting.com/2012/06/getting-started-with-mobile-testing-a-mindmap/

[PeppleAppStore] Pebble App Store, http://apps.getpebble.com/en_US/application/52d30a1d19412b4d84000025

[PeppleUXDesign] Pebble Designer Guide, http://developer.getpebble.com/guides/best-practices/design

[PW] Combinatorial Test Case Generation, http://www.pairwise.org/

[QSTag] Software-QS-Tag 2015, Session „Non-Functional Testing“, http://www.qs-tag.de/abstracts/tag-2/session-based-testing-als-methode-fuer-non-functional-tests/

[SBTM] Ministry of Testing, http://www.ministryoftesting.com/2012/06/session-based-test-management-the-mindmap/

[Selenium] Selenium-Grid, Selenium Documentation, http://www.seleniumhq.org/docs/07_selenium_grid.jsp