Unit Testing

Lukas Panni
TINF18B5

DHBW Karlsruhe
Vorlesung Advanced Software Engineering Semester 5/6

Inhaltsverzeichnis

1 Analyse und Begründung für Umfang der Tests

Um die Funktionalität der einzelnen Komponenten gewährleisten zu können werden Unit-Tests eingesetzt. Dabei werden für die einzelnen Tests nur die, für diesen Test, relevanten Teile des Systems verwendet. Da Abhängigkeiten zu anderen Komponenten die Tests nicht beeinflussen sollen werden alle anderen Komponenten durch Fake-/Mock-Objekte ersetzt. Das Zusammenspiel mit den anderen Komponenten kann in Integrationstests getestet werden. Außerdem tragen Unit-Tests auch zur Dokumentation bei, indem das gewünschte Verhalten der Komponente für Regel- und Ausnahmefälle in den Testfällen dokumentiert ist.
Für dieses Projekt wird *JUnit* als Framework für die Erstellung und Ausführung von Java-Unit-Tests verwendet. Bei der Implementierung der Tests wurde darauf geachtet die ATRIP-Regeln (*Automatic, Thorough, Repeatable, Independent, Professional*) möglichst zu beachten.
Teile des Codes in diesem Projekt werden für Android-Spezifische UI-Aufgaben benötigt und können deshalb nur schlecht getestet werden. Das führt auch dazu, dass die Code-Coverage über das gesamte Projekt vergleichsweise klein sein kann und in diesem Fall keine sinnvolle Aussage über die Genauigkeit der Tests ermöglicht. Stattdessen sollte hier zur Beurteilung der Testabdeckung nur die Code-Coverage der nicht-UI-Klassen betrachtet werden.
Allgemein wird darauf verzichtet triviale Funktionen, wie zum Beispiel Getter zu testen. Tests dieser Funktionen würden bei großem Aufwand nur einen minimalen Mehrwert bringen, da keine echte Funktionalität getestet wird. Stattdessen sollen sich die Tests auf relevante Funktionalität des Systems fokussieren. Das bedeutet, dass vor allem die Klassen getestet werden sollen, die häufig verwendet werden und auch die Klassen, die für die Funktion der Anwendung unerlässlich sind. UI-Klassen, die aufgrund starker Abhängigkeiten zu Android-Klassen schwer testbar sind sollen weniger ausführlich getestet werden.
Für die Tests innerhalb dieses Systems wurden drei Kernfunktionalitäten identifiziert, die von einer hohen Testabdeckung am stärksten profitieren. Diese sind das Abrufen von Daten von der API, die Konvertierung von API-Datentypen zu Datentypen der Anwendung und in diesem Zusammenhang auch das Caching von abgerufenen Daten um die Netzwerklast zu reduzieren.
Diese Funktionalitäten verteilen sich auf die Packages *client*, *data* und *repository*. Im Folgenden wird genauer auf die Tests dieser Packages eingegangen

Das data-Package enthält alle Klassen, die als Nutzdaten von der Anwendung verarbeitet werden. Antworten, die von der API empfangen werden, werden in einem ersten Schritt in einen solchen Datentyp konvertiert. Diese Konvertierungen werden in den Konstruktoren

der Klassen durchgeführt. Ansonsten werden diese Klassen hauptsächlich zur Speicherung von Anwendungsdaten verwendet und enthalten nur vergleichsweise wenig weitere Funktionalität. Da die Konvertierung von API-Datentyp in Anwendungsdatentyp sehr häufig verwendet wird und die Anwendungsdatentypen generell an vielen verschiedenen verwendet werden, sind hier ausführliche Tests notwendig. Der Fokus der Tests liegt dabei auf den Konstruktoren, alle weiteren Funktionalitäten der Klassen sind weniger wichtig und meist trivial.

Im repository-Package, sind einerseits Repository-Klassen enthalten, die den Zugriff auf Anwendungsdaten abstrahieren und dabei das Caching der Daten transparent machen. Andererseits finden sich hier auch die Klassen, die für die Umsetzung des Cachings verwendet werden. Die Repository-Klassen übernehmen die Funktion, die Daten aus dem Cache abzurufen, wenn diese dort vorhanden sind und ansonsten die Daten über Funktionalität, die im client-Package implementiert ist von der API-abzurufen. Als zentrale Komponente, um auf Daten zuzugreifen sind die Repository-Klassen deshalb entscheidend für die Funktionalität der gesamten Anwendung. Aus diesem Grund ist es wichtig hier eine möglichst hohe Testabdeckung zu erreichen.
Die Komponente zum Caching ist ebenfalls wichtig, da ohne funktionierendes Caching sehr viele Anfragen über das Netzwerk gesendet werden müssen, was nicht nur die Menge der übertragenen Daten unnötig erhöht, sondern auch die Reaktionszeit der Anwendung verschlechtert. Deshalb ist auch im cache-Package eine hohe Testabdeckung wünschenswert.

Das client-Package enthält alle Klassen, die benötigt werden, um Daten von der API abrufen zu können und steht damit, wie auch das cache-Package, im Zusammenhang mit dem Datenzugriff über die Repository-Klassen. Es ist offensichtlich, dass auch diese Funktionalität von großer Bedeutung für die Funktionalität der Anwendung ist, und deshalb auch in diesem Package eine möglichst hohe Testabdeckung angestrebt wird.

Im Gegensatz zu den genannten Bereichen ist eine hohe Testabdeckung im *ui*-Package weniger wichtig und bringt nur wenig Nutzen. Zusätzlich ist hier aufgrund der genannten Einschränkungen eine hohe Testabdeckung nur mit sehr hohem Aufwand möglich. Im Anbetracht des geringen Nutzens ist der Aufwand hier eine hohe Testabdeckung zu erreichen nicht gerechtfertigt, weshalb hier weitestgehend auf ausführliche Tests verzichtet wird.

2 Analyse und Begründung für Einsatz von Fake-/Mock-Objekten

Fake- und Mock-Objekte werden benötigt, um Abhängigkeiten einer Komponente zu anderen Komponenten in Unit-Tests zu reduzieren. Sie implementieren dafür zum Beispiel das benötigte Interface, aber davon nur die aktuell benötigte Funktionalität.