Aufteilen von Hilfsmethoden bei Software-Tests – Praxisbeispiel UI-Tests

Große Softwareprojekte benötigen zahlreiche Tests. Und so wächst mit dem Projekt auch die Anzahl der Testfälle sowie die dazugehörigen Codezeilenanzahl schnell an und erreicht einen unüberschaubaren Wert.

Neben den eigentlichen Testfällen stellen Hilfsmethoden, wie beispielsweise das Suchen von Elementen, das Vorbereiten von Testdaten und so weiter und so fort einen enormen Anteil der Codebasis der Tests dar. Oft beginnt man einfach damit die Hilfsmethoden irgendwie einzuordnen und das Ganze verliert dann mit zunehmender Zeit und stetig wachsender Projektgröße die Übersicht.

Daher gibt es auch zum Aufteilen von Test-Hilfsmethoden entsprechende Pattern an die man sich halten kann. Das Single Responsible Pattern – jede Klasse trägt exakt eine Verantwortung – aus der „normalen Softwareentwicklung“ kann man meiner Meinung nach einwandfrei auch zum Strukturieren von Hilfsklassen verwenden.

Vorteile?

Das Aufteilen von Hilfsmethoden in entsprechende Klassen dient erstens zur Steigerung der Übersichtlichkeit und zweitens trägt es dazu bei, dass Dopplungen vermieden werden. Die Wartung der Tests wird vereinfacht und die Code-Basis kann – geschickt ausgelagert – in jedem weiteren Projekt ebenfalls verwendet werden. Die Aussagekraft der Tests wird durch geschickt Kapselungsmethoden gesteigert und auch die Produktivität nimmt zu, wenn die Testprojekte gut strukturiert sind und jede Hilfsmethode einen festen Platz hat.

Gerrard Meszaros, Agile Software Development Consultant, stellt in seinem Buch xUnit Test Patterns: Refactoring Test Code einen meiner Meinung nach gelungenen Ansatz bereit, welchen ich gern in meinen Projekten verwende.

Dabei teilen sich die Hilfsmethoden seiner Ansicht nach so gut wie immer in folgende vier Bestandteile auf:

* Creation Methods
* Finder Methods
* Encapsulation Methods
* Verification/Assert Methods
* Cleanup Methods

Auf alle möchte ich kurz eingehen und anschließend ein kleines UI-Tests-Szenario unter Berücksichtigung dieser Aufteilung aufzeigen.

Creation Methods

Wie der Name schon sagt, dienen diese Hilfsmethoden zum Generieren und Bereitstellen von Daten. Darunter fällt beispielweise auch das Instanziieren der entsprechenden Testdaten, beziehungsweise der Abhängigkeiten für das Testobjekt. Auch das Instanziieren des Testobjekts selbst kann in diesen Bereich fallen.

Finder Methods

Auch hier gibt der Name bereits Aufschluss über die entsprechende Aktivität. Diese Methoden dienen zum Suchen, Ermitteln und Bereitstellen von Ressourcen. Gerade bei UI-Tests sind das zahlreiche Methoden, wie beispielsweise das Suchen von Elementen nach Id, Name, Inhalt oder sonstigen Sucheigenschaften.

Encapsulation Methods

Die Kapselungsmethoden dienen zum Zusammenfassen von Aktionen. Ein geeigentes Beispiel sind hier immer wieder auftretende Prozesse, die mehrere Codezeilen benötigen um ausgeführt zu werden. Beispielsweise muss ein Benutzer einer Anwendung vor jedem Test eingeloggt werden, ehe er interne Funktionen nutzen kann. Also wird das Suchen und Füllen der Textfelder für den Login, sowie das Abschicken des Formulars zusammengefasst ausgelagert.

Verification / Assert Methods

Darunter fallen Methoden zum Validieren von Objekteigenschaften, sowie Custom-Assert-Methoden.

Cleanup Methods

Um die Testumgebung sauber zurückzulassen, ist es oft von Nöten dass Überbleibsel des Tests noch beseitigt werden. All solche Teardown Methoden werden in dieser Kategorie zusammengefasst.

Folgendes Beispiel ist auch wie immer auf Github bereitgestellt. Zu Beginn möchte ich das Szenario kurz beschreiben. Ein Anwender soll bei der zu testenden Anwendung angemeldet werden und anschließend wird verifiziert, dass die Home-Site mit einer Willkommensmeldung angezeigt wird.

Alles nacheinander geschrieben sieht dieser Testfall so aus. Der Vorteil hier ist, dass noch nicht viel von Nöten ist um diesen simplen Test auszuführen.

[Code here]

Schnell wird deutlich, wie viel Methoden sich schon in diesem kleinen Test doppeln und daher Hilfsmethoden notwendig werden. Korrekt ausgelagert sieht der übrige Code dann wie folgt aus:

[CODE HERE]

Erstes ist der Test nun deutlich lesbarer gestaltet. Es sind bereits Basisstrukturen für die weiteren Testfälle vorhanden, da beispielsweise beim Test mit falschen Logindaten der Loginprozess der gleiche bleibt, und nur das Überprüfen der Fehlermeldung eine andere ist. Weiterhin wurde die Wartbarkeit deutlich erhöht. Das Beispiel steht, wie bereits erwähnt, auf Github bereit und da kann nachgeschaut werden wie ich die Methoden in die einzelnen Klassen aufgeteilt habe. Die Struktur möchte ich hier nur noch kurz aufzeigen.

[BILD HERE]

Aber Vorsicht!

Zu vernachlässigen ist aber nicht, dass diese Planung der Aufteilung nicht zu einem Overhead führen soll. Das heißt, es soll nicht bereits im Vorfeld entschieden werden, wie das ganze Aufzuteilen ist, und welche Hilfsmethoden benötigt werden. Es sollen also auch wirklich nur die Methoden implementiert werden, die wirklich benötigt werden – Ansonsten wird wertvolle Zeit in unnötige Implementierung gesteckt. Das sinnvollste Vorgehen ist hier für:

1. Schreiben des Testfalls
2. Code-Dopplungen der gleichen Methode / gleichen Klasse zusammenfassen
3. Auslagern der Methoden in die entsprechenden Kategorien
   1. Falls noch nicht vorhanden, so zuerst anlegen
4. Bei großen Projekten und zahlreichen Methoden: Auslagern in eigene Bibliothek

Fazit

Auch bei den Hilfsmethoden gibt es diverse Ansätze diese korrekt aufzuteilen und so dafür zu sorgen, dass eine gute Struktur vorliegt und so die Wartung vereinfacht wird. Weiterhin kann bei richtiger Umsetzung die entstandene Hilfsmethoden-Bibliothek sofort für das nächste Testprojekt verwendet werden.