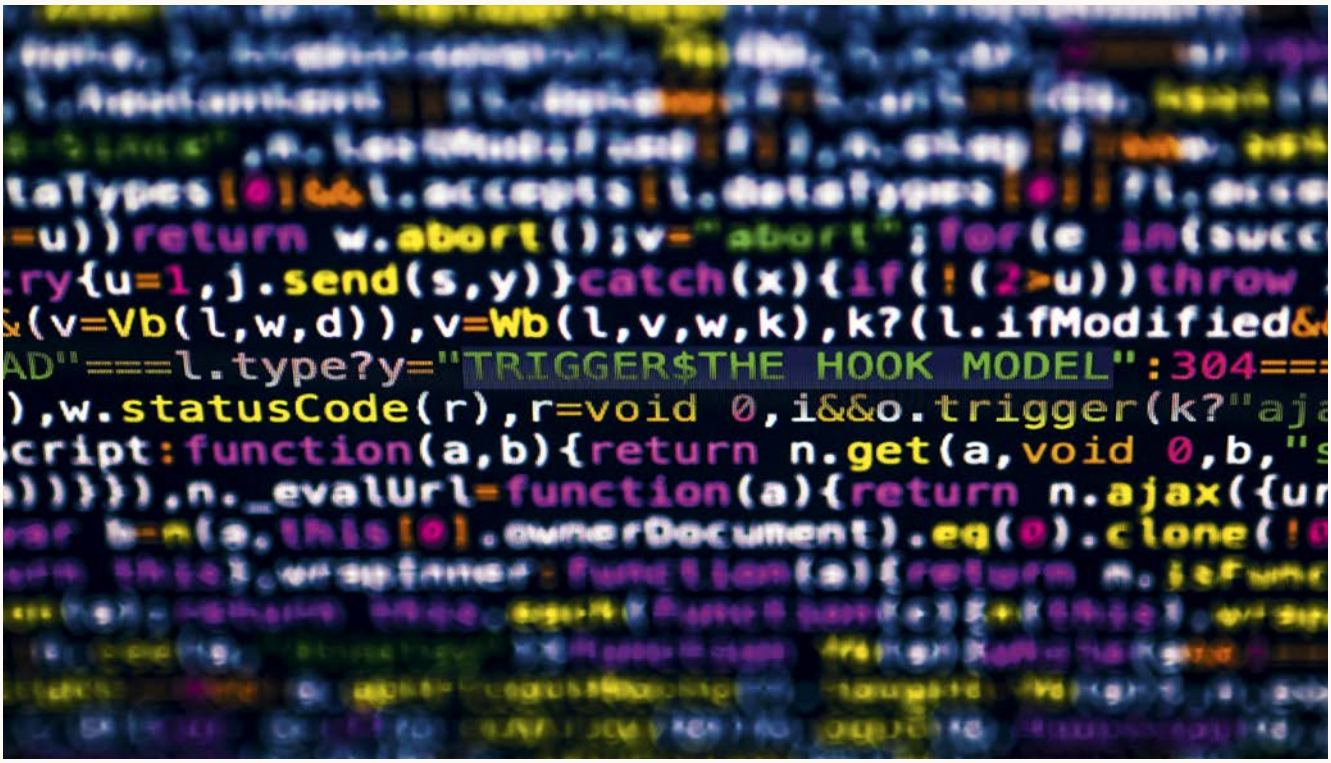
»Durchgeklickt – Automatisiertes GUI-Testen mit Squish«

Umfangreichere grafische Benutzeroberflächen (GUI) sind nur mit großem Aufwand manuell zu testen. Mit GUI-Test-Tools lassen sich detaillierte Test-Szenarien definieren, automatisiert ausführen und durch Screenshots dokumentieren. Dabei können der Zustand und die Eigenschaften sämtlicher grafischer Elemente überprüft und beliebige Eingaben und Button-Klicks simuliert werden. Wir berichten über den Einsatz von Squish zum Testen einer in Qt/QML geschriebenen Anwendung in der Medizintechnik.



Grafische Benutzeroberflächen bestehen häufig aus mehr als nur einer Handvoll Seiten. Kommen dann Verzweigungen hinzu und kann zwischen entfernten Seiten hin und her navigiert werden, sind die Abläufe schnell sehr komplex. Manuelles Testen der Oberfläche ist dann im Allgemeinen sehr zeitaufwendig.

Hier schaffen automatisierte GUI-Test-Tools beziehungsweise -Frameworks Abhilfe. Mit ihnen lassen sich Test-Szenarien definieren, die voneinander unabhängig unter festgelegten Anfangsbedingungen ausgeführt werden. Einzelne Schritte, zum Beispiel das Drücken eines Buttons, müssen dabei nur einmal definiert und können in verschiedenen Szenarien verwendet werden. Screenshots kann man zu beliebigen Zeitpunkten erstellen und ablegen.

Testen mit Squish

Ein solches Tool ist Squish der Firma Froglogic. Es unterstützt die meisten großen GUI-Technologien (z. B. Qt, Java AWT, Windows MFC, Webkit) und lässt sich für Desktop-, Mobile-, Web- und Embedded-Anwendungen einsetzen. Squish hat eine eigene Entwicklungsumgebung (IDE), ist aber auch vollständig über die Kommandozeile steuerbar.

Das Grundprinzip beim Testen mit Squish entspricht einem Konzept, das aus dem Behavior-Driven Development (BDD) bekannt ist: Die Anforderungen an das Programm werden mittels sogenannter Szenarien abgebildet und getestet. Dabei kann eine Anforderung auch durch mehrere Szenarien abgedeckt sein oder umgekehrt ein Szenario mehrere Anforderungen abdecken.

Jedes Szenario besteht aus einer Folge von Testschritten (Steps) und bildet eine überschaubare inhaltliche Einheit. Bietet ein Programm beispielsweise die Möglichkeit, zwischen den Sprachen Deutsch und Englisch umzuschalten, gibt es dazu folgendes Szenario:

Scenario: Language should be switchable from German to English
Given that my program is running
Then the label of the print button is "Drucken"
When I press the language button to switch to English
Then the label of the print button is "Print"

Diese vier Testschritte sind dabei in der Beschreibungssprache Gherkin formuliert. Sie bedient sich natürlicher Sprache sowie einiger weniger Schlüsselwörter wie Given, When oder Then, mit denen sich die Testschritte verknüpfen lassen. Auf diese Weise können auch Personen Szenarien definieren und verstehen, die sich in den implementierenden Programmiersprachen nicht zu Hause fühlen. An Aktionen und Überprüfungen ist grundsätzlich alles möglich, was menschliche Nutzer drücken, eingeben, lesen und sehen können.

Mehrere inhaltlich zueinander passende Szenarien werden üblicherweise in einer sogenannten Feature-Datei zusammengefasst. Die Szenarien einer oder mehrerer dieser Feature-Dateien bilden zusammen eine Test-Suite. Für die Implementierung der Steps unterstützt Squish mehrere Programmiersprachen. Dies sind derzeit Python, JavaScript, Ruby, Perl und TCL. Jeder Gherkin-Step entspricht einer Funktion, die in einer dieser Sprachen implementiert wurde. Die für die Step-Implementierung gewählte Sprache ist dabei unabhängig von der Sprache, in der die Anwendung geschrieben ist (z. B. Java). **Abbildung 1** zeigt ein Beispiel für den Zusammenhang zwischen Feature-Dateien und Implementierungsdatei.

Für den Test der Anwendung wird üblicherweise die komplette Test-Suite mit allen Szenarien durchlaufen. Dabei lassen sich die durchgeführten Aktionen direkt am Bildschirm mitverfolgen. Squish übernimmt alle Nutzerinteraktionen wie das Drücken der Buttons, das Ausfüllen der Textfelder oder das Wechseln der Seiten.

Testautomatisierung für die Anwendung in der Medizintechnik

Die Tests lassen sich auch automatisch mithilfe der Squish-Kommandozeilen-Tools ausführen. So können die UI-Tests in die Continuous Integration (CI) eingebunden und Fehler im UI-Verhalten zur Entwicklungszeit frühzeitig erkannt werden. Die Testresultate kann Squish in verschiedensten Formaten ausleiten, zum Beispiel als HTML-Report oder im JUnit-Format. Regularien für medizintechnische Anwendungen fordern die Rückverfolgbarkeit von Tests zu den Anforderungen an das Programm (Traceability). Um diese Verknüpfungen herzustellen, versieht man jedes Szenario mit der Identifikationsnummer der entsprechenden Anforderung. Die Identifikationsnummern tauchen im Testreport auf und können dann mit der Liste der Anforderungen abgeglichen werden.

Squish bietet an, zu bestimmten Zeitpunkten der Testausführung Screenshots der zu testenden Applikation (Application Under Test, kurz AUT) abzuspeichern. So lässt sich eine aktuelle Sammlung von verschiedenen Zuständen der Applikation sehr leicht automatisch erstellen. Diese Sammlung kann beispielsweise für das Handbuch der Applikation verwendet oder als aktueller Implementierungsstand an die UI-Designer verschickt werden. Ebenfalls kann diese für die Dokumentation oder die Gebrauchstauglichkeitsanalyse verwendet werden, um die gesetzlichen Bestimmungen zu erfüllen.

Anlegen eines neuen Test-Szenarios

Im Folgenden beschreiben wir beispielhaft, wie ein neues Szenario mithilfe der Squish-IDE erstellt werden kann. Zu Beginn muss in der Squish-IDE eine neue Test-Suite angelegt werden. Diese enthält die einzelnen Feature-Dateien und definiert deren Randbedingungen, wie die AUT und ihr verwendetes UI-Framework, zum Beispiel Qt. Auch die Sprache, in der die Steps implementiert werden sollen, wird bei der Erstellung der Test-Suite festgelegt. Sobald diese angelegt ist, können Testfälle erstellt werden.

Um nun das erste Szenario zu verfassen, beschreibt man das gewünschte Verhalten der Applikation in Bezug auf eine bestimmte Nutzerinteraktion mithilfe der Gherkin-Syntax. Als Beispielszenario dient hier das bereits kurz erwähnte Umschalten der Sprache von Deutsch auf Englisch. Wurde das Szenario beschrieben, erkennt die Squish-IDE, welche Steps bereits implementiert sind und welche nicht. Die entsprechenden Funktionen werden in einer Implementierungsdatei zusammengefasst.

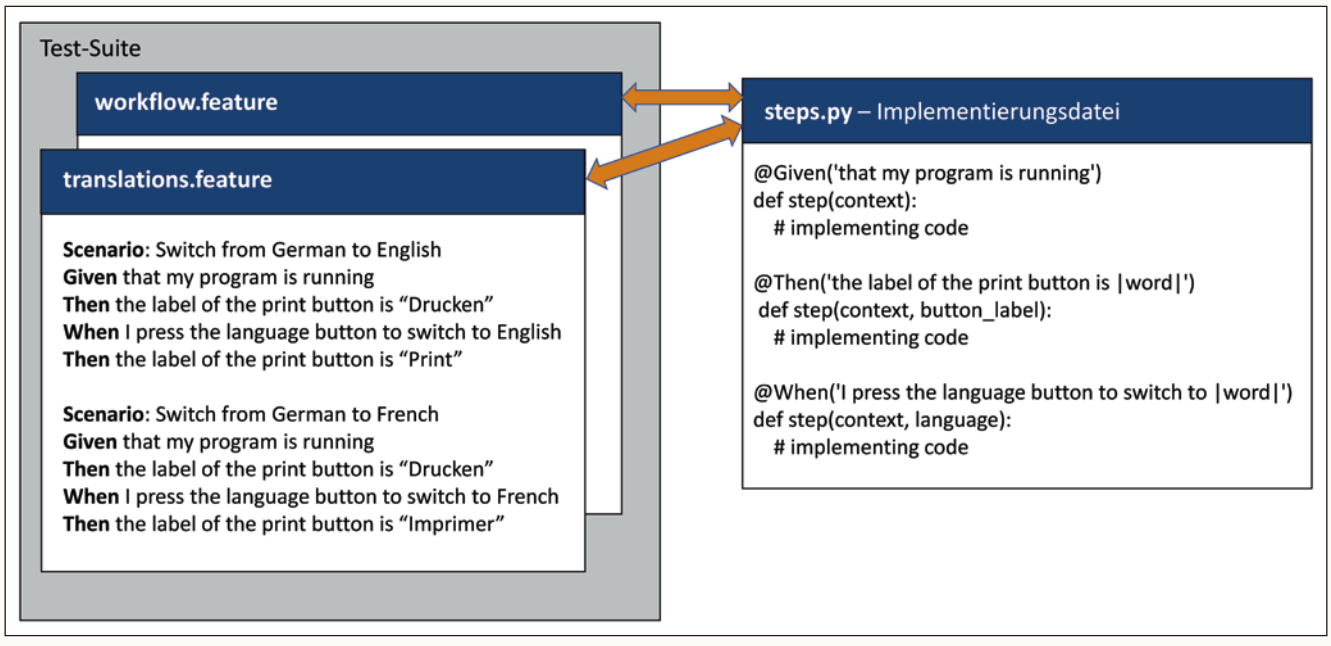


Abb. 1: Beispiel einer Test-Suite mit zwei Feature-Dateien und einer Implementierungsdatei

Um die fehlenden Step-Implementierungen hinzuzufügen, bietet Squish eine Aufnahmefunktion an, mit der sich Mausklicks und weitere Interaktionen des Nutzers und die benötigten Verifikationsschritte an einer laufenden AUT aufzeichnen lassen. Grafische Elemente werden dabei nicht anhand ihrer Koordinaten auf dem Bildschirm identifiziert (Screenshot Verification), sondern anhand ihrer Eigenschaften (Property Verification). Somit lassen sich auch interne, nicht sichtbare Elementeigenschaften überprüfen.

Durch das Starten der Aufnahme wird die Applikation ausgeführt; ein Dialogfenster zeigt an, welcher Step des Szenarios gerade aufgezeichnet wird. Dabei setzt Squish Interaktionen, wie Mausklicks oder Tastatureingaben, in Code um. Um zu verifizieren, dass der „Print"-Knopf beim Start der Applikation den Text „Drucken" anzeigt, muss ein Verifikationspunkt eingeführt werden. Squish bietet dazu einen sogenannten „Verification Point Creator" an. Mit diesem kann festgelegt werden, in welchem Zustand sich die Applikation beziehungsweise ihre UI-Elemente zu diesem Zeitpunkt befinden müssen. Eine Liste der aktuell sichtbaren UI-Elemente und ihrer Eigenschaften ist dafür in Squish auswählbar. In unserem Beispiel soll die „Text"-Eigenschaft des „Print"-Buttons mit dem Text „Drucken" überein stimmen.

Um zur Laufzeit der AUT an die entsprechenden Informationen zu gelangen, überschreibt Squish bestimmte Funktionen der verwendeten UI-Bibliothek. Die entsprechende Implementierung würde ungefähr wie in **Listing 1** aussehen.

```
@Then('the label of the print button is "Drucken"')
def step(context):
  test.compare(str(waitForObjectExists(print_button.text)), "Drucken")
```

Listing 1: Verifikations-Step

```
@When('I press the language button to switch to English')
def step(context):
  mouseClick(waitForObjectExists(language_button))
```

Listing 2: Interaktions-Step

Der nächste Step kann ausgewählt und aufgezeichnet werden. Hier muss der Nutzer auf den Sprachbutton in der AUT klicken. Wenn dieser Knopf gedrückt wurde, zeichnet Squish diese Interaktion wieder als Teil dieses Steps auf und erzeugt den Python-Code aus **Listing 2**.

Im letzten Testschritt des Szenarios wird geprüft, ob sich der Text des Buttons nun entsprechend der Sprachauswahl geändert hat. Dieser Step muss nicht erneut implementiert werden, wenn die Implementierung des quasi identischen Steps Nummer 2 angepasst wird. Mithilfe der Step-Parametrisierung können die beiden Schritte, die sich nur durch den Inhalt des Buttontextes unterscheiden, zusammengefasst werden.

Damit ist das Szenario vollständig. Von jetzt an kann Squish diese Abfolge der Interaktionen und Verifikationen automatisch durchführen, wenn der Entwickler die Test-Suite ausführt.

Squish Embedded: Testen auf dem Zielgerät

Die Entwicklungsumgebung auf dem Desktop-PC und das CI-System sind so eingerichtet, dass die verwendeten Bibliotheken mit denen auf dem Zielgerät möglichst übereinstimmen. So sind die regelmäßig während der Entwicklung ausgeführten Tests auch aussagekräftig für das Zielgerät. Trotzdem bleiben oft Unterschiede, die sich nicht beseitigen lassen, etwa ein anderer Prozessor mit unterschiedlicher Leistung oder eine andere Compiler-Version. Deswegen ist es wünschenswert, die Tests auch auf dem Zielgerät auszuführen. Dies ermöglicht die Erweiterung Squish for Qt Embedded, die als Embedded SDK and Support Package für Lizenzkunden separat zu erwerben ist. Squish for Qt Embedded unterstützt unter anderem Embedded Linux, WinCE und QNX.

Auf dem Zielgerät wird eine für die Zielarchitektur kompilierte Minimalinstallation von Squish eingerichtet. Diese interagiert zum einen mit der auf dem Zielgerät laufenden AUT und kommuniziert zum anderen über das Netzwerk mit einer auf dem Desktop-PC laufenden Squish-Instanz. Diese Squish-Instanz übernimmt wie gewohnt die Teststeuerung. Auf dem Desktop-PC läuft weiterhin die Squish IDE.

Projekterfahrungen und Fazit

Wir verwenden Squish mit dem oben beschriebenen Workflow im Rahmen der Entwicklung einer in Qt geschriebenen Medizintechnik-Anwendung auf einer Embedded-Plattform. Squish erweist sich dabei als ein sehr mächtiges Tool für die Organisation und Durchführung der GUI-Tests, das sich gut in unser CI-System integrieren lässt. Besonders hilfreich ist es, die Tests direkt auf der Zielplattform ausführen zu können.

Das Einrichten der Embedded-Umgebung ist nicht übermäßig aufwendig. Die Squish-Dokumentation haben wir als ordentlich bis gut empfunden, den Support stets als sehr kompetent und zügig. Bedienung und Erscheinung der auf Eclipse basierenden IDE ist Geschmackssache, sie funktioniert aber gut.

Für die Step-Implementierungen verwenden wir Python, was den Einstieg aufgrund bereits vorhandener Kenntnisse erleichtert hat. Leider wird Python im Moment nur in der Version 2 mitgeliefert. Mit etwas manueller Anpassung lässt sich jedoch auch eine neuere Python-Version verwenden; eine entsprechende Anleitung findet sich in der Squish-Dokumentation.

Auch wenn die Anwendung mit der Zeit schnell größer und komplexer wird, verliert man nicht den Überblick und der Aufwand für das Anlegen neuer Szenarien hält sich im Rahmen. Durch die große Anzahl an unterstützten Frameworks, Plattformen und Sprachen sollte es nur wenige Fälle geben, in denen Squish nicht anwendbar ist.

Tobias Bliem

tobias.bliem@methodpark.de
ist Software Engineer bei Method Park. Im Team Medical Devices entwickelt er Anwendungen für die Medizintechnik. Dabei unterstützt er seine Kunden bei Entwurf, Umsetzung und Test.

Simon Kuhnle

simon.kuhnle@methodpark.de
ist als Software Engineer bei Method Park tätig. Im Team Internet of Things unterstützt er die Medical-Kunden des Unternehmens bei der agilen Softwareentwicklung für Medizingeräte. Dabei konzentriert er sich speziell auf die UI-Entwicklung mit Qt/QML auf Embedded Linux.

Sebastian Kern

sebastian.kern@methodpark.de
ist bei Method Park als Teamleiter verantwortlich für das Team Medical Devices. Er befasst sich mit den Themen Agiles Testen, Projekt-Testmanagement und den regulatorischen Anforderungen in der Medizintechnik.