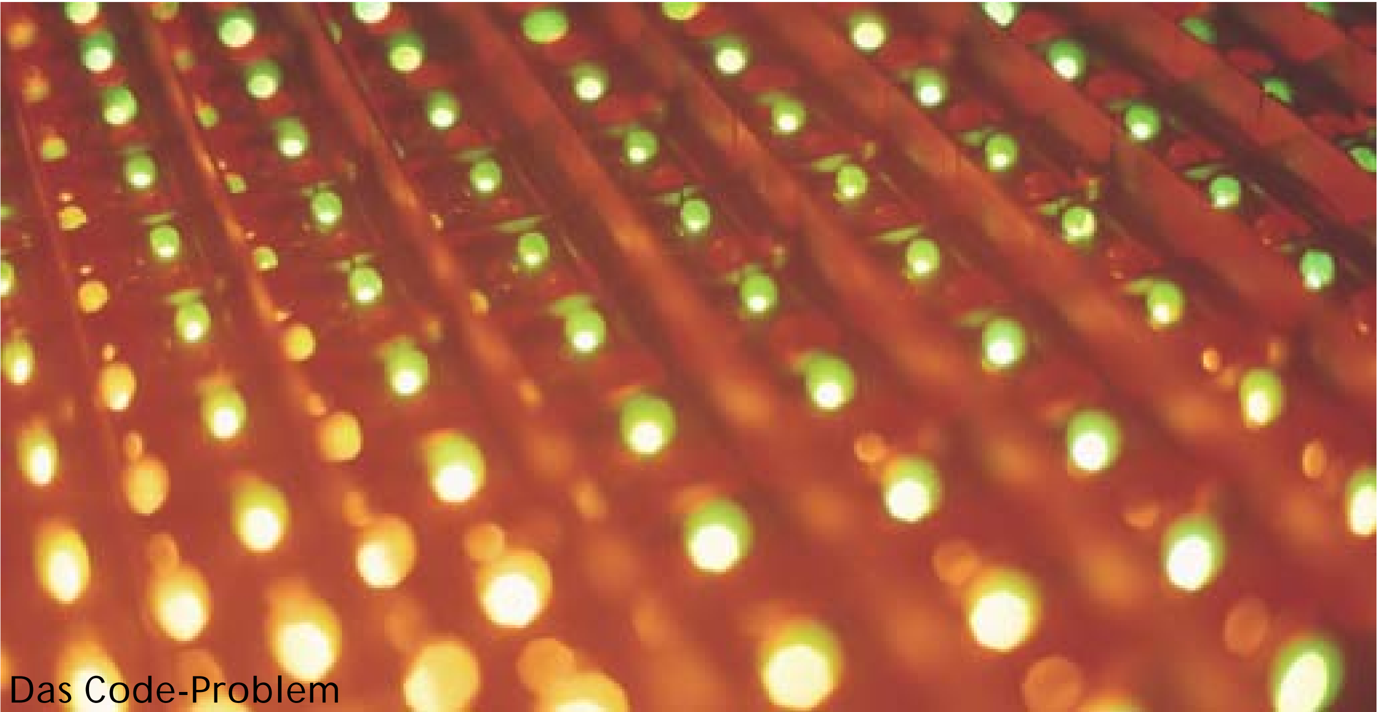

Das Code-Problem

Die Automatisierung von funktionalen Tests

In den Jahren 1996 bis 2001 fand das Thema „Testautomatisierung" in der Literatur und Praxis besonders viel Beachtung. Trotzdem zeigen aktuelle Projekte, dass die Automatisierung von funktionalen Tests nur selten erfolgreich umgesetzt wird. Ist die Krise im IT-Sektor allein dafür verantwortlich zu machen oder gibt es auch andere Gründe für die Dissonanz zwischen Theorie und Praxis?

Im Bereich „Last- und Performancetest" wirkt die „schwarz-weiß Malerei" immer noch erstaunlich gut. Zum größten Teil haben wir das den kollabierenden Web-Seiten und den gigantischen ERP-Projekten zu verdanken. Andrerseits aber auch den Testtools, die in diesem Fall eine wirkliche und unmittelbare Verbesserung bieten. Das haben gleichermaßen Toolhersteller als auch Toolanwender erkannt, was zu einem regelrechten Boom in diesem Segment führte.

Im Bereich „funktionales Testen" scheint diese Art der Schwarzmalerei aber nicht zu wirken. Wir glauben nicht, dass es an Lösungen mangelt, sondern an deren effizienter Umsetzung. Zudem ist der Nachweis, dass ausgerechnet funktionale Tests zum eventuellen Projekterfolg etwas wichtiges beigetragen haben, nach wie vor nur schwer zu erbringen. Welche verheerende Wirkung unrealistische Terminvorgaben haben, die für Softwareprojekte so symptomatisch sind, beweist eindrucksvoll das Debakel bei der Einführung der LKW-Maut. Uns fällt es schwer zu glauben, dass Tester von den Problemen nichts wußten. Nur welchen Einfluss konnten sie noch auf die Projektplanung nehmen?

Um den Unterschied zu verdeutlichen, wenden wir uns wieder den Last- und Performancetests zu. Betrachten wir hierzu eine einzelne Operation, die von, etwa 1000 Benutzern gleichzeitig ausgeführt werden soll. Um dies zu testen, stehen uns zwei Optionen zur Verfügung: a) 1000 echte Benutzer, die auf 1000 echten Rechnern die gleiche Operation gleichzeitig durchführen oder b) ein Testskript, das auf einem oder nur wenigen Rechnern 1000 mal synchronisiert ausgeführt wird. Stellen wir einfach fest, dass ohne die passenden Testtools, solche Tests realistisch gar nicht durchführbar sind.

Dazu kommen noch zwei weitere Aspekte. Erstens, keiner verheimlicht uns, dass ein spezialisiertes, sehr hohes Niveau an Know-how notwendig ist, um solche Tests zu gestalten beziehungsweise solche Testtools effizient zu bedienen. Zweitens, Last- und Performancetests sind in der Regel keine Daueraufgabe, sondern werden aufgesetzt, wenige Male durchgeführt und dann,

bis zur nächsten Kampagne, nicht mehr berührt. Dies passt gut zu der überaus technokratisierten Softwareentwicklung, die in puncto Organisation und Disziplin nicht gerade glänzt.

Funktionale Tests andererseits sind eine Daueraufgabe. Sie leben von einer guten Organisation. Obwohl sie nicht unbedingt automatisiert werden müssen, spielt die Automatisierung hier eine entscheidende Rolle. Das Volumen der Aufgaben, die hier zu bewältigen sind, lässt uns einfach keine Alternative.

Zur Automatisierung von funktionalen Tests werden allzu gern die so genannten GUI (Graphical User Interface) beziehungsweise Capture-Replay Tools eingesetzt. Diese Werkzeuge sind in der Lage, Aktionen eines Benutzers über die Benutzeroberfläche genau nachzubilden. Sie werden oft für die Testteams angeschafft, die fachliche Tests bisher manuell durchgeführt haben, um deren Effizienz zu steigern. Weniger bekannt scheint die Tatsache zu sein, dass diese Tools, genauso wie ihre Pendants im Last- und Performancebereich, ein besonderes Niveau an Know-how verlangen. Sie sind Programmiertools! Viele Fach- und Testspezialisten sind aber in der Regel keine Programmierer und wollen (sollen oder brauchen) auch keine werden.

Falsche Erwartungen

Praktisch alle GUI-Testtools haben die Fähigkeit, Testskripten automatisch aufzuzeichnen, während der Tester die Applikation bedient. Beim Abspielen der Skripts bildet das Tool die aufgezeichneten Aktionen exakt nach. Diese verführerische Eigenschaft verleitet schnell zu der Annahme, dass auch Leute ohne Programmierkenntnisse sie – mit etwas Schulung und Einarbeitung – beherrschen könnten.

Leider ist es mit der Aufzeichnung eines Skripts noch lange nicht getan. Alle, die es versucht haben, wissen, wie lange es dauert, bis ein Skript wirklich das tut, was man sich davon erhofft. Dabei ist die Wiederholbarkeit besser gesagt die Wiederausführbarkeit der Testskripte die Basis für alle weiteren Vorteile, die durch Automatisierung der funktionalen Tests zu erreichen sind.

Ein weiterer Mythos liegt in dem Glauben, dass die Testskripte mit minimalem Aufwand, so zu sagen „per Knopfdruck", wiederholbar sind. Testskripte kann man zwar per Knopfdruck starten, sie werden auf Dauer nur leider nicht laufen. Testskripte muss man wiederholbar machen und wiederholbar halten!

Unserer Ansicht nach sind daher falsche Erwartungen als „Killer Nr. 1" aller Testautomatisierungsversuche einzustufen.

Wartungsaufwand

Es gibt zwei Hauptgründe für die schlechte Wiederholbarkeit der Testskripte. Die Testabläufe sind entweder so komplex, dass sie von Grund auf schlecht wiederholbar sind, oder wir leiden an dem sogenannten Moving-Target-Syndrom.

Im ersten Fall haben wir einfach Pech gehabt – es ist nun mal komplex. Ob wir solche Tests erfolgreich automatisieren können, hängt von etwas Glück, passenden Tools und viel Geschick ab. Am Anfang unserer Automatisierungsversuche sollten wir am besten die Finger davon lassen.

Das Moving-Target-Syndrom andererseits ist ein organisatorisches Problem. Oft lässt es sich aber schwerer lösen als das erste. Auf unserer Liste steht es als „Killer Nr. 2". Worum geht es?

Damit sich ein Skript auszahlt, muß es oft wiederholt werden. Dafür muss man es früh aufsetzen. Je früher wir es aufsetzen, desto höher ist leider die Wahrscheinlichkeit, dass sich Teile der Applikation, die wir damit testen, ändern. Dies führt in der Regel zu Fehlern im Skript, die eigentlich keine Fehler sind, da die Applikation bewußt geändert wurde. GUI-basierte Tests sind eben anfällig für die Änderungen an der GUI!

Den Testern ist diese Abhängigkeit längst bekannt – vielen Managern, allen voran Projektmanagern, nicht. Wie lässt sich sonst erklären, dass bei so vielen Projekten an der Benutzeroberfläche dauernd herumgebastelt wird und sie erst kurz vor der Auslieferung fest steht? Packen wir das Problem direkt an der Wurzel.

Es wird oft gepredigt, die Endbenutzer in die Projektentwicklung stärker einzubinden, um Missverständnisse zu vermeiden beziehungsweise möglichst früh zu erkennen. Gleichzeitig wird gejammert, dass sich Endbenutzer und Entwickler so schlecht verstehen. Von alldem, was der IT-Branche bisher dazu einfiel, ist die Benutzeroberfläche nach wie vor eines der besten Mittel, um solchen Missverständnissen vorzubeugen. Warum in den Projekten die Benutzeroberfläche so spät stabilisiert wird, ist uns ein Rätsel. Technische Mittel und organisatorische Formen sind längst bekannt und verfügbar. Der „Schwarze Peter" wird aber meistens den Testern zugeschoben.

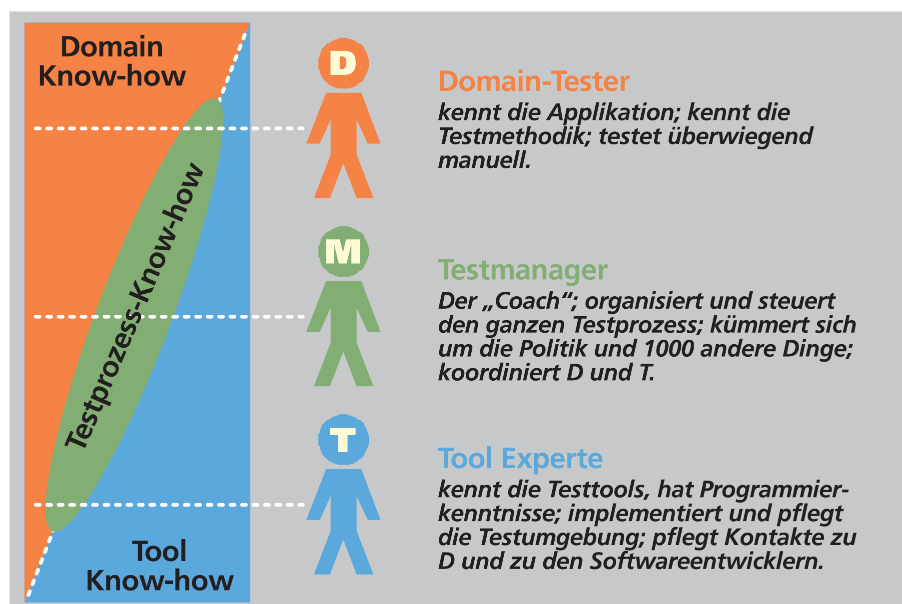

Bild 1: Die Rollenverteilung als Kernstück einer erfolgreichen Testorganisation.

Der Ausweg liegt in der Ausgewogenheit. Einerseits müssen alle gemeinsam aktiv mitwirken, um die Situation langsam zu verbessern. Andererseits muß man sich Mittel ausdenken, um mit den Änderungen leichter Schritt halten zu können. Was die Testskripte angeht, bleiben den Testern wenige Alternativen:

- sie werfen die Skripte weg (und bauen sie neu oder nie wieder auf),
- sie passen die Skripte (mühsam) an,
- sie bauen die Skripte um, so dass die Anpassung leichter wird.

Schlechte Erfahrungen

Einer Schätzung nach [1] verstauben 40 bis 50% aller Lizenzen für GUI-Testtools in den Regalen. Die Enttäuschung ist groß, wenn man feststellt, dass:

- in der Regel ein drei- bis zehnfacher Aufwand notwendig ist, um einen vergleichbaren manuellen Test zu automatisieren (damit er zum ersten Mal überhaupt läuft!),
- die Wartung aufgezeichneter Skripte oft umständlicher ist als deren Neuauflage,
- man viel vom Programmieren verstehen muss, um wartbare Testskripts zu erstellen,
- der Aufbau und die Betreuung automatisierter Tests keine Nebenaktivitäten sind,
- sich Testtools in der Regel erst in den Folgeprojekten richtig auszahlen.

Ein ROI, der kurz nach der Einführung von Testtools gesucht wird, lässt sich objektiv nur schwer ausrechnen und offenbart selten die tatsächlichen Probleme. Paradoxerweise lässt sich aber der Schaden, der durch den Ausfall von erfolgreich eingesetzten Testtools verursacht wird, wesentlich leichter ausrechnen. Leider bekommen jene, die sich zu früh die Finger verbrannt haben, selten die Chance es ausrechnen zu dürfen.

Ist es denn wirklich so schwer?

Ja und nein! Ja, weil es schwierig und teuer ist, eine gute Lösung von Grund auf zu entwickeln. Nein, weil es fertige Lösungen gibt und man keinesfalls von Null anfangen muss. Die Lösung muss natürlich auch zum Problem passen.

Man hat schon vor Jahren festgestellt, dass die effiziente Nutzung von GUI-Testtools immer wieder ähnliche Züge aufweist. Man fängt mit der Modifizierung der aufgezeichneten Skripte an und baut unangenehme Aspekte nacheinander aus bzw. um. Mit der Zeit entsteht ein System, das sich von den ursprünglich aufgezeichneten Skripten sehr unterscheidet. Obwohl im Detail oft stark von einander verschieden, sind die Hauptmerkmale in der Regel gleich:

- Aufzeichnung wird nicht zur Erstellung der Skripte genutzt, sondern nur um einzelne Aspekte zu prüfen,
- Testfälle werden in großem Umfang als datengesteuerte Tests entwickelt (das heisst Testdaten werden vom Testcode getrennt),
- eine Menge mehr oder weniger wiederverwendbarer Funktionen werden in den Bibliotheken gepflegt, um wiederkehrende Probleme besser behandeln zu können,
- viele andere Aspekte werden besonders behandelt wie etwa:
- Namenskonventionen,
- Programmierstandards (einschliesslich Kommentare),
- Testreporting und -protokollierung,
- Testregime (disziplinierte Testdurchführung),
- uvm.

Diese Entwicklung mündet im Laufe der Zeit in die Phasen, die gerne als „data-driven“ bzw. als „keyword-driven“ bezeichnet werden. Bei erster geht es um datengesteuerte Skripte. Bei zweiter um die so genannten „Testframeworks“.

Datengesteuerte Tests

Bei den datengesteuerten Tests handelt es sich um eine Technik zur Skripterstellung. Dabei ist es ausschlaggebend, dass die Testdaten vom Testcode getrennt werden. Dieser Stil entwickelt sich als natürliche Reaktion, nachdem man feststellt, dass viele automatisierte Tests oft einen sehr ähnlichen Ablauf haben. Die Unterschiede ergeben sich aus den Abweichungen in den Testdaten. Dabei bleibt die Testlogik so gut wie unverändert. Der unmittelbare Vorteil dieser Technik liegt in der Reduzierung des Codebestandes, der dadurch effizienter und wartbarer wird.

Eine viel weitreichendere Folge der Trennung von Testdaten und Testcode ist die Trennung von Kompetenzen. Man hat schnell erkannt, dass die Pflege der Testdaten, die jetzt aus dem Testcode in externen Dateien (Textdateien, Excel-Tabellen, Datenbanken) ausgelagert sind, keine nennenswerten Programmierkenntnisse verlangt. Sie können an fachliche Tester abgegeben werden. In der Tat, der Schlüssel für den Erfolg liegt in der Ausgewogenheit zwischen den Programmier- und den Testkenntnissen. Anders ausgedrückt: mit minimalem Programmieraufwand eine Plattform für viele Testfälle schaffen und pflegen.

Nun, wie so oft im Leben, steckt der Teufel im Detail. Datengesteuerte Tests sind zweifellos ein exzellentes Mittel – ja sogar die Voraussetzung – für eine effiziente Nutzung moderner GUI-Testtools. Aber sie sind kein Konzept, sondern nur eine Technik. Wenn man nicht aufpasst, werden sogar ähnliche Tests auf unterschiedliche Art und Weise implementiert. Außerdem ist es bei komplexen Vorgängen gar nicht so leicht, die Daten vom Code zu trennen.

Trotzdem stammen einige der elegantesten automatisierten Lösungen aus der Kategorie der datengesteuerten Tests. Sie gehören zum Standard-Repertoire jeder Testautomatisierung. Sie sind aber Insellösungen und daher schwer auf andere Testaufgaben oder auf andere Tester übertragbar. Auf der Suche nach dem „heiligem Gral“ wagen viele oft einen weiteren Schritt – den Schritt zum Testframework.

Testframeworks

Testframeworks sind datengesteuerte Testumgebungen, in denen Testdaten auf eine uniforme Art und Weise dargestellt und von den darunterliegenden Skripten ausgeführt werden. Die Betonung liegt auf „uniform“, denn der Wunsch ist allzu oft alle Testfälle auf die gleiche Art und Weise automatisieren zu wollen.

Die Testdaten werden so gestaltet, dass neben der reinen Datenübergabe auch die Steuerung der Skripte ermöglicht wird. Dahinter verbirgt sich der Wunsch, es Testern zu ermöglichen, ihre Testfälle in einer „Meta-Sprache“ artikulieren zu können ohne dafür programmieren zu müssen.

Der dritte – und einzig wirklich erreichbare – Wunsch ist, mit dem Testframework viele andere Aspekte einer Testumgebung einheitlich zu gestalten wie etwa Struktur, Namenskonventionen, Protokollierung, Dokumentation uvm.

Nehmen wir als Beispiel ein Modul für die Benutzerverwaltung einer Applikation X. Man kann sich leicht vorstellen, dass dort Aufgaben wie „Benutzer anlegen“, „Benutzer ändern“, „Benutzer löschen“, „Anmelden“, „Abmelden“ usw. eine Rolle spielen. Was fachliche Tester in solchen Fällen gerne tun, ist: sie legen Benutzer an, sie ändern sie, sie löschen sie, sie melden sich immer wieder an und ab und provozieren Fehler wo immer sie können. Damit sie das auch automatisiert machen können, müssen sich die Programmierer von Testframeworks einiges einfallen lassen.

Die gängigste Methode ist die Umwandlung der „Fachsprache“ in die „Testsprache“. Für Fachbegriffe wie „Benutzer anlegen“ entwirft man zunächst eine oder mehrere Testfunktionen, die die gewünschten Vorgänge durchführen. Damit Funktionen ähnliche Vorgänge mit unterschiedlichen Daten durchführen können, werden die variablen Informationen üblicherweise durch Parameter weitergegeben. In unserem Fall könnten die Funktionen etwa so aussehen:

```
BenutzerAnlegen( benutzerID, name, pwd, pwdBestaetigung, ...)
BenutzerAnlegen( benutzerID, name, pwd, pwdBestaetigung, ...)
BenutzerLoeschen ( benutzerID )
Anmelden ( benutzerID, password )
Abmelden ( )
```

Die Funktionen kümmern sich um viele Details, die den Testern „nach außen“ verborgen bleiben. Für die Tester wird eine eigene Meta-Sprache entworfen. Das Mindeste was diese Testsprache können muß, ist, die Funktionen mit einem vernünftigen Begriff anzusprechen und ihnen die Parameter zu übergeben. In unserem Fall könnte die Testsprache folgendermaßen aussehen:

```
BENUTZER_ANLEGEN <benutzerID> <name> <pwd> <pwd_bestaetigung>
BENUTZER_AENDERN <benutzerID> <name> <pwd> <pwd_bestaetigung>
BENUTZER_LOESCHEN <benutzerID>
ANMELDEN <benutzerID> <password>
ABMELDEN
```

Ein Test, der einen Benutzer anlegt und anschließend wieder löscht, kann so formuliert werden:

```
ANMELDEN    "administrator"   "topsecret"
BENUTZER_ANLEGEN    "hans"   "Hans Müller"    "xxx"    "xxx"
ABMELDEN
ANMELDEN    "hans"    "xxx"
ABMELDEN
ANMELDEN    "administrator"   "topsecret"
BENUTZER_LOESCHEN    "hans"
ABMELDEN
```

Das Testsystem im Hintergrund wird schon „wissen“, wie diese Vorgaben zu interpretieren sind. Man spricht in diesem Zusammenhang von den Keyword-Driven-Systemen. Die Steuerskripten lesen die Testdaten aus und rufen für jedes Schlüsselwort (zum Beispiel ANMELDEN) die passende Funktion mit den entsprechenden Parametern auf. Die Funktion führt dann ihre Dinge durch, erstellt darüber ein Protokoll und liefert den Indikator über Erfolg bzw. Misserfolg an das aufrufende Steuerskript zurück.

Der Vorteil dieser Technik liegt darin, dass die Terminologie den Testern vertraut klingt und eine Art der Modularisierung verlangt, die der des systematischen Testens entgegenkommt. Darüber hinaus können Testfälle relativ „neutral“ ausgedrückt werden. Man kann sich also vorstellen, dass die Funktion für „BENUTZER_ANLEGEN“ den tatsächlichen Vorgang sowohl über die Benutzeroberfläche als auch off-line oder auf irgend eine andere Weise durchführen könnte. Weiterhin muss auch überlegt werden, wie gültige und ungültige Daten (also Daten, die Fehler provozieren sollen) behandelt werden und wie die Reaktionen des Systems überprüft werden.

Nun, auch hier liegt der Teufel im Detail. Der größte Feind eines Testframeworks sind überambitionierte Testprogrammierer. Die Komplexität eines solchen Systems wächst sehr schnell. Es muß zunächst eine Menge an Testkomponenten implementiert werden, damit überhaupt etwas läuft. Nicht wenige Testframeworks sind bereits gestorben, bevor sie auch nur einen einzigen Test durchführen konnten. Der Entwurf eines Testframeworks ist ein Softwareprojekt! Die Tatsache, dass wir ihn als eine Testaktivität durchzuschmuggeln versuchen, erhöht keinesfalls seine Erfolgschancen. Was gefragt ist, ist viel Können und sehr viel Feingefühl!

Einfach ist es wirklich nicht!

Das Kernproblem liegt in einem nicht trivialen Widerspruch. Einerseits versuchen wir die Testfälle so zu artikulieren, dass fachliche Tester damit etwas anfangen können. Andererseits versuchen wir, die Wartbarkeit unserer Skripte zu steigern. Je mehr wir uns an die „fachlichen Konzepte“ herantasten, um so weiter entfernen wir uns von den Problemen, die wir eigentlich bekämpfen müssen. Ein Testtool bleibt nicht stehen, weil die Transaktion „BENUTZER_ANLEGEN“ nicht ausgeführt werden konnte, sondern, weil ein Push-Button nicht mehr da ist, wo er einmal war, oder ein Menü-Item plötzlich anders heißt oder ein Fenster zu spät erschienen ist, oder ...

Es ist daher kein Wunder, dass sich Hersteller vieler Testtools nicht trauen, eigene Frameworks zu entwickeln. Damit würden sie ihre Tools möglicherwei-

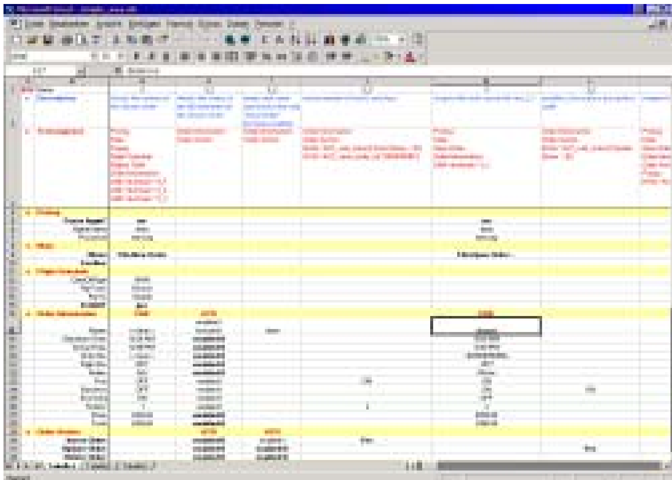

Bild 2: Die Definition von Testfällen im EMOS Framework.

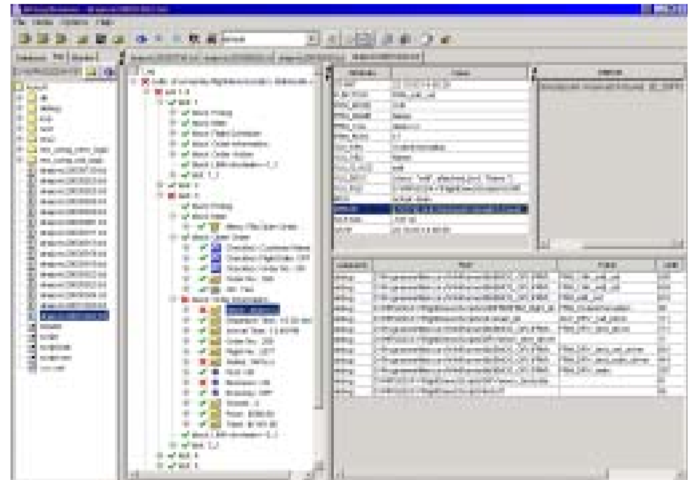

Bild 3: Die Testprotokolle von EMOS Framework-Tests.

se nur einschränken. Sie bleiben weiter auf dem sicheren Terrain von Push-Buttons, Edit-Feldern und Listboxen, denn höhere Konzepte mit dem gleichen Wiederverwendungswert sind sehr schwer zu finden.

So absurd es klingt – für einen effizienten Einsatz von GUI-Testools erweisen sich Testframeworks beziehungsweise „höhere Konzepte" immer wieder als unerlässlich. Die Toolhersteller machen aber nicht mit, so dass wir uns unsere Frameworks weiterhin selbst ausdenken müssen.

Muß es denn so sein?

Wir haben die Hoffnung nicht aufgegeben, dass die Toolhersteller diesbezüglich mehr tun werden. Bis dahin bleibt uns nichts anderes übrig, als weiterhin zu improvisieren und sich genau anzusehen, was andere schon versucht haben. Es wurde über Testframeworks viel geschrieben [1, 2, 3]. Berichte über erfolgreiche Projekte und Tipps was man beachten muß, gibt es ausreichend [4, 5, 6, 7, 8, 9]. Was es nicht gibt, ist Code.

Unternehmen, die ihre eigenen Frameworks entwickelt haben, geben sie ungern ab, ohne dafür Geld zu verlangen. Angesichts der dargestellten Problematik irgendwie verständlich. Testframeworks sind jedoch kein Allheilmittel. Sie lösen einige Probleme, produzieren dafür aber andere. Ob sie jeweils passen, muss genau untersucht werden. In der Regel sind auch Anpassungen notwendig. Uns fällt dazu kein besseres Modell ein, als das der Open Source-Gemeinde. Deswegen erwähnen wir hier zwei freie – konzeptionell ganz unterschiedliche – Frameworks für WinRunner.

SAFS (Software Automation Framework Support) wurde ursprünglich von Carl Nagle für Rational Robot (RRAFS) entwickelt und anschließend für WinRunner (WRAFS) portiert. SAFS ist ein Repräsentant der Frameworks, die sich an der Funktionsmodellierung, wie oben beschrieben, orientieren.

Weitere Informationen über SAFS sind unter http://safsdev.sourceforge.net/ zu finden.

EMOS Framework wurde 1999 von EMOS Computer entwickelt und seitdem ständig erweitert und betreut. Das zugrundeliegende Konzept weicht bewußt vom eingebürgerten (Funktions-) Modellierungsansatz ab und konzentriert sich ausschließlich auf die Modellierung der Benutzeroberfläche. Diese Vorgehensweise erlaubt eine wesentlich engere Verbindung zum eingesetzten Testtool (WinRunner) und löst damit elegant einige der erwähnten Konflikte. Weitere Informationen über das EMOS Framework sind unter **http://www.emos.de/ und http://groups.yahoo.com/group/EMOS_frame** zu finden.

Dipl.-Inf. Dean Rajovic / Dipl.-Ing. (FH) Robert Neulinger
drajovic@emos.de / rneulinger@emos.de

Literatur

1. Mark Fewster & Dorothy Graham, „Software Test Automation", Addision-Wesley, 1999
2. Elfride Dustin, Jeff Rashka & John Paul, „Automated Software Testing", Addison-Wesley, 1999
3. Daniel Mosley & Bruce Posey, „Just Enough Software Test Automation", Prentice Hall PTS, 2002
4. Cem Kaner, James Bach & Bret Pettichord, „Lessons Learned in Software Testing", John Wiley & Sons, 2001
5. Cem Kaner, „ Improving the Maintainability of Automated Test Suites", http://www.kaner.com/lawst1.htm
6. Karl E. Wiegers, „Read My Lips: No New Models", http://www.processimpact.com/articles/no_new_models.pdf
7. Carl Nagle, „Test Automation Frameworks", http://safsdev.sourceforge.net/DataDrivenTestAutomationFrameworks.htm
8. Bret Pettichord, „Seven Steps to Test Automation Success", http://www.io.com/~wazmo/papers/seven_steps.html
9. Keith Zambelich, „Totally Data-Driven Automated Testing" http://www.sqa-test.com/White_Paper.doc