Dr. Ernst Ambichl

(E-Mail: ernst.ambichl@microfocus.com)
ist Chief Scientist Continuous Quality Assurance Strategies bei Micro Focus.

Verfahren zur Optimierung der Testbarkeit bestehender Anwendungen für agile Entwicklungsteams

Dieser Artikel beschreibt Methoden zur Optimierung der Testbarkeit von Legacy-Anwendungen für die Fälle, in denen ein komplettes Redesign der Anwendung ausgeschlossen ist und spezielle Testschnittstellen nicht ohne Weiteres integriert werden können. In diesen Fällen muss zum Testen der Anwendung auf die bestehenden Anwendungsschnittstellen zur Steuerung und Kontrolle der Anwendung zurückgegriffen werden. Anhand einer konkreten Fallstudie wird im Folgenden gezeigt, wie solche Optimierungen in einer bestehenden, komplexen (keiner „Hello World"-Anwendung) Anwendung vorgenommen werden können und welche Vorteile durch die so verbesserte Testbarkeit entstehen. Der Artikel beleuchtet Testbarkeitsoptimierungen im Hinblick auf Funktionstests und im Hinblick auf Performance- und Lasttests und zeigt auf, welche Änderungen vorgenommen werden müssen, um beidem gerecht zu werden. Schließlich setzt der Artikel auch den Aufwand, der bei Verbesserung der Testbarkeit entsteht und die Mehrwerte, die durch eine verbesserte Testabdeckung, Testautomatisierung und eine verminderte Testwartung entstehen, in Relation zueinander. In der Fallstudie wird auch dargestellt wie agile Entwicklungsprozesse die erfolgreiche Realisierung von Testbarkeitsoptimierungen in der Anwendung unterstützen.

Definition von Testbarkeit

Zum Begriff Testbarkeit gibt es viele Definitionen – einige sind sehr allgemein und vage, andere sehr spezifisch und in unserem Kontext nicht zutreffend. Aus dem Blickwinkel der Testautomatisierung habe ich die folgende nützliche Definition gefunden, die Testbarkeit als Transparenz und Steuerung definiert: *Die Transparenz ermöglicht es, Metriken wie Status, Durchsatz, Ressourcenauslastung und andere Nebenerscheinungen der zu testenden Software zu beobachten. Die Steuerung ermöglicht es, Eingaben vorzunehmen und spezielle Stati herbeizuführen. Im Wesentlichen heißt Testbarkeit, angemessene Schnittstellen zum Steuern und Verifizieren von Software bereitzustellen* [PET02].

Testbarkeit – oft vernachlässigt in der Anwendungsarchitektur

Die Wahrscheinlichkeit, dass Entwicklungsteams daran denken und gewillt sind, Testschnittstellen bereitzustellen, unterliegt starken Schwankungen [PET02].

Wenn Sie beim Entwurf Ihrer Anwendung die Testbarkeit von vorneherein mitberücksichtigen, optimieren Sie das Design, die Modularisierung und vermindern die Kopplung von Softwaremodulen. Bei modernen Entwicklungsansätzen wie z. B. Extremprogrammierung (Extrem Programming) oder testgesteuerter Entwicklung (Test Driven Development) ist die Testbarkeit ein Grundpfeiler für ein gutes Anwendungsdesign.

Testbarkeit galt jedoch leider bei vielen bestehenden Anwendungen nicht von Anfang an als wichtiges Projektziel. Bei solchen Anwendungen ist es oft unmöglich, neue spezielle Testschnittstellen zu integrieren.

Testschnittstellen müssen im ursprünglichen Design und in der ursprünglichen Architektur der Anwendung vorgesehen sein. Sie in eine bestehende Anwendung zu integrieren, kann schwerwiegende Änderungen der Architektur nach sich ziehen, die oft dazu führen, dass größere Teile der Anwendung neu geschrieben werden müssen und dabei Kosten entstehen, die niemand übernehmen möchte. Testschnittstellen brauchen außerdem eine Schichtenarchitektur. Eine weitere Schicht für Testschnittstellen einzufügen, ist bei monolithischen Anwendungsarchitekturen oftmals unmöglich.

Testschnittstellen für Performancetests müssen in der Lage sein, eine Anwendung auch unter Mulituser-Aspekten hinreichend testen zu können. Das kann eine sehr komplexe Aufgabe werden, da dafür normalerweise eine remotefähige Testschnittstelle benötigt wird. Spezielle Testschnittstellen für Performancetests sind also noch seltener zu finden als Testschnittstellen für Funktionstests.

Um bei Anwendungen, in deren Design Testbarkeitskriterien nicht berücksichtigt wurden, Tests automatisieren zu können, müssen bestehende Anwendungsschnitt-

stellen zum Testen genutzt werden. In den meisten Fällen bedeutet das, dass die grafische Benutzerschnittstelle (GUI) für Funktionstests und die Protokollschnittstelle (wie z. B. HTTP/HTML, HTTP/SOAP, HTTP/XML, HTTP/JSON) oder die Remote-Anwendungsprogrammierschnittstelle (API) (wie z. B. RMI, JDBC, CORBA) für Performancetests genutzt werden. Auf diesen Ansatz greifen herkömmliche Funktions- und Lasttesttools zurück.

Wenn man allgemein verfügbare Anwendungsschnittstellen zum Testen nutzt, kann dies natürlich zu Problemen führen und Sie werden zahlreiche Belege dafür finden, besonders wenn Sie Artikel zum Thema agile Testprozesse lesen:

- Bestehende Anwendungsschnittstellen sind normalerweise nicht fürs Testen ausgelegt. Wenn sie zum Testen genutzt werden, werden sie ggf. so genutzt, wie der bestehende Client der Schnittstelle, diese Schnittstelle niemals nutzen würde. Wenn Tests über diese Schnittstellen durchgeführt werden, kann dies zu einem unvorhersehbaren Anwendungsverhalten führen. Bestehende Anwendungsschnittstellen bieten oft auch nur ungenügende Möglichkeiten zur Steuerung und Verifizierung der Ausgaben der Anwendung.
- Besonders GUIs mit Custom Controls (GUI-Steuerelemente werden vom Testtool nicht automatisch erkannt) sind problematisch, da sie nur eine begrenzte Transparenz und Steuerung für das Testtool bieten.
- Darüber hinaus bieten GUI-Steuerelemente mit schlecht ausgeprägten UI-Objekterkennungsmechanismen (wie Bildschirm- oder Fensterkoordinaten, UI-Indizes oder Captions) nur eine begrenzte Steuerbarkeit, besonders im Hinblick auf Wartungsaspekte. Dies führt zu instabilen Tests, die jedes Mal fehlschlagen und geändert werden müssen, wenn die Benutzerschnittstelle (UI) der Anwendung sich minimal verändert.
- Protokollschnittstellen, die für Performancetests genutzt werden, weisen vergleichbare Probleme auf. Daten zum Status der Arbeitssitzung bzw. zur Steuerung bleiben oftmals in undurchsichtigen Datenstrukturen verborgen, die für Testzwecke völlig ungeeignet sind. Oft ist auch die semantische Bedeutung des Protokolls nicht dokumentiert.

Dennoch muss der Einsatz von bestehenden Schnittstellen im Vergleich zu speziellen Testschnittstellen nicht automatisch die Effizienz und Effektivität beeinträchtigen. Oft führen schon kleinste Veränderungen in den Schnittstellen zu einer signifikant verbesserten Testbarkeit. Und wie bereits erwähnt, oft ist das Zurückgreifen auf die bestehenden Anwendungsschnittstellen, die einzige Möglichkeit bestehende Anwendungen zu testen.

Testbarkeit optimieren durch Modifizierung bestehender Schnittstellen mit Testability-Hooks

Ein Dreh- und Angelpunkt bei der Optimierung der Testbarkeit über bestehende Schnittstellen, ist die Fähigkeit, Steuerelemente der Anwendung anhand von stabilen Bezeichnern zu benennen und zu unterscheiden [PET02]. Oft sind es die fehlenden stabilen Bezeichner für Steuerelemente der Anwendung, die uns das Leben als Tester so schwer machen.

Ein stabiler Bezeichner für ein bestimmtes Steuerelement bedeutet, dass der Bezeichner für ein Steuerelement immer gleich bleibt – sowohl für mehrere Aufrufe des Steuerelements als auch über verschiedene Versionen der Anwendung selbst hinweg. Ein stabiler Bezeichner muss darüber hinaus auch im Anwendungskontext eindeutig sein, d. h. dass kein weiteres Steuerelement über denselben Bezeichner zur gleichen Zeit ansteuerbar sein darf.

Das bedeutet nicht notwendigerweise, dass Sie Bezeichner nach dem GUID-Ansatz verwenden müssen, die in jedem vorstellbaren Kontext eindeutig sind. Bezeichner für Steuerelemente sollten verständlich sein und sinnvolle Namen ergeben. Durch die Einhaltung von Namenskonventionen für diese Bezeichner wird die Zuordnung des einzelnen Bezeichners zum jeweiligen Steuerelement erheblich erleichtert.

Die Verwendung der Steuerelementhierarchie führt dazu, dass einzelne Steuerelemente nur schwer identifizierbar sind. Dies wird durch den Einsatz von definierten Bezeichnern vermieden. Steuerelementhierarchien kommen oft dann zum Einsatz, wenn schwache Steuerelementbezeichner, die im aktuellen Kontext der Anwendung nicht eindeutig sind, verwendet werden. Wenn Sie Steuerelemente hierarchisch ordnen, beschreiben Sie einen „Weg“, der Sie zum einzelnen Steuerelement führt. Wenn Sie sich nun darauf verlassen, Steuerelemente allein über diesen Weg zu identifizieren, führt dies dazu, dass Steuerelemente nur über andere Steuerelemente identifiziert werden können und wenn die Hierarchie sich ändert, sehen Sie sich einem gewaltigen Wartungsaufwand gegenüber.

Eine der einfachsten und wirksamsten Methoden, die Testbarkeit Ihrer Anwendungen zu optimieren, ist stabile Steuerelementbezeichner einzuführen und diese über bestehende Schnittstellen Ihrer Anwendung bereitzustellen. Diese Methode funktioniert nicht nur bei Funktionstests über die GUI, sondern kann auch auf Performancetests über die Protokollschnittstelle übertragen werden. Im Folgenden wird beschrieben, wie Sie dies im konkreten Fall umsetzen.

Fallstudie – Optimierung der Testbarkeit einer Unternehmensanwendung

Einer der Schwerpunkte bei der Arbeit an unserer neuen Version unserer Test-Management-Software war, die Anwendung für den unternehmensweiten, globalen und verteilten Einsatz zu verbessern. Daraus ergaben sich zwei Schlüsselaufgaben:

- Eine lokalisierte Version der Anwendung bereitzustellen.
- Die Performance und Skalierbarkeit der Anwendung in verteilten Umgebungen deutlich zu erhöhen.

Unsere Anwendung ist eine webbasierte Multi-User-Anwendung mit einem HTML/AJAX Frontend, das auf einer Java EE-Infrastruktur mit einem Datenbank-Backend basiert. Es dauerte mehrere Jahre, die Anwendung zu entwickeln und heute umfasst sie mehrere hunderdtausend Codezeilen.

Um eine gute Testabdeckung für alle lokalisierten Versionen der Anwendung zu garantieren, war es unser Ziel, lokalisierte Versionen der Anwendung ohne – oder wenn nur mit minimalen – Änderungen der Funktionstestskripte, die wir für die englische Version des Produktes einsetzen, testen zu können.

Darüber hinaus wollten wir die Skalierbarkeit und Performance der bestehenden Funktionalität der Anwendung über den gesamten Entwicklungszyklus hinweg nachts durch routinemäßige Tests auf

Build-Basis testen können. Diese routinemäßigen Performancetests sollten es uns ermöglichen, Performancerückgänge so früh wie möglich identifizieren zu können. Neue Funktionen sollten direkt nach der Verfügbarkeit der Funktion auf Performance getestet werden – auch in Kombination mit bestehenden Funktionen, um sicher zu gehen, dass die Anforderungen an Performance und Skalierbarkeit erfüllt werden. Die regelmäßige Durchführung von Performancetests sollte uns in die Lage versetzen, über den gesamten Verlauf zu messen, in wie weit wir die definierten Skalierbarkeits- und Performanceziele erreichen.

Optimierte Testbarkeit bei Performancetests

Problemstellungen – Fehleranfällige Performancetestskripte

Wenn man den Anteil automatisierter Performancetests erhöhen möchte, sind stabile und leicht zu wartende Performancetestskripte von großer Bedeutung. Bei Vorgängerversionen mangelte es den Performancetestskripten deutlich an Stabilität und Wartbarkeit. Es kam häufig vor, dass minimale Änderungen der Anwendung Fehler in der Ausführung der Performancetestskripte verursachten. Es war sehr mühsam die Ursachen der fehlerhaften Testausführungen zu finden. Anstatt die Ursachen wirklich zu suchen, wurden Performancetestskripte neu aufgenommen und neu konfiguriert. Komplexe Testfälle bildeten wir nicht ab (im speziellen Szenarien, die Daten ändern), da sie zum einen schwer zu entwickeln waren und zum anderen die Gefahr groß war, dass sie Änderungen der Anwendung zu Änderungen in den Testskripten führen werden. Darüber hinaus war die Anpassung der Tests kompliziert und machte die Tests im Ergebnis noch fehleranfälliger.

Um besser verstehen zu können, warum die Anwendung so schwer zu testen war, müssen wir uns der Anwendungsarchitektur und den Schnittstellen, die wir um Testen einsetzen, zuwenden.

Bei webbasierten Anwendungen werden Performancetests am häufigsten auf der HTTP-Protokollebene abgebildet. Werfen wir also einen Blick auf das HTTP-Protokoll der Anwendung.

Da unsere Anwendung hochgradig dynamisch ist, sind auch die HTTP-Anfrage (Request)und -Antwort (Response) dynamisch – d. h., dass sie sowohl dynamische Sitzungsdaten, als auch dynamische Daten zu den aktiven UI-Steuerelementen enthalten. Während Sitzungsdaten (wie bei vielen anderen Webanwendungen) über Cookies transportiert werden und daher automatisch von den meisten HTTP-basierten Testtools behandelt werden, werden Daten über Aktionen und Steuerelemente in Form von URL-Parametern im HTTP Request oder im Http-Nachrichtenkörper (Body) bei HTTP-POST Operationen wie folgt abgebildet:

Jede Anfrage, die eine Aktion auf einem Steuerelement auslöst (z. B. ein Tastendruck, Anklicken eines Links, Expandieren eines Baumstruktur-Steuerelements) nutzt einen *Steuerelement-Handler*, um das Steuerelement auf dem Server zu identifizieren:

```
http://host/borland/mod_1?control_handle
=68273841207300477604!...
```

Auf dem Server dient dieser Steuerelement-Handler dazu, den eigentlichen Steuerelementvorgang und die aufgerufene Aktion zu identifizieren. Da der Handle sich auf die jeweilige Instanz des Steuerelementes bezieht, ändert sich der Wert des Handles nach jeder Serveranfrage, da mit ihr eine neue Instanz desselben Steuerelementes erstellt wird. Die einzige Information, die der Server über seine Schnittstelle preisgibt, ist der Handle auf die Instanz des Steuerelements. Im Hinblick auf die Funktionalität einer Anwendung ist das auch völlig in Ordnung – im Hinblick auf Testtools und Tester allerdings, ist das ein wahrhafter Albtraum.

Welche Auswirkungen hat das auf die Identifizierung von Aktionen von Steuerelementen in einem Testtool?

Da die Anfragen dynamisch sind (Handles ändern sich fortlaufend), wird ein aufgezeichnetes, statisches HTTP-basiertes Testskript niemals funktionieren, weil es Handles aufruft, die nicht mehr gültig sind. Daher müssen Tester also zu allererst statische Handles im Testskript durch dynamische Handels ersetzen, die vom Server während der Laufzeit erstellt werden. Mit ausgereiften Performancetestlösungen, wie z. B. SilkPerformer, können Sie Parsing- und Austauschregeln festlegen, die diese Aufgabe für Sie übernehmen und die notwendigen Änderungen im Testskript automatisch vornehmen.

So parsen Sie den dynamischen Steuerelement-Handler aus einer HTTP-Antwort:

```
WebParseDataBound(<dynamic control
handle>, "control_handle=", "!" …)
```

Die Funktion "WebParseDataBound" parst die Antwortdaten einer HTTP-Anfrage, indem sie "control_handle=" als linke Begrenzung und "!" als rechte Begrenzung definiert und das Ergebnis in <dynamic control handle> returniert. Wenn Sie SilkP erformer kennen, wird Ihnen die oben stehende Darstellungsart vielleicht bekannt vorkommen, da sie der API von SilkPerformer ähnelt.

Der dynamische Steuerelement-Handle in einer HTTP-Anfrage:

```
http://host/borland/mod_1?control_handle
=<dynamic control handle>!...
```

Anwendungen haben natürlich viele verwendbare Steuerelemente auf einer Seite, die in unserem Fall alle das oben dargestellte Format ("control_handle=") zum Spezifizieren des Steuerelement-Handle benutzen. Man kann also nur durch den Vergleich des aktuellen Wertes des Steuerelement-Handle mit anderen Steuerelement-Handle in den Antwortdaten (die in unserem Fall entweder in HTML/JavaScript oder JSON sind) feststellen, welcher Steuerelement-Handle der des betreffenden Steuerelements ist.

Ein Beispiel: Der Steuerelement-Handle für die Login-Schaltfläche auf der Login-Seite ist der 6. Steuerelement-Handle auf der Seite. Für die Login-Schaltfläche sieht die Parsing-Anweisung zum Parsen des Wertes des Handles für die Login-Schaltfläche z. B. so aus:

```
WebPage("http://host/borland/login/");
WebParseDataBound(hBtnLogin,
"control_handle=", "!", 6);
```

Um die eigentliche Login-Schaltfläche anzusprechen, beispielsweise so:

```
WebPage("http://host/borland/mod_1?
control_handle=" + hBtnLogin + "!");
```

Schon dieses kurze Beispiel verdeutlicht, dass die Nutzung des Steuerelement-Handle zur Identifizierung von Steuerelementen alles andere als eine stabile Erkennungsmethode ist. Dieser Ansatz birgt die folgenden Probleme:

- *Schlechte Lesbarkeit:* Skripte sind schwer lesbar, da sie Steuerelemente anhand der Anordnung des Steuerelements (z. B. sechstes Steuerelement in der Seite) identifizieren.
- *Fehleranfällig bei Änderungen:* Das Hinzufügen von neuen Steuerelementen oder auch nur die Neuanordnung von Steuerelementen führt zur fehlerhaften Ausführung des Tests oder löst im schlimmsten Fall unerwartetes Anwendungsverhalten aus, das Folgefehler provoziert, die schwer zu finden sind.
- *Schlechte Wartbarkeit:* Skripte müssen häufig geändert werden. Änderungen in der Anordnung oder Anzahl der Steuerelemente sind nur unter großem Aufwand aufzudecken, da man alle Antwortdaten durchsuchen muss, um die neue Anordnung des Steuerelemente zu finden.

Die Lösung – Stabile Steuerelementbezeichner

Unser Versuch, die Erkennungsmethode innerhalb der Testskripte durch intelligentere Parsing-Regeln zu optimieren (anstatt lediglich nach dem Vorkommen von „control_handle“ zu suchen), stellte sich als nicht erfolgreich heraus. Letztlich sogar als kontraproduktiv, denn je eindeutiger wir die Parsing-Regeln gestalteten, desto instabiler wurden sie.

Daher entschlossen wir uns, das Problem an seiner Ursache anzugehen und stabile Bezeichner (Control Identifier oder kurz CID) für Steuerelemente einzuführen. Hierfür mussten wir natürlich den Code der Anwendung ändern, doch dazu kommen wir später.

Damit CIDs dem Testtool zugänglich gemacht werden konnten, erweiterten wir das auf HTTP basierende Protokoll der Anwendung. Da wir den CIDs sinnvolle Namen zugeordnet haben, können die Tester Steuerelemente sowohl in Anfrage- und Antwortdaten als auch in Testskripten leicht erkennen. Darüber hinaus können Entwickler nun auch CIDs schnell und einfach dem Code zuordnen, der das Steuerelement implementiert. CIDs dienen einzig und allein dazu, dem Testtool, den Tester und den Entwicklern mehr Kontextdaten zur Verfügung zu stellen. Das Anwendungsverhalten ändert sich dadurch nicht – die Anwendung nutzt noch immer den Steuerelement-Handle und ignoriert den CID.

Das HTTP-Anfrageformat änderte sich von:

```
http://host/borland/mod_1?control_handle
=68273841207300477604!...
```

auf:

```
http://host/borland/mod_1?control_handle
=*tm_btn_login.68273841207300477604!...
```

Da CIDs als Teil ihres Steuerelement-Handle enthalten sind, kann man ohne weiteres eine Parsing-Regel erstellen, die den Steuerelement-Handle auf eindeutige Weise parst.

Damit sieht das Skript von vorhin nun so aus:

```
WebPage("http://host/borland/login/");
WebParseDataBound(hBtnLogin, "control_
handle=*tm_btn_login.","!", 1);
WebPage("http://host/borland/mod_1?
control_handle==*tm_btn_login." +
hBtnLogin + "!");
```

Durch die Einführung der CIDs und dem Herausfiltern der CIDs auf HTTP-Protokollebene, sind wir nun in der Lage, absolut stabile Testskripte zu entwickeln. Da CIDs bestehender Steuerelemente sich nicht ändern, fällt kein Wartungsaufwand an, wenn Änderungen vorgenommen werden, wie z. B. beim Einfügen neuer Steuerelemente auf einer Seite, beim Einfügen dynamischen Inhalts, der Links zu Steuerelement-Handlers enthält (z. B. eine Linkliste) oder beim Umstrukturieren von Steuerelementen auf einer Seite. Auch sind Skripte jetzt lesbar und Tester und Entwickler können sich besser verständigen, da sie dieselben Begriffe verwenden, wenn sie sich über die Steuerelemente der Anwendung austauschen.

Kommen wir nun zu den Änderungen, die wir zur Optimierung der Funktionstests vorgenommen haben.

Optimierte Testbarkeit bei Funktionstests

Problemstellung – Sprachabhängige Testskripte

Unsere bestehenden Funktionstestskripte stützten sich hauptsächlich darauf, GUI-Steuerelemente und Fenster anhand ihrer *Captions*1*) zu erkennen.

**1) Zum Beispiel: Für Schaltflächen ist die Caption der angezeigte Name der Schaltfläche, für Links der angezeigte Text des Links, für Textfelder der Text-Label, der dem Textfeld vorausgeht.*

Um die Funktionstests der lokalisierten Versionen der Anwendung automatisieren zu können, mussten wir die Abhängigkeiten zwischen den Testskripten und den verschiedenen lokalisierten Versionen der Anwendung weitestgehend minimieren. Captions sind bekanntlich sprachabhängig und eignen sich daher ganz und gar nicht als stabile Steuerelementbezeichner.

Eine Möglichkeit wäre gewesen, die Testskripte selbst zu lokalisieren, die Captions in eigene Dateien zu exportieren und lokalisierte Versionen der exportierten Captions bereitzustellen. Dieser Ansatz hätte jedoch einen großen Wartungsaufwand mit sich gebracht, nämlich dann wenn Captions geändert oder neue Sprachen eingebunden werden müssen.

Identifizierung eines HTML-Links (HTML-Fragment einer HTML-Seite) anhand von Captions:

```
<A …  HREF=" http://...control_handle
=*tm_btn_login.6827..." >Login</A>
```

Ein Aufruf des eigentlichen Logins könnte dann so aussehen:

```
MyApp.HtmlLink("Login").Click();
```

Da wir das Konzept stabiler Steuerelementbezeichner (CIDs) bereits für Performancetests etabliert hatten, wollten wir diese Bezeichner auch für Tests auf GUI-Ebene verwenden. Durch die Verwendung der CIDs werden Testskripte sprachenunabhängig und müssen daher nicht lokalisiert werden (zumindest nicht für die Erkennung von Steuerelementen – die Verifizierung von Steuerelementinhalten kann ggf. sprachenabhängig bleiben). Damit unsere Funktionstestlösung (*2) auf den CID zugreifen konnte, erweiterten wir die Eigenschaften der HTML-Steuerelemente unserer Anwendung um ein anwendungsspezifisches HTML-Attribut namens „CID“. Dieses Attribut wird vom Browser ignoriert, ist aber für unsere Funktionstestlösung über das DOM des Browsers zugänglich.

**2) Wir setzten SilkTest für unsere Funktionstests ein. Die Lösung kann HTML-Steuerelemente mittels eines konfigurierten HTML-Attributs erkennen.*

Identifizierung eines Links anhand des CID:

```
<A …  HREF=" http://...control_handle
=*tm_btn_login.6827..."
CID="tm_btn_login" >Login</A>
```

Ein Aufruf des eigentlichen Logins über den CID könnte dann so aussehen:

```
MyApp.DomLink("CID=tm_btn_login").
Click();
```

Übertragung der verbesserten Testbarkeit auf bestehende Testskripte

Da wir bestehende Funktionstestskripte hatten, in die wir die neuen Mechanismen zur Identifizierung von Steuerelementen übertragen mussten, war es absolut notwendig, dass wir die Deklaration der UI-Steuerelemente und die Methode, über die die Testskripte die Steuerelemente erkennen, separat behandelten. Wir mussten also lediglich die Deklaration der Steuerelemente ändern, jedoch nicht die Art und Weise, in der die verschiedenen Testskripte auf sie zugriffen (*3).

**3) Die Datenkapselung von spezifischen Daten der UI-Steuerelemente ist ein Konzept, das von SilkTest optimal unterstützt wird. Das unten stehende Skriptfragment ist SilkTest in der Darstellungsart sehr ähnlich.*

Skript das Daten von Steuerelementen von Aktionen abkapselt:

```
// declaration of the control BrowserChild
MyApp { …  DomLink Login {
locator "CID=tm_btn_login"  // before: tag
"Login"     …  } }  // action on the
control MyApp.Login.Click();
```

Funktioniert das auch außerhalb von Webanwendungen?

Dieser Ansatz ist in ähnlicher Weise auch auf andere GUI-Toolkits übertragbar – z. B. auf Java SWT, Java Swing/AWT und auf Toolkits, die MSUIA (Microsoft User Interface Automation) oder Adobe Flex Automation unterstützten. Entwickler, die mit diesen GUI-Toolkits arbeiten, können ein eigenes Attribut (Custom Attribute) zu Steuerelementen hinzufügen, bzw. diese Toolkits stellen ein bestimmtes Attribut (Automation Attribute) zur Verfügung. Dieses Attribut kann dann verwendet werden, um CIDs für Testtools zugänglich zu machen. Natürlich müssen Sie vorher sicherstellen, dass Ihr GUI-Testtool mit Custom Attributes bzw. Automation Attributes des jeweiligen Toolkits arbeiten kann.

Code-Änderungen durch das Hinzufügen von Testability-Hooks

Notwendige Änderungen bei Funktionstests

Eine der bemerkenswertesten Erfahrungen dieses Projekts war, wie einfach es sich gestaltete, die Testability-Hooks in den Anwendungs-Code zu integrieren. Als wir mit den Entwicklern des UI-Frameworks der Anwendung sprachen und Ihnen erklärten, dass wir CIDs zur Erkennung der Steuerelemente einsetzten müssten, fanden sie sofort heraus, dass die Programmierschnittstellen des Frameworks bereits über die Funktion verfügten, individuelle HTML-Attribute zu den Steuerelementen des UI-Frameworks zu hinterlegen. So mussten wir nicht einmal den Code des UI-Frameworks ändern, um CIDs für Funktionstests einführen zu können. Selbstverständlich kam aber einige Arbeit auf uns zu, als wir die CIDs für jedes UI-Steuerelement der Anwendung erstellten. Doch wir näherten uns der Aufgabe Schritt für Schritt und führten zunächst die CIDs für die Teile der Anwendung ein, die wir schwerpunktmäßig testen wollten.

Notwendige Änderungen im Anwendungs-Code, um einen CID einzufügen:

```
Link login = new Link("Login");
login.addAttribute("CID", "tm_btn_login");
// additional code line to
// create CID the control
```

Notwendige Änderungen bei Performancetests

Für Performancetests mussten wir das Protokoll der Anwendung so erweitern, dass es den CID zusätzlich zum Steuerelement-Handle beinhaltet. Nachdem die Entwickler des UI-Frameworks verstanden hatten, was wir brauchten, konnten Sie die Lösung sofort umsetzen. Auch hier waren die Änderungen minimal und mussten nur in einer Basisklasse des Frameworks vorgenommen werden.

Notwendige Änderungen im UI-Framework des Anwendungs-Codes, um CIDs in Steuerelement-Handlers einfügen zu können:

```
private String generateControlHandle() {
String cid = this.getAttribute("CID");   if
(cid != null)     return "*" + cid + "." +
this.generateOldControlHandle();  else
return this.generateOldControlHandle(); }
```

Die veränderte „generateControlHandle"-Methode, die in der Basisklasse aller UI-Steuerelementklassen implementiert ist, generiert nun den folgenden Steuerelement-Handle, der den CID als Teil des Handle enthält:

- *tm_btn_login.6827384 I 207300477604!

Gesammelte Erfahrungen

Die Kosten für die Verbesserung der Testbarkeit über bestehende Anwendungsschnittstellen waren minimal.

Eine der bemerkenswertesten Erfahrungen dieses Projekts war, wie einfach es sich gestaltete, die Testability-Hooks zu integrieren als wir den Entwicklern unser Problem darlegen konnten. Um die Testability-Hooks in die bestehenden Anwendungsschnittstellen (GUI und Protokoll) integrieren zu können, bedurfte es nur minimaler Änderungen im Anwendungs-Code. Wir mussten keine speziellen Testschnittstellen entwickeln, was größere Überarbeitungen der Anwendung bedeutet hätte. Auch die durch die Testability-Hooks verursachte Laufzeitverlängerung fiel nicht ins Gewicht. Der größte Aufwand entstand im Zusammenhang mit der Einführung der stabilen Bezeichner (CIDs) für alle relevanten Steuerelemente der Anwendung. Alle notwendigen Änderungen ergaben einen Aufwand von ca. 4 Mannwochen – das ist weniger als 1 % der Ressourcen die jährlich an dieser Anwendung arbeiten.

Die Vorteile waren überwältigend.

Der Wartungsaufwand für Testskripte sank in erheblichem Maße. Besonders im Hinblick auf Performancetests haben wir heute – zum ersten Mal – eine Reihe von wartbaren Tests. Früher mussten wir häufig Testskripte neu erstellen, wenn die Anwendung geändert wurde. Jetzt haben wir Performancetestskripte, die extrem stabil laufen und, wenn die Anwendung geändert wird, schnell und einfach angepasst werden können.

Auch die Änderungen in den bestehenden Funktionstestskripten, die wir vornehmen mussten, um die verschiedenen lokalisierten Versionen der Anwendung testen zu können, waren geringfügig. Hier kam uns zugute, dass wir bereits die Deklarationen der UI-Steuerelemente und die Methode, über die die Testskripte die Steuerelemente erkennen, separat behandelt hatten. Da wir

nun auch die lokalisierten Versionen der Anwendung automatisch testen können, konnten wir nicht nur die Testabdeckung vergrößern, sondern auch den Zeitaufwand für die Bereitstellung von lokalisierten Versionen der Anwendung reduzieren. Aktuell stellen wir alle lokalisierten Versionen gleichzeitig mit der englischen Version bereit.

Performancetests werden nun nachts routinemäßig mit verschiedenen Lasten bei verschiedenen Konfigurationen für jeden Build durchgeführt. So haben wir zu jeder Zeit einen Überblick wie die Performance und die Skalierbarkeit der Anwendung sich verbessern (oder verschlechtern). Darüber hinaus werden Programmänderungen, die die Performance oder Skalierbarkeit beeinflussen, sofort aufgedeckt. Da Performanceprobleme normalerweise extrem schwer zu finden sind, ist das ein enormer Vorteil. Wenn man weiß, wie die Performance sich zwischen zwei Builds verändert hat, kann man sich bei der Ursachensuche viel Zeit und Aufwand sparen.

Der Umstieg auf agile Entwicklungsprozesse diente als Auslöser.

Wenn es denn so einfach ist, die Testbarkeit bestehender Anwendungen zu verbessern und gleichzeitig die Testautomatisierung voranzutreiben und die die Qualität so deutlich zu optimieren, warum tut es dann nicht jeder?

Bei getrennten Entwicklungs- und Qualitätssicherungsabteilungen, wie in herkömmlichen Entwicklungsorganisationen üblich, findet der notwendige Austausch zwischen Entwicklern und Testern nicht statt. Für die Tester ist und bleibt die Anwendung eine „Back-Box“ und sie konzentrieren ihre Bemühungen darauf, die Anwendung von „außen“ zu testen. Für die Entwickler spielt Testbarkeit keine Rolle, da das Testen nicht ihr Problem ist und oftmals informieren sie die Tester nicht, über Schnittstellen, die beim Testen nützlich sein könnten. Auch wissen Entwickler oft nicht, was Tester brauchen und Tester wissen nicht, welche Möglichkeiten es gibt, ihnen ihre Aufgaben zu erleichtern.

Was in diesen herkömmlichen Entwicklungsorganisationen mit getrennten Qualitätssicherungs- und Entwicklungsabteilungen also häufig eine Verbesserung der automatisierten Testprozesse hemmt, ist die fehlende Kommunikation zwischen Testern und Entwicklern, die fehlende Verantwortungsbereitschaft der Entwickler, die fehlende Handlungskompetenz der Tester und die Tatsache, dass kein Wissenstransfer zum Thema Testbarkeit und Testautomatisierung stattfindet.

In den 15 Jahren, in denen ich an der Entwicklung von Funktions- und Performancetesttools arbeite, gibt es einen Satz, den ich von unseren Kunden immer wieder zu hören bekomme: „Wir können nicht unsere Anwendung ändern, damit sie mit Ihrem Tool besser testbar ist.“ Wenn Unternehmen nicht anfangen, diese Haltung zu überdenken, wird Testautomatisierung immer nur begrenzt erfolgreich sein.

In unserem Forschungs- und Entwicklungslabor führte der Übergang zu agilen Entwicklungsprozessen zu einem Umdenken. Damals bildeten wir SCRUM-Teams, die sich aus Testern und Entwicklern zusammensetzen und übertrugen dem Team die Verantwortung für die Qualitätssicherung und das Testen. Daraufhin entwickelte sich ein offener Dialog zwischen Testern und Entwicklern darüber, wie man die Anwendung besser testen könnte. Das Zusammenführen des Know-hows über die Testtools und ihre Funktionen und über die Anwendung und ihre Architektur, die direkte Kommunikation innerhalb des Teams und die gemeinsame Teamverantwortung führte innerhalb von kurzer Zeit zu den Optimierungen der Testbarkeit, die in diesem Artikel beschrieben sind.

Referenzen

[FEW1999] Mark Fewster, Dorothy Graham, Addison Wesley: Software Test Automation, 1999

[PET02] Bret Pettichord: Design for Testability, Paper, 2002

[KAN2002] Cem Kaner et. al.: Lessons Learned in Software Testing, John Wiley & Sons, 2002

[SAI2008] Agile Tester, Internet Blog: Why GUI tests fail a lot? (From tools perspective), http://developer-in-test.blogspot.com/2008/09/problems-with-gui-automation-testing.html