

Frameworks zum Testen von Oberflächen einsetzen

Gelungene Verbindung

Bei Akzeptanztests von Anwendungen kann das Open-Source-Framework FitNesse.NET durch Automatisierung den Aufwand erheblich reduzieren. Leider bringt es keine GUI-Bindung mit. Hier springt ein zweites Framework namens White ein und behebt das Defizit. Zusammen sind beide ein gutes Team und ermöglichen GUI-Tests von WPF-Anwendungen.

Auf einen Blick



Dr. Dehla Sokenou arbeitet als Senior Software Engineer bei Gebit Solutions. Neben der Entwicklung großer objektorientierter Softwaresysteme umfassen ihre Schwerpunkte modellgetriebenes Requirements Engineering und modellbasiertes Testen.



Erwin Tratar beschäftigt sich mit modellbasierter Softwareentwicklung. Er verbindet langjährige Praxis in Anwendungsentwicklung und Projektleitung mit seiner Erfahrung als Entwicklungsleiter für das MDD-Applikationsframework Trend.

Inhalt

- Übliche GUI-Testwerkzeuge erkennen Veränderungen an der Oberfläche oft nicht.
- FitNesse.NET kann dies, unterstützt jedoch keine GUI-Tests.
- Die Kombination aus FitNesse.NET und White gleicht dieses Manko aus.

dnpCode
A1005GUITest

Automatisierte GUI-Tests reduzieren den Aufwand in Softwareprojekten erheblich, besonders in agilen Entwicklungsprojekten und in Softwareprojekten, bei denen mehr als eine Version der Software ausgeliefert wird. Sie verhindern als Regressionstests, dass bereits funktionierende Programmteile durch neue Funktionen Fehler produzieren.

Als Werkzeuge für GUI-Tests dienen oft sogenannte Capture/Replay-Tools; sie zeichnen Abläufe auf der Benutzeroberfläche als Skripte auf, die sich zum Testen wieder abspielen lassen. Diese Skripte sind jedoch nicht besonders robust gegenüber Änderungen der Benutzeroberfläche, da sie direkt GUI-Ereignisse auslösen und GUI-Komponenten oft direkt adressieren. Daher lässt bereits ein Verschieben eines GUI-Elements von links oben nach rechts unten in der Benutzeroberfläche den Test fehlschlagen. Besser sind hier rein skriptgesteuerte Werkzeuge, die GUI-Elemente auf abstraktere Weise referenzieren. Dazu gehört zum Beispiel FitNesse [1], ein Framework für den Akzeptanztest speziell aus Kundensicht, entwickelt von Robert und Micah Martin und Michael Feathers; die Umsetzung des Frameworks auf .NET ist Mike Stockdale von Syterra Software zu verdanken [2].

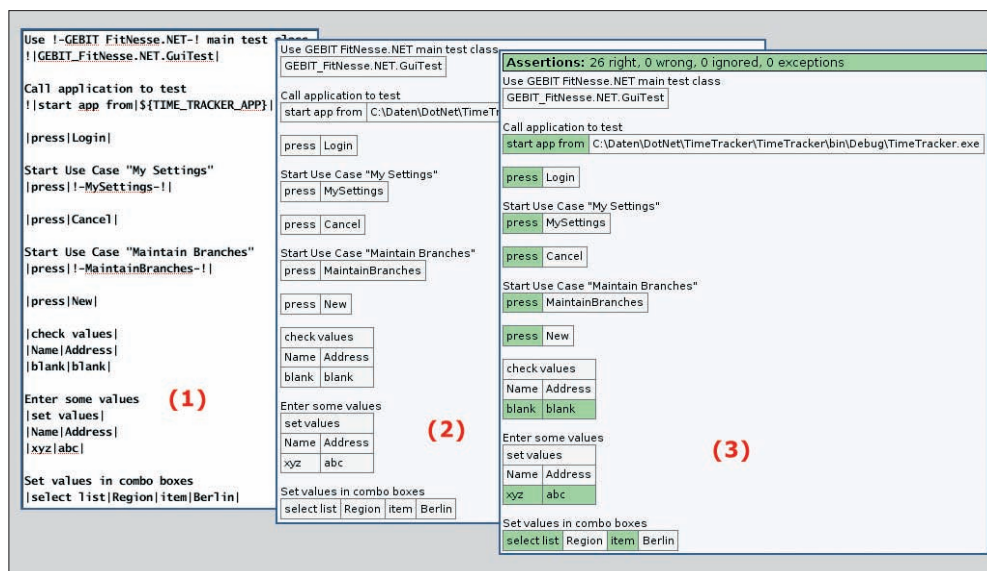
FitNesse selbst unterstützt GUI-Tests jedoch nicht; allgemein gibt es nur wenig Funktionen, die FitNesse testet. Erweiterungen sind also nötig.

Auf der anderen Seite steht das ursprünglich für Unit-Tests entwickelte Framework White von Vivek Singh [3], das auf Basis der Microsoft UI Automation Library [4] und von Windows-Nachrichten das Testen von GUI-Anwendungen erlaubt. White ist ein reines Programmierer-Tool, also für den Kunden als Testwerkzeug in der Regel ungeeignet. Bei der Kombination beider Werkzeuge ergibt sich eine Reihe von Vorteilen. FitNesse lässt sich auf diese Weise mit wenig Aufwand zu einem GUI-Test-Werkzeug erweitern, das den Test aller Windows-basierten Anwendungen erlaubt, von Windows-Forms- über WPF- und Silverlight- bis hin zu SWT-Anwendungen.

Hier soll es einzig um WPF-Anwendungen gehen, da es bei ihnen eine Reihe von Besonderheiten aufgrund der Struktur der GUI-Elemente gibt. Alle Beispiele sind jedoch auch für Tests etwa mit Windows-Forms-Anwendungen geeignet.

FitNesse für Tests beim Kunden

FitNesse ist speziell auf Akzeptanztests ausgelegt. Dies sind Tests auf Kundenseite, bei denen es im



[Abb. 1] FitNesse-Testtabellen im Editiermodus (1), Ansichtsmodus (2) und Testmodus (3).

Gegensatz zu Unit-Tests vor allem darum geht, zu prüfen, ob die Anforderungen korrekt und in der vom Kunden gewünschten Weise umgesetzt wurden. FitNesse steht als Open Source unter der GNU General Public License (GPL). Stefan Lieser hat es bereits in [5] ausführlich vorgestellt.

Das Framework stellt einen Webserver und eine webbasierte Clientoberfläche zur Verfügung, die zur Definition der Tests dient. Es basiert auf dem FIT-Framework von Ward Cunningham [6] und ermöglicht wie dieses die Definition von Tests in HTML, in Excel oder Word. Tests werden in Tabellenform mithilfe einer Wiki-Syntax definiert und intern als HTML-Tabellen verwaltet. Der Vorteil ist, dass keine Programmierkenntnisse nötig sind, sodass auch Nicht-Softwareentwickler Tests schreiben können. Zudem stehen die Testergebnisse in der gleichen Tabellenform zur Verfügung. Technisch wird dies realisiert, indem das Framework die ursprünglichen HTML-Tabellen während des Testlaufs mit den entsprechenden Ergebnissen anreichert. Einzelne Testseiten mit den definierten Testtabellen lassen sich durch Strukturierung zu Testsuiten zusammenfassen.

Um FitNesse an das eigene System, das getestet werden soll, zu binden, ist ein wenig manuelle Arbeit nötig. Dazu stellt FitNesse ein mächtiges Konzept zur Verfügung, die sogenannten Fixtures: Rahmenbedingungen, die eine Testsituation festlegen, wie etwa das Einbinden einer Datenbank mit definierten Testdaten. FitNesse-technisch gesehen ist eine Fixture eine Klasse, die eine Testtabelle auswerten kann, und dies in sehr generischer Form. Je nach Art der Fixtures haben Tabellen eine unterschiedliche Semantik.

Die vordefinierten Fixtures (*ColumnFixture*, *IntFixture*, *StringFixture* und so weiter) kann der Anwender durch eigene Fixture-Klasse ergänzen oder erweitern. Erst die Anpassung an die eigenen Bedürfnisse, das heißt die Interpretation einer Testtabelle durch eine selbst implementierte Fixture-Klasse, macht FitNesse zu einer praktikablen Testumgebung. Es gibt zwei Wege, eigene Fixtures zu implementieren:

- Der erste ist die spezielle Implementierung von Fixtures für jeweils eine zu testende Einheit, ähnlich dem Unit-Test. Dies ist jedoch nicht zu empfehlen, da es einen unverhältnismäßigen Aufwand bedeutet; in diesem Fall sind für jede neue zu testende Einheit die Fixtures zu ergänzen.
- Der zweite Weg ist die Implementierung von generischen Fixtures. Diese sollten

Listing 1

Beispielcode für das White-Framework.

```
Application application = Application.launch("myApp.exe");
Window window = application.GetWindow("myWindow");
if (window != null) {
    TextBox textBox = window.Get(
        SearchCriteria.ByAutomationId("mTextBoxId").
        AndControlType(typeof(TextBox)));
    textBox.Text = "New Text";
    Button button = window.Get<Button>("myButtonId");
    button.Click();
}
```

Tabelle 1

Implementierte Testbefehle für die Verbindung von FitNesse und White.

Befehl	Parameter	Erläuterung
start app from	Pfad zur Anwendung	Startet die angegebene Anwendung.
press	Name/ID/Beschriftung eines Buttons	Klickt auf den gegebenen Button.
set values	Name(n)/ID(s) eines/mehrerer Textfelder	Setzt in alle gegebenen Textfelder die entsprechenden Werte.
select list index	Name/ID der Liste/ComboBox, Index des Listen-Items	Wählt ein Listen-Item anhand seines Index aus.
select list item	Name/ID der Liste/ComboBox, Name des Listen-Items	Wählt ein Listen-Item anhand seines Namens aus.
double click list item	Name/ID der Liste/ComboBox, Index des Listen-Items	Doppel-Klick auf das gegebene Listen-Item.
unselect list	Name/ID der Liste/ComboBox	Deselektiert alle Items der Liste.
check values	Name/ID der GUI-Komponente (Liste, ComboBox, Textfeld, Kontrollkästchen)	Überprüft den Wert, den die gegebene GUI-Komponente anzeigt.
menu path (popup)	Vollständiger Name des Menüpfads, kommasepariert	Wählt den gegebenen Eintrag aus einem Menü oder Popup-Menü.

möglichst allgemein gehalten sein. Im Regelfall reicht eine Zahl von etwa zehn bis 30 generischen Fixtures aus, um eine spezifische Klasse von Anwendungen zu testen. Am einfachsten ist es, wenn auch das zu testende System auf einem Framework beruht, das einen einheitlichen Zugriff für die Fixtures anbietet, sodass andere Systeme auf Basis des gleichen Frameworks ebenso testbar sind. Zu nennen sind hier zum Beispiel modellbasierte Entwicklungs-Frameworks. Nur für Spezialfälle ist dann noch die Ergänzung einiger weniger Fixtures notwendig.

Im Falle des GUI-Tests von WPF-Anwendungen ist die Entscheidung hier für die Implementierung generischer Fixtures mithilfe eines weiteren Frameworks gefallen, das bereits einen einheitlichen Zugriff auf Windows-GUI-Elemente anbietet, jedoch keine Abstraktion der Testfälle analog zu FitNesse kennt: White.

Nach Möglichkeit sollten Sie den zweiten Weg generischer Fixtures wählen.

White verbindet GUI- mit Unit-Tests

Das Test-Framework White ist Open Source und unter der Apache License verfügbar. Es automatisiert GUI-Tests für Windows-Applikationen in Unit-Tests – egal ob es sich dabei um Anwendungen in Win32, Windows Forms, WPF oder auch SWT handelt. Dazu stellt White einen gekapselten Zugriff über die Microsoft UI Automation Library auf die GUI-Komponenten bereit, der einfache Identifikation und Steuerung per Programm erlaubt [7].

Dabei wird zunächst die zu testende Applikation gestartet, anschließend werden die zu steuernden Komponenten gesucht. Komponenten lassen sich durch ihren Typ, ihre ID, ihren Namen oder auch durch ihr Label identifizieren. Dies wird ermöglicht durch kombinierbare Suchkriterien. White liefert dabei immer die erste gefundene Komponente zurück. Gibt es mehrere Komponenten mit gleichem Typ/Namen/ID, so lässt sich zusätzlich ein Index zur genaueren Identifikation übergeben.

Ist die gesuchte Komponente gefunden, lässt sie sich durch einfache Befehle manipulieren, etwa durch Eingabe von Texten in ein Textfeld oder durch das Klicken auf eine Schaltfläche wie in Listing 1. Für alle Standardkomponenten stellt White die entsprechenden Methoden bereits zur Verfügung.

Die FitNesse-White-Kombi

Zur Kombination beider Frameworks ist eine Schicht aus Fixtures nötig, welche die zuvor definierten Testbefehle aus FitNesse-Tabellen in White-Testbefehle übersetzt und ausführt – Drücken eines Buttons, Auswahl eines Elements in einer Liste et cetera. Dabei ist zunächst zu ermitteln, welche Testbefehle nötig sind, um die Oberfläche einer Anwendung möglichst umfassend zu testen, gleichzeitig aber die Anzahl der Testbefehle möglichst klein zu halten. Tabelle 1 zeigt die bis jetzt implementierten Testbefehle der hier vorgestellten FitNesse-White-Bindung.

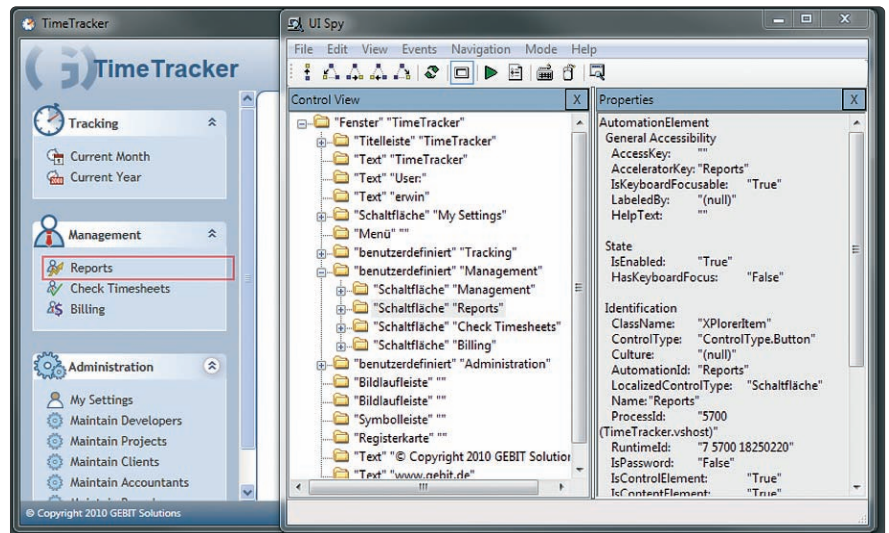
Einige Komponenten sind bisher nicht unterstützt, zum Beispiel die Auswahl von Elementen in einem Baum. Sie sind jedoch mit dem White-Framework in ähnlicher Form zu implementieren wie die bereits vorhandenen Funktionen.

Als globales Verhalten ist Folgendes festgelegt: Pro Testseite wird eine Instanz der Klasse *GUITest* erzeugt. Jeder Testbefehl ist als Methode dieser Klasse realisiert. Dieses Verhalten ist ein Standardverhalten aller Fixtures, die von der Klasse *DoFixture* erben. Die vorgestellte *GUITest*-Klasse ist deshalb als Subklasse von *DoFixture* implementiert.

Der erste Testbefehl ist naturgemäß der Start der zu testenden Applikation. Dabei muss einfach nur der White-Testbefehl *Application.launch* durch einen Testbefehl in FitNesse angesprochen werden. Dies leisten die Befehle *StartAppFrom* und *start app from* – beides ist in FitNesse zulässig – mit der entsprechend implementierten Methode *StartAppFrom* in der Klasse *GUITest* (Listing 2).

Hier sollte bereits klar sein, ob und in welcher Form die gestartete Applikation weiterhin verfügbar sein muss; für den Zugriff mithilfe von White ist dies notwendig. Soll eine Applikation über mehrere FitNesse-Testseiten hinweg verfügbar sein, etwa um Anwendungstestsuiten zu definieren, muss sie global erreichbar sein. Dies ist im genannten Beispiel auch umgesetzt.

Das Schema der Übersetzung ist für die meisten weiteren Testbefehle relativ ein-



[Abb. 2] UI Spy aus dem Windows SDK zeigt den UI-Baum einer Anwendung an.

fach: Zunächst wird eine entsprechende Komponente anhand ihres Typs und eines vom Benutzer übergebenen Strings gesucht. Der Typ ergibt sich aus dem aufgerufenen Testbefehl. So gilt beim Befehl *Auswahl aus einer Liste*, dass es sich um eine *ComboBox*, eine *ListBox* oder eine *ListView* handelt.

Die Suche nach einem bestimmten Typ vermeidet die Rückgabe der falschen Komponente – etwa wenn zwei Komponenten den gleichen Namen haben –, schließt diese jedoch nicht aus. Deshalb startet die Suche erst mit der ID als Suchkriterium. Erst wenn sich damit keine Komponente findet, verwendet die Suche andere Merkmale wie Name oder Beschriftung. Anschließend wird der entsprechende Testbefehl ausgeführt, in dem der korrespondierende White-Befehl aufgerufen wird.

Problematisch wird es, wenn der Tester keine internen Kenntnisse der Anwendung hat. In diesem Fall kann er nicht wissen, wie die einzelnen Elemente benannt sind. Dies gilt besonders für Elemente ohne Beschriftung; diese wird zur Suche herangezogen, wenn die ID nicht zu einem Ergebnis geführt hat. Hier hilft jedoch das Tool UI Spy [8] aus dem Windows SDK, das den gesamten UI-Automatisierungs-Baum einer laufenden Anwendung anzeigen kann (Abbildung 2).

Geschachtelte UI-Komponenten in WPF-Anwendungen

Mit WPF ist es möglich, eine Komponente aus verschiedenen weiteren Komponenten aufzubauen – zum Beispiel eine Schaltfläche, die selbst wiederum aus Komponenten besteht, unter denen das *Label-Control*

mit der gesuchten Beschriftung nur eines von vielen ist. Die Suche nach einem solchen Button landet im Leeren. Um dieses Problem zu umgehen, werden zunächst alle Buttons des aktuellen Fensters in Betracht bezogen; der *AutomationElementFinder* von White soll dabei innerhalb der Buttons nach einer Komponente mit der gegebenen Beschriftung suchen. Hat eines der Elemente innerhalb der Schaltfläche eine solche, so wird der Button als das gesuchte Element zurückgeliefert.

Eigene UI-Komponenten testen

Beim Verwenden eigener GUI-Komponenten muss der Entwickler zusätzliche Hürden nehmen – besonders im Fall von WPF-Anwendungen. Im einfachsten Fall ist das GUI-Element aus bestehenden Elementen zusammengesetzt, etwa aus einer Schaltfläche und einem Eingabefeld. Dann baut die WPF in der Regel bereits intern den Automatisierungsbaum so auf, dass die beiden Teile auftauchen und ansprechbar sind.

Manchmal ist es auch notwendig, nachzuhelfen. Das ist etwa der Fall, wenn die Unterkomponenten zwar enthalten, aber nicht einzeln identifizierbar sind. Dann kommt der Entwickler nicht umhin, für seine Komponente ein sogenanntes UI Automation Peer zu schreiben. Ein weiterer Fall für ein eigenes Automatisierungs-Peer ist, wenn eigene Oberflächen-Interaktionen abzubilden sind, die über Texteingabe und Auslösen eines Buttons hinausgehen.

Um die eigenen Komponenten auf den Automatisierungs-Peer-Typ *Custom* abzubilden, sind zusätzliche Maßnahmen auf der Testseite erforderlich. In diesem Fall ist in White ein Mapping anzulegen, um eine

Identifikation der entsprechenden Komponente zu erreichen. Dazu ist es notwendig, für den jeweiligen Komponententyp eine Subklasse der *White*-Klasse *UIItem* zu implementieren, die Methoden für den Zugriff auf die neue Komponente anbietet. Alternativ kann es erforderlich sein, die neue Komponente den *CustomUIItemTypes* von *White* hinzuzufügen und das Mapping bekannt zu machen. Mehr Informationen dazu enthält die Dokumentation von *White*.

Bekannte Probleme

Beim Starten der Testapplikation setzt *FitNesse* das Arbeitsverzeichnis automatisch auf den *FitNesse*-eigenen Arbeitsbereich. Dadurch kann es passieren, dass die Anwendung relativ zu ihr liegende Konfigurationsdateien oder Ähnliches nicht findet. Um das zu vermeiden, ist es notwendig, beim Start der Anwendung ein entsprechend konfiguriertes *ProcessStartInfo*-Objekt mitzugeben wie in Listing 2.

Unerklärliche Probleme können sich außerdem unter einem 64-Bit-Windows ergeben. In diesem Fall müssen Sie dafür sorgen, dass die zu testende Anwendung im 32-Bit-Modus ausgeführt wird, etwa durch Angabe von *x86* statt *Any-CPU* in der Kompilierungsoption *Ziel-CPU*.

Aber selbst das reicht nicht, da noch ein weiterer Prozess beteiligt ist: Neben der zu testenden Anwendung startet *FitNesse.NET* den *FitServer/TestRunner*, der die Tests steuert. Die veränderte Einstellung zur Ziel-CPU wirkt sich jedoch darauf nicht aus, sodass die betreffenden Prozesse im 64-Bit-Modus laufen. Das fällt nur auf, da manche Merkmale nicht korrekt funktionieren. So lassen sich keine WPF-Menüs mehr steuern. Abhilfe schafft eine Neukompilation von *FitNesse.NET* mit den entsprechenden Einstellungen oder die Änderung des Modus mithilfe des Tools *corflags.exe*. Dieses Tool ist Teil des .NET Framework SDKs [9].

Die Trennung von *TestRunner* und zu testendem Prozess wirft ein weiteres Problem auf: Während sich Aktionen in der Oberfläche relativ einfach anstoßen lassen, ist eine Überprüfung des Ergebnisses auch nur in der GUI möglich. Für ernst zu nehmende Tests sind aber gegebenenfalls weitere Validierungen notwendig, zum Beispiel ob korrekte Werte in die Datenbank geschrieben wurden. Dazu wäre ein paralleler Zugriff auf die Datenbank nötig, was allerdings intime Kenntnisse der internen Architektur der zu testenden Anwendung voraussetzt.

Besser ist es, über die UI-Automatisierungsschnittstelle *Control-Entwurfsmus-*

Listing 2

Starten der Applikation in der Klasse *GuiTest*.

```
public class GuiTest : DoFixture {

    public static Application _application;

    public bool StartAppFrom(String anAssembly) {
        // reset application
        _application = null;
        // try to reinitialize application
        ProcessStartInfo psi = new ProcessStartInfo(anAssembly);
        psi.WorkingDirectory = new FileInfo(anAssembly).Directory.FullName;
        // launch application
        _application = Application.Launch(psi);
        return true;
    }
    ...
}
```

ter für die Applikation bereitzustellen, die Zugriff auf das *White*-Framework gewähren. Gerade dann ist es von unschätzbarem Wert, wenn Applikationen auf einem gemeinsamen Framework basieren, um wiederverwendbare Muster zu ermöglichen.

Fazit

Die Kombination von *FitNesse.NET* und *White* bietet einen einfachen und schnellen Weg, GUI-Tests zu automatisieren. Die Vorteile beider Test-Frameworks ergänzen sich: *FitNesse* bietet die Oberfläche zur Testdefinition und die Testausführung an, *White* ermöglicht den gekapselten Zugriff auf die Oberfläche einer Testanwendung und eignet sich gut für Programmierer. Andererseits ist bei *FitNesse* vieles Handarbeit, insbesondere fehlen Fixtures zum Test einer konkreten Anwendung; bei *White* fehlen essenzielle Dinge wie Testauswertung – etwa durch eine Anbindung an *NUnit* – sowie eine einfache Eingabesprache. Es gibt zwar einen Skriptrecorder, der befindet sich aber noch im Alphastadium.

FitNesse erlaubt eine hohe Abstraktion der Testbefehle und der Nutzer muss nicht viele Kenntnisse der internen Vorgänge einer Komponente besitzen; andererseits beschränken sich die Testbefehle auf essenzielle Dinge, sodass Spezialverhalten nicht mehr einfach testbar ist und zusätzliche manuelle Programmierung erfordert. Dies betrifft sowohl *FitNesse* als auch *White* und damit auch die Kombination der beiden. Allerdings lässt sich Spezialverhalten durch die Kopplung wiederum abstrahiert vom konkreten Fall in *FitNesse* implementieren und wird so einfach wiederverwendbar.

Nicht unerwähnt bleiben soll, dass die Definition von Tests in *FitNesse*-Tabellen gewöhnungsbedürftig ist und dass diese sich bei Änderungen der Applikation nicht einfach zusammen mit dieser refaktorisieren lassen. Immerhin sind sie jedoch robuster gegenüber einfachen Änderungen als GUI-Recorder.

Andererseits sind *FitNesse*-Tabellen auch für Nichtprogrammierer einfach zu schreiben und zu verstehen, sodass beispielsweise auch Kunden in den Testprozess einbezogen werden können. Dies wäre bei der alleinigen Verwendung von *White*, selbst unter Nutzung eines Recorders, nicht so einfach möglich. Das liegt insbesondere daran, dass der Recorder auch lediglich Testcode erzeugt, der für einen Nichtprogrammierer weder verständlich noch eigenständig anpassbar ist. [jp]

[1] *FitNesse*, <http://fitnesse.org>

[2] *FitNesse.NET*, www.dotnetpro.de/SL1005GUITest1

[3] *White-Framework*, www.codeplex.com/white

[4] MSDN, UI Automation Overview, www.dotnetpro.de/SL1005GUITest2

[5] Stefan Lieser, Die gleiche Sprache sprechen, Automatisiertes Testen von Akzeptanzkriterien, *dotnetpro* 11/2009, S. 98 ff.

[6] FIT, <http://fit.c2.com>

[7] Gojko Adzic, Test Driven .NET Development with *FitNesse*,

www.dotnetpro.de/SL1005GUITest3

[8] MSDN, UI Spy (UISpy.exe), www.dotnetpro.de/SL1005GUITest4

[9] MSDN, Corflags Conversion Tool (CorFlags.exe), www.dotnetpro.de/SL1005GUITest5