HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFT UND KUNST
Fakultät Naturwissenschaft und Technik
Göttingen

-

Diplomarbeit über GUI Tests für WPF

Entwicklung einer GUI Testsoftware mit Record & Replay Funktionalität für WPF

Eingereicht von:
Sebastian Basner
Matr.Nr.:381392

Erstprüfer:
Prof. Dr.-Ing. Bernd Stock

Zweitprüfer:
Dipl.-Ing. Tobias Bürmann

28. Mai 2009

Inhaltsverzeichnis

Abkürzungsverzeichnis

API ..... Application Programming Interface
CodeDOM ..... Code Domain Object Model
GUI ..... Graphical User Interface
IDE ..... Integrated Development Environment
MSAA ..... Microsoft Active Accessibility
MSDN ..... Microsoft Developer Network
SDK ..... Software Development Kit
UML ..... Unified Modelling Language
UP ..... Unified Process
WPF ..... Windows Presentation Foundation
XAML ..... eXtensible Markup Language

1 Einleitung

Bei der (Weiter-)Entwicklung von Software kommt es häufig zu Änderungen und Erweiterungen von bestehendem Quellcode. Funktionen werden verändert oder vielleicht auch gänzlich durch neue ersetzt. Bei diesen Entwicklungsprozessen kann es schnell passieren, dass man vergisst z.B. den Namen einer Funktion überall im Quellcode anzupassen oder einen sonstigen Fehler macht, der den korrekten Ablauf des Programms dann beeinflusst.

Um die korrekte Funktion eines Programms zu garantieren, sollten deshalb nach jeder Änderung Funktionstests durchgeführt werden. Da Softwareprojekte sehr schnell an Umfang zunehmen können, ist es wenig sinnvoll manuelle Tests durchführen zu wollen, da dies viel zu zeitaufwändig und fehleranfällig wäre. Es bietet sich viel mehr an automatisierte Tests nach wichtigen Änderungen durchlaufen zu lassen. Grundsätzlich lassen sich alle Testverfahren in die Kategorien Blackbox-, Whitebox- und Regressionstests einordnen, wobei einem Verfahren durchaus auch zwei dieser Kategorien zugeordnet werden können.

Bei Blackbox-Tests wird der Test ohne Quellcodezugriff durchgeführt. Bekannt sind nur die Funktionsschnittstellen sowie die Eingabeparameter und die zurückgegeben Werte. Im Gegensatz hierzu basieren Whitebox-Tests auf einem Test der zugrunde liegenden Programmlogik [whi]. Für das Erstellen solcher Tests ist es unerlässlich einen Einblick in den Quellcode der Software zu haben um das Verhalten der zu testenden Funktionen analysieren zu können. Der Vorteil gegenüber den Blackbox-Tests ist, dass hier nicht nur das Ergebnis, sondern auch der Ablauf des Programms mit getestet wird. Das führt allerdings auch dazu, dass der Aufwand sehr schnell sehr groß wird und der Einsatz von Whitebox-Tests deshalb nur in kleinen Programmen oder nur in sehr begrenzten Teilabschnitten möglich ist.

Unter einem Regressionstest versteht man einen Test, der jederzeit wiederholbar ist, solange keine so grundlegenden Veränderungen am System durchgeführt werden, dass die Tests angepasst werden müssen. GUI-Tests und „Unit-Tests“ fallen im Allgemeinen in dieses Schema. Ziel dieser Tests ist es nach Änderungen im Quellcode die Funktionstüchtigkeit der Software sicherzustellen und vor unerwünschten Fehlern zu warnen, die ein Compiler nicht bemerkt, weil der Quellcode zwar korrekt ist, aber einfach die Berechnungen nicht mehr stimmen.

Die verbreitetste Möglichkeit von Software-Tests ist der sogenannte Unit-Test (Komponententest), der z.B. nach jedem Kompilieren automatisch einmal durchlaufen wird und jeweils ein bestimmtes Modul, oder eine bestimmte Komponente, auf Ihre korrekte Funktion hin testet [Her08] . Hierbei ist Voraussetzung, dass die Entwickler für jede neue Komponente im Programmcode auch wieder einen eigenen Test schreiben. Der Test ruft die entsprechende Komponente mit definierten Parametern auf und vergleicht die Rückgabewerte mit den erwarteten Werten. Unit-Tests können sowohl Whitebox- als auch Blackbox-Tests sein, werden in der Regel aber als Blackbox-Tests durchgeführt. Ein populäres Werkzeug zur Durchführung von Unit-Tests bringt Microsoft in seinem Visual Studio mit NUnit bereits mit.

Der für diese Arbeit gewählte Ansatz ist den Test am GUI durchzuführen. Bei UI-Tests werden automatisiert Eingaben am GUI vorgenommen und mit den Rückgabewerten, die die Software liefert, abgeglichen. Wird eine fehlerhafte Komponente in ein Projekt implementiert, wird der Test negativ ausfallen, da nicht mehr die erwarteten Ergebnisse zurückgeliefert werden. Diese Art des Testens ist ein reiner Blackbox-Test, da der Tester oder die Testsoftware nur Zugriff auf das User Interface hat. Vorteile dieses Vorgehens sind, dass keine Kenntnis von der internen Programmstruktur vorhanden sein muss, um einen Test durchzuführen, dass Tests sehr schnell erweitert oder verändert werden können und dass nicht, wie bei den Unit-Tests, für jede neue Funktion auch ein neuer Test geschrieben werden muss.

1.1 Ziel dieser Arbeit

Im Rahmen der Arbeit soll eine universell einsetzbare Testsoftware für auf Windows Presentation Foundation (WPF) basierenden Anwendungen entwickelt werden (Einführung und Überblick in WPF s. Kap. 2.1). Die Testsoftware „GUI Tester“ soll in beliebigen WPF Applikationen die Eingaben des Testers im GUI in einem Testskript speichern, diese generierten Skripte dann beliebig oft auf der zu testenden Software wieder abspielen und die Rückgabewerte, der zu testenden Software, mit denen zum Zeitpunkt der Skripterzeugung generierten abgleichen können. Das hier beschriebene Vorgehen wird auch als „Capture & Replay“ bezeichnet. Erreicht werden sollte dieses Ziel mit Hilfe der von Microsoft im .NET-Framework 3.0 eingeführten UI Automation Technologie, die in Kapitel 2.2 näher erläutert wird. In Kapitel 5 wird gezeigt wie der GUI Tester erweiterbar wäre um auch andere Technologien als nur WPF zu testen.

1.2 Entwicklungsschritte

Da es sich beim GUI Tester um eine komplette Neuentwicklung handelt, ging der tatsächlichen Implementierung eine längere Planung voraus. Dabei wurde weitestgehend nach den Vorgaben des Unified Process gearbeitet. In [Oes05a] wird der Unified Process als ein Vorgehensmodell für die objektorientierte Softwareentwicklung beschrieben, das sich in folgende Phasen unterteilt:

- Vorbereitung
- Entwurf und Architektur
- Konstruktion
- Einführung
- Betrieb

Während des Entwurfes entstehen nach diesem Entwicklungskonzept verschiedene Modelle, wie das Anwendungsfallmodell und das Fachklassenmodell, die mit Hilfe der Unified Modelling Language (UML) erstellt werden. Dieses Vorgehen hat zur Folge, dass man sich frühzeitig genauestens Gedanken um alle Möglichkeiten, die die Software abdecken soll, und um die grobe Struktur bei der Implementierung machen muss. Man gibt sich somit selbst eine Richtlinie vor, die dann nur noch umgesetzt werden muss.

Während der Implementierungsphase (ggf. auch während der anderen Phasen) sieht der UP ein iteratives Vorgehen vor. Während einer Iteration, die einen festgelegten Zeitraum umspannt, werden realistische Ziele definiert, die in dieser Zeit umgesetzt werden sollen. Üblicherweise beginnt man mit den Anwendungsfällen, die Kernfunktionalitäten betreffen, und arbeitet sich nach und nach zu unwichtigeren vor. Am Ende einer Iteration findet ein Resümee über das Erreichte statt und es wird eine neue Iteration gestartet [Oes05b] .

In der Implementierung wurden zuerst drei Teilprojekte mit den Namen „UI Tester“ , „TestController“ und „AutomationWrapper“ erstellt, die dann zu einer Projektmappe vereinigt wurden. Hiermit wird sichergestellt, dass einzelne Teile des Projekts sehr einfach ausgetauscht und getrennt bearbeitet werden können. Das Teilprojekt UI Tester beinhaltet das GUI mitsamt der Ausgabetexte und wurde in WPF realisiert. Der AutomationWrapper ist für das Abspielen der Skripte zuständig und hält die dazu benötigten UI Automation Control Patterns bereit (Erklärung dazu in Kapitel 2.2), auf die noch genauer in Kapitel 2.2 eingegangen wird. Außerdem werden hier die erwarteten und die tatsächlichen Ausgaben miteinander verglichen und entsprechend die Fehler gezählt. Der TestController schließlich

identifiziert über Events die genutzten GUI Elemente, der zu testenden Anwendung, und erstellt automatisch das Testskript. Als zweite wichtige Komponente ist die CodeDOM Funktionalität (näheres zu CodeDOM s. Kap. 2.3) im TestController enthalten, mit der das Testskript kompiliert und ausgeführt wird.

In einem ersten Schritt wurde die Klassen $AutomationWrapper$ und $TestController$ erstellt. Der AutomationWrapper bekam zunächst der Einfachheit halber vom TestController ein statisches Testskript übergeben (auf Einzelheiten und Probleme wird im Kapitel 3.3 näher eingegangen).

Im zweiten Schritt der Entwicklung wurde dann dem TestController die Möglichkeit der Skriptgenerierung implementiert und im abschließenden dritten Schritt zur Ausführung der Skripte zur Laufzeit die CodeDOM Funktionalität hinzugefügt.

Bei der Entwicklung wurde besonders auf leichte Erweiterbarkeit des Quellcodes geachtet. So können z.B. im AutomationWrapper sehr leicht weitere UI Automation Control Patterns für das Steuern weiterer WPF Steuerelemente hinzugefügt werden, oder auch eine Erweiterung für die Kompatibilität mit Forms GUIs vorgenommen werden (mehr hierzu in Kapitel 5).

2 Grundlagen

Microsoft veröffentlichte die erste Version des .NET Frameworks im Jahr 2002 [Kue08a]. Es handelt sich dabei um eine Entwicklungsplattform, die verschiedene Klassenbibliotheken bereitstellt, die von sämtlichen unterstützten Programmiersprachen (z.B. VB.NET oder J#) gemeinsam genutzt werden können. Neu in .NET war, dass es zu 100 Prozent objektorientiert ist. Es gibt somit kein Element mehr, dass sich nicht auf ein Objekt zurückführen ließe. Eines der wichtigsten Ziele des .NET Frameworks soll es sein, die inzwischen in die Jahre gekommene, Win32 API komplett zu ersetzen.

Das .NET Framework wurde in den letzten Jahren immer weiterentwickelt und um neue Komponenten erweitert. Seit der Version 3.0 –der aktuelle Stand ist die Version 3.5– enthält .NET die UI Automation Technologie, wobei es sich somit um eines der neuesten Features handelt. Im Folgenden sollen die Grundlagen von UI Automation, sowie den für dieses Projekt wichtigsten sonstigen Technologien, erläutert und ein Einblick gegeben werden.

2.1 WPF

Die WPF (Windows Presentation Foundation) ist seit der Version 3.0 des .NET-Frameworks die Technologie, mit der in .NET-Anwendungen grafische Oberflächen erstellt werden [Kue08b]. Es handelt sich somit um die Nachfolgetechnologie für Forms, die zwar immer noch eingesetzt wird, aber nicht mehr die propagierte Standardlösung ist. Dass es sich beim Übergang der beiden Technologien um einen deutlichen Schnitt handelt, wird sofort deutlich, wenn man sich anschaut, dass sich WPF von der Basisklasse Window und nicht mehr von Forms ableitet. Mit der WPF hat Microsoft auch einige Konzepte übernommen, wie es sie schon länger z.B. bei Java gibt. So wird die Ausrichtung und Größe der einzelnen Elemente über „Panels“ und „Grids“ an die Fenstergröße gekoppelt. Damit lassen sich von vornherein unschöne Effekte, wie z.B. beim Maximieren eines Fensters, wo sich zwar das Fenster, aber nicht dessen Inhalt auf die gesamte Bildschirmfläche ausdehnte, verhindern.

Die Auswirkungen auf die Softwareentwicklung durch die WPF sind gravierend,

denn Design und Quellcode wurden beinahe rückwirkungsfrei voneinander getrennt. Selbstverständlich ist keine komplette Trennung möglich, da es natürlich einen Zusammenhang zwischen Quellcode und Design geben muss, doch dieser wurde minimiert und beschränkt sich nur noch auf die Namen der Steuerelemente und die zugewiesenen Events.

Durch diese Umstrukturierung ist es möglich, dass ein Entwickler, oder Designer, Änderungen am Layout der Software vornehmen kann, während ein anderer Entwickler gleichzeitig an der Logik arbeitet, oder dass ein GUI(Graphical User Interface) mit möglichst wenig Aufwand ausgetauscht werden kann. Dementsprechend befinden sich Programmcode und Design auch in getrennten Dateien. Während der Quellcode für die Programmlogik wie gewohnt in den C# Quellcodedateien abgelegt wird, befinden sich die Informationen zum Layout in den sogenannten XAML (eXtensible Markup Language) Dateien. Bei XAML handelt es sich um einen XML Dialekt mit eigenem Namespace.

In der Praxis können XAML Dateien also mit einem handelsüblichen Editor wie jede andere XML Datei erstellt werden. Für Grafik Designer werden aber auch extra Softwarepakete (z.B. Microsofts Expression Blend) zur Verfügung gestellt die am ehesten mit einem Grafikbearbeitungsprogramm vergleichbar sind und zu deren Nutzung man keine Kenntnisse im Programmieren benötigt [Was08]. Ein auf das Nötigste beschränkter Designer wird auch im Visual Studio schon mitgeliefert. Dieser ist aber in seiner Ausstattung keines Wegs mit den Werkzeugen von Expression Blend vergleichbar.

2.2 UI Automation

Wenn man im Internet den Suchbegriff „UI Automation“ recherchiert, fällt sehr schnell auf, dass es sich dabei um eine offensichtlich noch recht neue und bislang etwas vernachlässigte Technologie handelt. Deutsche Quellen sind extremst rar und auch im Englischen gibt es nur wenige gute. Die wichtigste Anlaufstelle war für mich der entsprechende Abschnitt im Microsoft Developer Network (MSDN) [Mic08], da hier vor allem auch die UI Automation Control Patterns aufgelistet sind. Eine gute Einführung in die grundlegenden Prinzipien von UI Automation bieten die Artikel „UI Automation Framework using WPF“ auf der Website „The Code Project“ [Sac09] und „Test Run“ aus dem englischen „MSDN Magazin“ [McC08]. In deutscher Sprache ist noch ein Artikel aus der Dezemberausgabe des „dot.net magazin“ zu empfehlen [Her08], in dem sehr grundlegend auf die Anwendungen und den Aufbau von GUI Tests eingegangen wird. Demonstriert werden diese Möglichkeiten hauptsächlich am Beispiel des UI Testframeworks Ranorex,

das von dem gleichnamigen Hersteller entwickelt wird (mehr zu Ranorex unter Kap. 2.2.1). Doch auch auf Microsofts UI Automation wird eingegangen.

Bei UI Automation handelt es sich um einen Bestandteil der Windows Automation API zur Steuerung von Benutzeroberflächen unter .NET. Eingeführt wurde es mit der .NET Version 3.0 und wird seitdem gepflegt und weiterentwickelt. In Windows 7 soll UIA Teil des neuen „Windows Automation API Framework“ sein und ist somit integraler Bestandteil des kommenden Microsoft Betriebssystems [Kan08]. Trotzdem beschränkt sich UIA nicht ausschließlich auf die aktuelle WPF Technologie, sondern kann sowohl Forms als auch Win32 Programme steuern. Darüber hinaus hat Microsoft bei der grundlegenden Architektur von UIA Wert darauf gelegt es „Cross Plattform“ fähig zu machen. Inzwischen sind auch Mono Portierungen –die Open Source Variante des .NET Frameworks– erschienen, die UIA unter Linux nutzbar machen (siehe [Pro08] und [Mic07]). UI Automation kann sowohl mit den Standard Benutzerelementen eines GUIs, wie auch mit selbst erstellten umgehen, auch wenn hierfür dann sogenannte „Server Side Provider“ notwendig sind um UIA um die nötigen Funktionalitäten zu erweitern.

Der Ursprüngliche Ansatz bei der Entwicklung von UIA war es eine Hilfe für körperbehinderte Menschen zu schaffen, die einen Rechner nicht völlig normal bedienen können, sondern auf Hilfsprogramme angewiesen sind –z.B. Screenreader oder On Screen Keyboards–, die bestimmte Dinge automatisiert ausführen [msd08]. Die Entwicklung solcher Software sollte durch UIA vereinfacht und vereinheitlicht werden. Der zweite, und für diese Arbeit wichtigere, Grundgedanke bei UIA war eine Basis für den immer größeren Bedarf an Oberflächentests zu schaffen.

Vorläufer von UIA war die „Microsoft Active Accessibility“ (MSAA) [msd09]. Diese wurde 1997 für Windows95 eingeführt und war bisher das Hilfsmittel von Microsoft, wenn es um Steuerung von Benutzeroberflächen durch Software ging. MSAA war selber nie in der Lage mit .NET zu interagieren und floss mit dem .NET Framework 3.0 als zweiter großer Kernbestandteil neben UIA in die Windows Automation API ein. Um UIA und MSAA miteinander kommunizieren zu lassen, wurden der „MSAA-to-UI Automation Proxy“ und die „UI Automation-to-MSAA Bridge“ entwickelt. Über diesen Umweg kann MSAA dann auch auf WPF GUIs zugreifen.

Um mit UI Automation auf ein bestimmtes Element im User Interface zuzugreifen, braucht man dessen Automation ID. Diese kann der GUI Tester sich selbst auslesen und schreibt sie auch in das generierte Testskript. Im Normalfall kennt man selbst diese Automation ID allerdings nicht und kann daher weder im Quellcode des Skriptes kontrollieren ob die richtigen Elemente angesprochen werden, noch

kann man das Skript so anpassen wie man es selber gerade braucht. Für dieses Problem hat Microsoft mit dem „UI Spy“ ein sehr nützliches Tool entwickelt, mit dem man die zu testende Software auf alle benötigten Parameter hin untersuchen kann (siehe Abbildung 2.1). Der UI Spy ist Teil des Windows Software Development Kit

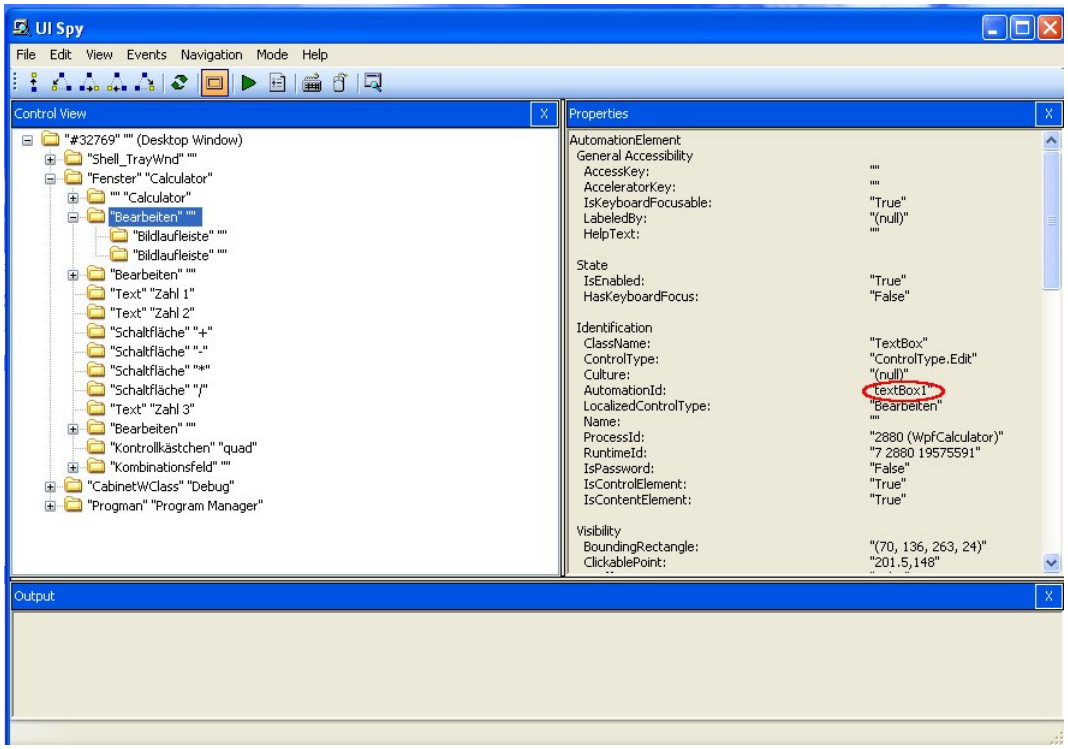

Abbildung 2.1: UI Spy von Microsoft

(SDK) und somit nicht im .NET Framework enthalten. Er kann auch separat von den Microsoft Seiten heruntergeladen werden.

Der eigentliche Zugriff auf ein Steuerelement wird über sogenannte „UI Automation Control Patterns“ (im Folgenden als Automation Pattern bezeichnet) bereitgestellt. Für jedes unterstützte Steuerelementmuster existiert ein eigenes AutomationPattern, von dem für den Zugriff nur eine Instanz erstellt werden muss [msd07a]. Beispiele hierfür sind z.B. das $InvokePattern$, das für das Drücken von Buttons auf WPF Oberflächen zuständig ist, oder das $SetValuePattern$, mit dem der Value Wert von Steuerelementen die über einen solchen verfügen, verändert werden kann. Ein Beispiel für den Einsatz eines solchen Patterns findet sich in Kapitel 3.3.5.6.

2.2.1 Vergleich mit Ranorex

> *„Ranorex is a Windows GUI test and automation framework for C#, VB.NET and Python. Ranorex doesn't have a scripting language of its own like other test tools. The user (e.g. the software tester) can use the functionalities of powerful*

programming languages like Python or C# as a base and expand on it with the GUI automation functionality of Ranorex.“ [ran09]

Da der Bedarf für ein Produkt wie den UI Automation Namespace schon vor dessen Einführung bestand, kamen einige Softwarepakete von Drittanbietern auf den Markt, die die Lücke füllen wollten. Eine der besten Entwicklungen in dieser Hinsicht stellt meiner Meinung nach Ranorex dar. Bei Ranorex handelt es sich um ein komplettes Framework zum automatisierten GUI Testen. Ranorex unterstützt dafür eine Vielzahl von Programmiersprachen und kann zum Testen von .NET, Java (nur SWT), Flash und Web GUIs benutzt werden. Es wird nicht, wie andere Ansätze von Drittanbietern, in einer eigenen Skriptsprache programmiert sondern wird als Funktionsbibliothek in .NET eingebunden und kann dann mit C#, Visual Basic oder IronPython genutzt werden. Ranorex versteht sich als Komplettpaket und bringt daher neben der eigentlichen Kernfunktionalität auch so gut wie alles mit, was man braucht, um mit dem Entwickeln der eigenen Testsoftware beginnen zu können. Zum Paket gehört z.B. „Ranorex Studio“ , eine eigene vollwertige IDE für C#, VB.NET und XML, die optisch stark an ein verschlanktes Visual Studio erinnert (siehe Abbildung 2.2), „Ranorex Spy“ –ein Ersatz für Microsofts UI Spy–

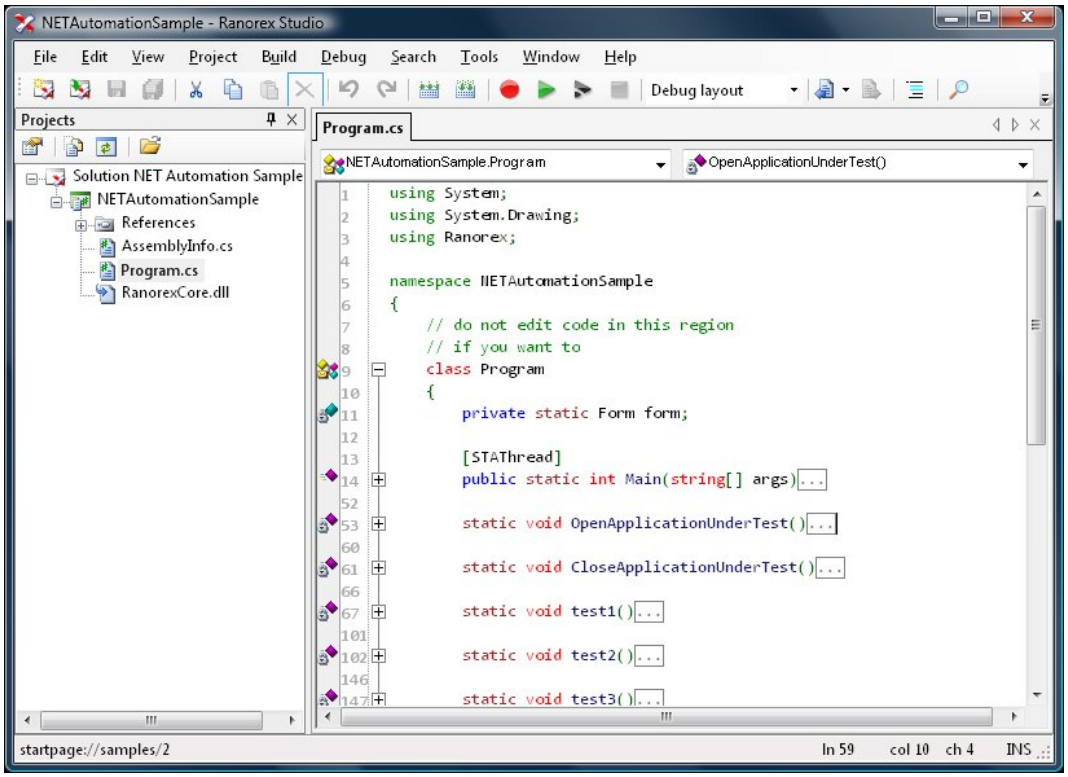

Abbildung 2.2: Ranorex Studio; eigene IDE von Ranorex

und der „Ranorex Recorder“ (siehe Abbildung 2.3). Der Recorder übernimmt ähnliche Aufgaben wie der im Rahmen dieser Arbeit erstellte GUI Tester, ist aber leider

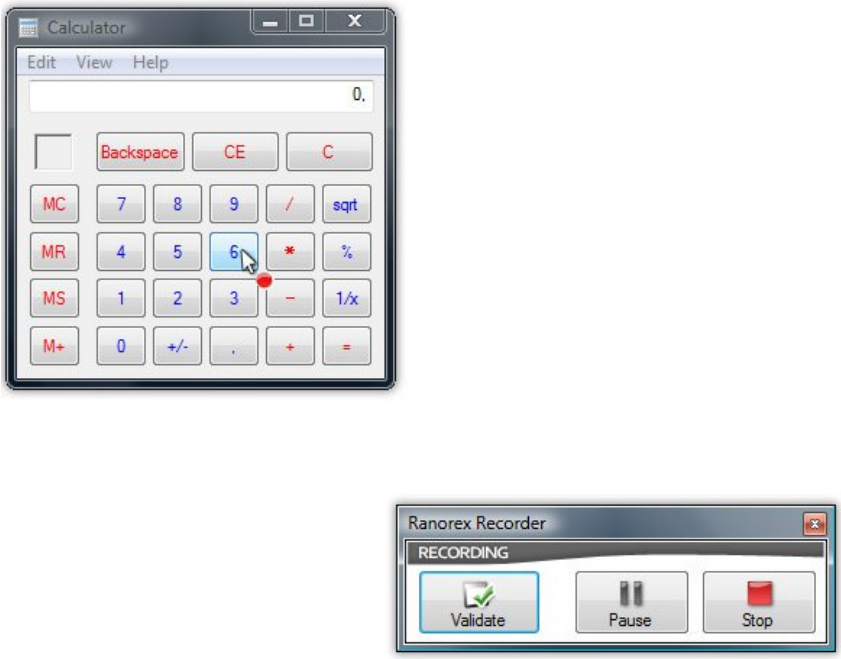

Abbildung 2.3: Ranorex Recorder; kommerzieller UI Recorder

komplett closed Source und daher nicht in seiner Funktionalität erweiter- oder anpassbar. Des Weiteren ist er nur in der €890,- teuren Premium Lizenz mit enthalten, was es äußerst problematisch machen dürfte das Tool einfach an mehrere Entwickler auszuteilen ohne viel Geld dafür ausgeben zu müssen. Mit dem Recorder ist es möglich Benutzereingaben in einem User Interface aufzuzeichnen, diese werden dann im XML Format abgelegt und können auch direkt wieder über den Recorder abgespielt werden. Alternativ wird die Möglichkeit geboten aus den XML Recorderdaten C# Testcode zu generieren, der dann in ein bestehendes Softwareprojekt eingebunden oder für Unit-Tests genutzt werden kann.

Anders als bei Microsofts UI Automation werden beim Einsatz des Ranorex Recorders die Bedienelemente über die Maus angesteuert. So ist beim Abspielen der Ranorex Skripte der Mauszeiger auf dem Bildschirm sichtbar, der nach und nach alle Elemente durchklickt.

Da Ranorex ein proprietäres Produkt ist ist leider nicht viel darüber bekannt wie die eigentliche UI Automatisierung intern funktioniert. Das Konzept schient aber ein anderes als das von UI Automation zu sein. So ist es zwar möglich mit Ranorex auch über die Automation/Control IDs eine Software, oder ein Steuerelement, zu identifizieren, aber Ranorex bietet auch noch die Möglichkeit über eine „RanoreX-Path“ genannte Technologie Steuerelemente auf eine ganz andere Art aufzufinden. So sieht dann z.B. der Aufruf zum Auffinden des Buttons 1 im Windows Taschenrechner aus wie folgt:

```
/form[@title='Calculator']/button[@text='1']
```

2.3 CodeDOM

Sucht man in den einschlägigen Suchmaschinen nach dem Suchbegriff „CodeDOM“ stellt man schnell fest, dass, auch wenn es in Büchern bislang ein stiefmütterliches Thema zu sein scheint, diese Technologie bereits wesentlich verbreiteter ist als UI Automation und es somit auch wesentlich mehr Material darüber zu finden gibt. Gerade im deutschen Bereich gibt es besonders in Programmierforen immer mal wieder interessantes Material. Eine hervorragende Anlaufstelle ist da z.B. das myCSharp Forum im Bereich „Diskussionsforum“ > „Entwicklung“ > „Basistechnologien und allgemeine .NET-Klassen“ [myc09].

Einen sehr guten beispielorientierten Einstieg in deutscher Sprache findet man im „DotNetBlog“ [Bau08] von Sascha Baumann. Noch etwas ausführlicher und –dank einer kompletten Namespace Übersicht–, auch als Nachschlagewerk zu gebrauchen, ist aber ein Artikel auf der englischen Programmierplattform „15 seconds“ [Kor02].

Bei CodeDOM handelt es sich um eine .NET API, mit deren Hilfe es möglich wird, ohne den Umweg über eine externe Skriptsprache zu gehen, zur Laufzeit .NET Quellcode zu kompilieren und auszuführen. Zu diesem Zweck wird ein sogenannter „CodeDOM Objektbaum“ erstellt, der dann mit dem CodeProvider –jede .NET Sprache stellt einen CodeProvider bereit, der seinerseits Compiler Interfaces implementiert– kompiliert werden kann (näheres hierzu in den Kapiteln 3.3.5.3, 3.3.5.4 und 3.3.5.5).

Da der CodeDOM Namespace Teil des .NET Frameworks ist, werden auch sämtliche .NET Programmiersprachen unterstützt. Quellcode aus jeder dieser Sprachen lässt sich mittels des entsprechenden Code Providers kompilieren. Um anstatt eines C# Quellcodes beispielsweise ein Visual Basic Programm zu kompilieren, reicht es aus einzig den Code Provider zu tauschen. Der Rest der CodeDOM Methode kann komplett unangetastet bleiben. So könnte z.B. der GUI Tester Skripte aus verschiedenen Programmiersprachen abspielen, indem man mehrere Code Provider implementiert, die dann im UI, vor dem Übersetzen eines Skriptes, per Dropdownmenü auswählbar sind.

3 GUI Tester für WPF

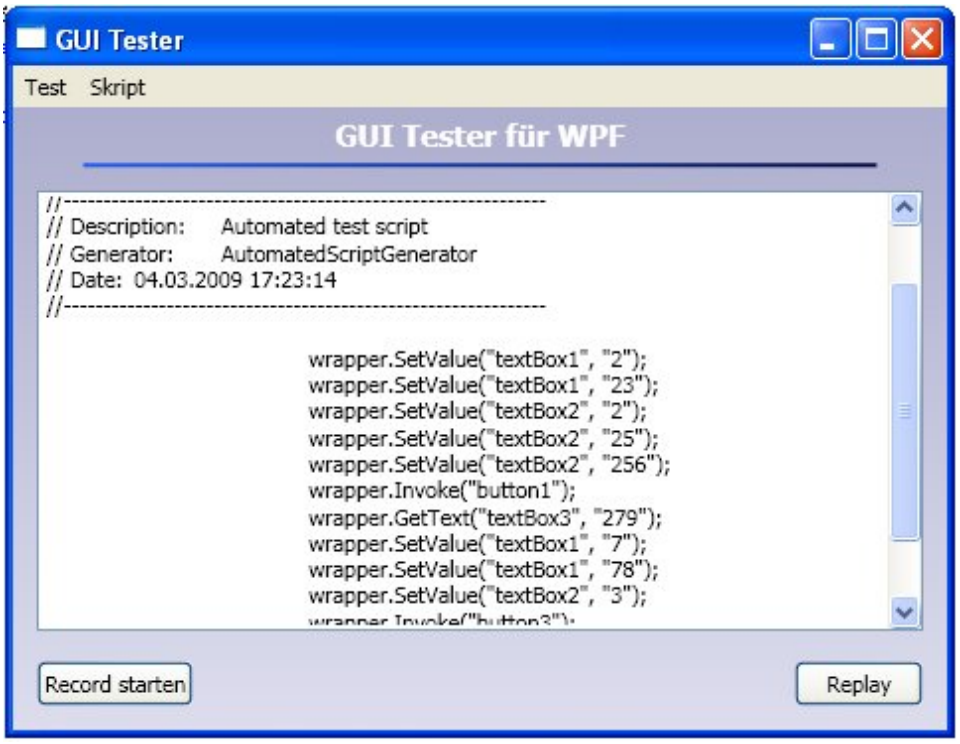

Abbildung 3.1: Der GUI Tester, der im Rahmen dieser Arbeit entstanden ist

Im Rahmen dieser Arbeit wird der GUI Tester soweit entwickelt, dass er Eingaben auf grundlegende Benutzerelemente aus beliebigen WPF User Interfaces aufnehmen und daraus Testskripte in C# generieren kann. Diese Skripte können mit jedem einfachen Texteditor beliebig abgeändert, erweitert und wieder abgespielt werden. In der Textbox wird ein Testlog ausgegeben, das am Schluss die fehlerhaften Rückgabewerte des getesteten Programms zusammenfasst.

Das User Interface soll möglichst einfach und selbsterklärend sein (siehe Abbildung 3.1), da es sich um ein Tool zum automatisierten Testen handelt und die Einstiegshürde möglichst gering sein sollte, damit ein schneller Einsatz ohne lange Einarbeitungszeit möglich ist.

3.1 Analyse

> *„Innerhalb der Analysephase wird die Gesamtheit der Anforderungen an die zu entwickelnde Software festgelegt. Gebräuchliche Bezeichnungen sind auch Anforderungs- oder Produktdefinition. Ziel der Analyse ist es, die Wünsche und Anforderungen eines Auftraggebers zu ermitteln und zu beschreiben... Die Aspekte der technischen Realisierung sind auszuklammern...“* [Kry08]

Der erste Schritt ist eine Anforderungsanalyse, die genau festlegt, was die Software können muss. Anhand dessen können dann bereits Informationen zur einzusetzenden Technologie eingeholt werden. Anschließend folgen verschiedene Modelle, die den späteren Aufbau der Software und der Klassenstruktur visualisieren. Durch die frühzeitige Planung dieser Dinge wird die Zeit für die tatsächliche Implementierung stark verkürzt, da man zu jedem Zeitpunkt weiß was noch getan werden muss und eine ungefähre Vorgabe hat wie das Programm umgesetzt wird. Am Ende der Analysephase wird ein Prototyp des GUI erstellt, der nicht dem Endprodukt entsprechen muss, aber an dem man sich orientieren und der einem Kunden bereits präsentiert werden kann.

3.1.1 Anforderungen

Gefordert ist eine Software, mit der es möglich sein soll beliebige WPF GUIs per Capture & Replay Funktionalität automatisiert und beliebig oft wiederholbar über, von der Software generierte, Testskripte testen zu können. Die Software selbst soll in C# mit einer WPF GUI erstellt werden.

Da mir die Auswahl einer Skriptsprache freigestellt war, habe ich mich nach einiger Recherche entschieden keine echte Skriptsprache wie z.B. Lua oder jScript zu verwenden. CodeDOM deckt alle hier benötigten Anforderungen ab und bringt den Vorteil mit, dass sich der User nicht in die Skriptsprache einarbeiten muss, nur um ein paar händische Änderungen an einem Testskript vorzunehmen, da es als Teil des .NET Frameworks in jeder beliebigen .NET Sprache –in diesem Fall C#– ausprogrammiert werden kann. Als Automatisierungsframework habe ich mich aufgrund der sehr guten Integriertheit im .NET Framework und der damit verbundenen Weiterentwicklungsaussichten für UI Automation und gegen Ranorex entschieden. Gerade im Zusammenspiel mit CodeDOM gibt es für mich auch kein wirklich treffendes Argument mehr, das die Ausgabe von bis zu €850,- für Ranorex rechtfertigt. Die Werkzeuge, die das .NET Framework in der aktuellen Version bietet stehen dem in so gut wie nichts mehr nach. Weiterer Vorteil für mich ist, dass UI Automation wesentlich besser dokumentiert und offener als Ranorex ist. Ranorex ist an einigen, für mich entscheidenden, Stellen leider völlig proprietär und scheidet deshalb aus.

3.1.2 Anwendungsfälle

Als erster wichtiger Punkt wurden die Anwendungsfälle identifiziert und ein entsprechendes Diagramm erstellt (s. Abb. 3.2).

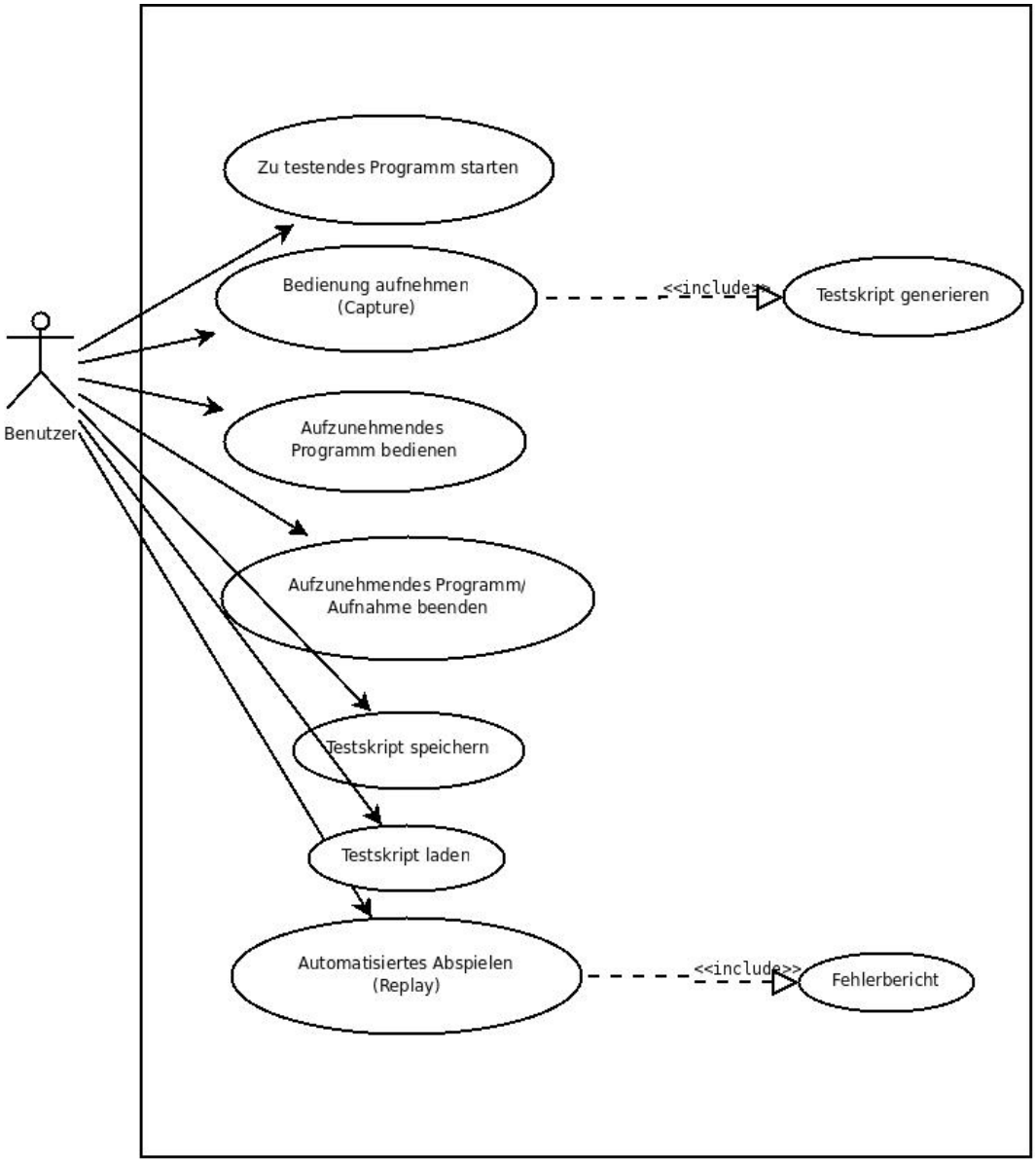

Abbildung 3.2: Das Anwendungsfalldiagramm

Über die Anwendungsfälle wird genau festgelegt welche Interaktionen zwischen Nutzer und Software auftreten können und welche Aktionen das System ausführen muss.

1. **Zu testendes Programm starten:** Das zu testende Programm wird durch den GUI Tester gestartet

2. **Bedienung aufnehmen (Capture):** Es wird eine Aufnahme gestartet. Die auf-

zunehmende Anwendung muss gestartet sein. Der Rekorder ist bereit das Testskript zu generieren.

3. **Aufzunehmendes Programm bedienen:** Der Tester führt alle möglichen, oder alle zu testenden, UI Operationen einmal aus. Eingaben werden vom Rekorder erkannt und gespeichert.

4. **Aufzunehmendes Programm/Aufnahme beenden:** Indem das aufzunehmende Programm geschlossen wird, ist die Aufnahme beendet und das Skript kann in eine entsprechende Skriptdatei auf die Festplatte geschrieben werden.

5. **Testskript speichern:** Das Testskript wird als C# Quellcodedatei abgespeichert.

6. **Testskript laden:** Ein beliebiges Testskript wird geladen.

7. **Automatisiertes Abspielen (Replay):** Nach Drücken des Replay Knopfes wird das Testskript automatisiert durchlaufen und die durchgeführten Tests in einem Logfenster dargestellt. Abschließend wird eine Fehlerzusammenfassung ausgegeben.

3.2 Entwurf

> *„Im Entwurf wird die Architektur der Anwendung festgelegt. Die Gesamtaufgabe wird in überschaubare Teilaufgaben zerlegt, die technischen Festlegungen erfolgen. Dazu gehören Angaben zur Hardware und zum Betriebssystem. Handelt es sich um eine betriebliche Anwendung, sind oft Angaben zur Datenhaltung (Datenbanksystem) notwendig...“* [Kry08]

Für dieses Projekt war, wie bereits im vorhergehenden Abschnitt erwähnt, die Programmiersprache bereits von vornherein auf C# und die zu verwendende Oberfläche auf die Windows Presentation Foundation festgelegt. Dennoch wird diese Festlegung formal hier getroffen. Als Entwicklungsumgebung (IDE) entschied ich mich aufgrund der Verfügbarkeit an der Fachhochschule für Microsofts Visual Studio 2008 Professional. Als Automatisierungsframework habe ich mich für den Einsatz von UI Automation und gegen Ranorex entschieden. Ebenfalls in dieser Phase wird auch die Form der Datenhaltung der Skripte festgelegt. Da ich mich mit CodeDOM gegen eine externe Skriptsprache entschieden habe, kommt für die Skripte das Format einer C# Quellcodedatei zum Einsatz.

3.2.0.1 Klassenstruktur

Das Entwurfsklassendiagramm in Abbildung 3.3 wurde anhand der Anwendungsfälle erstellt. Es zeigt die geplante Struktur des fertigen Programms mit seinen drei Teilprojekten, die jeweils in einer eigenen Klasse implementiert werden. Innerhalb der Klassen sind bereits die Methoden eingefügt von denen zu diesem Zeitpunkt anhand der Anwendungsfälle klar war, dass sie gebraucht würden. Die Pfeile zwischen den Klassen zeigen an in welcher Richtung die Klassen miteinander kommunizieren können. Weder die Klassen, noch die Methoden müssen zu diesem Zeitpunkt vollständig sein und können sich natürlich mit dem Projekt weiterentwickeln. Dennoch geben die Entwurfsklassen einen guten Rahmen vor an dem man sich in der Implementierung orientieren kann.

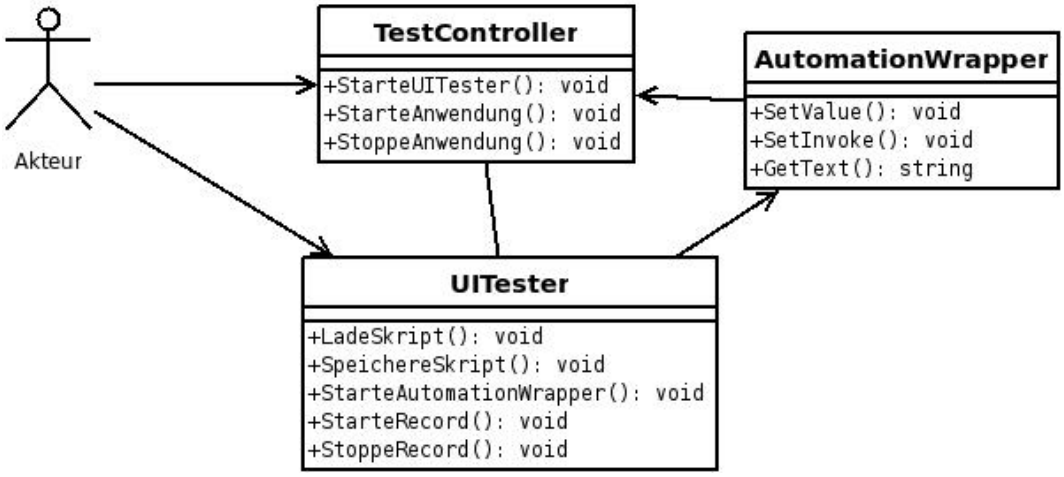

Abbildung 3.3: Das Entwurfsklassendiagramm

In Kapitel 4 ist ein weiteres Klassendiagramm zu sehen, das nach Abschluss der Implementierung mittels eines Tools automatisch generiert wurde. Es zeigt die tatsächliche Klassenstruktur, wie sie während der Implementierung entstanden ist.

3.3 Implementierung

Der GUI Tester besteht aus den drei Teilprojekten UITester, TestController und AutomationWrapper. Jedes dieser Teilprojekt verfügt über einen eigenen Namespace und wird teilweise sogar in eigenen Threads ausgeführt. Das Teilprojekt UITester beinhaltet das User Interface in WPF und entstand in seiner Grundlage bereits während der Analysephase als Prototyp. Dieser Prototyp wird im Laufe der Entwicklung kontinuierlich weiter ausgebaut, da ihm noch die Funktionalitäten fehlen.

Der TestController enthält die Funktionalität zum Referenzieren der zu testen-

den Applikationen, sowie der Erstellung der Testskripte und die Kompilierung der Skripte über das CodeDOM.

Der AutomationWrapper übernimmt vor allem die Aufgabe der Steuerung der Testapplikationen über die entsprechenden Automation Pattern.

3.3.1 Das GUI im Überblick

Das User Interface (siehe Abbildung 3.1) ist so schlicht und einfach wie möglich gehalten um die Einstiegshürde für die Software möglichst niedrig zu halten.

Egal ob ein Skript aufgenommen oder abgespielt werden soll, muss als erstes mit dem Menüpunkt „Test“ > „Anwendung festlegen und starten“ das zu testende Programm aufgerufen werden. Das Programm muss unbedingt auf diesem Weg aufgerufen werden, da nur so der GUI Tester das zu testende Programm identifizieren und referenzieren kann. Soll nun ein Skript aufgenommen werden, genügt ein Klick auf den „Record starten“ Button. Der GUI Tester wartet nun auf Eingaben am zu testenden Programm. Wird dieses nun bedient, zeigt der GUI Tester die erstellten Skriptbefehle in seinem Logfenster an. Nach Abschluss der Eingabe kann über den Menüpunkt „Skript“ > „Speichern unter“ das erzeugte Testskript abgespeichert werden.

Zum Abspielen eines Skriptes wird ebenfalls erst, wie bereits oben beschrieben, die zu testende Anwendung über das Menü gestartet. Nun kann über den Menüpunkt „Skript“ > „öffnen“ ein vorher erzeugtes Skript geöffnet werden. Sobald jetzt der Button „Replay“ geklickt wird, wird das Skript abgespielt. Währenddessen wird im Logfenster jede gerade abgespielte Zeile und das Ergebnis darauf ausgegeben. Entspricht ein Ergebnis nicht dem Erwartungswert, wird dies mit einer entsprechenden Meldung kommentiert. Nach Abschluss des Skriptes wird die Summe der aufgetretenen Fehler zusammengezählt und das zu testende Programm wird geschlossen.

3.3.2 Kommunikation mit dem GUI

Sowohl die Methoden der Klasse $TController$, wie auch die der Klasse $AWrapper$, geben im Betrieb oft Statusmeldungen –wie z.B., dass die Testsoftware gefunden wurde, oder die gerade abgespielte Skriptzeile– an das GUI zurück, die dann im Logfenster ausgegeben werden sollen. Hierbei ergab sich das Problem, dass das User Interface, z.B. solange ein Skript abläuft, blockiert wurde und erst wieder nach Beendigung des Skriptes aktualisiert werden konnte. Es wurde also das ganze Log für ein Skript erst in das Logfenster eingetragen, nachdem das Skript vollständig

durchlaufen wurde. Dieses Problem liegt darin begründet, dass mehrere Methoden sich einen Thread teilten. Um das zu umgehen, war es notwendig den TestController aus dem GUI als eigenen sogenannten Workerthread aufzurufen. Die zwei Threads können nun parallel ablaufen (Multithreading) und blockieren sich daher nicht gegenseitig. Ein entsprechender Aufruf einer Skriptkompilierung in einem eigenen Thread sieht z.B. so aus:

```
private TController TController = new TController();
.
.
.
Thread skript = new Thread(new ThreadStart(TController.
 skript));

skript.Start();
```

Zuerst wird eine neue Instanz des Controllers mit dem Namen $TController$ angelegt. Anschließend wird ein neuer Thread mit dem Namen $skript$ erstellt, der als Argument die aufzurufende Methode $TController.skript$ übergeben bekommt. Die letzte Zeile startet den Thread und somit auch die Methode. Auf diese Art blockiert das GUI nicht mehr und das Logfenster kann bei jeder durchgeführten Aktion aktualisiert werden.

Wenn man eine Methode allerdings auf diese Weise aufruft, ergibt sich ein anderes Problem. Das Schreiben in ein GUI Fenster ist so nämlich für den TestController nicht mehr möglich, da nur der GUI Thread auch das GUI aktualisieren darf.

3.3.3 GUI Aktualisierung aus dem Workerthread

Bis zu diesem Punkt habe ich mein Logfenster einfach per $return$ in der ausführenden Methode aktualisiert. Ein Methodenaufruf sah dann z.B. so aus:

```
logfenster.Text += TestController.Methode();
```

Wenn in der TestController Methode dann per $return$ ein String zurückgegeben wurde, wurde dieser in das Logfenster geschrieben. Da nun aber der Methodenaufruf nicht mehr im GUI Thread stattfindet, ist dies so nicht mehr möglich. Um die beiden Threads aber trotzdem miteinander kommunizieren lassen zu können, habe ich die Kommunikation zwischen den Klassen auf Events umgestellt.

Um die gewünschten Textausgaben in die Events zu verpacken, wird eine eigene Implementierung von EventArgs benötigt.

> *„EventArgs ist die Basisklasse für Klassen, die Ereignisdaten enthalten. ...*
> *Wenn für einen Ereignishandler Zustandsinformationen erforderlich sind, muss*

die Anwendung von dieser Klasse eine Klasse ableiten, die die Daten enthält. “ [msd07b]

Diese muss für jede Klasse, die ein Event werfen soll, implementiert werden. Hier beispielhaft die Implementierung für den Wrapper:

```
public class GetTextEventArgs: EventArgs
    {
        public GetTextEventArgs(string result)
        {
           this.result = result;
        }

        public string Result
        {
            get
            {
                return (this.result);
            }
        }
        string result;
     }
```

Die Klasse $GetTextEventArgs$ leitet sich von EventArgs ab und kann einen String setzen sowie diesen dann wieder bereitstellen. Als nächstes wird ein Delegat gebraucht um damit den eigentlichen Eventhandler aufzurufen.

```
public delegate void GetTextEventHandler(object sender,
 GetTextEventArgs e);
```

Der Delegat gibt vor, dass der Eventhandler einen Sender und das selbst implementierte $GetTextEventArgs$ übergeben bekommen muss. Anschließend wird das eigentliche Event mit dem Namen $onGetText$ und der entsprechende Eventhandler definiert.

```
public event GetTextEventHandler onGetTextHandler;

        protected virtual void onGetText(GetTextEventArgs e)
        {
            if (onGetTextHandler != null)
            {
                onGetTextHandler(this, e);
            }
        }
```

Bei den beiden Argumenten $this$ und e handelt es sich um die im Delegaten geforderten $sender$ und $GetTextEventArgs$.

Das Event kann nun beliebig oft und aus jeder Methode der Klasse geworfen werden. Dies sieht für den Wrapper so aus:

```
string result = "Text der ans GUI gehen soll";
GetTextEventArgs e = new GetTextEventArgs(result);
onGetText(e);
```

Zuerst wird der eigentliche String, der auf dem GUI ausgegeben werden soll, in die Variable $result$ geschrieben. Eine Instanz von $GetTextEventArgs$ wird erstellt und bekommt den String übergeben. Zum Schluss wird die Methode $onGetText$ aufgerufen und bekommt die erstellte $GetTextEventargs$ Instanz e, in der jetzt der Ergebnis-String verpackt ist, übergeben.

Damit das User Interface auf die Events reagieren kann, müssen diese mit dem $+=$ Operator abonniert werden.

```
wrapper.onGetTextHandler += new GetTextEventHandler
 (AWrapper_GetTextHandler);

TController.onGetRecordHandler += new GetRecordEventHandler
 (TController_GetRecordHandler);
```

Auch wenn ich es nicht verwendet habe, sei noch kurz erwähnt, dass das Abonnement eines Events entsprechend mit $-=$ wieder gekündigt werden kann. Wird nun ein Event geworfen, wird die Methode $AWrapper\_GetTextHandler$ aufgerufen.

```
private delegate void MethodInvoker(object sender,
 GetTextEventArgs e);

public void AWrapper_GetTextHandler(object sender,
 GetTextEventArgs e)
{
   if (!txt_logger.Dispatcher.CheckAccess())
   {
      txt_logger.Dispatcher.Invoke(System.Windows.
       Threading.DispatcherPriority.Normal, new
       MethodInvoker(this.AWrapper_GetTextHandler),
       sender, e);

      return;
```

```
    }

    txt_logger.Text += "\r\n";
    txt_logger.Text += e.Result.ToString();
    txt_logger.ScrollToEnd();
}
```

In der If-Bedingung liefert die $Dispatcher.CheckAccess$ Bedingung den Wert true zurück, wenn der Aufruf aus dem GUI Thread erfolgt ist, und false, falls er aus in einem anderen Thread erfolgt. Der Dispatcher ist eine Neuerung, die mit der WPF eingeführt wurde. Der Zugriff auf das GUI ist ausschließlich über diesen Weg möglich, denn

> *„In WPF kann nur der Thread auf das Objekt zugreifen, der das betreffende DispatcherObject erstellt hat. Ein Hintergrundthread außerhalb des Haupt-UI-Threads kann beispielsweise nicht den Inhalt eines Button aktualisieren, der dem UI-Thread zugeordnet ist. Um im Hintergrundthread auf die Content-Eigenschaft des Button zugreifen zu können, muss der Hintergrundthread die Arbeit an den Dispatcher delegieren, der dem UI-Thread zugeordnet ist. Dies erreichen Sie entweder mit Invoke oder mit BeginInvoke. Invoke ist synchron und BeginInvoke ist asynchron. Die Operation wird mit der angegebenen DispatcherPriority in die Warteschlange des Dispatcher eingefügt.“* [msd07c]

Beim GUI Tester erfolgt der Aufruf der Funktion $AWrapper\_GetTextHandler$ im Normalfall immer aus einem Workerthread. Um in den richtigen Thread zu wechseln, muss der Methodenaufruf also per Invoke an den Dispatcher delegiert werden. Hierzu wird der vorher definierte Delegat $MethodInvoker$ zu Hilfe genommen, der als Argumente den Sender und GetTextEventArgs übergeben bekommen muss. Nach diesem Aufruf findet der Zugriff im GUI Thread statt. Die If-Abfrage liefert ein false und es wird der Wert $Result$ über die $ToString$ Methode aus $GetTextEventArgs$ in die Textbox des GUIs geschrieben.

3.3.4 Capture/Record

Für das Aufnehmen und Abspeichern eines neuen Testskripts werden nur die Teilprojekte UITester und TestController benötigt. Der Wrapper hat hierfür keine Funktion.

Das Aufnehmen eines neuen Skripts läuft grob so ab, dass über das GUI ein beliebiges Programm gestartet wird. Dazu wird der Menüeintrag „Test“ > „Anwendung festlegen und starten“ ausgewählt. Behandelt wird dieser Aufruf von der $menu\_loader\_Click$ Methode der Klasse $UITester$. Hier wird nun ein Open Filedialog erzeugt, über den dann die gewünschte Anwendung ausgewählt werden kann. Der Pfad zur gewählten Anwendung wird in der Variablen $Applicationpath$ gespeichert. Diese Variable wird dann an die Methode $Loader$ des Controllers übergeben, von wo aus sie gestartet wird. Um jetzt die Skriptaufnahme zu starten, muss der Button „Record starten“ gedrückt werden. Das Button-Clickevent in $UITester$ gibt nun ein wenig Text im Logfenster aus und erstellt ein neues AutomationElement $aeLoader$, das mit der Rückgabe der Methode $Referrer$ im Controller gefüllt wird. Die Methode $Referrer$ ist dafür zuständig Referenzen auf das Desktop und das Testprogramm anzulegen (s. Kap. 3.3.4.1). Nachdem geprüft wurde, ob die Referenzen erstellt wurden, wird die Controllermethode $Recording$ aufgerufen, die die Automation Eventhandler bereitstellt (s. Kap 3.3.4.3). Benutzereingaben werden nun im Controller über Events erkannt und in einer Liste gespeichert. Nach Beendigung der Testaufnahme kann nun das Skript über den Menüeintrag „Skript“ > „Speichern Unter“ im aufpoppenden Save Filedialog gespeichert werden. Dazu ruft das Clickevent in $UITester$ die Methode $SaveSkript$ des Controllers auf, wo dann das Skript an der angegebenen Position gespeichert wird (s. Kap. 3.3.4.4).

3.3.4.1 Referenzieren der Testapplikation

Für den Einsatz von UI Automation ist es zunächst erforderlich die zu steuernde Applikation zu referenzieren. Hierzu wird ein UI Automation Tree aufgebaut –abgebildet wird dieser Baum z.B. im UI Spy in Abbildung 2.1–, als dessen Wurzelelement der aktuelle Desktop festgelegt wird.

```
AutomationElement aeDesktop = AutomationElement.RootElement;
```

Jedes Kind-Element repräsentiert eine geöffnete Applikation und deren Kinder sind wiederum Steuerelemente [msd]. Als nächstes müssen nun in diesem Baum das zu steuernde Programm und dessen Elemente ausfindig gemacht und referenziert werden. Hierbei stellte sich leider heraus, dass UI Automation für den Einsatz mit beliebigen zu testenden Programmen an dieser Stelle nicht so einfach zu handhaben ist.

Für erste Tests habe ich dem Wrapper vom Controller ein statisches Testskript übergeben. Zum Steuern einer WPF Anwendung via UI Automation ist es bislang noch notwendig die AutomationID der Anwendung zu kennen, da nur darüber eine Anwendung identifiziert wird. Da beim GUI Tester aber beliebige Anwendungen testbar sein sollen, ist diese ID im Voraus nicht bekannt. Somit funktioniert die von Microsoft propagierte Methode des Auffindens, der zu testenden Software im Baum über das $NameProperty$, leider nicht:

```
AutomationElement ae = null;
int numWaits = 0;
do
{
    ae = aeDesktop.FindFirst(TreeScope.Children, new
     PropertyCondition(AutomationElement.NameProperty,
     "StatCalc"));

    ++numWaits;
    Thread.Sleep(100);
}
```

In diesem Fall wird so lange die Do-Schleife durchlaufen bis ein Element mit dem Namen StatCalc gefunden wird. Versuche diese Methode einfach in sofern abzuändern, dass nicht nach dem $NameProperty$, sondern dem gestarteten Prozess gesucht wird, scheiterten leider daran, dass die Schleife sofort mit einer Fehlermeldung abbricht, wenn sie einmal durchlaufen wird und der Prozess noch nicht gestartet ist. Da aber auch nicht bekannt ist wie lange ein Prozess zum Starten braucht, ist es ebenfalls wenig sinnvoll den Thread für eine bestimmte Zeit über die Funktion $Thread.Sleep()$ warten zu lassen. Wie ich in einem Mailwechsel mit einem Entwickler aus dem .NET Team erfahren habe, ist man sich dieses Problems bereits bewusst und will hier, auch auf Grund vieler Nachfragen, in einer der kommenden .NET Versionen Abhilfe schaffen. Ich habe mich daher dafür entschieden das zu testende Programm über einen Menüeintrag zu starten und seine Identifizierung erst nach dem Klicken des Record Buttons durchzuführen. Dadurch ist das Starten der Testsoftware ein isolierter Prozess und ich kann den Benutzer per MessageBox darauf hinweisen auf das Starten des Programms zu warten. Ist das Programm einmal fertig gestartet, ist es kein Problem es anhand des Prozesses zu identifizieren und zu referenzieren.

```
aeForm = AutomationElement.FromHandle(p.MainWindowHandle);
```

Als Referenz wird hier einfach vom Hauptfenster der Applikation ausgegangen. p ist das Prozessobjekt, mit dem die Anwendung gestartet wird.

3.3.4.2 Identifikation der Bedienelemente

Nachdem die zu testende Anwendung identifiziert und in einem AutomationElement referenziert ist, müssen nun noch die Bedienelemente auf dem GUI gefunden und ebenfalls referenziert werden. Hierzu gibt es die beiden Möglichkeiten des „Scoping“ und des „Filtering“ . Die Unterschiede liegen darin, dass man über das Filtering die Bedienelemente nach Gruppen sortieren kann, während das Scoping die Elemente definierter anspricht, wie z.B. nur die Kindelemente der Wurzel, oder (wie im konkreten Fall) direkt nach dem Namen des Elements.

Die eigentliche Referenzierung eines Bedienelements erfolgt direkt im Wrapper, erst kurz bevor es auch tatsächlich angesprochen werden soll. Im Wrapper sind für jedes Bedienelement, das der GUI Tester momentan ansprechen kann, Methoden definiert, die vom Testskript aus aufgerufen werden. Jede dieser Methoden muss mindestens die Variable $ControlID$ vom Skript übergeben bekommen (s. Kap.3.3.4.4) –Textfelder brauchen noch den eigentlichen Value und Ergebnisfelder einen Prüfwert–, die das Äquivalent zur AutomationID für Bedienelemente darstellt und diese eindeutig identifizieren kann. Mit Hilfe dieser ControlID kann dann das entsprechende Bedienelement im Automation Tree über das $AutomationIdProperty$ gefunden und in einem neuen AutomationElement ae abgelegt werden.

```
AutomationElement ae = aeForm.FindFirst(TreeScope.
 Descendants, new PropertyCondition(
 AutomationElement.AutomationIdProperty, ControlID));
```

$aeForm$ ist hierbei die Referenz auf das zu testende Programm, die der Wrapper, nachdem sie gefunden wurde, übergeben bekommt. Gesucht wird das erste Element im Teilbaum von $aeForm$, das die übergebene ControlID besitzt. Wichtig ist dabei zu wissen, dass eine ControlID theoretisch nicht eindeutig sein muss. Auch wenn sie automatisch nie doppelt vergeben wird, so kann sie vom Entwickler manuell doppelt gesetzt werden. In diesem Fall wäre es Zufall welches Element zu erst gefunden wird und man müsste sich über weitere PropertyConditions absichern, dass man auch tatsächlich das Element bedient, das gewünscht war. Ist das Bedienelement referenziert, wird die gewünschte Funktion über das entsprechende Automation Pattern realisiert. Dies wird einmal beispielhaft in Kapitel 3.3.5.6 beschrieben.

3.3.4.3 Eventbasiertes Abfangen von Useraktionen

Die Kernfunktionalität des Recordings besteht im Abfangen der Benutzereingaben an der zu testenden Software. Dies ist, ähnlich wie die Kommunikation mit dem

GUI, in der Klasse $TController$ über Events realisiert. Doch im Unterschied zu den vorher vorgestellten Events gibt es für UI Automation bereits fertige AutomationEventHandler. Von diesen wird in der $Recording$ Methode jeweils erst eine Instanz erstellt und anschließend festgelegt wie das auslösende Element identifiziert wird und welche Methode es auslösen soll:

```
AutomationPropertyChangedEventHandler textchangedHandler =
 new AutomationPropertyChangedEventHandler(OnTextChanged);

Automation.AddAutomationPropertyChangedEventHandler(aeForm,
 TreeScope.Descendants, textchangedHandler,
 ValuePatternIdentifiers.ValueProperty);
```

Der $textchangedHandler$ wird ausgelöst, wenn sich der Wert eines Textfeldes ändert. Dabei ist es völlig egal ob die Änderung durch eine Usereingabe, oder einen automatisch zurückgegebenen Wert ausgelöst wird. Es wird eine Referenz auf das Event auslösende Bedienelement nach dem Scope-Verfahren angelegt und die Methode $OnTextChanged$ aufgerufen.

```
 private void OnTextChanged(object source,
  AutomationEventArgs e)
        {
            AutomationElement ae = source as
             AutomationElement;

            ValuePattern vp = ae.GetCurrentPattern
             (ValuePattern.Pattern) as ValuePattern;
```

Das AutomationElement wird über das Source Argument übergeben und einfach wieder einem AutomationElement ae zugewiesen. Danach holt sich ae über die $GetCurrentPattern$ Eigenschaft das gerade ausgeführte Pattern und legt es in einem neuen ValuePattern vp ab.

```
            if (!ae.Current.HasKeyboardFocus)
            {
                events.Add("wrapper.GetText(\"" +
                 ae.Current.AutomationId + "\", \""
                 + vp.Current.Value + "\");");

                string result = "Prüfung auf Richtigkeit
                 von: " + ae.Current.AutomationId + " =
                 " + vp.Current.Value + "\r\n";
```

```
                GetRecordEventArgs es = new
                 GetRecordEventArgs(result);

                onGetRecord(es);
            }

            else
            {
                events.Add("wrapper.SetValue(\"" +
                 ae.Current.AutomationId + "\", \"" +
                 vp.Current.Value + "\");");

                string result = "Es wurde der Wert " + vp.
                 Current.Value + " in das Textfeld ''" +
                 ae.Current.AutomationId + "''
                 geschrieben.";

                GetRecordEventArgs es = new
                 GetRecordEventArgs(result);

                onGetRecord(es);
            }

        }
```

Die If-Abfrage gibt true zurück, wenn das Textfeld keinen Keyboardfokus hat, also keine Usereingabe stattgefunden hat. In diesem Fall handelt es sich um einen Rückgabewert und keine Benutzereingabe. Diese Unterscheidung ist für den Skriptablauf später wichtig. Innerhalb der If-Abfrage werden zwei Strings zusammengebaut. Bei $events$ handelt es sich um eine Liste vom Typ String, die als Klassenvariable definiert wird.

```
private List<string> events = new List<string>();
```

Mit $events.add$ kann der Liste ein neuer String hinzugefügt werden. Aus dieser Liste wird später das Skript erzeugt, während $result$ der String ist, der zur Kontrolle an die Logausgabe des User Interfaces weitergereicht wird. Die Eigenschaften $ae.Current.AutomationID$ und $vp.Current.Value$ sind bei der späteren Skriptausführung dafür zuständig dem Wrapper mitzuteilen welches Textfeld angesprochen werden soll und welcher Wert hineingeschrieben, bzw. auf welchen Wert geprüft werden muss.

Ähnliche Methoden wie die hier vorgestellte $OnTextChanged$, gibt es mit $OnInvoke$ und $OnToggle$ noch für das Klicken eines Buttons unter WPF und für das Betätigen einer Checkbox.

3.3.4.4 Schreiben des Testskriptes

Die Testskripte werden ebenfalls im Teilprojekt TestController in der Klasse $TController$ erstellt. Nach dem Auswählen des Menüeintrags „Speichern“ > „Speichern Unter“ im GUI (s. Abb. 3.4) öffnet sich ein Save Filedialog, mit dem Speicherort und Name der Skriptdatei ausgewählt werden können. Ist diese Auswahl getrof-

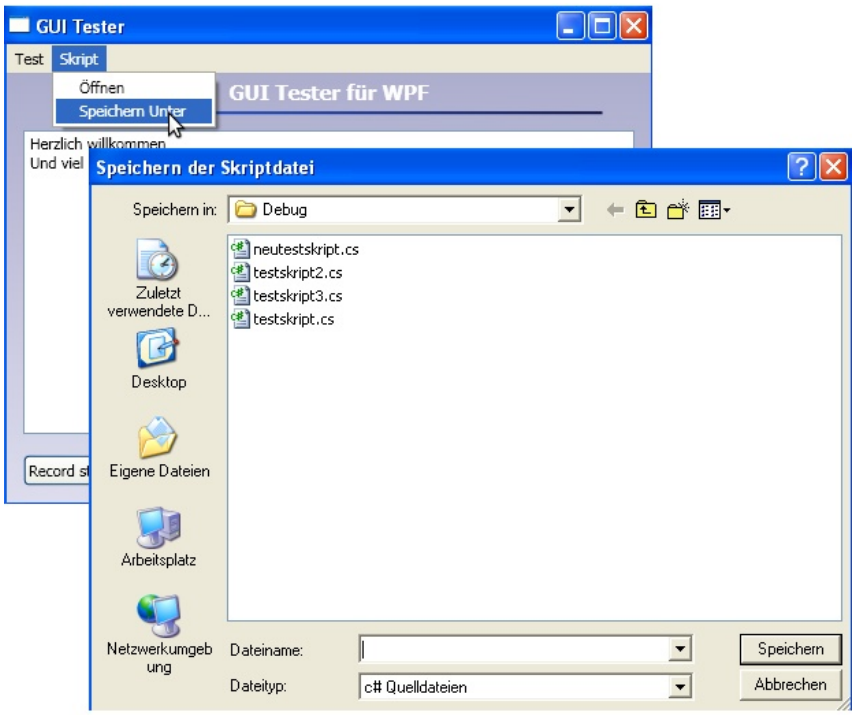

Abbildung 3.4: Save Filedialog

fen, wird aus der Klasse $UITester$ die Methode $saveSkript$ des TestControllers aufgerufen und ihr die, im Save Filedialog getroffene, Auswahl mit übergeben. In der $saveSkript$ Methode wird jetzt zunächst ein neuer Stringbuilder mit dem Namen $UIASkript$ erstellt.

```
StringBuilder UIASkript = new StringBuilder();
```

$UIASkript$ kann nun mittels $AppendLine$ Strings hinzugefügt werden. Auf diese Art und Weise wird zunächst ein wenig Standardtext als Header definiert bevor dann die vorher angelegte Liste $events$ ausgelesen wird.

```
foreach (string s in events)
{
```

```
        UIASkript.AppendLine("\t\t\t" + s);
}
```

Die foreach Anweisung durchläuft die Liste in der Reihenfolge, in der sie angelegt wurde, und entnimmt jeweils einen String, der dann per $Appendline$ zu $UIASkript$ hinzugefügt wird. Die \t Anweisungen dienen nur der Formatierung der Skriptfiles und fügen jeweils einen Tabulator ein. Anschließend werden noch ein paar Zeilen statischer Text hinzugefügt.

```
UIASkript.AppendLine("");
UIASkript.AppendLine("\t\t\t wrapper.errors();");
UIASkript.AppendLine("\t\t\t Controller.CloseApp();");
```

Die Methode $wrapper.errors$ erstellt eine Fehlerzusammenfassung nachdem das Skript durchlaufen wurde und $Controller.CloseApp$ tut nichts weiter als das zu testende Programm zu beenden.

Nachdem das Skript zur optischen Rückmeldung jetzt nochmal über die entsprechenden Events auf dem GUI ausgegeben wurde, kann es als C# Quellcodedatei auf die Festplatte geschrieben werden. Dazu wird als erstes überprüft ob sich am ausgewählten Pfad evtl. schon eine Datei mit dem gleichen Namen befindet.

```
if (File.Exists(sfd.FileName))
            {
                File.Delete(sfd.FileName);
            }
```

Wenn es die Datei bereits gibt, wird diese zunächst gelöscht, da davon ausgegangen wird, dass im save Filedialog einfach eine alte Skriptdatei angeklickt wurde, anstatt einen neuen Namen einzugeben. Gibt es die Datei bislang noch nicht, wird diese mittels $File.Create$ angelegt.

```
File.Create(sfd.FileName).Close();
```

Jetzt kann über einen neuen Streamwriter sw und der $ToString$ Methode des Stringbuilders der Inhalt von $UIASkript$ in die neu angelegte Datei geschrieben werden.

```
StreamWriter sw = new StreamWriter(sfd.FileName);
sw.Write(UIASkript.ToString());
sw.Close();
```

Nun muss lediglich noch der StreamWriter durch die $close$ Eigenschaft wieder geschlossen werden um die Erstellung des Testskriptes abzuschließen.

3.3.5 Replay

Für die replay Funktion des GUI Testers kommt nun zusätzlich zu den beiden Klassen $UITester$ und $TController$ erstmalig der $AWrapper$ zum Einsatz. Hier findet das eigentliche Ansteuern der Bedienelemente statt.

Zum Abspielen eines bereits aufgenommenen Skriptes muss, wie bereits beim Aufnehmen, über das Menü die zu testende Software gestartet werden. Anschließend wird nun über den Menüeintrag „Skript“ > „Öffnen“ das entsprechende Clickevent der Klasse $UITester$ ausgelöst. Dieses öffnet einen Open Filedialog über den nun das entsprechende Skript ausgewählt werden kann. Um das Skript zu starten, ist nur noch ein Klick auf den „Replay“ Button nötig. Nun werden auch hier über die $Referrer$ Methode des Controllers die benötigten Referenzen angelegt. Diese Referenzen werden nun auch an den Wrapper übergeben, da erst von dort die Benutzerelemente angesprochen werden. Da sie sehr zeitintensiv ist, wird die Controller Methode $Skript$ jetzt in einem neuen Thread gestartet. In dieser Methode wird das Skript ausgelesen und kompiliert (s. Kap. 3.3.5.2 – 3.3.5.5). Anschließend kann das Skript abgespielt werden.

3.3.5.1 Aufbau der Testskripte

Durch die Entscheidung den kompletten Header, den eine Quellcodedatei benötigt, in der CodeDOM Methode $TController.Skript$ zu definieren, beinhalten die Skripte nur noch die tatsächlichen Methodenaufrufe mit den zu übergebenen Argumenten.

Soll z.B. die Invoke Methode des Wrappers aufgerufen werden um den WPF Button mit der ControlID $button1$ zu drücken, lautet die entsprechende Skriptzeile:

```
wrapper.Invoke("button1");
```

Die letzten beiden Zeilen jedes Skriptes sind gleich und beinhalten neben dem Aufruf der Error Methode des Wrappers noch den einzigen Methodenaufruf im Controller, den ein Skript ausführen muss.

```
Controller.CloseApp();
```

Dieser Methodenaufruf schließt das zu testende Programm. Ist ein Schließen nicht erwünscht weil evtl. mehrere Testläufe kurz hintereinander durchgeführt werden sollen, muss diese Zeile im Skript entfernt werden.

Ein Beispiel für ein komplettes Testskript findet sich in Kapitel 6.1.

3.3.5.2 Sicherheitsaspekte in Hinsicht auf das Skripten

Die Methoden, die ein Skript aufrufen können muss, lassen sich gut eingrenzen. Es handelt sich dabei ausschließlich um die Methoden des Wrappers, die die Automation Patterns implementieren, sowie die Error Methode und die CloseApp Methode des Controllers. Um zu verhindern, dass die Skripte einfach dahingehend abgeändert werden können, dass sie andere Methoden wie z.B. $TController.Skript$, die für das Kompilieren der Skripte zuständig ist, aufrufen und dadurch unvorhergesehene Effekte und einen Absturz des GUI Testers verursachen, habe ich mich dazu entschieden den Skripten keinen direkten Zugriff auf den Controller und den Wrapper zu geben, sondern die Skriptausführung über Interfaces zu realisieren. Zu diesem Zweck wurde den Klassen $AWrapper$ eine Datei IAWrapper.cs und $TController$ entsprechend ITController.cs hinzugefügt. Die Skripte bekommen über den Compiler nur Instanzen dieser Interfaces (näheres hierzu s. Kapitel 3.3.5.3), nicht aber der eigentlichen Klassen, übergeben. Die Interfaces implementieren die Methoden, auf die den Skripten der Zugriff gestattet wird. Ein Interface selbst enthält überhaupt keine Quellcodeimplementierung, sondern ausnahmslos nur abstrakte Definitionen [Kue08c]. Das Controller Interface sieht beispielsweise wie folgt aus:

```
using System;
namespace TestController
{
    public interface ITController
    {
        void CloseApp();
    }
}
```

Es läuft im gleichen Namespace wie die Klasse $TController$ und besitzt ein einziges Objekt, das als Interface deklariert ist. Darin werden dann die Methoden, auf die der Zugriff über das Interface möglich ist, implementiert. Beim Controller ist das nur eine einzige Methode, die zum Schließen der zu testenden Applikation dient.

Damit die Interfaces funktionieren, müssen nun noch die Klassen, für die sie als Schnittstellen dienen, von Ihnen abgeleitet sein.

```
public class TController : TestController.ITController
```

3.3.5.3 Anlegen einer Code CompileUnit

Die Methode „skript“ , der Klasse $TController$, beinhaltet die CodeDOM Funktionalität. Hier wird das Skript zur Laufzeit kompiliert und anschließend auch gestartet. Ein CodeDOM Objekt ist in einer Baumstruktur aufgebaut und besteht aus

einer Code CompileUnit, verschiedenen Compilerparmetern und dem Code Provider.

Die Code CompileUnit ist der Stammcontainer eines CodeDOM Baumes. Hier werden Verweise auf Attribute, Namespaces und Assemblys, die für den Kompiliervorgang benötigt werden, abgespeichert.

```
CodeCompileUnit compU = new CodeCompileUnit();
CodeNamespace nameSpace = new CodeNamespace("UIASkript");
CodeTypeDeclaration className = new CodeTypeDeclaration
 ("Skript");

CodeMemberMethod methodName = new CodeMemberMethod();
```

Hier wird zuerst eine neue CompileUnit $compU$ erzeugt, dann wird ein Namespace für das Testskript mit Namen $UIASkript$ definiert und schließlich wird ein neues CodeTypeDeclaration Objekt mit dem Namen $className$ erstellt, das den Namen der Klasse für das Testskript beinhaltet. Dieses Objekt kann dann später in einem zweiten Schritt als Klasse deklariert werden. Zuletzt wird noch eine Methode für das Skript angelegt. Namespace, Klasse und Methode könnten alternativ auch direkt im Testskript, wie bei jeder anderen C# Datei auch, angelegt werden. ich habe mich aber dagegen entschieden, da so die Skripte nur noch den Quellcode enthalten, der wirklich dem Testen dient und es deshalb nicht zu Fehlern durch ein Editieren der Files und versehentliches Löschen einer Headerzeile kommen kann. Außerdem muss der Header so nur ein einziges Mal zentral im Quellcode definiert werden und verschlankt so die Skripte.

Über eine foreach Anweisung werden nun die Assemblys des GUI Testers in die Stringliste $assemblyNames$ geschrieben. Diese Liste wird später gebraucht um diese Assemblys für das Skript verfügbar zu machen.

```
List<string> assemblyNames = new List<string>();
foreach (Assembly asm in AppDomain.CurrentDomain.
 GetAssemblies())
{
   if ((asm.Location != null) && (asm.Location.
    Length > 0))
   {
       assemblyNames.Add(new Uri(asm.Location).
        LocalPath);
   }
}
```

Als nächstes wird das angelegte Objekt $nameSpace$ als Namespace zur CompileUnit hinzugefügt. Anschließend können die konkret benötigten Namespaces über die $Imports.Add$ Anweisung in $nameSpace$ importiert werden. Dies sieht exemplarisch für System und System.IO aus wie folgt:

```
compU.Namespaces.Add(nameSpace);
nameSpace.Imports.Add(new CodeNamespaceImport("System"));
nameSpace.Imports.Add(new CodeNamespaceImport("System.IO"));
```

Diese Namespaces können nun nach dem gleichen Muster beliebig vervollständigt werden. Es steht hier alles zur Verfügung, was auch vom Visual Studio aus genutzt werden kann. Um den Baum weiter zu vervollständigen, wird jetzt $className$ zum Namespace hinzugefügt.

```
nameSpace.Types.Add(className);
```

Um nun aus dem $className$ Objekt auch wirklich eine Klasse zu machen, muss darauf der $IsClass$ Operator angewendet werden.

```
className.IsClass = true;
```

Diese Klasse soll nun noch als public deklariert werden und bekommt als Membermethode $methodName$ zugeordnet, die ebenfalls public sein soll.

```
className.TypeAttributes = TypeAttributes.Public;
className.Members.Add(methodName);
methodName.Attributes = MemberAttributes.Public;
```

Abschließend fehlt $methodName$ nur noch ein return Typ und ein Name:

```
methodName.ReturnType = new CodeTypeReference(typeof(void));
methodName.Name = "testSkript";
```

Da die Methode nichts zurück gibt, bekommt sie den return Typ $void$ und als Name $testSkript$. Damit ist der CodeDOM Baum auch bereits komplett und kann genutzt werden. Im nächsten Schritt müssen der Methode sowie dem Compiler noch ein paar Parameter übergeben werden.

3.3.5.4 Die Compiler- und Methodenparameter

Über die Parameter der Methode wird jetzt festgelegt auf welche Klassen das Skript bei der Ausführung Zugriff erhält. Ich erlaube den Zugriff wie in Kapitel 3.3.5.2 beschrieben aus Sicherheitsaspekten nur auf die Interfaces der Klassen $AWrapper$ und $TController$.

```
methodName.Parameters.Add(new
 CodeParameterDeclarationExpression(typeof(IAWrapper),
 "wrapper"));

methodName.Parameters.Add(new
 CodeParameterDeclarationExpression(typeof(ITController),
 "Controller"));
```

Um die Skripte möglichst gut lesbar zu machen, ist eine eindeutige Namensgebung der Klassen von Vorteil. Ich habe mich für $wrapper$ und $Controller$ als Namen für die Interface Instanzen entschieden. Anschließend muss die Methode aus dem Skriptfile eingelesen werden. Hierfür habe ich eine Funktion $ReadFile$ geschrieben, die einen String erwartet, der den Pfad des auszulesenden Files enthält. Dieser wird dann über einen StreamReader ausgelesen, der Inhalt in einem neuen String gespeichert und wieder an die aufrufende Methode zurückgegeben. In den Baum lässt sie sich über die $Statements.Add$ Funktion als sogenannte $CodeSnippetExpression$ einbinden.

```
methodName.Statements.Add(new CodeSnippetExpression(ReadFile
 (content)));
```

Bei $content$ handelt es sich um den benötigten String, der das Ergebnis des Open Filedialogs übergeben bekommt.

Der Compiler braucht nun ebenfalls noch ein paar Randparameter. Diese Parameter werden an ein $CompilerParameters$ Objekt übergeben, das zunächst erstellt werden muss. Diesem Objekt können dann die zu verwendenden Assemblys in einem Array übergeben werden. Hierfür verwende ich den $ToArray$ Operator auf die oben angelegte Liste $assemblyNames$ an.

```
CompilerParameters compilerParameters = new
CompilerParameters((string[])assemblyNames.ToArray());
```

Jetzt muss eine Entscheidung über das Ausgangsformat des kompilierten Skriptes getroffen werden. Ich hatte mich zunächst dafür entschieden eine Assembly im DLL Format auf C: zu erstellen. Da man aber im Voraus nicht wissen kann in welchen Ordnern der User Schreibrechte hat, und die Assembly ja nur während des Tests gebraucht wird, lasse ich sie nun nur noch im RAM erstellen. Auf diese Art benötigt der GUI Tester keine Schreibrechte auf der Festplatte.

```
compilerParameters.GenerateInMemory = true;
```

Gerade wenn die Datei nur im Speicher erstellt wird, ist es wichtig, dass diese möglichst klein, schnell und effizient ist. Um das zu erreichen, wird dem Compiler die

Option $Optimize$ mitgegeben. Weiterhin wird über die $target$ Option festgelegt in welchen Dateityp das Skript kompiliert werden soll. Das hier eingesetzte $library$ sorgt für die Kompilierung in eine DLL Library.

```
compilerParameters.CompilerOptions = "/target:library
 /optimize";
```

Möchte man statt einer DLL beispielsweise lieber eine Exe erzeugen, würde es genügen das Target auf $/target : exe$ zu ändern.

Um gegebenenfalls ein Debugging zu ermöglichen, muss die Methode $IncludeDebugInformation$ den Wert true übergeben bekommen. Hiermit werden der Assembly die benötigten Debug Informationen übergeben.

```
compilerParameters.IncludeDebugInformation = true;
```

3.3.5.5 Der Code Provider und die Kompilierung zur Laufzeit

Der Code Provider ist die Klasse, die den Zugriff auf den Compiler bereitstellt. Hier wird erst die Entscheidung getroffen in welcher Sprache das Programm übersetzt werden soll. Je nachdem welcher Code Provider angelegt wird, ist ein einfacher Wechsel zwischen z.B. C# oder Visual Basic möglich. Da die Testskripte des GUI Testers bislang nur in C# geplant sind, wird lediglich eine Instanz eines $CSharpCodeProvider$ benötigt.

```
Microsoft.CSharp.CSharpCodeProvider csc = new Microsoft.
 CSharp.CSharpCodeProvider();

CompilerResults res = null;
res = csc.CompileAssemblyFromDom(compilerParameters, compU);
```

Zum Auslesen eventueller Programmierfehler dient die Variable res vom Typ $CompilerResults$. Über eine If-Bedingung kann nun abgefragt werden ob res noch leer ist. Falls dem nicht so ist, werden die Compilerfehler nacheinander über $res.ToString$ an das Logfenster des GUIs weitergeleitet.

Gab es keine Fehler wird ein Objekt der Klasse $Skript$ im Namensraum $UIASkript$, also dem Testskript, erstellt.

```
objFunc = res.CompiledAssembly.CreateInstance("UIASkript.
 Skript", true);
```

Nun muss ein neuer Delegat auf die Methode des Testskriptes erstellt werden, um diese aufrufen zu können.

```
runDelegate = (RunMacro)Delegate.CreateDelegate(typeof
 (RunMacro), objFunc, methodName.Name);
```

Wenn der Delegat erfolgreich erstellt werden konnte, müssen ihm nun nur noch die Instanzen der beiden Interfaces des Wrappers und des Controllers übergeben werden, damit das Skript ausgeführt wird.

```
if (runDelegate != null)
{
   runDelegate(wrapper, Controller);
}
```

3.3.5.6 Aufbau einer Automatisierungsmethode im Wrapper

Die Hauptaufgabe des Wrappers ist es Methoden bereitzustellen, mit denen der tatsächliche Zugriff auf ein GUI für die Testskripte bereitgestellt wird. Anhand der $SetValue$ Methode soll hier einmal exemplarisch dargestellt werden, wie so eine Methode aufgebaut ist. $SetValue$ ist für das Beschreiben von Textfeldern verantwortlich und daher bereits eher komplex.

Die Methode bekommt durch das Skript mit der $ControlID$ den Namen des zu beschreibenden Textfeldes und durch das $Value$ den eigentlichen zu schreibenden Inhalt in Stringform übergeben. Anschließend wird das Textfeld im Baum über seinen Namen gesucht und einem AutomationElement ae zugewiesen.

```
public void SetValue(string ControlID, string Value)
{
     AutomationElement ae = aeForm.FindFirst(TreeScope.
      Descendants, new PropertyCondition(AutomationElement
      .AutomationIdProperty, ControlID));
```

Für jeden Zugriff auf ein Bedienelement ist ein spezielles „ControlPattern“ nötig. In diesem Fall ist es das $ValuePattern$, das die Value Eigenschaft eines GUI Elementes verändern kann. Es muss nur noch über ein $SetValue(Value)$ der Value String übergeben werden und schon wird dieser in das Textfeld geschrieben.

```
ValuePattern vp = (ValuePattern)ae.GetCurrentPattern(
 ValuePattern.Pattern);

vp.SetValue(Value);
```

Abschließend wird noch der Ausgabestring $result$ für das Logfenster im GUI zusammengebaut und an die Klasse $GetTExtEventArgs$ übergeben, bei der es sich um eine Klasse handelt, die sich von $EventArgs$ ableitet und den ihr übergebenen String für das anschließend aufgerufene Event $onGetText$ verfügbar macht.

```
string result = "Es wurde der Wert " + Value + " in das
 Textfeld ''" + ControlID + "'' geschrieben.";

GetTextEventArgs e = new GetTextEventArgs(result);
onGetText(e);
```

Nach diesem Prinzip sind alle Steuermethoden im Wrapper aufgebaut. Sie unterscheiden sich nur in der Anzahl der übergebenen Argumente (da man z.B. für einen Invoke keinen Value braucht, sondern einzig den Namen des zu klickenden Buttons) und den eingesetzten Pattern Objekten.

3.4 Test

Natürlich muss auch ein Testprogramm während seiner Entwicklung selbst getestet werden. Da das Programm bis zum jetzigen Entwicklungsstand sehr übersichtlich gehalten ist und ich mich aus Zeitmangel nicht mehr in das Erstellen von Unittests mit NUnit einarbeiten wollte, habe ich mich dazu entschieden den GUI Tester manuell zu testen. Dafür habe ich einen in C# und WPF erstellten Taschenrechner genommen, den ich immer gerade so erweitert habe, dass er die Bedienelemente bietet, die der GUI Tester bedienen kann (Quellcode s. Kap. 6.2).

3.4.1 GUI Test eines WPF Taschenrechners

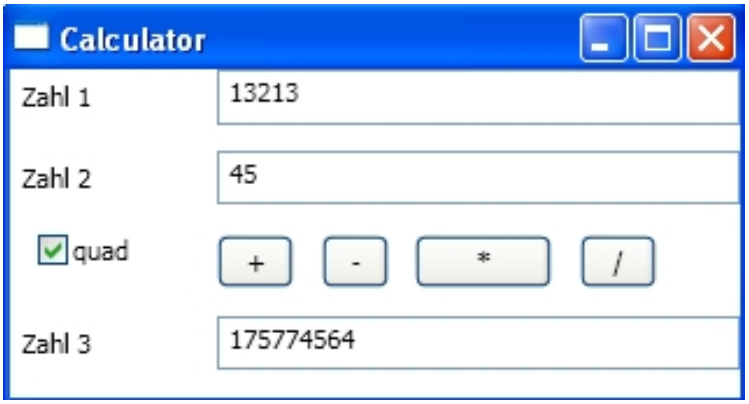

Abbildung 3.5: WPF Taschenrechner als Testobjekt für den GUI Tester

Das Testskript in Kapitel 6.1 wurde für den Taschenrechner in Abbildung 3.5 erstellt. Er verfügt über drei Texteingabefelder. In die oberen beiden Felder können die Operanden eingegeben werden. Ein Klick auf das entsprechende Symbol führt die mathematische Berechnung aus. Das Aktivieren der Checkbox quadriert das Ergebnis. Wird der Haken wieder entfernt wird auch die Quadrierung rückgängig gemacht (es wird also die Wurzel gezogen). Das dritte Textfeld ist das Ergebnisfeld. Dieses wird vom GUI Tester während eines Aufnahmevorgangs ausgelesen, um die dort eingetragenen Werte als Soll Werte zur Überprüfung bei einem Abspielvorgang zu speichern.

Zur Überprüfung der Fehlererkennung des GUI Testers habe ich ein Testskript per Hand so abgeändert, dass der erwartete Wert nicht mehr mit dem tatsächlichen übereinstimmte. Dadurch fällt die Überprüfung beim Abspielen des Testskriptes negativ aus und es wird im Logfenster entsprechend ein Fehler angezeigt.

Ein Listing des Quellcodes für den Taschenrechner findet sich im Anhang unter Punkt 6.2. Die Bedienelemente wurden einfach in der Standard Benamung belassen

und auch die Automation- sowie ControlIDs wurden nicht per Hand verändert.

4 Zusammenfassung

In dieser Arbeit wurde erfolgreich ein Tool erstellt, mit dem es möglich ist Testskripte für User Interface Tests beliebiger WPF Anwendungen aufzunehmen und wieder abzuspielen.

Als Automatisierungsframework kam Microsofts UI Automation zum Einsatz, da dies ein kostenloser und gut dokumentierter Bestandteil des .NET Frameworks ist. Es steht somit für jeden .NET Entwickler zur Verfügung und muss nicht einzeln zugekauft werden. Als Bestandteil des .NET Frameworks ist es sehr gut integriert und soll auch in Zukunft durch Microsoft beständig weiterentwickelt werden.

Für die Skriptfunktionen kam mit dem CodeDOM Namespace ebenfalls eine standard .NET Komponente anstelle, einer externen Skriptsprache, zum Einsatz. Dies hat den entscheidenden Vorteil, dass die Skripte in C# geschrieben sein können, was sie für den Tester sehr viel leichter zu lesen und gegebenenfalls auch einfach veränderbar macht. Ein weiterer Vorteil ist das Dateiformat einer C# Quellcodedatei. Diese sind unter anderem mit dem Visual Studio von Microsoft bearbeitbar, was es einem durch Funktionen wie „Intelli Sense“ sehr einfach macht sie manuell zu verändern und viel Unterstützung bietet. Durch die Möglichkeiten von CodeDOM den kompletten Header des Skriptes im Quellcode aufzubauen, können die Skriptdateien extrem schlank gehalten werden und sich im wesentlichen auf die Methodenaufrufe beschränken.

Auftretende Probleme durch zu lange Operationen, die ein blockierendes GUI zur Folge hatten, wurden mit Multithreading gelöst. Dadurch wurde es nötig eine eventbasierte Kommunikation zwischen den einzelnen Klassen aufzubauen, da ein GUI in WPF Anwendungen nur noch durch seinen eigenen Thread aktualisiert werden kann. Die durch die Events aufgerufen Methoden können per $Dispatcher.Invoke$ in den GUI Thread wechseln und somit dieses Problem umgehen.

Das Klassendiagramm, des fertigen GUI Testers in Abbildung 4.1, wurde mit Hilfe der Software „ObjectIF“ der Firma microTOOL als Repräsentation des fertigen Programms erstellt. Die Pfeile zeigen an in welcher Richtung die Klassen miteinander kommunizieren. Die Struktur der Entwurfsklassen in Abbildung 3.3 wur-

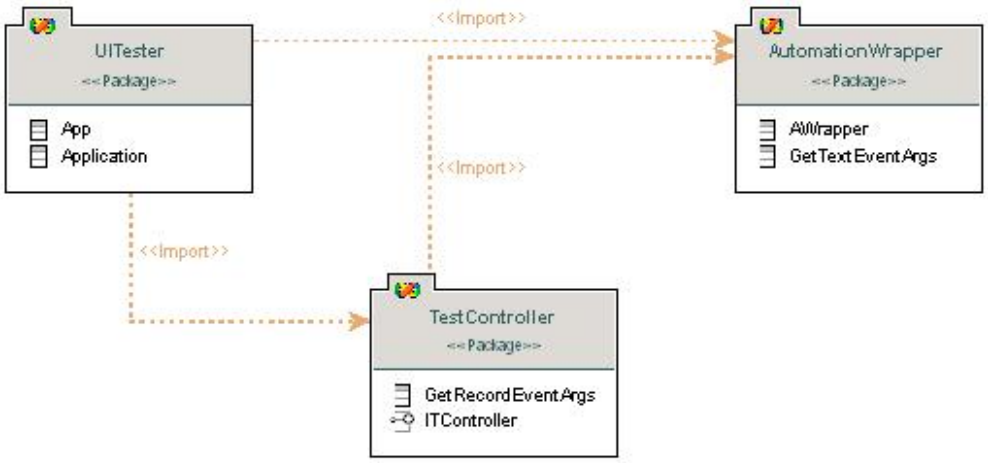

Abbildung 4.1: ObjectIF Klassendiagramm

de weitestgehend beibehalten. Die Anzahl der Methoden stimmt allerdings nicht mit dem Endergebnis überein. Hier kamen noch einige Dinge hinzu, die sich erst im Implementierungsprozess ergaben und im vornhinein nicht zu berücksichtigen waren.

5 Erweiterungsmöglichkeiten

Der GUI Tester bietet momentan sehr rudimentäre Testmöglichkeiten an da er noch nicht besonders viele WPF Steuerelemente ansprechen kann. Die Methoden zum Steuern der Bedienelemente befinden sich vollständig in der Wrapper Klasse und sind nach einem immer gleich bleibenden Muster aufgebaut. Hier müssten die fehlenden Steuerelemente mit den entsprechenden ControlPatterns hinzugefügt werden.

Um auch das Aufnehmen der Skripte entsprechend auf neue Steuerelemente auszuweiten, ist es notwendig die $Recording$ Methode der Klasse $TController$ entsprechend samt den zugehörigen OnXYZ Methoden auszubauen. Dies ist aufgrund des hohen zeitlichen Aufwands aber eine Aufgabe für eine separate Studienarbeit.

Eine weitere denkbare Ausbaustufe wäre dann die Testfunktionalität auf Forms und Win32 GUIs auszubauen. Hierfür gilt im Prinzip das Gleiche wie für die restlichen WPF Steuerelemente. Es müssen die, zur Behandlung von Forms/Win32 Steuerelementen geeigneten, Automation Methoden im Wrapper und Controller hinzugefügt werden.

Sobald das .NET Framework um diese Möglichkeit erweitert wird, wäre es Wünschenswert das Starten der zu testenden Applikationen wieder aus dem Menü zu entfernen und es mit in die Testskripte zu integrieren. Dies würde den immer wiederkehrenden separaten Schritt des Startens der Anwendung überflüssig machen und mehr Komfort bieten. Außerdem kann es so nicht zu möglichen Verwechslungen beim Abspielen eines Testskriptes und der dazugehörigen Anwendung kommen.

Erweiterungspotential besteht ebenfalls noch in der eingesetzten „Skriptsprache“ . Hier könnte man eine Option einbauen seine Skripte in C#, Visual Basic, Python oder einer anderen beliebigen .NET Sprache zu speichern und zu kompilieren. Dazu wäre es nur nötig im TestController die Skriptsyntax auf die entsprechenden Sprachen anzupassen und anschließend in der $skript$ Methode die benötigten CodeProvider anzulegen und auswählbar zu machen.

6 Anhang

6.1 Testskript

```
//-----------------------------------------------
// Description: Automated test script
// Generator: AutomatedScriptGenerator
// Date: 05.03.2009 11:25:47
//-----------------------------------------------

wrapper.SetValue("textBox1", "1");
wrapper.SetValue("textBox1", "12");
wrapper.SetValue("textBox1", "123");
wrapper.SetValue("textBox2", "2");
wrapper.SetValue("textBox2", "25");
wrapper.SetValue("textBox2", "2565");
wrapper.Invoke("button1");
wrapper.GetText("textBox3", "2687");
wrapper.SetValue("textBox1", "1");
wrapper.SetValue("textBox1", "12");
wrapper.SetValue("textBox1", "123");
wrapper.SetValue("textBox1", "12");
wrapper.SetValue("textBox1", "125");
wrapper.SetValue("textBox2", "1");
wrapper.SetValue("textBox2", "12");
wrapper.Invoke("button4");
wrapper.GetText("textBox3", "10,4166666666667");
wrapper.SetValue("textBox1", "4");
wrapper.SetValue("textBox1", "448");
wrapper.SetValue("textBox2", "6");
wrapper.SetValue("textBox2", "65");
wrapper.SetValue("textBox2", "656");
wrapper.Invoke("button2");
wrapper.GetText("textBox3", "-207");
```

```
 wrapper.errors();
Controller.CloseApp();
```

6.2 WPF Calculator

```
public partial class Window1 : Window
{
   public Window1()
   {
      InitializeComponent();
   }

   private void button1_Click(object sender,
    RoutedEventArgs e)
   {
      textBox3.Text = (Double.Parse(textBox1.Text)
       + Double.Parse(textBox2.Text)).ToString();
   }

   private void button2_Click(object sender,
    RoutedEventArgs e)
   {
      textBox3.Text = (Double.Parse(textBox1.Text)
       - Double.Parse(textBox2.Text)).ToString();
   }

   private void button3_Click(object sender,
    RoutedEventArgs e)
   {
      textBox3.Text = (Double.Parse(textBox1.Text)
       * Double.Parse(textBox2.Text)).ToString();
   }

   private void button4_Click(object sender,
    RoutedEventArgs e)
   {
      textBox3.Text = (Double.Parse(textBox1.Text)
       / Double.Parse(textBox2.Text)).ToString();
```

```
    }

    private void chck_quad(object sender,
     RoutedEventArgs e)
    {
       textBox3.Text = (Double.Parse(textBox3.Text)
        * Double.Parse(textBox3.Text)).ToString();
    }

    private void unch_quad(object sender,
     RoutedEventArgs e)
    {
       textBox3.Text = (Math.Sqrt(Double.Parse
        (textBox3.Text))).ToString();
    }
}
```

7 Literaturverzeichnis

[Bau08] Sascha Baumann. C#-Code dynamisch kompilieren mit CodeDom. *DotNetBlog*, August 2008. http://dotnetblog.saschabaumann.com/2008/08/c-code-dynamisch-kompilieren-mit-codedom-ein-beispiel/, Stand 23.03.09.

[Her08] Jenö Herget. UI Test-Automatisierung. *dot.net magazin*, pages 39–44, 12.2008.

[Kan08] Masahiko Kaneko. Windows Automation API 3.0 Overview. *CoDe Magazine*, 2008. http://www.code-magazine.com/Article.aspx?quickid=0810042, Stand 23.03.09.

[Kor02] Brian J. Korzeniowski. Microsoft .NET CodeDom Technology. *15 seconds*, September 2002. http://www.15seconds.com/Issue/020917.htm/, Stand 23.03.09.

[Kry08] Veikko Krypczyk. Von der Idee zum fertigen Programm, Teil1. *entwickler magazin*, page 54, November/Dezember 2008.

[Kue08a] Andreas Kuehnel. *Visual C# 2008*, page 32 ff. Galileo Press, 2008. http://www.galileocomputing.de/openbook/csharp/.

[Kue08b] Andreas Kuehnel. *Visual C# 2008*, page 1037 ff. Galileo Press, 2008. http://www.galileocomputing.de/openbook/csharp/.

[Kue08c] Andreas Kuehnel. *Visual C# 2008*, pages 265 – 266. Galileo Press, 2008. http://www.galileocomputing.de/openbook/csharp/.

[McC08] Dr. James McCaffrey. Test Run. Februar 2008. http://msdn.microsoft.com/en-us/magazine/cc163288.aspx, Stand 23.03.09.

[Mic07] Microsoft. Microsoft and Novell Celebrate Year of Interoperability, Expand Collaboration Agreement. November 2007. http://www.microsoft.com/presspass/press/2007/nov07/11-07MSNovell1YearPR.mspx, Stand 08.04.09.

[Mic08] Microsoft. MSDN Accessibility. 2008. http://msdn.microsoft.com/en-us/library/ms747327.aspx, Stand 23.03.09.

[msd] UI Automation Tree Overview. *msdn*. http://msdn.microsoft.com/en-us/library/ms741931.aspx, Stand 01.05.09.

[msd07a] AutomationPattern-Klasse. *msdn*, November 2007. http://msdn.microsoft.com/de-de/library/system.windows.automation.automationpattern.aspx, Stand 16.05.09.

[msd07b] EventArgs-Klasse. *msdn*, November 2007. http://msdn.microsoft.com/de-de/library/system.eventargs.aspx, Stand 14.05.09.

[msd07c] UI Automation Tree Overview. *msdn*, November 2007. http://msdn.microsoft.com/de-de/library/system.windows.threading.dispatcher.invoke.aspx, Stand 04.05.09.

[msd08] Accessibility Information for Developers. *Microsoft Accessibility*, August 2008. http://www.microsoft.com/enable/developer.aspx, Stand 23.03.09.

[msd09] Windows Automation API: Microsoft Active Accessibility. *msdn*, April 2009. http://msdn.microsoft.com/en-us/library/dd373592(VS.85).aspx, Stand 02.05.09.

[myc09] Basistechnologien und allgemeine .NET-Klassen. *myCSharp*, 2009. http://www.mycsharp.de/wbb2/board.php?boardid=23.

[Oes05a] Bernd Oestereich. *Analyse und Design mit UML2*, pages 21–24. Oldenbourg, 2005.

[Oes05b] Bernd Oestereich. *Analyse und Design mit UML2*, page 14 ff. Oldenbourg, 2005.

[Pro08] Mono Project. Mono Accessibility. 2008. http://www.mono-project.com/Accessibility/, Stand 08.04.09.

[ran09] Ranorex Professional. *Ranorex Hersteller Website*, 2009. http://www.ranorex.de/, Stand 03.04.09.

[Sac09] Saki Sachin. UI Automation Framework using WPF. März 2009. http://www.codeproject.com/script/Articles/ArticleVersion.aspx?aid=34038&av=0, Stand 23.03.09.

[Was08] Tobias Wassermann. Grundlagen von XAML und WPF im schnelldurchlauf für Umsteiger. *Entwicklermagazin*, pages 14–18, Mai/Juni 2008.

[whi] Software Testing : Black box and White box Techniques. *Exforsysy Inc. Website*. http://www.exforsys.com/tech-articles/testing/software-testing-black-box-and-white-box-techniques.html, Stand 26.05.09.

8 Abbildungsverzeichnis

Erklärungen

Eidesstattliche Erklärung

Hiermit versichere ich, diese Arbeit selbständig verfasst und nur die angegebenen Quellen verwendet zu haben. Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt.

…….…….…… ……
(Datum) (Unterschrift/en)

Einverständniserklärung

Ich bin / wir sind damit einverstanden, dass von meiner / unserer Diplomarbeit /Masterarbeit gem. §§ 16 und 17 UrhG Vervielfältigungsstücke erstellt werden können, um sie an Dritte weiterzugeben.
Mein / unser Einverständnis erstreckt sich auch darauf, dass die Diplomarbeit zu diesem Zweck an Dritte weitergegeben werden kann.

☐ einverstanden

☐ nicht einverstanden

Dauer der Sperre:

☐ Jahre

☐ unbefristet

…….…….…… ……
(Datum) (Unterschrift/en)

Einverständniserklärung.doc