Diplomarbeit zum Thema

Entwurf und Realisierung eines Protokollmodelleditors

vorgelegt dem
Lehrstuhl für Softwaretechnologie des
Instituts für Software- und Multimediatechnik
an der Fakultät Informatik, TU Dresden

von **Christoph Schmidt**
31. Januar 2010

Verantwortlicher Hochschullehrer: Prof. Dr. rer. nat. habil. Uwe Aßmann

Betreuer: Dipl.-Inform. Falk Hartmann, ubigrate GmbH

Inhaltsverzeichnis

Abstract 1

1 Einführung 2
1.1 Vorstudie zu MDMLEdit . . . 2
1.2 Ziele dieser Arbeit . . . 4
1.3 Einführung in das ubigrate Integrationsframework . . . 6
1.3.1 Allgemeine Funktionsweise . . . 6
1.3.2 Metamodel-Mapping und verwandte Ansätze . . . 6

2 Schwerpunkte der MDMLEdit-Weiterentwicklung 11
2.1 Modellgetriebene Generierung von Teilen des Editors . . . 11
2.1.1 Kandidatenanalyse . . . 12
2.1.2 Generierungskontext . . . 21
2.1.3 Generierungstechniken – Überblick und Auswahl . . . 24
2.1.4 Proof-of-Concept: Generierung von PropertiesSheetAdaptern . . . 26
2.1.5 Auswirkung auf den Implementierungsaufwand . . . 38
2.2 Umschaltung der PDU-Ansicht zwischen Abstrakter Syntax und Konkreter Syntax . . . 44
2.2.1 Einführung Abstrakte vs. Konkrete Syntax (AS vs. CS) . . . 44
2.2.2 Realisierung der Sichtumschaltung in MDMLEdit . . . 46
2.3 Dynamische (Meta-)Modellerweiterungen . . . 50
2.3.1 Dialog für Einstellungen am Nachrichtenmodell selbst . . . 50
2.3.2 Einbindung von TypeLibraries und Bindable-Typwechsel . . . 52
2.3.3 Unterstützung für EncodingLibraries . . . 56
2.4 Abkehr von geisterhaften Konnektoren – SemiApparitionalConnections 58
2.5 Fortgesetzte Implementierung . . . 61
2.5.1 Einfügen von Komponenten . . . 61
2.5.2 Einfügen in komplexen Komponenten . . . 63
2.5.3 Nachrichtenwechsel via Command . . . 69
2.5.4 Weitere Verbesserungen . . . 69

3 Testsuite für die grafische Benutzeroberfläche 71
3.1 Existierende Werkzeuge für den Oberflächentest von SWT 71
3.2 Evaluation von QF-Test und SWTBot 73
3.2.1 QF-Test 74
3.2.2 SWTBot 82
3.2.3 Fazit und Auswahl 88
3.2.4 Testautomatisierung 90
3.3 Die Testsuite für den Oberflächentest 92
3.3.1 Prozeduren 93
3.3.2 Testfälle 93
3.4 Ausblick 95

4 Validierung: Tutorial zur Erstellung eines Nachrichtenprotokollmodells 97
4.1 Das Gerät: Devantech SRF02 97
4.2 Die Erstellung des SRF02/I2C-Protokolls mit MDMLEdit 100
4.2.1 Ein Protokollmodell anlegen 100
4.2.2 Die Oberfläche von MDMLEdit 105
4.2.3 Grundeinstellungen am Protokollmodell 106
4.2.4 Eine PDU anlegen 112
4.2.5 Bearbeitung der PDU 114
4.2.6 Speichern des Protokolls 123
4.3 Fazit 125

5 Zusammenfassung und Ausblick 129

Literaturverzeichnis 136

Abbildungsverzeichnis 140

Tabellenverzeichnis 143

Glossar 144

3 Testsuite für die grafische Benutzeroberfläche

Die Editoroberfläche soll mittels eines Capture-Replay-Tools einem Regressionstest unterworfen werden. Zentraler Punkt des Oberflächentests ist die Prüfung, ob das Programm korrekt auf Nutzereingaben reagiert und ob der Editor korrekt in Eclipse integriert ist.

Zu diesem Zweck wird im Folgenden zuerst ein Überblick über die verfügbaren Werkzeuge für den Java-Oberflächentest gegeben. Zwei dieser Tools werden ausgewählt und näher evaluiert. Abschließend erfolgt eine Beschreibung der mit dem ausgewählten Werkzeug erstellten Testsuite.

3.1 Existierende Werkzeuge für den Oberflächentest von SWT

Gleich zu Beginn der Marktanalyse wird klar, dass die Mehrzahl der GUI-Testwerkzeuge auf Swing bzw. das AWT-Toolkit ausgerichtet sind. Da Oberflächen unter Java traditionell eher mit Swing assoziiert werden, sind Tools für den Test von SWT-Oberflächen vergleichsweise schwer zu finden und zudem entweder kommerzieller Natur oder in einem Entwicklungsstadium, das nicht für den Produktiveinsatz geeignet ist. Tabelle 3.1 bietet einen Überblick über die bei meiner Recherche vorgefundenen Werkzeuge für den Test von Java-Oberflächen.

Aus der Menge der vorgefundenen Tools werden im Folgenden zwei vom Ansatz her eher unterschiedliche Werkzeuge ausgewählt und gegenübergestellt. Im Anschluss erfolgt die Entscheidung für eines der beiden. Freundlicherweise stellte QFS für diese Diplomarbeit eine kostenfreie akademische Lizenz zur Verfügung, so dass QFS' *QF-Test* die professionellen Capture-Replay-Werkzeuge vertreten kann. Auf der anderen Seite soll ein aktuelles Open-Source-Werkzeug, *SWTBot*, die Position der zur Verfügung stehenden freien und stabilen GUI-Test-Tools verteidigen. Zwar ist in der Aufgaben-

Werkzeug	Lizenz	SUT-GUI-Toolkit-Unterstützung			
		SWT	AWT	Web	sonst.
Abbot 1.0.2 [abb]	OSS CPL [lic06b]	⊠[1]	⊠	□	
Bredex GUIdancer 3.1 [bre]	kommerziell (ab 3900 EUR)	⊠[2]	⊠	⊠	
Froglogic Squish for Java 3.2.3 [squ]	kommerziell (ab 2400 EUR)	⊠	⊠	□	
Instantiations WindowTester 3.9.1 [win]	kommerziell (ab 629 USD)	⊠	⊠	□	
Jemmy 2.3 [jem]	OSS CDDL1 [lic06a]	⊠	⊠	□	Scenegraph, Java FX
QFS QF-Test 3.1.1 [qfs]	kommerzielle (ab 1595 EUR) und akademische Lizenzen	⊠[2]	⊠	⊠	
SWTBot 2.0.0.371-dev-e34 [swtb]	OSS EPL [lic09]	⊠	□	□	
TPTP 4.1 Automated GUI Recorder [tpt]	OSS EPL [lic09]	(⊠)	□	□	Testframework für Eclipse-Interna
IBM Rational Functional Tester 8.1 [rft] (ehemals XDETester)	kommerziell (ab 2985 EUR)	□?	□?	⊠	„Java“, SAP, Siebel
Jacareto 0.7.12 [jac]	OSS GPLv3 [lic07]	□	⊠	□	
Jalian Marathon 1.2.1.1 [mar]	OSS LGPL [lic99]	□	⊠	□	
JFCunit 2.08 [jfc]	OSS LGPL [lic99]	□	⊠	□	
Pounder 0.95 [pou]	OSS LGPL [lic99]	□	⊠	□	
UISpec4J 2.0 [uis]	OSS CPL [lic06b]	□	⊠	□	

Tabelle 3.1: Testwerkzeuge für den Java-Oberflächentest. Die Lizenzangabe bezieht sich auf *eine* Entwicklerlizenz für mindestens ein Jahr in der günstigsten, in der Quelle ausdrücklich aufgeführten Ausführung. Preisangaben zzgl. MwSt. Die Angaben über die API-Unterstützung wurden den öffentlich verfügbaren Dokumentationen der Tools entnommen. AWT-Unterstützung impliziert Swing-Unterstützung. Stand: Juli/August 2009.

1 „Undersupported“ [Roc06].

2 Ausdrücklich mit GEF-Unterstützung.

stellung ausdrücklich ein Capture-Replay-Tool gefordert, doch sollte trotzdem auch eine einfache JUnit-Erweiterung ohne eine solche Funktionalität ob ihres Potenzials geprüft werden.

Grundsätzlich überzeugten alle anderen SWT-fähigen Tools bereits bei der Vorauswahl nicht. Sie waren entweder in einem frühen oder schlecht unterstützten Entwicklungsstadium (Abbot SWT, AGR), boten keine kostenfreie Lizenz (GUIdancer, Squish for Java, WindowTester) oder keinen offensichtlichen Vorteil gegenüber SWTBot – etwa was die Dokumentation betraf. Die Tools ohne SWT-Unterstützung schieden naheliegenderweise ebenfalls aus.

3.2 Evaluation von QF-Test und SWTBot

Das Testwerkzeug muss eine Reihe von Anforderungen erfüllen. Die zentrale Annahme ist hierbei, dass diese Arbeit nicht die Entwicklung oder Verbesserung eines Oberflächentestwerkzeuges für SWT zum Gegenstand hat, sondern eben dieses Werkzeug als funktionierendes Hilfsmittel einsetzen möchte, um die Qualität des eigentlichen Kernpunktes der Arbeit – des Editors – zu prüfen und zu verbessern.

Dementsprechend soll das Tool primär die Erstellung von Tests mit geringem Aufwand und geradlinigem Workflow ermöglichen und wenigstens mittelbar GEF unterstützen.

Das Hauptproblem von Oberflächentests ist das Wiederfinden der Elemente und Controls der Oberfläche in unabhängigen Durchläufen. Selbst unter identischen algorithmischen Umständen gibt es keine Garantie, dass der Objektbaum, der die GUI repräsentiert, in den nachfolgenden Durchläufen identisch zum aufgezeichneten Zustand ist (etwa aufgrund von Objektpooling [Kir03]).

Aus diesem Grund muss das Tool einen Mechanismus besitzen, der einerseits von den vorgefundenen Objekten abstrahieren kann (z.B. indem die Hashes aus der standardmäßigen Objektidentifizierung herausgeschnitten werden) und andererseits die Eigenschaften der Objekte aufzeichnet, die ihm erlauben, zwei Objekte gleicher Art akkurat unterscheiden zu können (z.B. zwei Buttons der selben Klasse, aber mit verschiedenen Beschriftungen). Der entsprechende Mechanismus muss sehr gut funktionieren oder mangelhaftes Wiederfindeverhalten auf einfache Weise korrigierbar sein.

Zuguterletzt schreibt die Aufgabenstellung den Einsatz eines *Capture*-Replay-Tools vor, jedoch basiert die folgende Evaluation in Übereinstimmung mit *ubigrate* auf der

Annahme, dass ein Fehlen der Capture-Komponente bei anderweitig überzeugender Leistung des Werkzeuges von nachrangiger Bedeutung wäre.

Die meisten Tools, so auch die beiden für die Evaluation ausgewählten Werkzeuge, unterstützen das `setData`-Interface[1] von SWT, welches mittels key-value-Datenpaaren das Wiederfinden von Komponenten erleichtern kann. Beispielsweise empfiehlt QF-Test, die `name`-Property zu setzen, damit es diese als zusätzliches Unterscheidungsmerkmal im Komponentenauffindungsmechanismus verwenden kann: `splitCanvas.setData("name", "MDMLEdit.Mainview.Sashform");`

3.2.1 QF-Test

Als erstes Werkzeug wird *QFS' QF-Test 3.1.0-p1 (build 3363)* betrachtet[2]. QFS stellt jedem Interessierten eine vollumfängliche, zeitlich begrenzte Evaluationslizenz zur Verfügung.

Die Installation der binären Distribution erfolgt über ein eigenständiges Setup-Programm, welches im Verlauf der Installation das zu benutzende JDK und den verfügbaren Speicher mithilfe eines Einstellungsassistenten für die Verwendung mit QF-Test konfiguriert. Die gewählte Java-Installation wird von QF-Test mittels Injektion einer jar-Datei via `accessibility.properties`-Property des JDK instrumentiert[3]. Über diese Instrumentierung liest das Testtool während der späteren Testaufzeichnung alle Vorgänge innerhalb der Oberfläche mit.

QF-Test untersteht einer proprietären, kommerziellen Lizenz [QFS07], die zwar umfangreich, aber in gut verständlichem Deutsch formuliert ist. Es werden kostenlose Probierversionen mit eingeschränktem Funktionsumfang, eine kostenlose 4-Wochen-Evaluationslizenz mit vollem Funktionsumfang und kostengünstige bzw. kostenlose akademische Lizenzen mit vollem Funktionsumfang angeboten. Die kommerzielle Entwicklerlizenz kostet ab 1595 EUR netto. Die Preise sind gestaffelt nach Anzahl der unterstützten APIs und Anzahl der angeforderten Lizenzen. Neben den Entwicklerlizenzen werden zudem ab 1036,75 EUR Runtime-Lizenzen angeboten, die nur für die Ausführung von gespeicherten Testsuiten geeignet sind[4].

1 Siehe `org.eclipse.swt.widgets.Widget#setData(java.lang.String, java.lang.Object)` in [swta].
2 Im Verlauf der Arbeit wurde dann die Nachfolgeversion 3.1.3 eingesetzt.
3 Siehe [Sch09a, Kapitel 25].
4 Stand: 10.08.2009

Einführung anhand eines Tutorials

Das mitgelieferte Handbuch [Sch09a] ist detailliert und mit 667 Seiten sehr ausführlich. Ein enthaltenes „learning-by-doing“-Tutorial [Kel09] führt in die Erstellung und Ausführung von Testsuiten ein. Neben Tutorial und Handbuch wird von QFS noch ein sehr schneller Mailsupport und eine recht aktive Mailingliste bereitgestellt, auf der auch der Support des Herstellers mitliest und -schreibt. Neben dem empfehlenswerten Tutorial werden noch die Information aus dem Artikel von Markus Stäuble [Stä09] herangezogen.

Intern verwendet QF-Test Jython und Groove als Skriptsprache. Mit diesen sind auch eigene Skripte möglich – QF-Test bietet hierfür eine spezielle Schnittstelle.

Am Anfang jeder Testsuitenerstellung steht die Einbindung des SUT. Dies geschieht über einen Schnellstartassistenten, der den Testgegenstand zusätzlich mit einer modifizierten SWT-Version instrumentiert, da SWT nicht Teil des JDK, sondern Teil der Eclipse-Plattform ist und deshalb die ursprüngliche Instrumentierung der JVM nicht ausreicht.

Das grundlegende Vorgehen der Testerstellung besteht aus dem Aufnehmen von benutzergesteuerten Eventsequenzen (also Clicks und Tastatureingaben) direkt im SUT und der anschließenden Erweiterung der aufgezeichneten Sequenz um sogenannte Checks. Diese Checks stellen Prüfbedingungen, vergleichbar mit `assert` aus JUnit [jun], dar und prüfen die korrekte Funktion der SUT-Oberfläche während der Ausführung der Eventsequenzen.

Die große Frage, abgesehen vom Test der GEF-Unterstützung, die mit QF-Test im Rahmen der Evaluation geklärt werden sollte, war, ob auch in Entwicklung befindliche (i.S.v. nicht installierte) Eclipse-Plug-Ins getestet werden können – also Plug-Ins, die in einem Eclipse laufen, das erst aus einem anderen Eclipse heraus gestartet wird. Die optimale Variante ist hierbei, dass das Werkzeug auch die Kindinstanz erkennt und darin regulär testen kann. Weniger günstig wäre es, das Plug-In vor jedem Testlauf erst deployen zu müssen.

Die Evaluation beginnt mit der Einarbeitung mittels Tutorial „Teil I – Java testen mit QF-Test“ [Kel09]. Teil II des Tutorials beschäftigt sich mit dem Testen von Webanwendungen und spielt hier keine Rolle. Das Tutorial beginnt mit einer Beispielsuite, „Options.qft“, anhand derer QF-Test, insbesondere Testausführung und Ergebnisinspektion, eingeführt wird.

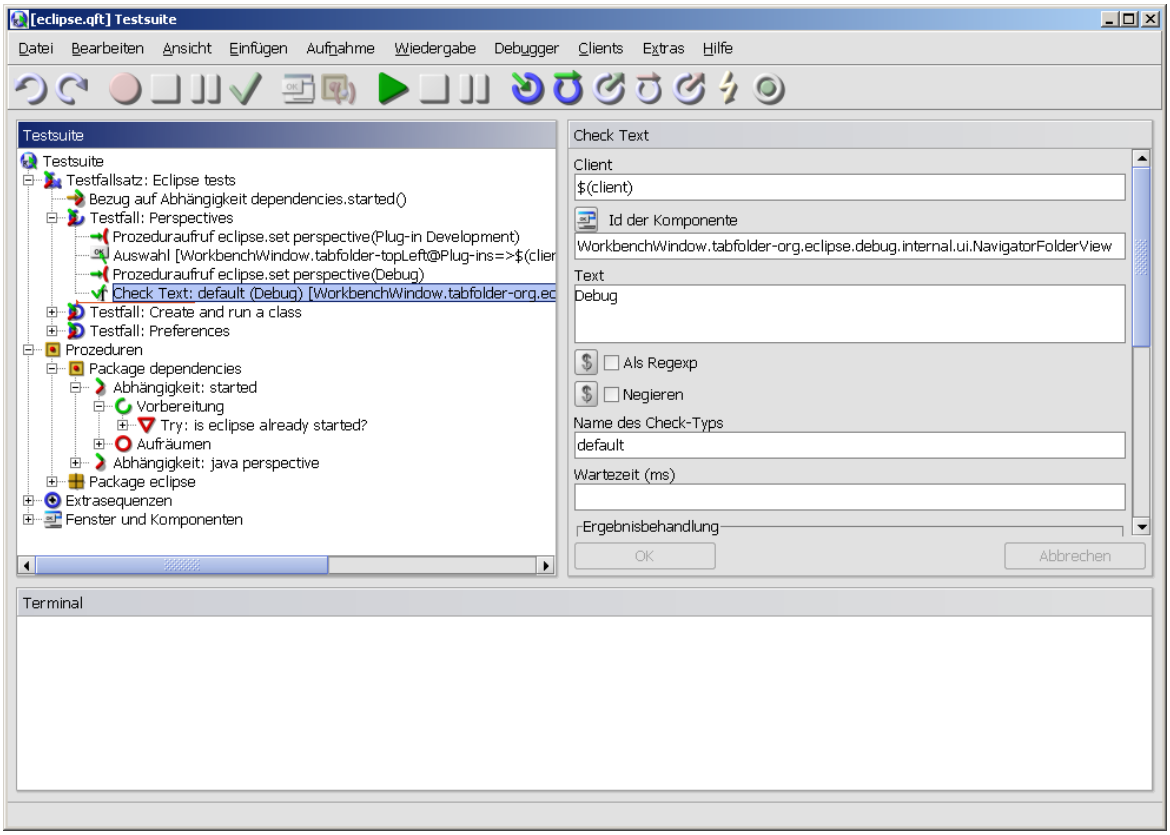

Abbildung 3.1: Die Oberfläche von QF-Test.

Die Oberfläche des Testwerkzeuges, dargestellt in Abbildung 3.1, zentriert sich um eine Baumstruktur auf der linken Seite, die den Aufbau der Suite (z.B. Testfälle, Setup/Teardown,..) repräsentiert. Dieser Baum besteht aus einer Reihe von Knoten verschiedenen Typs, die unter anderem die einzelnen Schritte im Verlauf des Testfalls repräsentieren. Auf der rechten Seite findet sich das Eigenschaftenblatt des jeweils ausgewählten Knotens. Der untere Bereich der Oberfläche wird von einer Konsole eingenommen.

Das Tutorial setzt sich mit einer Vorstellung der einzelnen Knoten der Beispielsuite fort:

- Setup „Vorbereitung“: Startet das SUT und wartet darauf, dass die Verbindung zur JVM hergestellt wird. Wartet danach auf das Erscheinen der GUI-Komponente – etwa die Anzeige des Hauptfensters.

- Testfall „Clickstream“: Enthält eine Mausklicksequenz und Texteingaben. Testfälle bestehen in der Regel aus mehreren Eventsequenzen.

- Sequenz „Table“ und Sequenz „Tab“: Sequenzen sind Sammelknoten für mehrere einzelne Aktionen bzw. Events (Tastendruck, Texteingabe, Mausklick, Auswahl, Fensterevent, sonstige Events). Die Sequenz „Table“ enthält zwei Klicks bezüglich relativer Koordinaten innerhalb der Zielkomponente. Positiv fällt an dieser Stelle die kontextsensitive Hilfe auf, die live auf eine HTML-Version des Handbuchs verlinkt.

- Testfall „Text Check“: Führt das „Check“-Konzept ein. Hier werden SUT-GUI-Elemente im Sinne des eigentlichen Regressionstests überprüft, z.B. ob die Beschriftung eines Textlabels den korrekten Wert zeigt. Mittels eines absichtlich falschen Wertes stellt das Tutorial das Protokoll des Testtools vor. Dieses Protokoll ist nach jedem Testlauf aufrufbar und listet sehr genau die vom vorgefundenen Fehler betroffene Komponente, Wertabweichung, einen Stacktrace, einen Screenshot des sichtbaren Bildschirminhalts und einen Screenshot des SUT-Fensters zum Zeitpunkt des Fehlers auf. Dadurch ist es auch zum Debugging geeignet.

- Testfall „Selected Test“: Erweitert das „Check“-Konzept auf nichttextuelle Komponenten, wie etwa Radiobuttons, und prüft hier, ob die korrekte Auswahl wiedergegeben wird.

- Tear-Down „Clean up“: Soll das SUT planmäßig beenden. Zuerst wird versucht, das SUT regulär zu schließen – das Tool wartet dann eine gewisse Zeit auf

den Shutdown. Falls das nicht klappt, wird über eine Try-Catch-Struktur die harte Terminierung des SUT ausgelöst. Anhand des Try-Catch werden Knoten eingeführt, die den Kontrollfluss der Testabarbeitung steuern. Insgesamt sind elf Kontrollflusskonstrukte verfügbar[1], ergänzt um die Möglichkeit, eigene Skripte auszuführen.

Setup und Tear-Down umschließen dabei jeden einzelnen Testfall. Bei Ausführung der gesamten Suite wird ein Protokoll angelegt, aus dem zum einen etwaige Fehler und Fehlschläge ersichtlich sind und zum anderen ein Bericht („Report“) als HTML bzw. XML mit XSLT generiert werden kann[2].

In der zweiten Hälfte des Tutorials erstellt der Nutzer seine eigene Testsuite. Das verwendete SUT ist dabei die Swing-Demo „FileChooserDemo“[3]. Es wird detailliert erklärt, wie der SUT-Start geskriptet wird und erste, einfache Tests aufgenommen und organisiert werden können. Ab diesem Zeitpunkt werden auch Tests für kausale Abhängigkeiten („Business Logic“) innerhalb des SUT vorgestellt, die sich nicht nur auf das korrekte Feedbackverhalten von Oberflächenprimitiven beschränken.

Der Superknoten „Extrasequenzen“ fungiert dabei als „Spielwiese“, in die neu aufgenommene Sequenzen einsortiert werden. An der roten Einfügemarke – in Abbildung 3.1 direkt unter dem markierten Knoten zu sehen – sind immer nur die dort erlaubten Knotentypen einfügbar. Statt dem Schnellstartassistenten wird in diesem Teil der Startknoten „Java SUT Client starten“ manuell befüllt (JVM-Aufruf mit `-jar ...`). Der Schnellstartassistent unterstützt folgende Arten von SUT-Kontexten: Startskript bzw. Programm (bat/exe), Java Webstart/Applet im Webbrowser, Jar, Class und Website im Browser.

Um einfache Mausklick- und Tastatursequenzen aufzunehmen wird das SUT, etwa über eine entsprechende Vorbereitungssequenz in QF-Test, gestartet und in QF-Test der Record-Button aktiviert. Danach kann sich der Tester normal im SUT bewegen. Nach Abschluss der Aufnahme wird diese in QF-Test gestoppt, woraufhin das Tool die fertige Sequenz als weiterverwendbare „Extrasequenz“ ablegt. Mit der reinen Aufnahme ist bereits ein Test der Oberflächenprimitiven möglich. Sie kann weiterhin als automatische Programmdemo verwendet werden.

1 Siehe [Sch09a, Seite 307ff]: Schleife, While, Break, If, Else, Elseif, Try, Catch, Finally, Throw, Rethrow.
2 Siehe [Sch09a, Kapitel 28.3]
3 `http://java.sun.com/docs/books/tutorial/uiswing/components/filechooser.html`

Mit diesen aufgezeichneten Sequenzen kann dann die eigentliche Suite zusammengestellt werden. Ein Testfallsatz gruppiert dabei mehrere Testfälle. Diese bestehen wiederrum aus Event- bzw. Aktionssequenzen. Dazu kommen noch Prozeduren. Diese enthalten wiederverwendbare Sequenzen, die über spezielle Aufrufknoten angesprochen werden können. Neben den oben beschriebenen „Extrasequenzen“ gibt es noch den Superknoten „Fenster und Komponenten“, der alle aufgezeichneten Oberflächenkomponenten mit ihren jeweiligen Eigenschaften auflistet.

Das Tutorial führt aus, dass zum Zwecke einer sauberen Testbasis zwischen Tests oder Untertests (Dialog öffnen/schließen) das SUT möglichst immer wieder geschlossen und neu gestartet werden soll. Dies ist jedoch zeitaufwändig, so dass eine saubere Testbasis möglichst auf andere Weise erreicht werden sollte.

Zum Clean-Up wird ein „Vorbereiten“- bzw. „Aufräumen“-Knoten verwendet, der als Abhängigkeit zwischen Testfällen eingefügt werden kann. In diesen kann man den Start des SUT ganz einfach per Drag-Drop oder Cut-Paste verschieben. Nach dem Start wird auf den Client gewartet, der dann über eine Variable referenziert werden kann. Danach wird ein „Warte-auf-Client“- bzw. „Warte-auf-Komponente“-Knoten eingesetzt, um den Test aufzuhalten, bis die Verbindung zum SUT hergestellt ist und das GUI dargestellt wird.

Der Testfall wird ordentlich benannt und mit der vorher aufgenommenen Sequenz aus den Extrasequenzen befüllt. Er ordnet sich im Testfallsatz nach der „Vorbereiten“-Referenz ein. Analog zum „Vorbereiten“-Knoten wird dann im Tutorial ein „Aufräumen“-Knoten eingeführt, der versucht, das Programm zu beenden und auf dessen Ende wartet. Alternativ kann man hier auch intern einen Mausklick auf „Beenden“ bzw. das Schließen-Symbol in Fensteranwendungen aufzeichnen und nach einer gewissen Wartezeit das SUT terminieren.

Das Prüfen des Clientverhaltens mittels Checks funktioniert ähnlich wie das Aufnehmen von Sequenzen: Während der Sequenzaufnahme in QF-Test den Check-Aufnehmen-Modus aktivieren (Button oder [F12]), dann im SUT über einen Rechtsklick auf der markierten Komponente die Art des Checks auswählen[1] und danach den Check-Aufnehmen-Modus beenden.

Die so gewonnene Testsuite wird ausgeführt und findet keine Fehler – das Tutorial präsentiert im Anschluss noch einmal den Fehlerfall, indem die erwarteten Testwerte absichtlich verändert werden.

1 Siehe [Sch09a, Abschnitt 22.9]: Text, Enabled-Status, Abbild, Elemente, Geometrie, ...

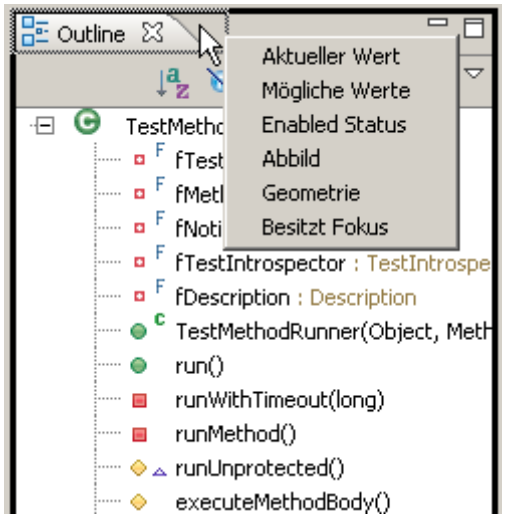

Abbildung 3.2: Visuelles Feedback beim Aufnehmen von Checks in QF-Test.

Den Abschluss des Tutorials bildet eine Präsentation von Tests der Geschäftslogik: In einer Sequenz wird ein Radiobutton aktiviert, der ein Textfeld freischaltet. An dieser Stelle wird ein Check auf dem Textfeld eingebaut, der dessen Enabled-Status prüft. Danach folgt ein Klick auf einen anderen Radiobutton, so dass das Textfeld wieder abgeschaltet wird.

Es zeigt sich, dass der Workflow zur Erstellung von Testsuiten ausgesprochen linear ist. Nach dem Start der Aufzeichnung können die Aktionen, die der Testfall prüfen soll, einfach live im SUT aufgezeichnet werden. Dies ermöglicht eine sehr flüssige Arbeitsweise ohne Unterbrechungen.

Live-Test am Prototypen

Nach dem erfolgreichen Einblick in das Werkzeug durch das Tutorial geht es nun im Folgenden um die beiden wichtigsten Fragestellungen, die das Tool beantworten muss: Ist es in der vorhandenen Entwicklungsumgebung auch ohne Deploy benutzbar? Unterstützt es GEF? Zu diesem Zweck wird die Anwendung von QF-Test auf den bestehenden Prototypen des Editors aus [Sch09b] geprüft.

Auf dem naiven Wege wird eine Sequenz aufgenommen, in der innerhalb des „äußeren“ Eclipse, mit dem das Editor-Plug-In entwickelt wird, das „innere“ Eclipse mit dem virtuell deployten Plug-In gestartet wird. Die Aufnahme funktioniert! Das innere Eclipse taucht als Child-Prozess des eigentlichen SUT auf und kann danach wie ein

eigenständiges SUT weiterverwendet werden.

Auf dieser Grundlage ist es ein leichtes, einen entsprechenden, modularisierten Testkontext zu erzeugen und auch zu zerstören, um daraus wiederverwendbare Prozeduren zu konstruieren. Der Testkontext besteht aus einem Projekt im inneren Eclipse, das eine Datei mit der für den Protokolleditor interessanten Endung *.mdml* enthält.

Im nächsten Schritt wird geprüft, ob das Werkzeug basierend auf dem erzeugten Testkontext auch innerhalb der konkreten GEF-Umgebung noch alle Oberflächenelemente erkennt. Zu diesem Zweck muss in einem „Skript"-Knoten der GEF-Support explizit importiert werden. Dies geschieht über ein zweizeiliges Skript ([RL08]) aus der Dokumentation des Werkzeugs. Das Ergebnis ist vielversprechend: Die GEF-basierten Oberflächenelemente werden aus einem sonst für das Testtool gleichförmigen, großflächig unidentifizierten Oberflächenelement aufgelöst und können individuell verwendet werden.

Es fällt jedoch auf, dass es Probleme mit der Wiedererkennung der Palette gibt – der Standardresolver schneidet hier offenbar die Namen der einzelnen Sashes der SashForm[1] ab, die intelligenterweise einfach im Postfix durchnummeriert werden. Diese Nummerierung wird vom Resolver scheinbar als Hash aufgefasst und ohne Zögern weggekürzt. Dadurch überlagern sich praktisch Editorhauptfläche und Palette.

Es ist jedoch bereits an diesem Punkt ersichtlich, wie dieser Umstand einfach behoben werden kann: Entweder kann das durch Verfeinerung des Resolvers[2] gelöst werden oder die einzelnen Sashes werden besser benannt, indem deren `toString()`-Methode modifiziert oder eben via `setData()`-Interface deren „name" Property gesetzt wird. Die beiden letzten Punkte würden die Verwendung einer eigenen, in jeweils diesem Punkt veränderten `SashForm`-Klasse erfordern.

Tests lassen sich nach dem Erreichen der GEF-Oberfläche sehr einfach zusammenklicken und ermöglichen hiernach umfangreiche Checks, die sich bereits in der Evaluation leicht einsetzen ließen. Der Workflow für die Testerstellung ist extrem linear und der Instrumentationsansatz zum Mitschneiden der GUI-Events ist schnell und robust.

Einige implementatorische Details des für die Evaluation verwendeten Editorprototypen aus [Sch09b] erwiesen sich jedoch als testungeeignet. So wurden etwa frisch angelegte Komponenten automatisch mit einer Zufallszahl benannt, wodurch sich eine

1 Sashes sind die Container einer komplexen Containerkomponente namens SashForm, vgl. [swta].
2 Siehe [Sch09a, Seite 546-580].

Sequenzaufnahme nicht replizieren ließ. Auch die Verwendung von `setData()` zur Verfeinerung der Erkennung konnte an dieser Stelle bereits geprüft und ausgewertet werden und wirkte problemlindernd.

Offen bleibt an dieser Stelle, inwiefern andere Testtools, wie etwa *XMLUnit* eingebunden werden können – vermutlich ergibt sich, falls dies notwendig werden sollte, ein Einstieg über die „Skript"-Knoten.

3.2.2 SWTBot

Als Vertreter der Open Source JUnit-Erweiterungen wird *SWTBot* von ThoughtWorks Entwickler Ketan Padegaonkar näher betrachtet.

Die Installation des Tools erfolgt über dessen Updatesite[1]. Gegenstand der Untersuchung ist Version *2.0.0.371-dev-e34*. Es gibt zwei Ausprägungen von SWTBot: Eine für Eclipse-Testing und eine für SWT-Testing; Projekte im Rahmen der PDE benötigen dabei die Edition zum Testen von Eclipse. Das nach dem Neustart auf der Willkommensseite des Tools verlinkte Tutorial zeigt lediglich auf die Homepage, auf der sich kein Tutorial findet, so dass der Artikel von Ralf Ebert [Ebe09] als Einstieg genutzt werden muss, ergänzt durch einen Artikel von Zeljko Markovic [Mer09] und den SWTBot Userguide [swtc].

Einführung anhand eines Tutorials

Der Artikel beginnt mit der Einrichtung des Beispielprojektes „Adressbuch". In Eclipse muss, um die korrekte Funktionsfähigkeit des Werkzeugs sicherzustellen, die „Target Platform" auf ein Eclipse umgeändert werden, dass eine Reihe von testwerkzeugspezifischen Plug-Ins enthält. Eine solche Plattform ist auch im Beispielprojekt enthalten. Eine selbst erstellte Target Platform benötigt folgende zusätzliche Plugins:

- org.eclipse.swtbot.swt.finder,
- org.eclipse.swtbot.eclipse.finder,
- org.eclipse.swtbot.eclipse.core,
- org.junit4,

1 `http://download.eclipse.org/technology/swtbot/ganymede/dev-build/update-site`

- org.junit,
- org.hamcrest,
- org.apache.log4j

Bei näherer Betrachtung dieser Liste fällt auf, dass das von SWTBot geforderte *log4j* auch in der ubigrate-Abhängigkeitshierarchie[1] Verwendung findet. Damit besteht zumindest theoretisch ein Konfliktpotential zwischen Werkzeug und SUT, wenn dieses auch in der Praxis durch OSGi selbst unterbunden wird.

Der Artikel beschreibt zuerst den Import und das Setup des SUT, das schließlich via `plugin.xml` gestartet wird.

SWTBot-Tests sollen grundsätzlich in einem eigenen Plug-In verwaltet werden, das dann in einer speziellen SWTBot-Konfiguration ausgeführt wird. Das entsprechende Container-Plug-In definiert zu diesem Zweck Abhängigkeiten zu JUnit4, SWT, Eclipse-Core, Hamcrest (für deklarative Matcher) und SWTBot. Auffällig hierbei ist, dass diese Dependencies in die Target Platform sich nicht mit dem Manifest-Editor eintragen lassen, da dieser die Bibliotheken der Target Platform ignoriert und diese deshalb nicht zur Verfügung stehen. Der Eintrag muss letztlich manuell erfolgen.

Zum Zwecke der Funktionsprüfung des Setups bis hierhin wird nun ein neuer JUnit-Test inklusive einer leeren Testmethode angelegt. Unter *Startkonfiguration* → *Main* wird das zu startende Produkt ausgewählt: das Adressbuch-Plug-In. An dieser Stelle hat sich ein Fehler in den Artikel [Ebe09] eingeschlichen: Der Paketname der Applikation ist falsch wiedergegeben (`.product`, richtig ist: `.application`). Die Ausführung mit einer leeren Testmethode funktioniert, so dass an dieser Stelle sicher ist, dass die vorher vom Manifest-Editor nicht gefundenen Abhängigkeiten in der Target Platform auch tatsächlich aufgelöst werden.

Nun werden richtige Testfälle implementiert. Das API, basierend auf `SWTBot`, insbesondere `SWTWorkbenchBot`, enthält Methoden zur Bedienung von SWT-Oberflächen. Mit diesen kann im aktuellen Kontext nach Steuerelementen gesucht und auf diesen Aktionen ausgelöst werden, indem entsprechende Events in die SWT-Eventqueue eingestellt werden. Das zentrale Namensschema der Methoden lautet dabei

```
[widget]With[Condition](Criterion);
```

z.B. `SWTBotText var = bot.textWithLabel(„Heinz").setText(„Heinzel");`

1 Siehe auch Abhängigkeitsdiagramm in [Sch09b], Abbildung 4.3.

Auf die so gefundenen und beeinflussten Komponenten kann sodann die bekannte `assert`-Methode aus JUnit angewendet werden, was für ein vertrautes Gefühl bei der Arbeit mit dem Werkzeug sorgt.

Die primäre Orientierung an den Labelbeschriftungen ist zwar anwendernah, führt aber bei Änderungen an der Beschriftung oder beim Umschalten der SUT-Sprache zu Problemen. Eine Alternative ist auch hier das `setData()`-Paradigma von SWT. Wie QF-Test erfordert auch SWTBot die Verwendung eines fixen Keys. Auf diesem Schlüsselsatz kann dann mittels `[widget]WithId()` die hoffentlich eindeutig vergebene Markierung wiedergefunden werden. Werden mehrere Komponenten mit demselben Unterscheidungsmerkmal angetroffen, so werden diese nur noch über eine fortlaufende ID unterschieden.

Jeder `Control`-Typ erfordert seine eigene Auffindungsmethode (z.B. Menü: *bot.menu*(..)-*.menu*(..).*click*()), generische Matcher – etwa zu Debug- oder Inspektionszwecken – sind nur über eigene Implementierungen möglich.

Während der Ausführung wartet SWTBot immer eine gewisse Zeit auf das Erscheinen des gesuchten Oberflächenelementes und wirft sonst nach einem Timeout eine `WidgetNotFoundException`. Views und Editoren sind via ID oder Caption aufrufbar. Auf diesen sind neue Bots erzeugbar, die dann auf diesem neuen Kontext arbeiten. Unmittelbar unterstützt werden laut Artikel auch Perspektiven und Commands.

Das Auffinden und Selektieren von Widgets muss jedoch nicht nur über die vorgegebenen Standardmethoden geschehen. Es ist möglich, mithilfe des Hamcrest-Frameworks einen eigenen Matcher zu implementieren. Dadurch sind auch generische Matcher (etwa für das Debugging) oder Spezialmatcher für eigene Oberflächenelemente erzeugbar. Der Matcher muss zu diesem Zweck aber im UI-Thread laufen, da außerhalb dessen der Zugriff auf die SWT-Controls nicht gestattet ist.

Es fällt auf, dass das Setup und Tear-Down des SUT nicht einfach zu realisieren ist. Theoretisch müsste das SUT zwischen einzelnen Tests neu gestartet werden. Dies ist jedoch in SWTBot nicht unmittelbar möglich, da dessen Methoden auf derselben Ebene laufen wie das SUT selbst, womit sich eine gewisse Abhängigkeit des Tests zum Laufzeitkontext ergibt. Zur Automatisierung ist deshalb die Verwendung eines externen Tools wie etwa *ant* notwendig. Lediglich die aus JUnit bekannten `@Before`- und `@After`-Methoden stehen zur Verfügung, um eine saubere Testumgebung innerhalb der SUT-Ausführungsebene herzustellen.

Da SWTBot eine reine JUnit-Erweiterung ist, lassen sich auch andere Erweiterungen wie etwa *DBUnit* oder *XMLUnit* einbinden, etwa um neben der Oberfläche in Form

eines monolithisierten End-to-End-Tests zeitgleich die Persistenzschicht explizit mit prüfen zu können. Da dazu jedoch in der Regel Zugang zum internen SUT-Zustand notwendig ist, müssen Abhängigkeiten aus den Tests in die SUT-Klassenstruktur eingeführt werden. Aus diesem Grund empfiehlt der Tutorialautor die Verwendung von Backend-Mocks auf dem Client, damit der GUI-Test das Backend ausdrücklich nicht mittestet.

Live-Test am Prototypen

Nach Abarbeitung des Tutorials wird versucht, den bestehenden Prototypen aus [Sch09b] mit SWTBot zu verbinden. Beim Import des Prototypen in das frische SWTBot-Eclipse hat dieses Probleme, das in der Target Platform mitgelieferte GEF zu erkennen – ein Update bzw. Install wird verweigert. Nach einem Backup der Produktionsumgebung wird umgekehrt probiert, SWTBot auf dem Produktiv-Eclipse zu installieren. Dabei stellt sich heraus, dass die mitgelieferte Target Platform den Prototypen nicht unterstützt – die ursprüngliche Target Platform muss um die geforderten Plug-Ins erweitert werden und wird im Anschluss weiterverwendet.

Es ist nicht sofort klar, wie der Prototyp für den Test in Eclipse eingebunden werden kann, denn im Gegensatz zum Anwendungsfall des Tutorials taucht das Plug-In nicht in der Applikationsliste der SWTBot-Startkonfiguration auf. Das Tutorial betraf an dieser Stelle eine RCP-Anwendung und nicht ein PDE-Plug-In, das nunmal nicht als unabhängige Anwendung funktioniert, sondern innerhalb der inneren Eclipse-Instanz läuft.

Der Anbindungstest scheint ab diesem Punkt unmöglich zu werden, da das SUT sofort wieder geschlossen wird – an dieser Stelle ist unklar, ob das von der SWTBot-Startkonfiguration gestartete Eclipse überhaupt das zu testende Plug-In enthält. Trotz `bot.wait(Wert)` terminiert das gestartete SUT unverständlicherweise sofort nach dem Start. Nach einigen Versuchen stellt sich heraus, dass mit einem harten `Thread.sleep()` das SUT wenigstens einmalig offengehalten werden kann, um zumindest dem OSGi-Framework die Chance zu geben, vollständig zu starten. Daraufhin bleibt endlich auch das SUT-Fenster für einige Zeit untersuchbar. Auf diese Weise kann zumindest ersteinmal verifiziert werden, dass MDMLEdit wenigstens geladen wird.

Es schließt sich die Frage an, wie jetzt im SUT ohne großen Aufwand ein Evaluations-Fixture geschaffen werden kann. In Analogie zur QF-Test-Evaluation soll der Test sich selbst ein Testprojekt anlegen und eine entsprechende Datei erstellen, die dann mit dem Editor geöffnet werden kann. Das Hauptproblem ist dabei, zeiteffizient die

richtigen Oberflächenelemente zu identifizieren. Wie kann herausgefunden werden, ob die GEF-Konstrukte bereits erkannt werden? Der Bot scheint Probleme zu haben, überhaupt erstmal die einfachsten Oberflächenmerkmale, wie etwa das Dateimenü der Entwicklungsumgebung, wiederzufinden.

Nach einigen fruchtlosen Versuchen, in denen scheinbar kein Element der Oberfläche gefunden wird, schafft ein selbst erstellter, generischer Matcher Abhilfe. Mit ihm können die Oberflächenelemente, die ihm vom Framework zum Abgleich gegeben werden, auf der Konsole aufgelistet werden. So kann zumindest schon einmal verifiziert werden, dass der Bot überhaupt irgendwelche Elemente erkennt. In der Auflistung ist die korrekte Shell zu finden, aber der Zugriff auf das Dateimenü scheint immer noch nicht zu funktionieren.

Es stellt sich heraus, dass ein Hauptproblem bei der Benutzung von SWTBot ist, dass die Events nicht auf der Nutzerebene abgespielt werden, sondern darunter. Das führt dazu, dass visuelles Feedback in großem Umfang verloren geht. Beispielsweise wird der Click auf einen Menüeintrag nicht visualisiert. Im günstigsten Fall erfolgt die unmittelbare Rückmeldung dadurch, dass der entsprechende Dialog geöffnet bzw. die entsprechende Aktion ausgeführt wird. Dies erschwert die zielgerichtete Testerstellung ungemein.

Der selbstgeschriebene, generische Matcher macht auch deutlich, dass das Framework die verfügbaren Oberflächenelemente nicht zuverlässig in den Matcher einbringt – offensichtlich immer dann nicht, wenn das SUT nicht den Fensterfokus besitzt. Dies passiert insbesondere, wenn sich das SUT-Fenster hinter dem äußeren Eclipse befindet.

Größtes Problem ist aber nach wie vor, dass das SUT unmittelbar nach dem letzten Matching hart terminiert wird – selbst dann, wenn der Matcher im UI-Thread arbeitet und versucht wird, die SUT-Terminierung in der Testmethode hart mit `Thread.sleep()` aufzuhalten.

Immerhin ist es mithilfe des generischen Matchers jetzt möglich, stückchenweise durch das SUT zu navigieren, da jetzt deutlich wird, welche Elemente sichtbar sind. Auf die Weise wird versucht, Stück für Stück ein neues Projekt anzulegen. Der Workflow für die Testerstellung gestaltet sich dabei denkbar unlinear. Nach dem Schreiben einer Zeile Bot-Code muss im (inneren) Eclipse der Test-Workflow bis zum gerade programmierten Schritt nachvollzogen und dann das Kriterium zum Auffinden des nächsten Oberflächenelementes gesucht werden. Dann muss der Test-Workflow abgebrochen werden und das gefundene Kriterium in einer weiteren Zeile Bot-Code verarbeitet werden usw. Nach wenigen Schritten muss pessimistischerweise geprüft werden, ob der Ablauf so funktioniert, da manche Widgets in manchen Situation gefunden werden, in anderen

wiederum nicht – in der Menüsequenz *File* → *New* → *File* wird zum Beispiel das letzte „File“ regelmäßig nicht gefunden.

```
1    @Test
2    public void testGEF1() throws Exception {
3        SWTWorkbenchBot b = new SWTWorkbenchBot();
4
5        System.out.println("Test1");
6        System.out.println("waiting with "+b);
7        Thread.sleep(20000);
8
9        System.out.println("Trying to find the shell...");
10       SWTBotShell sh = b.shell("Java - Eclipse SDK");
11       System.out.println(sh);
12
13       //FIXME: close "Welcome" automatically?!
14       b.menu("File").menu("New").menu("Project...").click();
15
16       SWTBotShell np = b.shell("New Project");
17       assertTrue("Shell vorhanden und aktiv..",np.isActive());
18       b.tree().expandNode("General").select("Project");
19       b.button("Next >").click();
20       b.textWithLabel("Project name:").setText("Test");
21       b.button("Finish").click();
22
23       b.menu("File").menu("New").menu("File").click();
24       b.textWithLabel("File name:").setText("Test.mdml");
25       b.button("Finish").click();
26
27       System.out.println("Handing in the generic matcher..");
28       b.widget(new GenMatch<Widget>());
29
30       System.out.println("waiting..");
31       Thread.sleep(60000);
32   }
```

Abbildung 3.3: Code-Beispiel für einen SWTBot-Test auf der Eclipse-Oberfläche.

Problematisch ist hier wieder, dass das SUT nach dem letzten Test terminiert, ohne etwaige Sleeps abzuwarten – die erwarteten Zwischeneffekte sind so oft nicht nachvollziehbar.

Es ist bereits an dieser Stelle sehr deutlich klar geworden, dass die Testspezifikation mit SWTBot recht aufwändig ist – pro Test dürfte ein Aufwand zu investieren sein, der mindestens dem Programmieraufwand zur Erstellung des SUT-Gegenstücks entspricht. Hauptgrund dafür ist die nicht vorhandene visuelle Unterstützung bei der Auswahl der aufzuzeichnenden Widgets – in dem Sinne gibt es kein Capture. Damit bleibt es

beim oben beschriebenen, schrittweise manuellen Erstellen der Tests, das durch seine Nichtlinearität vermutlich maximal fehleranfällig ist.

Nachteilig ist auch, dass für jede Anwendung eine spezielle Target Platform eingestellt werden muss, die im schlimmsten Fall neu zu instrumentieren ist, da die vorgefertigten Plattformen nicht immer unterstützt werden.

Selbst das Setup des SUT zur Laufzeit gerät zur Geduldsprobe, da jeder zu programmierende Schritt erst manuell nachvollzogen werden muss. Das Framework ist zugegebenermaßen noch jung und auch der Autor des Tutorials meint, dass es zwar robust und stabil ist, aber schnell an seine Grenzen stößt.

Zusätzlich enthält [Mer09] den Hinweis, dass GEF bisher eher mangelhaft bis gar nicht unterstützt wird. Ebert weist zwar in [Ebe09] auf die offene Weiterentwicklung bzw. Erweiterbarkeit von SWTBot im Sinne von Open Source Software hin, doch eine solche, möglicherweise notwendige Erweiterung des Tools würde den Rahmen dieser Arbeit sprengen, da es nicht deren Aufgabe ist, ein GUI-Test-Tool für SWT/GEF zu entwickeln, sondern eines für den Test des Editors zu *benutzen*.

Unter diesem Gesichtspunkt ist der Einsatz von SWTBot besonders im Vergleich mit QF-Test zu aufwändig und kleinteilig und der GEF-Support zu schlecht, als dass es einen effektiveren Ansatz als QF-Test bieten würde.

3.2.3 Fazit und Auswahl

Insgesamt wirkt das kommerzielle Werkzeug deutlich komfortabler und robuster. Nicht zu vernachlässigen ist auch der ausgesprochen schnelle und kompetente Support per E-Mail bzw. Mailingliste. Das ist etwas, was man bei einem Tool aus der Kategorie Open Source Software ganz einfach nicht erwarten kann und in der Regel auch nicht findet. Die Ausnahme bilden hier vielleicht die Foren von großen Open-Source-Produkten wie OpenOffice oder Eclipse, doch SWTBot kann mit diesen bei der Größe der Community einfach nicht mithalten. In Tabelle 3.2 werden die relevanten Unterschiede zwischen den beiden Tools abschließend zusammengefasst.

Im Ergebnis der Evaluation und unter Beachtung der Schwerpunkte, die die Aufgabenstellung setzt, wurde in Abstimmung mit *ubigrate* entschieden, dass aufgrund der gefundenen Unterschiede zwischen den Tools – besonders im Hinblick auf die Linearität der Testerstellung – im Rahmen der Arbeit *QF-Test* als Werkzeug für den Oberflächentest verwendet werden soll.

Kriterium	SWTBot	QF-Test
Lizenz	Open Source/EPL [lic09]	Kommerziell/Proprietär
Capture-Replay?	Keine Capture-Komponente	Ja
Notwendige API-Kenntnisse	Umfangreich (`SWTBot.*`, Hamcrest)	Begrenzt (Resolver)
Injektionsmethode	Eigenes Target Platform Plug-In	Instrumentiertes JDK, modifiziertes SWT
Aufwand Testerstellung	Sehr hoch (Zeile-für-Zeile mit Wechsel zwischen Testrealisierung und SUT)	Niedrig (Capture, schlimmstenfalls mit nicht wiedererkannten Komponenten)
Schwächen in der Testerstellung	Zeilenweises Scripten, nichtlinearer Workflow	Weniger Kontrolle über die SUT-Interna
Ausführungsgeschwindigkeit	Schnell (In-situ auf SUT-Ebene)	Schnell (Direkte Instrumentierung des JDK)
Stärke in der Testdurchführung	Introspektion	Deploy-Neutralität
Setup/Teardown	Nur auf SUT-Ebene od. via ant etc.	Prozeduren, SUT kann auch zwischen Testläufen neu gestartet werden
Anpassbarkeit	Eigene Hamcrest-Matcher verwendbar	Eigene Resolver verwendbar
GEF-Unterstützung	Schlecht [Mer09]	Ausgereift, per Import aktivierbar

Tabelle 3.2: Direkter Vergleich zwischen SWTBot und QF-Test.

3.2.4 Testautomatisierung

Wie jedes moderne Testwerkzeug bietet QF-Test Möglichkeiten, im Rahmen eines entsprechenden Entwicklungsumfeldes automatisierte Tests durchzuführen. Das betrifft zum einen die Einbindung der Tests etwa in Build-Skripte, zum Beispiel für regelmäßige, unbeaufsichtigte Regressionstests, und die Einbindung in Testverwaltungswerkzeuge, und zum anderen datengetriebenes Testen, bei dem die Testsuite wiederholt mit wechselnden Ein- und Ausgabedaten betrieben wird.

QF-Test bietet in diesem Bereich vier grundlegende Features: Datengetriebenes Testen [Sch09a, Kapitel 13], Batchmodus [Sch09a, Abschnitt 15.1], Daemonmodus [Sch09a, Abschnitt 15.2 bzw. Kapitel 33] und die Einbindung in Werkzeuge zum Testmanagement [Sch09a, Kapitel 18]. Diese Features sollen im Folgenden kurz vorgestellt werden.

Datengetriebenes Testen

Beim datengetriebenen Testen werden die in den Testfällen verwendeten Eingabedaten und die zugehörigen, erwarteten Ausgabewerte nicht fest im Testfall hinterlegt, sondern über eine externe, vom eigentlichen Testfall unabhängige Datenquelle eingespeist. Auf diese Art und Weise können etwa Domänenexperten Testdaten beisteuern, ohne Kenntnisse im Umgang mit dem Testwerkzeug oder der Erstellung von Tests allgemein zu haben.

Zudem ist es einfacher, Testdatengenerierungsmethoden wie modellgetriebenes Testen, Klassifikationsbaummethode und Grenzwertanalyse einzusetzen, wenn nicht für jede Instanz ein eigener Test geschrieben werden muss. Somit lassen sich bessere Testabdeckungen erzeugen, die die Qualität des SUT verbessern.

QF-Test unterstützt laut [Sch09a, Abschnitt 22.4] als Quelle toolinterne Tabellen, Excel-Sheets, CSV-Dateien, Datenbanken (JDBC-kompatibel [Sch09a, Tabelle 22.6]), gescriptete Datenbinder und Datenschleifen für die mehrfache Ausführung von Testfällen im Kontext eines Datentreibers, wobei lediglich eine Zählvariable gebunden wird. Diese Quellen können auch verschachtelt in Kombination verwendet werden, so dass sich bei der Testausführung kombinatorische Komplexitäten ergeben.

Um diese Datentreiber in der Testsuite verwenden zu können, wird einem Testfallsatz ein Datentreiberknoten untergeordnet. In jenem Knoten werden sodann beliebig viele Datenknoten registriert. Die Variablen von zuerst registrierten Datenknoten wer-

den dabei zuerst gebunden und können von späteren Datenbindungen überschrieben werden.

Batchmodus

QF-Test ermöglicht die Ausführung von Testsuiten über die Kommandozeile. Auf diese Weise lässt es sich auch in herkömmliche Testskripte – etwa auf Ant-Basis oder in Form eines Shell-Scripts – einbinden. Kapitel 24 in [Sch09a] beschreibt ausführlich sämtliche Kommandozeilenparameter.

Über den Parameter *−batch* wird der gleichnamige Modus initialisiert. Von einer übergebenen Suite werden sodann alle Testfallsätze und Testfälle in der obersten Ebene ausgeführt. Allerdings kann über den Parameter *−test* die Ausführung auf einzelne Testfälle eingeschränkt werden (Testfallindex oder -ID).

Nach der Ausführung wird ein Protokoll des Testlaufs erzeugt, das mittels Parameter *−runlog* auch an beliebigen Stellen im Dateisystem abgelegt werden kann. QF-Test stellt auch Platzhalter für Datum und Zeit zur Verfügung, so dass der Name oder Pfad des Protokolls durch Angaben zu Tag und Zeit der Testausführung ergänzt werden kann. Das Protokoll kann zudem durch Angabe von *−compact* soweit gekürzt werden, dass nur noch wichtige Knoten, beziehungsweise jene Knoten aufgezeichnet werden, die für die Fehleranalyse notwendig sind. Neben der technischen Auswertung in Form des Protokolls ist die Erstellung eines zusammenfassenden Testberichtes mittels Parameter *−report* möglich. Damit ist auch das Reporting von automatisierten Testausführungen vollumfänglich sichergestellt.

Der folgende Beispielaufruf erzeugt für einen Batchlauf des „TestfallA38“ aus der Testsuite „suiteA.qft“ entsprechend der Testausführung datierte Protokolle:

```
qftest -batch -runlog c:/mylogs/+b -report c:/mylogs/rep_+b_+y+M+d+h+m
              -test "TestfallA38" c:/mysuites/suiteA.qft
```

Der Batchmodus ermöglicht weiterhin die Ausführung von Lasttests[1]. Dazu wird QF-Test im Batchmodus zusätzlich mit dem Parameter *−threads* aufgerufen, der die Batchausführung in die angegebene Anzahl an Threads hineinmultipliziert, so

1 Näheres zum Thema „Lasttest mit QF-Test“ findet sich in [Sch09a, Abschnitt 19.2].

dass sehr viele SUT-Instanzen gleichzeitig getestet werden können. Auf diese Weise können etwaige Back-Ends mitgetestet werden und unter Umständen Race Conditions gefunden werden. Die verschiedenen Testläufe können über das Skriptinterface von QF-Test untereinander – selbst über mehrere Rechner hinweg – synchronisiert werden, was die Reproduzierbarkeit und damit die Aussagekraft der Ergebnisse der parallelen Ausführung erhöht.

Daemonmodus

Im Daemonmodus kann QF-Test von entfernter Stelle über RMI angesteuert werden und sich so an verteilten Tests beteiligen. Nach dem Start des Daemons kann diese Instanz von anderen Instanzen per Batchmodus-Parameter *−calldaemon* angesprochen oder über das Skriptinterface des Werkzeugs mittels einer speziellen API, die in [Sch09a, Kapitel 33] beschrieben ist, gesteuert werden.

Einbindung in Testmanagementwerkzeuge

Für den Einsatz in umfangreichen Testszenarios unterstützt QF-Test die Anbindung an Testmanagementwerkzeuge über den Batch- und Daemonmodus. Das Handbuch [Sch09a] bietet Musterlösungen für die Integration in *HP TestDirector for Quality Center*, *Imbus TestBench* und *TestLink*.

Mit diesen Tools wird es dem Entwickler ermöglicht, etwa den Entwicklungsstand von Testfällen zentral zu überwachen oder Testfälle mit Use-Cases zu verknüpfen. Auch kann die Auslastung von Testsystemen beim verteilten Testen überwacht werden.

3.3 Die Testsuite für den Oberflächentest

Die Testsuite besteht aus einem Satz mit mehreren Testfällen und benutzt eine Reihe von wiederverwendbaren Prozeduren, um jedem Testfall eine saubere Ausgangslage zu verschaffen.

Die Vorbereitungs- und Aufräumaktionen werden dabei zweistufig geschachtelt: In der ersten Stufe erfolgt als Vorbereitung der Start des äußeren (PDE-) und des inneren (MDMLEdit-) Eclipse sowie der Import der GEF-Unterstützung. Als Aufräumaktionen

werden zuerst das innere und dann das äußere Eclipse geschlossen. Diese Spezialsequenzen werden vor und nach jedem Lauf der gesamten Testsuite automatisch vom Testwerkzeug ausgeführt.

In der zweiten Stufe werden andere Vorbereitsungs- und Aufräumaktionen verwendet – die zugehörigen Knoten werden genau einmal vor und nach jedem einzelnen Testfall ausgeführt. Damit ist eine saubere Ausgangslage für jeden einzelnen Testfall garantiert, ohne das SUT jedesmal zeitaufwändig vollständig neu starten zu müssen. In der zweiten Stufe wird im Vorbereitungsknoten eine neue Modelldatei angelegt und geöffnet. Im Aufräumknoten werden hingegen alle offenen Editorfenster geschlossen und die für den Test erzeugte Modelldatei wird anschließend gelöscht.

3.3.1 Prozeduren

Die Aktionen der Vorbereitungs- und Aufräumknoten der zweiten Stufe sind in Prozeduren zusammengefasst und können so auch innerhalb der Testfälle mittels eines einfachen Prozeduraufrufs wiederverwendet werden.

- `CreateModel1()` – erzeugt innerhalb des SUT eine neue, leere MDML-Datei und öffnet diese.
- `CloseModels(boolean doSave)` – schließt alle offenen Editorfenster. Der Parameter entscheidet darüber, ob die eventuelle Frage danach, ob die Änderungen im Modell gespeichert werden sollen, mit „Ja“ oder „Nein“ beantwortet wird.
- `DeleteModel1()` – löscht die von CreateModel1() erzeugte Datei.

3.3.2 Testfälle

Abbildung 3.4 zeigt den Aufbau der Testsuite nach Art von QF-Test. Die folgenden Testfälle werden von der Testsuite erfasst.

MessageModel-Verwaltung

Änderungen der Properties des `MessageModel`s inklusive Hinzufügen von Encoding-Libraries und anschließende Effektprüfung (Gibt die Oberfläche im Tabellenkopf der Nachrichtenliste den korrekten Wert für Manufacturer und Devicename zurück?). Hinzufügen und Entfernen mehrerer Nachrichten. Mehrfacher Nachrichtenwechsel und Umbenennung von Nachrichten. Speichern, Schließen und Öffnen des Modells und anschließende Effektprüfung (Auswirkung des Serialisierens und Parsens: Sind noch alle Komponenten enthalten? Haben sich die Properties verändert, die die Oberfläche in das PropertiesSheet einliest?). Hinzufügen von Bindables und Effektprüfung des Imports der EncodingLibraries. Funktionsprüfung encodingbasierter Properties.

Bindable-Test

Hinzufügen und Löschen mehrerer `Bindables`. Properties-Änderungen in den erstellten Bindables und Umschaltung zwischen AS- und CS-Sicht. Erste Effektprüfung (Hat die Umschaltung den erwarteten Effekt auf die angezeigte Modellrepräsentation?). Speichern, Schließen und Öffnen des Modells mit anschließender zweiter Effektprüfung (Auswirkung des Serialisierens und Parsens: Sind noch alle Komponenten enthalten? Haben sich die Properties verändert, die die Oberfläche in das PropertiesSheet einliest?).

ByteBoundary-Test

Hinzufügen und Löschen mehrerer `ByteBoundaries`. Properties-Änderungen an den eingefügten Komponenten. Speichern, Schließen und Öffnen des Modells mit anschließender Effektprüfung (Auswirkung des Serialisierens und Parsens: Sind noch alle Komponenten enthalten? Haben sich die Properties verändert, die die Oberfläche in das PropertiesSheet einliest?).

Array-Test

Hinzufügen und Löschen mehrerer `Arrays`. Danach Schachtelung von Arrays und Properties-Änderungen an diesen. Speichern, Schließen und Öffnen des Modells mit anschließender Effektprüfung (Auswirkung des Serialisierens und Parsens: Sind noch

alle Komponenten enthalten? Haben sich die Properties verändert, die die Oberfläche in das PropertiesSheet einliest?).

Aufbauend darauf: Hinzufügen (auch geschachtelt) von `Bindables` und `ByteBoundaries` an mehreren Stellen im `Array`-Geflecht des ersten Schritts dieses Testfalls. Properties-Änderungen und Prüfung des automatischen Layouts (insbes. *SmartShelf*-Umbruch). Danach Speichern, Schließen und Öffnen des Modells mit anschließender Effektprüfung (Auswirkung des Serialisierens und Parsens: Sind noch alle Komponenten enthalten? Haben sich die Properties verändert, die die Oberfläche in das PropertiesSheet einliest?).

IncludeMessage-Test

Hinzufügen mehrerer Nachrichten, danach Hinzufügen und Löschen mehrerer `IncludeMessages` samt Properties-Änderungen. Speichern, Schließen und Öffnen des Modells mit anschließender Effektprüfung (Auswirkung des Serialisierens und Parsens: Sind noch alle Komponenten enthalten? Haben sich die Properties verändert, die die Oberfläche in das PropertiesSheet einliest?).

Zuletzt wird die aktive Verlinkung der referenzierten PDUs geprüft: Wird nach einem Doppelklick auf die `IncludeMessage`-Komponente auch tatsächlich die referenzierte PDU geöffnet?

3.4 Ausblick

Der Test des Editors kann zukünftig mit den in Abschnitt 3.2.4 beschriebenen Möglichkeiten automatisiert werden und sich so etwa in den nächtlichen Regressionstest bei *ubigrate* einfügen.

Weiterhin ist denkbar, die Testsuite stärker in Richtung Roundtrip-Test zu entwickeln, indem etwa eine fertige MDML-Datei als Referenz hinterlegt wird und das vom Testfall erzeugte Modell dann mit dieser Vorgabe vergleichen wird.

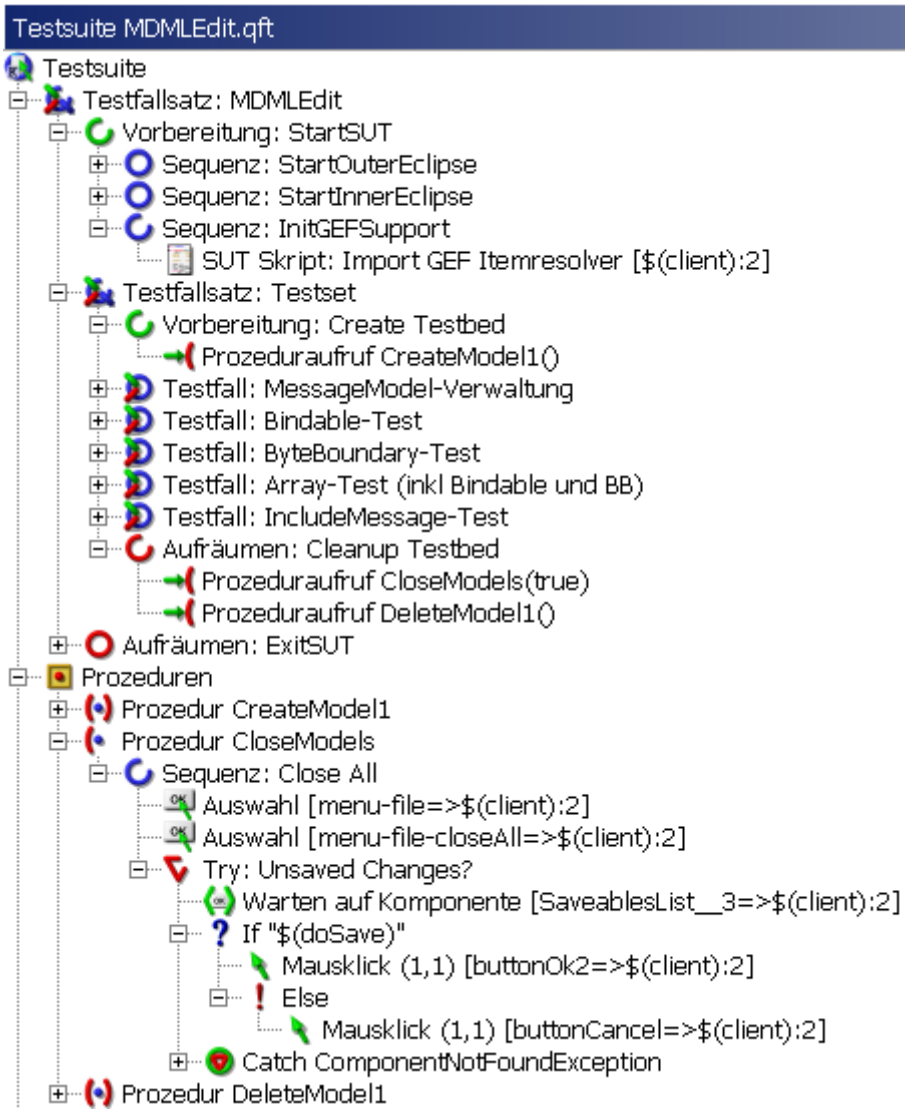

Abbildung 3.4: Der Aufbau der Testsuite für MDMLEdit