Hochschule
Ravensburg-Weingarten
Technik | Wirtschaft | Sozialwesen

Entwicklung eines Prototyps einer Test-Bibliothek zum GUI Framework Jo Widgets für automatisierte Tests

Lukas Groß
18116
Weingarten, 31. Mai 2011

Bachelor-Arbeit

Bachelor-Arbeit

zur Erlangung des akademischen Grades

Bachelor of Science

an der

Hochschule Ravensburg-Weingarten
Technik | Wirtschaft | Sozialwesen

Fakultät Elektrotechnik und Informatik
Studiengang Angewandte Informatik

Thema: **Entwicklung eines Prototyps einer Test-Bibliothek zum GUI Framework Jo Widgets für automatisierte Tests**

Verfasser: Lukas Groß
Ried 16
88371 Ebersbach

1. Prüfer: Prof. Dr.-Ing. Silvia Keller
Hochschule Ravensburg-Weingarten
Doggenriedstraße
88250 Weingarten

2. Prüfer: Dipl.-Inf. Michael Grossmann
innoSysTec GmbH
In Oberwiesen 16
88682 Salem-Neufrach

Abgabedatum: 31. Mai 2011

Abstract

Thema:	Entwicklung eines Prototyps einer Test-Bibliothek zum GUI Framework Jo Widgets für automatisierte Tests
Verfasser:	Lukas Groß
Betreuer:	Prof. Dr.-Ing. Silvia Keller Dipl.-Inf. Michael Grossmann
Abgabedatum:	31. Mai 2011

Diese Bachelorarbeit geht der Frage nach, wie ein Prototyp für eine Test-Bibliothek zum GUI-Framework Jo Widgets für automatisierte Tests aussehen kann. Im Grundlagenteil dieser Arbeit wird zunächst Jo Widgets kurz erläutert und die Probleme beim Testen von GUIs beschrieben. Weiterhin geht es um das Test-Framework JUnit und wie Test-getriebene Entwicklung durchgeführt werden kann.

Im ersten Schritt werden die Anforderungen an die Test-Bibliothek definiert. Diese besteht aus dem Test-Tool und der Test-Architektur. Nach den Anforderungen werden die verschiedenen Techniken für die Testautomatisierung an Beispielen erläutert und anschließend verglichen. Im Fazit wird Stellung dazu genommen, welche Technik für das Test-Tool verwendet wird.

Die Test-Architektur erweitert die Jo Widgets Architektur um Testaspekte. Zum Verständnis wird die Architektur von Jo Widgets vorgestellt und deren wichtigsten Mechanismen erläutert.

Der Entwurf der Test-Bibliothek befasst sich mit der Frage, wie die Kernprobleme der ausgewählten Test-Technik umgesetzt werden können und entwickelt mehrere alternative Lösungsmöglichkeiten. Auf diese Lösungsalternativen und die damit verbundenen Schwierigkeiten wird in der Realisierung eingegangen. Anschließend werden mögliche Erweiterungen und deren Lösungsmöglichkeiten für die Test-Bibliothek vorgestellt und erklärt.

Eidesstattliche Erklärung

Hiermit versichere ich, die vorliegende Arbeit selbstständig und unter ausschließlicher Verwendung der angegebenen Literatur und Hilfsmittel erstellt zu haben. Die Arbeit wurde bisher, in gleicher oder ähnlicher Form, keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Unterschrift Ort, Datum

Inhaltsverzeichnis

Abstract (Zusammenfassung) **5**

Inhaltsverzeichnis **9**

1 Einleitung **13**

1.1 Zielsetzung und Motivation 13

1.2 Inhaltsüberblick 14

2 Grundlagen **15**

2.1 GUI-Framework Jo Widgets 15

2.2 Test-getriebene Entwicklung 15

2.3 JUnit-Tests 16

2.4 Allgemeine Probleme bei GUI-Tests 17

2.5 Besonderheiten automatisierter GUI-Tests 17

3 Anforderungen **19**

3.1 Test-Architektur 19

3.2 Test-Tool 19

4 Techniken für die Testautomatisierung **21**

4.1 GUI-Unit-Test-Technik 21

4.1.1 Beispiel eines Testablaufs 22

4.1.2 Vorteile/Nachteile 22

4.2 Capture/Replay Technik 23

4.2.1 Beispiel eines Testablaufs 24
4.2.2 Vorteile/Nachteile 25
4.3 Data-Driven-Technik 26
4.3.1 Beispiel eines Testablaufs 26
4.3.2 Vorteile/Nachteile 26
4.4 Keyword-Driven-Technik 27
4.4.1 Beispiel eines Testablaufs 27
4.4.2 Vorteile/Nachteile 28
4.5 Vergleich der Techniken 28
4.6 Fazit 29

5 Architektur von Jo Widgets 31

5.1 Einführung in die Architektur 31
5.2 Die Architektur 33
5.2.1 Api 34
5.2.2 Spi 34
5.2.3 Impl 34
5.2.4 Swt/Swing Impl 34
5.3 Wie ein Widget erstellt wird 34
5.3.1 Die „BluePrintFactory“ 35
5.3.2 Die „WidgetFactory“ 36

6 Entwurf 37

6.1 Test-Architektur 37
6.1.1 Automatisiertes Ablaufen der Tests 37
6.1.2 Erweiterung der Widgets um HCI-Aspekte 37
6.2 Test-Tool 38
6.2.1 Die Test-Tool-GUI 38
6.2.2 Das Test-Tool einbinden 39
6.2.3 Ein Widget identifizieren 39

6.2.4 Wie das Test-Tool die GUI „kennen lernt“ 40

7 Realisierung 41

7.1 Entwicklung der Test-Architektur 41

7.1.1 Die Projektstruktur 41

7.1.2 Umsetzung der HCI-Aspekte 42

7.1.3 Erstellen eines GUI-Unit-Tests 43

7.2 Entwicklung des Test-Tools 44

7.2.1 Einbinden und Starten des Test-Tools 44

7.2.2 Die Benutzeroberfläche des Test-Tools 45

7.2.3 Generierung der Widget-ID 46

7.2.4 Ein Widget wiederfinden 47

7.2.5 Aufzeichnen einer Nutzeraktion 47

7.2.6 Serialisierung des aufgezeichneten Materials 48

8 Erweiterungen/Ausblick 51

8.1 Weitere Nutzeraktionen 51

8.2 Checkpoints 51

8.3 Mausanzeigeunterstützung 52

8.4 Eigene Properties festlegen 52

9 Fazit 53

Abkürzungsverzeichnis 55

Abbildungsverzeichnis 57

Listings 59

Tabellenverzeichnis 61

Literaturverzeichnis 63

Kapitel 1

Einleitung

In komplexen Software-Projekten gibt es stets einen sich wiederholenden Arbeitsaufwand beim Erstellen und Ausführen von Tests. Ein Großteil dieses Arbeitsaufwandes kann automatisiert werden. Durch die Automatisierung der Tests können diese häufiger durchgeführt und die Software dadurch gründlicher getestet werden. Außerdem fällt das erneute, manuelle Ausführen von Tests im Testzyklus weg und kann für andere Aktivitäten genutzt werden. Dadurch sinkt nicht nur der Aufwand und damit die Kosten für das Testen der Software, sondern es ist auch langfristig eine Strategie, um einen hohen Qualitätsstandard zu sichern.

In der Praxis ist das Erstellen von automatisierten GUI-Tests häufig von vielschichtigen Problemen begleitet. Eine der Schwierigkeiten ist, dass die Erzeugung der GUI-Komponenten von UI-Technologien (wie z.B. Swing oder SWT) auf einem CI-System, wie z.B. dem Hudson, zu sogenannten „Headless Eceptions“[1] führt. Weiterhin kapseln diese ihre GUI-Komponenten nicht mit Hilfe von Schnittstellen, was die Erstellung von Dummy[2]- oder Fake[3]-Objekten erschwert. Weitere Probleme entstehen dadurch, dass grafische Bedienoberflächen aus einer Darstellungsschicht und einer Objektebene bestehen und sich daraus ein komplexes Gebilde ergibt. Dieses zu durchschauen ist nicht immer leicht und macht das Automatisieren von GUI-Tests zu einer großen Herausforderung.

1.1 Zielsetzung und Motivation

Bisher wurden bei der Firma innoSysTec GmbH GUI-Tests manuell durchgeführt. Dies soll in Zukunft durch die Automatisierung der GUI-Tests unterstützt werden.

[1] Headless Exception ≙ Eine Fehlermeldung die „geworfen“ wird, wenn auf einem System ohne Anzeige-, Tastatur- oder Mausunterstützung versucht wird, ein Code auszuführen der diese benötigt. (frei nach [Ora11, Javadoc zu “Headless Exception“])

[2] „Dummy-Objekte stellen eine Schnittstelle zur Verfügung, verfügen aber über keine Implementierung. Sie werden meist eingesetzt, falls das zu testende Objekt keinen Rückgabewert erfordert.“ [SBD+10, s.143]

[3] „Fake-Objekte beinhalten Implementierungen und simulieren Funktionalitäten, in dem vordefinierte Werte zurückgegeben werden.“ [SBD+10, s.143]

Ziel dieser Arbeit ist es mit Hilfe der zu entwickelnden Test-Bibliothek, automatisierte GUI-Tests mit dem GUI-Framework Jo Widgets zu ermöglichen.

Dabei geht es nicht darum, ein voll funktionsfähiges Test-Tool zu erstellen, sondern viel mehr um das entwickeln eines Prototyps, der eine mögliche Lösung zeigt. Hierfür sollen die verschiedenen Techniken für die Testautomatisierung miteinander verglichen werden.

Das Test-Tool soll es ermöglichen HCI-Aspekte wie Klicken, Drücken, Sehen oder Finden zu simulieren. Weiterhin ist ein Ziel, eine Möglichkeit zu bieten, die GUI-Tests automatisiert auf CI-Systemen laufen zu lassen.

1.2 Inhaltsüberblick

Die einzelnen Kapitel dieser Arbeit bauen aufeinander auf und sollten deshalb nacheinander gelesen werden. Im ersten Kapitel, „Grundlagen“ werden die Technologien und Techniken für Testautomatisierung erläutert. Danach werden die Anforderungen an die Test-Bibliothek aufgelistet. Im anschließenden Vergleich der einzelnen Techniken für die Testautomatisierung wird die geeignetste Technik ausgewählt.

Das Kapitel „Architektur von Jo Widgets“ erläutert die Konzepte in Jo Widgets die für das Verständnis wichtig sind. Das Kapitel „Entwurf“ befasst sich mit der Planung und Umsetzung der Test-Bibliothek. Anschließend wird im Kapitel „Realisierung“ die konkrete Umsetzung des Prototyps besprochen.

Zum Abschluss wird im Kapitel „Erweiterungen“ ein Ausblick auf die möglichen Erweiterungen der Bibliothek gegeben. Das Kapitel “Fazit“ bildet den Abschluss dieser Arbeit.

Kapitel 2

Grundlagen

Im folgenden Kapitel werden die Grundlagen, Techniken und Technologien erläutert, die für diese Arbeit wichtig sind.

2.1 GUI-Framework Jo Widgets

Jo Widgets[1] ist ein GUI-Framework, das von der Firma innoSysTec GmbH entwickelt wird. Die Ziele des Open-Source-Projektes sind:

- Single Sourcing[2]
- Eine einfach zu verwendende API[3]
- Erweiterbarkeit

Mit Hilfe von Jo Widgets können Enterprise-Applikationen erstellt werden. Dadurch sind diese unabhängig von UI-Technologie wie Swing oder SWT. In Kapitel 5 "Architektur von Jo Widgets" werden die wichtigsten Abläufe in Jo Widgets erläutert.

2.2 Test-getriebene Entwicklung

Unter „Test-getriebene Entwicklung" versteht man den Vorgang, zuerst Tests für eine Komponente[4] zu schreiben und die Komponente selbst erst danach zu implementieren. Während diesem Vorgang werden folgende Schritte durchgeführt:

1. Erstellen eines Tests, der eine Komponente testet, die noch nicht erstellt ist. Dieser Test schlägt deshalb zuerst fehl.

[1] Jo Widgets ≙ Java open Widget Api. [Gro11b]

[2] Single Sourcing ≙ „Single source publishing, also known as single sourcing, allows the same source to be used in different runtime environments" [Chr11, Folie 25]

[3] ≙ Application Programming Interface.

[4] Eine Komponente ist eine Klasse oder eine Methode.

2. Überprüfen, ob der erstellte Test fehlschlägt, um zu zeigen, dass er durchgeführt wurde.
3. Die Komponente implementieren und die Tests erneut ausführen.
4. Überprüfen, ob die Tests fehlerfrei durchlaufen werden.
5. Den Quellcode der Komponente verbessern und unnötigen Code entfernen.

Durch das Erstellen von Tests mit dieser Methode, werden Fehler mit einer hohen Wahrscheinlichkeit sehr früh erkannt. Ein weiterer Vorteil ist, dass nahezu jede Komponente getestet ist und damit eine hohe Test-Abdeckung des Quellcodes[5] erreicht wird. [MfMRWF07, s.198]

Im nachfolgenden Abschnitt wird das Framework JUnit vorgestellt, mit dem Test getriebene Entwicklung durchgeführt werden kann.

2.3 JUnit-Tests

JUnit ist ein Framework für das automatisierte Testen von einzelnen Einheiten (engl. Units) wie Klassen oder Methoden in Java. Das Framework soll genutzt werden, um die mit der Test-Bibliothek erstellen Tests automatisiert ablaufen zu lassen.

Um einen Test mit JUnit zu erstellen, wird zuerst eine Klasse erstellt. Diese enthält eine oder mehrere Testmethoden. Eine Testmethode ist eine Methode, die den Quellcode für den Test beinhaltet und mit der Annotation [6] „@Test“ versehen ist. Diese Annotation ist notwendig, um dem Framework zu signalisieren, dass die Testmethode automatisiert ablaufen soll. Tests können entweder fehlschlagen oder erfolgreich sein. Dies wird z.B. in der Eclipse IDE durch ein grünes Symbol bzw. im Falle eines Misserfolges durch ein rotes Symbol dargestellt. Die erwarteten Testergebnisse können mithilfe von Assertions [7] überprüft werden. Ein einzelner Vergleich wird Assert genannt.

```
1  public class MyTestClass{
2
3      @Test
4      public void testFrameVisibility() {
5        JoFrame myFrame = new JoFrame();
6        myFrame.setVisible(true);
7
8        Assert.assertTrue(myFrame.isVisible());
9      }
10 }
```

Listing 2.1: Beispiel JUnit Test

[5] Die Test-Abdeckung eines Quellcodes (engl. Test Coverage) gibt an, wie viel Prozent des gesamten Quellcodes in Tests aufgerufen wird. Dies ist ein Hilfsmittel um festzustellen, in welchem Maße eine Software getestet ist.

[6] Annotations stellen Metadaten für Java-Programme zur Verfügung.

[7] Assertions ≙ „Eine Sammlung von statischen Methoden, mit denen Ergebniswerte mit erwarteten Werten verglichen werden können.“ [Bec05, s.26]

Das Listing 2.1 zeigt einen JUnit-Test in Jo Widgets. Dieser testet die Sichtbarkeit eines Frames. Hierzu wird das Frame nach der Instanziierung sichtbar gemacht. Die Sichtbarkeit wird anschließend mit einem Assert überprüft.

Nach jeder Änderung im Quellcode werden alle Tests ausgeführt. Schlägt ein Test fehl, müssen die Fehler behoben werden, bevor weitere Änderungen durchgeführt werden dürfen. Das erneute Ausführen aller bereits vorhanden Tests wird „Regression Testing“ genannt. [Mrs08, Kapitel 4.30] Mit dieser Methode kann überprüft werden, ob Änderungen im Quellcode unerwartete Seiteneffekte ausgelöst haben und ob Fehler wirklich behoben wurden.

2.4 Allgemeine Probleme bei GUI-Tests

Auf den ersten Blick erscheint es nicht schwer GUI-Elemente zu testen. Wie in Listing 2.1 zu finden in Abschnitt „JUnit Tests“ zu sehen ist, kann ein GUI-Element (in diesem Beispiel ein Frame) erzeugt werden und die Sichtbarkeit oder andere Eigenschaften getestet werden. Dadurch lassen sich einzelne Elemente der GUI wie Buttons, Fenster, Labels oder Texteingaben auf ihre Funktion testen.

In modernen GUI-Applikationen sind eine große Anzahl dieser GUI-Elemente vorhanden. Diese sind voneinander abhängig und reagieren je nach Status[8] der GUI anders. Zusätzlich werden sie durch Benutzerinteraktionen wie Mausbewegungen, Mausklicks oder Tastendrücke beeinflusst. Dies führt dazu, dass die erstellten Tests schnell unübersichtlich und kompliziert werden.

Neben diesem funktionalen Aspekt gibt es noch den zeitlichen und optischen Aspekt. Was ist, wenn der Nutzer eine DB-Anfrage startet? Wie kann der Test darstellen, dass auf eine Antwort eines Servers gewartet wird? Wie sieht ein GUI-Element aus? Hat das Label wirklich eine rote Hintergrundfarbe? Sind zusammengehörende Labels/Texteingaben symmetrisch angeordnet? Ist jedes GUI-Element an der richtigen Position? Beeinflusst wird dies durch das verwendete Layout, „Look and Feel“ und anderen Faktoren. Dies kann durch einen Test wie oben beschrieben nur schwer bis gar nicht überprüft werden, da es schwierig ist zu testen, was der Nutzer sieht.

In Kapitel 4 werden Testansätze beschrieben und untersucht, die ermöglichen moderne GUI-Applikationen zu testen und teilweise die Probleme lösen, die in diesem Abschnitt beschrieben wurden. [SHBA11, s.224,225],[Ham04, s.50,51]

2.5 Besonderheiten automatisierter GUI-Tests

Bei dem automatisierten Testen von GUIs bestehen neben den im vorherigen Abschnitt besprochenen Schwierigkeiten noch weitere. Wird für die kontinuierliche In-

[8] Der Status der GUI, wird durch die GUI mit all ihren Komponenten und deren Zustand (z.B. ist ein Fenster sichtbar oder nicht) definiert.

tegration (engl. Continuous Integration) ein CI- Server[9] wie Hudson [10] verwendet, muss darauf geachtet werden, dass dieser meist ein System ohne Monitor, Keyboard oder anderen Ein- und Ausgabegeräten ist. Wird auf einem dieser Systeme ein GUI Test ausgeführt, schlägt dieser z.B. unter Java mit einer „Headless Exception“ fehl.

Dies bedeutet, dass bei automatisierten GUI-Tests zusätzlich darauf zu achten ist, dass der Code auch auf einem System ohne Anzeige- und Eingabegeräte lauffähig sein muss. Hierfür wird gewöhnlich eine Dummy- oder Fake-Implementierung der verwendeten GUI-Elemente verwendet. Dadurch ist es möglich, die Funktionalität der Elemente zu testen, ohne sie grafisch darzustellen.

[9] CI Server ≙ „...a dedicated server machine responsible for „continuously“ building and testing all the code in a given project.“ [CH08, s.22]

[10] Hudson ist ein Open-Source-Continuous-Integration-Server.
Homepage: http://java.net/projects/hudson/

Kapitel 3

Anforderungen

In diesem Kapitel werden die Anforderungen an die GUI-Test-Bibliothek beschrieben. Die Bibliothek unterteilt sich in zwei Bereiche, der Test-Architektur und dem Test-Tool. Die Test-Architektur stellt eine Erweiterung der „Jo Widgets"-Architektur um Testaspekte dar. Das Test-Tool verwendet die Testaspekte um automatisierte GUI-Tests zu ermöglichen.

Die Anforderungen für die beiden Bereiche werden getrennt aufgelistet.

3.1 Test-Architektur

- Test-Architektur getrennt von „Jo Widgets"-Architektur in eigenen Projekten. Projektstruktur nach dem Vorbild von Jo Widgets.
- Tests sollen erstellt werden können, ohne dass das Test-Tool benutzt werden muss.

3.2 Test-Tool

- Die Nutzerinteraktionen mit der GUI sollen aufgezeichnet und simuliert werden können.
- Die Tests sollen sowohl grafisch als auch automatisiert abspielbar sein.
- Die aufgezeichneten Tests müssen in geeigneter Form gespeichert werden können.
- Das Test-Tool soll über eine eigenständige GUI verfügen. Diese muss unabhängig von der getesteten Anwendung sein.
- Aufgezeichnete Nutzer Aktionen müssen angezeigt und gelöscht werden können.

Kapitel 4

Techniken für die Testautomatisierung

Für das automatisierte Testen von GUIs werden verschiedene Ansätze, auch Techniken oder Strategien genannt, verfolgt. Dieses Kapitel gibt einen Überblick über die einzelnen Techniken, beschreibt diese und zeigt Beispiele. Anschließend wird im Vergleich eine geeignete Technik für das Test-Tool der zu entwickelnden Test-Bibliothek ausgewählt.

Für die Beispiele der Test-Techniken werden Open-Source-Projekte bevorzugt. Diese bieten den Vorteil, dass der Quellcode eingesehen werden kann und von Beginn an der volle Funktionsumfang zugänglich ist. Es wurden nur Projekte in Betracht gezogen, die über den Beta-Status hinaus sind und über eine gute Dokumentation verfügen.

4.1 GUI-Unit-Test-Technik

Mit der GUI-Unit-Test-Technik werden Komponenten auf ihre Funktionalität durch einen Unit-Test (s. Kapitel 2.3) überprüft. Ein Beispiel für diese Technik ist das Open-Source-GUI-Test-Framework Abbot[1]. Abbot kann dazu verwendet werden mit Unit-Tests funktionale Tests für AWT und Swing durchzuführen. Unter einem funktionalen Test versteht man das Testen einer Komponente auf ihre Funktion. Zum Beispiel: „Hat ein Button wirklich den richtigen Tooltip nach dem Aufruf der entsprechenden Methode?“
Dem Tester ist der Quellcode bei dieser Art des Testens nicht bekannt (Black-Box-Test). Der Tester überprüft, ob die Komponente wie erwartet funktioniert. In diesem Beispiel, ob der Button den richtigen Tooltip zurückliefert.

Um GUI-Unit-Tests durchzuführen, wird in der einfachsten Form eine Erweiterung für JUnit geschrieben, die es ermöglicht, Nutzeraktionen (z.B. „einen Button drücken“) zu simulieren.

[1] Abbot ≙ „Framework for automated testing of Java GUI components and programs.“[Wal11]

4.1.1 Beispiel eines Testablaufs

Das folgende Beispiel[2] zeigt einen GUI Unit Test mit Abbot.

```
private boolean invoked;

@Test
public void testButtonClick(){

  JButton button = new JButton();
  invoked = false;
  button.addActionListener(new ActionListener(){
  public void actionPerformed(ActionEvent e){
    invoked = true;
    }
  });
  ...
  // Den Button der GUI hinzufügen
  // Das Fenster anzeigen
  ...
  ComponentTester tester = new ComponentTester();
  tester.actionClick(button);
  // Überprüfen ob der Button geklickt wurde
  Assert.assertTrue(invoked);
}
```

Listing 4.1: Beispiel-GUI-Unit-Test mit Abbot

Bei diesem Test wird das, durch einen Nutzer simulierte, „Drücken" eines Buttons überprüft. Für das Simulieren von Nutzeraktionen bei Abbot sind die Tester-Klassen zuständig. Diese stellen Methoden zur Verfügung, die Nutzeraktionen wie „Eine Maustaste drücken", „Eine Maustaste loslassen" oder „Eine Taste drücken" simulieren.

In diesem Test wird die Methode „actionClick(...)" aufgerufen. Diese simuliert einen Mausklick durch den Nutzer. Wurde der Button gedrückt, erhält der „ActionListener" des Buttons ein „ActionEvent" und die Variable „invoked" wird verändert. Am Testende wird durch einen Assert überprüft, ob der „ActionListener" das Event erhalten hat.

4.1.2 Vorteile/Nachteile

Vorteile dieser Technik:

- Da in den Tests einzelne Komponenten getestet werden, sind diese bei Veränderungen der Anwendung noch lauffähig (robuste Tests).
- Einzelne Komponenten können gezielt getestet werden.
- Tests können von Anfang an durchgeführt werden, es wird keine vollständig funktionsfähige Anwendung benötigt (Stichwort „Test-getriebene Entwicklung").

[2] In Anlehnung an: http://abbot.sourceforge.net/doc/overview.shtml, besucht am: 19.4.2011

Nachteile dieser Technik:

- Hoher Aufwand bei der Erstellung der Tests.
- Zur Erstellung von Tests sind Kenntnisse über den Aufbau der GUI nötig (White-Box-Test).[3]

4.2 Capture/Replay Technik

CR-Tools [4] zeichnen die vom Nutzer durchgeführten Aktionen auf und können diese abspielen. Als Beispiel dient auch hier das Open-Source-Projekt Abbot. Dieses unterstützt das Aufzeichnen und Abspielen von GUIs, die auf AWT oder Swing basieren, und kann über Java Webstart ausgeführt werden. Die Abbildung 4.1 zeigt die Benutzeroberfläche von Abbot. Diese ist in vier Bereiche eingeteilt:

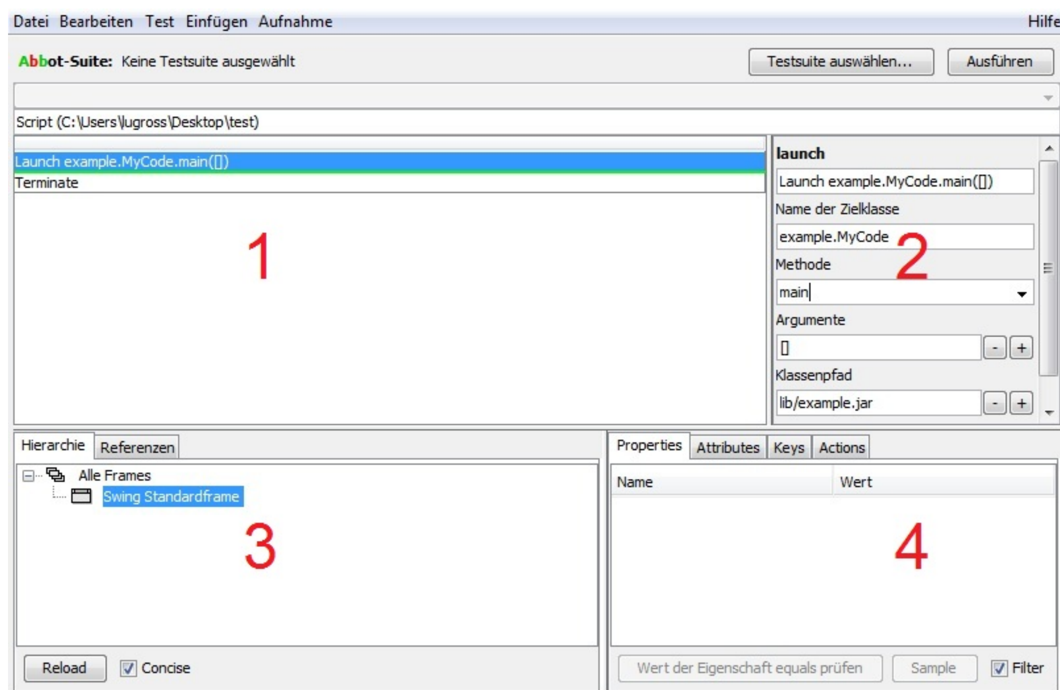

Abbildung 4.1: Die Benutzeroberfläche von Abbot

Der Bereich 1.) zeigt das aufgezeichnete Testskript. Zu Beginn besteht es aus dem Aufruf der Main-Methode und dem Terminieren der Anwendung. Um eine Anwendung zu starten, müssen Informationen, wie z.B. die Klasse mit der Main-Methode, bei 2.) angeben werden. Die Hierarchie der GUI-Elemente kann bei 3.) eingesehen werden. In 4.) sind die Eigenschaften der Auswahl aus 3.) zu sehen.

[3]Quelle: http//www.bea-projects.de/files/BlueNotes/BlueNote - GUI-Tests.pdf, besucht am 15.05.2011

[4]CR Tools ≙ Capture & Replay Tools

4.2.1 Beispiel eines Testablaufs

Für das Beispiel wird die im Webstart von Abbot enthaltene Beispielanwendung (siehe Abbildung 4.2) getestet.

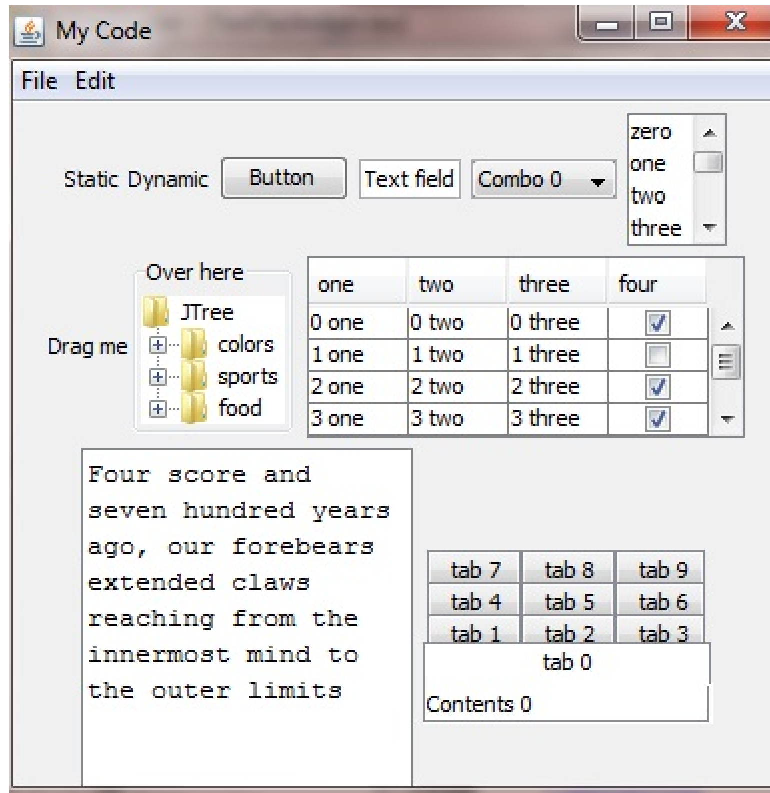

Abbildung 4.2: Die Beispielanwendung von Abbot.

Die Anwendung besteht aus Buttons, einer Tabelle, einem Textfeld und weiteren Standard-GUI-Elementen. Um die Anwendung zu testen, muss Abbot entsprechend konfiguriert werden. Das Aufzeichnen (Capture Mode) wird im Menü „Aufnahme“ aktiviert. Aufgezeichnet werden alle Aktionen des Nutzers, z.B. Mausklicks, Änderungen im Textfeld oder das Selektieren von Elementen. Je nach Einstellung beinhaltet das auch die Mausbewegungen. Dies kann zu unübersichtlichen Tests führen, da dem Testskript sehr viele Aktionen hinzugefügt werden. Deshalb wurde dies in diesem Beispiel deaktiviert. Um den Test übersichtlich zu gestalten, werden zusätzlich nur wenige Aktionen durchgeführt.

Ziel des Beispieltests ist es, den Inhalt des Textfeldes (unten links in der GUI zu sehen) zu löschen. Hierzu wird im Test die Maus zum Textfeld bewegt, der Text markiert und mit der Backspace Taste gelöscht. Abbot beendet automatisch das Aufzeichnen des Tests, wenn die Anwendung geschlossen wurde.

Nach dem schließen der Anwendung kann das Skript bearbeitet werden. Die Abbildung 4.3 zeigt das erstellte Skript.

Das Testskript stellt den sequenziellen Ablauf der durch den Nutzer durchgeführten Aktionen dar. Die Aktionen sind im Testskript ab Zeile 2 zu sehen. Sie zeigen die durchgeführten Aktionen:

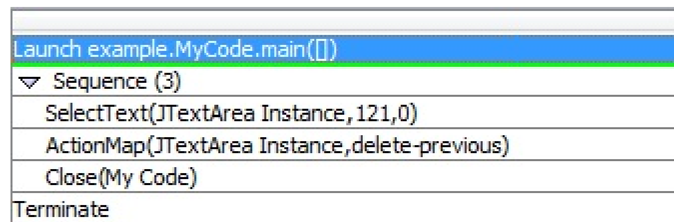

Abbildung 4.3: Das erstellte Testskript

1. Zeile 3 „SelectText(JTextArea Instance,121,0)“ der Inhalt des Textfeldes wird markiert.
2. Zeile 4 „ActionMap(JTextArea Instance,delete-previous)“ der Inhalt des zuvor ausgewählten Elementes wird gelöscht.
3. Zeile 5 „Close(My Code)“ die Anwendung wurde beendet.

Das Abspielen (Replay Mode) des Tests wird durch klicken von „Ausführen“ gestartet.

4.2.2 Vorteile/Nachteile

Vorteile dieser Technik:

- Tests können leicht und ohne Programmierkenntnisse erstellt werden.
- Es ist jederzeit nachvollziehbar welche Aktionen vom Tester durchgeführt wurden.
- Die GUI kann als Gesamtsystem getestet werden.

Probleme bei dieser Technik:

- Beim Aufzeichnen aller Events (wie z.B. Mausbewegungen) kann das Testskript sehr lange und dadurch unübersichtlich werden.
- Wird die GUI verändert, müssen die Tests eventuell angepasst werden (erfordert Programmierkenntnisse).
- Das Pflegen der Tests kann sehr arbeitsintensiv sein.

[LW04, s.28,29]

4.3 Data-Driven-Technik

Bei dieser Technik wird die Anwendung mit einem zuvor angefertigten Datensatz getestet. Die Idee ist, dass bei ähnlichen Eingaben (wie z.B. dem Benutzernamen und Passwort in einem Login-Dialog) nicht alle Daten manuell eingegeben werden müssen, sondern aus einem Datensatz geladen werden. Dieser enthält neben den Eingabedaten auch die erwartenden Resultate. Diese werden in einer Tabelle gespeichert und liegen entweder lokal z.B. als Excel-Dokument oder extern z.B. auf einem DB-Server. Mit dieser Technik alleine können keine GUI Tests durchgeführt werden. Sie ist viel mehr als ein Hilfsmittel zu verstehen, mit dem die Testdaten von den Tests getrennt werden können.

4.3.1 Beispiel eines Testablaufs

Der Ablauf eines Test mit diesem Ansatz sieht folgendermaßen aus:

Erstellen des Testdatensatzes. Getestet werden soll ein Login-Dialog in dem jeweils der Benutzername und das Passwort eingegeben werden. Ist die Eingabe gültig wird das Ergebnis „ok“ erwartet, im Falle einer ungültigen Eingabe „error“.

UserName	Password	Expected Result
lukas	123	ok
lukas	test	error
test	test	ok
admin	9xT123	ok
lukas		error

Tabelle 4.1: Beispiel Testdaten für einen Login-Dialog

Der erstellte Testdatensatz kann nun z.B. in einem angepassten GUI-Unit-Test verwendet werden, um die Funktionalität des Login-Dialogs zu testen. Hierzu müsste die Testmethode z.B. als Parameter die Werte aus dem Testdatensatz erhalten. Für jede Zeile der Tabelle des Testdatensatzes wird der Test einmal ausgeführt. Damit kann eine große Anzahl an Testfällen abgedeckt werden.

Für diese Technik gibt es als Hilfsmittel die Möglichkeit, Generatoren zu verwenden, die nach vordefinierten Regeln Testdatensätze erzeugen. Damit kann der Aufwand für das Erstellen der Testdaten gesenkt werden. Voraussetzung ist, dass die Testdaten über ein gemeinsames Muster verfügen.

4.3.2 Vorteile/Nachteile

Vorteile dieser Technik:

- Einfach und schnell zu implementieren.
- Die Testdaten werden von den Tests getrennt.

- Daten können zentral verwaltet werden.
- Automatisierte Tests müssen nicht aufgrund von Änderungen der Testdaten angepasst werden.

Nachteile dieser Technik:

- Testdaten sind „hart codiert“ und müssen umständlich gepflegt werden. Testdaten Generatoren können dies reduzieren, wenn die Testdaten ein gemeinsames Muster besitzen.
- Testdaten werden unbrauchbar, wenn die Anwendung verändert wird.

4.4 Keyword-Driven-Technik

Bei der Keyword-Driven-Technik, auch Table-Driven-Technik genannten, werden Tests in einer Tabelle erstellt. Als Beispiel dient das Open-Source-Framework SAFS (Software Automation Framework Support)[5]. Zum Erstellen der Tests wird ein Vokabular von Schlüsselwörtern (die Keywords) verwendet. Diese werden vom Framework ausgewertet und umgesetzt.

4.4.1 Beispiel eines Testablaufs

Document	Component	Action	Parameter
LoginPage	UserIDField	InputText	MyUserID
LoginPage	PasswordField	InputText	MyPassword
LoginPage	SubmitButton	Click	

Tabelle 4.2: Beispiel Test eines Login Dialogs

Das Beispiel in Tabelle 4.2[6] zeigt den Test eines Login-Dialogs mit SAFS. Der Dialog „LoginPage“ besteht aus den Textfeldern „UserIDField“ und „PasswordField“, sowie dem Button „SubmitButton“. Die Bezeichnungen der Komponenten sind in Spalte 1 und 2 zu sehen. In Spalte 3 der Beispieltabelle sind die Aktionen zu finden, die auf den Komponenten ausgeführt werden sollen. Durch das Schlüsselwort „InputText“ wird dem Framework mitgeteilt, dass ein Text eingeben werden soll. In der letzten Spalte sind Parameter zu finden die für die Aktionen benötigt werden.

Zeile 1 der Tabelle wäre also folgendermaßen zu interpretieren: „Füge in das Textfeld „UserIDField“ in dem Dokument „LoginPage“, den Wert von „MyUserID“ ein.“

[5] „SAFS is a tool-independent automation framework supporting keyword-driven functional test automation“ Quelle: http://sourceforge.net/projects/safsdev/, besucht am 19.4.2011

[6] Quelle: [Nag11]

4.4.2 Vorteile/Nachteile

Vorteile dieser Technik

- Die Tests können häufig wiederverwendet werden. Der Test für den Login-Dialog könnte z.B. durch eine geringe Anpassung für einen ähnlichen Dialog verwendet werden.
- Tests können ohne Programmierkenntnisse nach einer kurzen Einarbeitungszeit erstellt werden.

Nachteile dieser Technik

- Hoher Implementierungsaufwand [Nag11]
- Kein Test für ein Gesamtsystem, da nur einzelne Komponenten getestet werden.

4.5 Vergleich der Techniken

Die Tabelle 4.3 zeigt die Test-Techniken im Vergleich.

	ATD[1]	RT[2]	ATE[3]	GTA[4]
GUI-Unit-Test	hoch	hoch	hoch	funktional
Capture&Replay	hoch	gering	gering	regression
Data-Driven	-	-	-	beliebig
Keyword-Driven	hoch	hoch	mittel	funktional

Tabelle 4.3: Techniken für Testautomatisierung im Vergleich

Abkürzungen:

[1] Aufwand bei Änderung der Testdaten
[2] Robustheit der Tests
[3] Aufwand Test-Erstellung
[4] Geeignete Testart

Spalte 1, „ATD“, der Tabelle zeigt den Aufwand bei Änderung der Testdaten. Damit ist die Zeit und Arbeit gemeint, die benötigt wird, die Tests auf die Änderungen der Testdaten anzupassen. Dabei wird davon ausgegangen, dass die jeweilige Technik keine anderen Techniken als Hilfsmittel verwenden und die Testdaten „hart codiert“ in den Tests stehen. Dieser Aufwand kann reduziert werden, wenn eine Technik in Kombination mit der Data-Driven-Technik verwendet wird. Die Tests werden damit von den Testdaten getrennt und können zentral verwaltet werden (siehe Vorteile Data-Driven-Technik).

Die Robustheit der Tests wird in Spalte 2, „RT“, beschrieben. Sie gibt an, wie anfällig, die mit der jeweiligen Technik erstellten Tests gegenüber Änderungen der Anwendung sind. Der Wert „hoch“ bedeutet, dass die Tests nur sehr selten angepasst werden müssen, „gering“ hingegen, dass bereits eine Änderung innerhalb der Anwendung ausreichen kann, damit die Tests neu erstellt werden müssen. In der Tabelle ist zu sehen, dass bei den Techniken für funktionale Tests die Robustheit der Tests hoch ist. Der Grund dafür ist, dass bei diesen Techniken einzelne Komponenten getestet werden und kein Gesamtsystem. Wird ein solches getestet, sinkt fast immer automatisch die Robustheit der Tests, wenn die Anwendung verändert wird. Deshalb ist die Robustheit der Tests bei der Capture&Replay-Technik gering. Da mit der Data-Driven-Technik alleine keine Tests erstellt werden können, wird kein Wert angegeben.

Spalte 3, „ATE“, zeigt den Aufwand der betrieben werden muss um einen Test zu erstellen. Bei GUI-Unit-Tests besteht ein hoher Aufwand, da dort ähnlich viel Quellcode wie in einem Unit-Test geschrieben werden muss. Dafür ist der Test aber auch unabhängig von Anwendungen. Tests lassen sich mithilfe der Capture&Replay-Technik am einfachsten und schnellsten erstellen. Die Aktionen des Nutzers werden aufgezeichnet und als Skript gespeichert und lassen sich mit dem verwendeten Test-Tool automatisiert abspielen. Mit der Keyword-Driven-Technik lassen sich Tests schneller als mit GUI-Unit-Tests erstellen. Wie die Beispielabbildung. 4.2 im Abschnitt „Keyword-Driven Technik“ zeigt, genügt es den Test mit Schlüsselwörter zu formulieren.

In der letzten Spalte, „GTA“, wird angegeben für welche Testart sich die jeweilige Test-Technik am besten eignet.

4.6 Fazit

In der Praxis verknüpfen Test-Tools, wie z.B. das in den Beispielen beschriebene Abbot, mehrere Test-Techniken. Keine Test-Technik alleine eignet sich für alle Testarten. Erst in Kombination mit anderen Techniken wird ein Test-Tool wirklich vielseitig einsetzbar. Die Keyword-Driven-Technik oder GUI-Unit-Tests sind für funktionale Tests zu empfehlen, wenn einzelne Komponenten geprüft werden sollen. Für Regressionstests eignen sich Capture&Replay-Tools, da diese die vom Nutzer durchgeführten Aktionen genau wiedergeben. Sie unterstützen damit auch manuelles und exploratives[7] Testen, wenn nachvollzogen werden kann, welche Interaktionen stattfanden und wie oft diese ausgeführt wurden.

Bei der Firma innoSysTec ist das Ziel, das GUI-Testen durch Testautomatisierung zu erleichtern. Bisher wurden GUI-Tests manuell durchgeführt. Die durchzuführenden Schritte wurden handschriftlich, in einem mit dem Kunden festgelegten Prüfprotokoll, festgehalten und abgearbeitet. Durch die Capture&Replay-Technik kann dies erleichtert werden. Ein weiterer Vorteil ist, dass beim explorativen Testen oder dem Durchführen von Regressionstests der Testablauf stets nachvollziehbar bleibt. Sie ist auch die einzige Technik, die der Anforderung entspricht, dass Tests durch Aufzeich-

[7] Unter explorativen Testen ist das Testen einer Anwendung ohne Zielvorgabe gemeint. Der Tester probiert die Anwendung aus und versucht Fehler zu finden.

nung von Nutzeraktionen erstellt werden können. Deshalb wird das Test-Tool diese Technik verwenden.

Eine weitere Anforderung ist das Erstellen von Tests ohne das Test-Tool zu verwenden. Dazu soll JUnit verwendet werden. Für funktionale Tests eignen sich GUI-Unit-Tests und die Keyword-Driven-Technik. Da bereits in Jo Widgets funktionale Tests mit JUnit durchgeführt werden, wird als Technik für die funktionalen Tests die GUI-Unit-Test-Technik verwendet. Mit Hilfe der Data-Driven-Technik, werden die Testdaten von den Tests getrennt.

Kapitel 5

Architektur von Jo Widgets

In diesem Kapitel wird die Architektur von Jo Widgets beschrieben. Ziel ist es, dem Leser einen Überblick zu geben und die für die Test Bibliothek relevanten Konzepte und Mechanismen zu erklären. Dies dient als Vorbereitung und Nachschlagewerk für die Kapitel 6 „Entwurf“ und 7 „Realisierung“.

5.1 Einführung in die Architektur

Das Hauptziel von Jo Widgets ist Single Sourcing zu ermöglichen. Dadurch können Enterprise-Applikationen erstellt werden, ohne dass es nötig ist, sich auf ein bestimmtes GUI-Framework festzulegen. Dieser Abschnitt zeigt wie Jo Widgets dies ermöglicht. Hierzu wird an Beispielarchitekturen gezeigt, wie Jo Widgets entstanden ist.

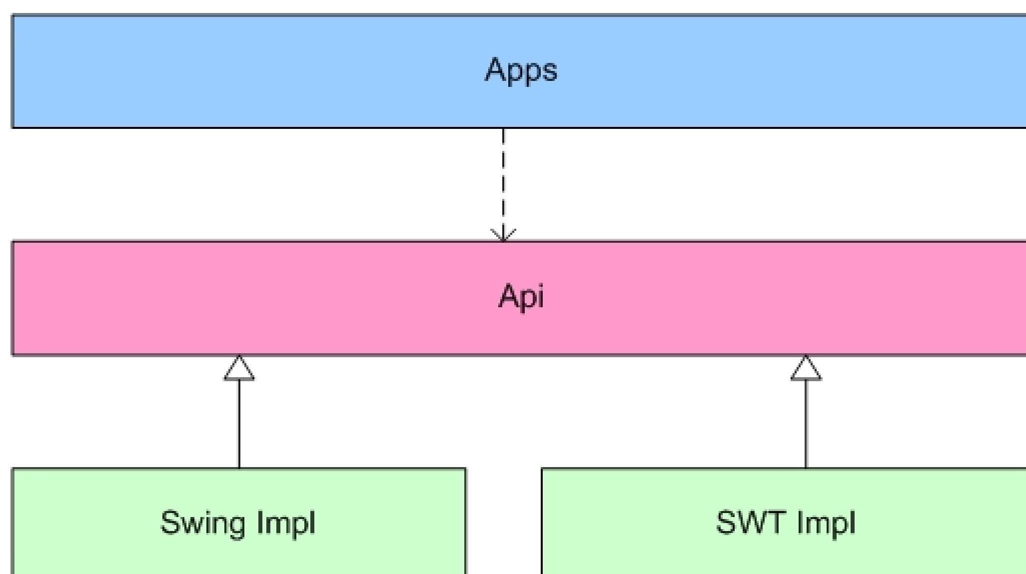

Abbildung 5.1: Beispiel einer einfachen Architektur für Single Sourcing

Abbildung 5.1 zeigt, wie eine Architektur für Single Sourcing aussehen könnte. Diese ist aufgeteilt in die grün dargestellten, konkreten Implementierungen der API „Swing Impl“ und „SWT Impl“, der API selbst (In der Abbildung in der Farbe Pink) und den „Apps“ (Anwendungen die diese API verwenden. In der Abbildung in blau zu sehen).

Das Problem bei dieser Architetur ist, dass wenn die API viele „convenience“-Methoden[1] enthält, diese in jeder Implementierung implementiert werden müssen und ein hoher Grad an Code Duplizierung entsteht. Weiterhin führen Änderungen in der API dazu, dass jede Implementierung angepasst werden muss. Aus diesem Grund ist der nächste logische Schritt die Architektur zu erweitern. Abbildung 5.2 zeigt dies:

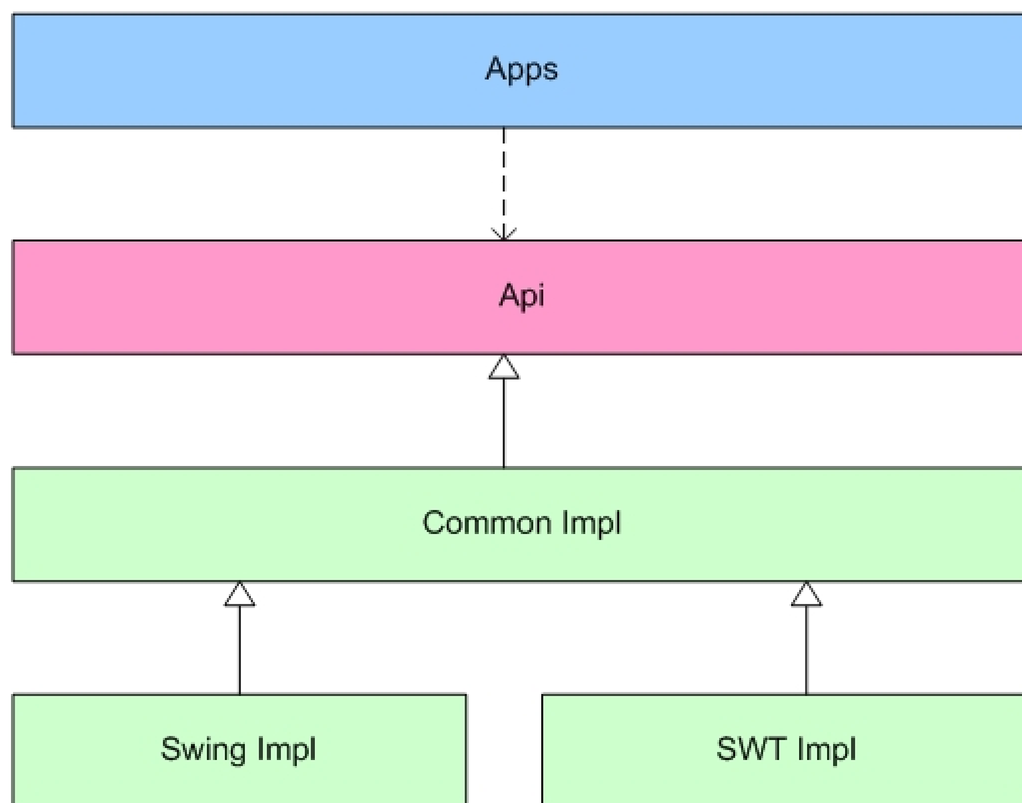

Abbildung 5.2: Die erweiterte Architektur

Durch das Erweitern der Architektur um „Common Impl“ als gemeinsame Basis für die API-Implementierungen, sinkt der Aufwand bei Änderungen um ein Vielfaches. Ein Nachteil ist jedoch, dass „Common Impl“ keine fest definierte Schnittstelle ist. Das bedeutet, sie enthält konkrete oder abstrakte Klassen. Dies führt dazu, dass Implementierungsdetails (Wissen über diese Klassen) notwendig sind um eine Erweiterung zu erstellen. Weiterhin führt eine Änderung dieser Klassen auch automatisch zu einem anderen Verhalten der Implementierungen, die diese Klassen verwenden. Aus diesem Grund erweitert Jo Widgets diese Architektur um eine gemeinsame Schnittstelle.

[1] „convenience“-Methoden sind Methoden, die das Programmieren erleichtern. Ein Beispiel ist die Methode „getChildren“ bei einem Container.

5.2 Die Architektur

Die Abbildung 5.3[2] zeigt die Jo Widgets Architektur. Abgebildet sind nur die zum Verständnis wichtigen Teile, sie ist daher nicht vollständig.

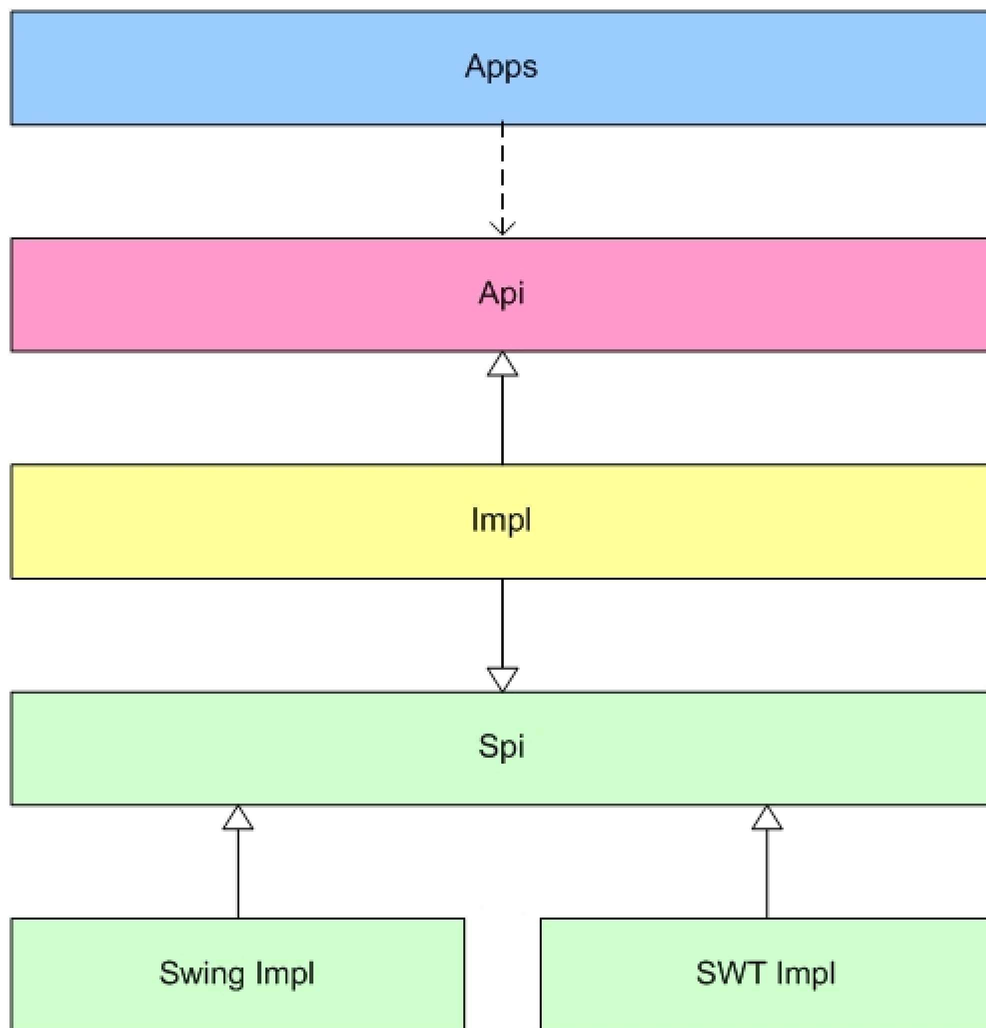

Abbildung 5.3: Die Architektur von Jo Widgets

[2] In Anlehnung an: [Gro11b]pic/jowidgets_modules_no_workbench.gif

5.2.1 Api

Die API ist der Teil des Frameworks der von Anwendungen verwendet wird und erweitert werden kann. Hierfür bietet die API viele „convenience“-Methoden an. Ein Beispiel ist die Methode „getText()“ auf einem Label oder „getChildren“ auf einem Container. Die API selbst besteht hauptsächlich aus Interfaces. Den Hauptteil bilden die Interfaces für Widgets, Setups und Descriptoren. Diese werden im Abschnitt 5.3 „Wie ein Widget erstellt wird“ weiter unten beschrieben.

5.2.2 Spi

Die SPI[3] ist die gemeinsame Schnittstelle von Jo Widgets, die es ermöglicht verschiedene UI-Technologien zu verwenden. Die SPI enthält keine „convenience“-Methoden wie die API, sondern nur grundlegende Funktionen. D.h. am Beispiel eines Labels, dass eine Methode wie „setText()“ exisiert, aber das Gegenstück „getText()“ nicht in der SPI zu finden ist. Die Methode „getText()“ wird als „convenience“-Methode angesehen, da der Text bereits durch den Aufruf von „setText“ gesetzt wird und keine weitere Schritte notwendig sind um das Widget wie gewünscht zu erstellen. Somit kann gesagt werden, dass die SPI die Low-Level-Funktionalitäten von Jo Widgets bereitstellt.

5.2.3 Impl

Die „impl“(in der Abbildung gelb dargestellt) ist die Implementierung von Jo Widgets. Diese verwendet die SPI (Low-Level) zur Implementierung der API (High-Level).

5.2.4 Swt/Swing Impl

Eine Implementierung der SPI nennt man Service Provider. Um z.B. das SPI-Interface eines Buttons zu implementieren, wird innerhalb des Service Providers für die Swing-Implementierung ein JButton verwendet. An diesen werden Methodenaufrufe wie „setText()“ delegiert. Gibt es in der Ziel-UI-Technologie kein vergleichbares Widget, muss dieses mit den Möglichkeiten der UI-Technologie nachgebaut werden.

5.3 Wie ein Widget erstellt wird

Beim Erzeugen eines Widgets[4] werden mehrere Schritte durchlaufen (s. Abbildung 5.4[5]). In den nachfolgenden Abschnitten werden die einzelnen Elemente der Darstellung erläutert.

[3] SPI ≙ Service Provider Interface

[4] Ein Widget ist ein GUI-Element, wie z.B. ein Button oder ein Label.

[5] In Anlehnung an [Gro11a]

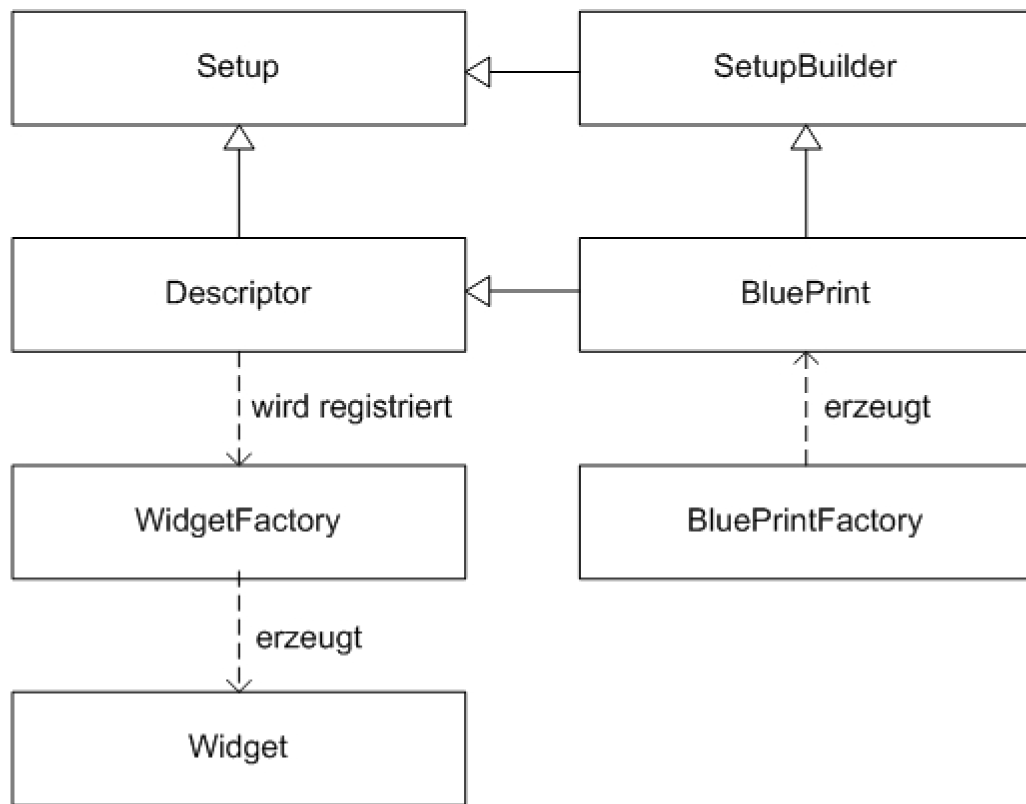

Abbildung 5.4: Wie ein Widget erstellt wird

5.3.1 Die „BluePrintFactory“

Mithilfe der „BluePrintFactory“ werden „Blueprint“s erzeugt, die für das Erstellen von Widgets notwendig sind. Ein „Blueprint“ (dt. Entwurf oder Plan) ist abgeleitet von einem „SetupBuilder“. In einem „SetupBuilder“ können über Set-Methoden Properties für das Widget festgelegt werden. Manche Properties, z.B. der Typ eines „FileChooserDialog“s, können nur hier gesetzt werden und nicht auf dem Widget selbst (diese Properties sind dadurch „immutable“[6]).

Um das Setzen der Properties zu erleichtern und die Lesbarkeit des Quellcodes zu erhöhen, sind die SetupBuilder nach dem Builder Pattern[7] aufgebaut. Dadurch ist es möglich, in einer einzelnen Codezeile mehrere Properties auf einem „Blueprint“ zu setzen. Beispiel (Listing 5.1) zeigt das Erstellen eines „Blueprint“s in Jo Widgets mit der „BluePrintFactory“.

```
1 IBluePrintFactory factory = Toolkit.getBluePrintFactory
    ();
2 IFrameBluePrint bpFrame = factory.frame();
3 bpFrame.setTitle("title").setVisible(true);
```

Listing 5.1: Beispiel erzeugen eines Blueprints mit der BlueprintFactory

[6] ≙ unveränderlich/unveränderbar

[7] Das Builder Pattern ist ein Entwurfsmuster, das die Erzeugen von Objekten mit viele optionale Konstruktor-Parametern vereinfacht. Für weitere Informationen, siehe http://javapapers.com/design-patterns/builder-pattern/

Über die Klasse „Toolkit“ kann die Default-Implementierung der „BluePrintFactory“ von Jo Widgets verwendet werden. In Zeile 2 wird ein „FrameBluePrint“ erstellt. Anschließend werden in Zeile 3 auf diesem „BluePrint“ die Properties (hier der Titel des Frames und die Sichtbarkeit) gesetzt.

5.3.2 Die „WidgetFactory“

Die „WidgetFactory“ ist der zentrale Ort an dem Factories für das Erzeugen von Widgets registriert werden. Anhand des „Descriptors“ wird entschieden, an welche Factory das Erzeugen delegiert wird. Der „Descriptor“ erfüllt noch einen weiteren Zweck: Er wird für das Initialisieren des Widgets verwendet. „Descriptoren“ sind deshalb von „Setups“ abgeleitet. Durch dieses verfügen sie über Get-Methoden für die Properties des Widgets.

Das Listing 5.2 zeigt wie ein Widget erstellt wird. Hierbei wird das Beispiel 5.1 erweitert.

```
1 IBluePrintFactory factory = Toolkit.getBluePrintFactory
      ();
2 IFrameBluePrint bpFrame = factory.frame();
3 bpFrame.setTitle("title").setVisible(true);
4 // Das Frame wird erstellt.
5 // Über den Descriptor wird eine passende Factory
      ausgewählt.
6 IFrame rootFrame = Toolkit.createRootFrame(bpFrame);
```

Listing 5.2: Beispiel erzeugen eines Widgets durch Übergabe eines Descriptors

Kapitel 6

Entwurf

Dieses Kapitel geht auf mögliche Lösungen für die Realisierung der Test-Bibliothek ein. Wie bereits im Kapitel 3, „Anforderungen“, werden die beiden Bereiche der Test-Bibliothek getrennt aufgelistet.

6.1 Test-Architektur

6.1.1 Automatisiertes Ablaufen der Tests

Sollen die mit dem Test-Tool aufgezeichneten Tests automatisiert ablaufbar sein, müssen sie auf Systemen ausführbar sein, die über keine Unterstützung für Ein- und Ausgabegeräte verfügen (s. Kapitel 2.5). Deshalb soll neben den Service Providern für Swing und SWT, ein Dummy-Service-Provider erstellt werden. In diesem werden die Widgets nach dem Vorbild der anderen Service Provider nachgebaut. Diese haben die gleichen Funktionalitäten wie „echte“ Widgets (≙ Widgets die auf eine UI-Technologie basieren), werden aber nicht grafisch dargestellt. Im GUI-Framework Jo Widgets wurde bereits mit einem Dummy-Service-Provider begonnen, dieser muss noch erweitert werden.

6.1.2 Erweiterung der Widgets um HCI-Aspekte

Die Widgets sollen um HCI[1]-Aspekte erweitert werden. HCI-Aspekte sind die Nutzerinteraktionen mit der GUI. Beispiele sind „einen Button drücken“, „ein Label anschauen“ oder „einen Text lesen“. Dafür sollen die Widgets über Methoden verfügen, die diese Aktionen simulieren. Bei einem Button wäre dies beispielsweise eine Methode mit dem Namen push(), für den HCI-Aspekt „klicken bzw. drücken“.

Wird in einer GUI ein Button durch den Nutzer gedrückt, werden Events erzeugt. Diese beinhalten Informationen über die ausgeführte Aktion und wie diese ausgelöst wurde (z.B. durch drücken der linken Maustaste). Die Events durchlaufen anschließend ein komplexes System, das von Faktoren wie dem verwendeten Look&Feel und

[1] HCI ≙ Human Computer Interaction.

der Komponente selbst beeinflusst wird. Deshalb muss bei den Methoden für das Simulieren der Nutzeraktionen beachtet werden, dass die Events möglichst gleich sind, wie die durch den Nutzer erzeugten.

6.2 Test-Tool

Wie der Vergleich in Kapitel 4, „Techniken für die Testautomatisierung“, ergeben hat, wird das Test-Tool die Capture&Replay-Technik verwenden. Um diese Technik umzusetzen, müssen folgende Probleme gelöst werden:

- Widgets müssen identifiziert werden können (z.B. anhand einer ID).
- Es muss einen Mechanismus geben, um Widgets innerhalb der GUI finden zu können. Hierzu muss das Test-Tool die GUI Komponenten „kennen“.

Bei der Capture&Replay-Technik gibt es mehrere Möglichkeiten für das Aufzeichnen der Nutzeraktionen. Ein Ansatz könnte z.B. sein, das Widget, mit dem der Nutzer interagiert, mit all seinen Properties und der durchgeführten Aktion (z.B. ein Mausklick) abzuspeichern. Beim erneuten Ausführen der Anwendung könnte das Test-Tool das Widget anhand seiner Properties wiederfinden und die Aktion abspielen.

Dies würde allerdings dazu führen, dass die Tests, die so erstellt werden, sehr anfällig gegenüber Änderungen der Anwendung sind. Die Tests sollen so robust wie möglich sein, was diese Methode nicht erfüllt. Deshalb ist die Idee, nur die ID für das Widget und die durchgeführte Aktion aufzuzeichnen und nicht das Widget selbst. Dadurch sind die Tests nicht mehr so stark von GUI-Änderungen abhängig. Der Nachteil ist, dass das Widget verändert oder ausgetauscht werden kann. Die Korrektheit des Tests kann in diesem Fall nicht gewährleistet werden. Dies wird zugunsten der höheren Test-Robustheit in Kauf genommen.

Um die Tests auf das wesentliche zu beschränken und übersichtlich zu halten, werden keine Mausbewegungen aufgezeichnet. Um Mausbewegungen beim grafischen Abspielen der Tests zu simulieren, soll die Maus direkt von der aktuellen Cursor-Position zum Ziel-Widget bewegt werden. Dies genügt, um die vom Nutzer durchgeführten Aktionen nachvollziehen zu können.

6.2.1 Die Test-Tool-GUI

Die GUI des Test-Tools soll möglichst einfach und intuitiv sein:

- Einzelne Test-Schritte sollen in einer Tabelle dargestellt werden.
- Eine Toolbar mit Funktionen für das Aufzeichnen und Abspielen von Tests.
- Funktionen für das Editieren der Tabelle, z.B. Löschen einzelner Test-Schritte (s. Kapitel 3 „Anforderungen“).

- Ein Menü mit Unterpunkten für das Laden/Speichern eines Tests und der Möglichkeit, das Test-Tool zu beenden.

Die Abbildung 6.1 zeigt, wie die GUI aussehen könnte.

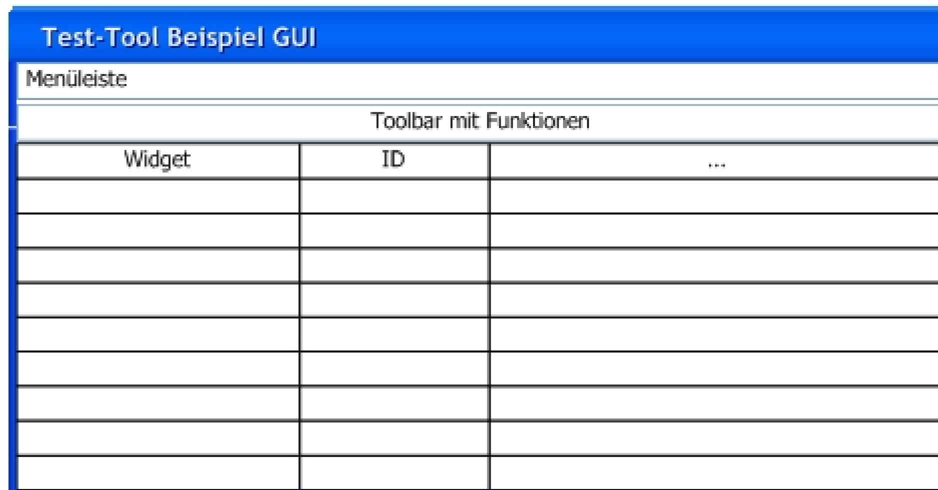

Abbildung 6.1: Beispiel einer möglichen Test-Tool-GUI

6.2.2 Das Test-Tool einbinden

Ziel ist es, die Verwendung des Test-Tools möglichst einfach zu gestalten. D.h. zur Verwendung sollen keine Änderungen im Quellcode der zu testenden Anwendungen notwendig sein. In Jo Widgets wird Apache Maven[2] zur Projektverwaltung verwendet. Nach Möglichkeit soll versucht werden, dass nur eine Abhängigkeit zum Test-Tool in die Projekt POM[3] der zu testenden Anwendung, eingefügt wird.

6.2.3 Ein Widget identifizieren

Eine ID soll verwendet werden, um die GUI-Elemente identifizieren zu können. Um die Objekte nach einem Neustart der Anwendung wiederzufinden ist es wichtig, dass die ID nicht zufallsgeneriert wird, sondern nach einem festgelegten Schema erzeugt oder gesetzt wird. Damit erhält ein bestimmtes Objekt stets die gleiche ID und kann wiedergefunden werden.

[2] „Apache Maven is a software project management and comprehension tool.“ [Tea11b]

[3] POM ≙ Projekt Object Model. „...is an XML file that contains information about the project and configuration details used by Maven to build the project.“ [Tea11a]

6.2.4 Wie das Test-Tool die GUI „kennen lernt“

Um Widgets wiederfinden zu könnten, muss das Test-Tool die Komponenten der GUI kennen. Hierfür gibt es verschiedene Lösungsansätze. Eine Möglichkeit wäre das Speichern aller GUI-Komponenten an einer zentralen Stelle z.B. in Form einer Map. Die ID könnte als „Key“ verwendet werden und die Komponente als „Value“. Dadurch können Widgets über den „Key“ schnell wiedergefunden werden. Viele Test-Tools verwenden ein ähnliches Prinzip.

Diese Methode hat den Nachteil, dass nicht unbedingt alle Widgets von Beginn an verfügbar sein müssen. In Enterprise-Applicationen mit vielen GUI-Elementen ist es oft sinnvoll, ein Teil der Widgets durch Lazy Initialization[4] zu erzeugen. Dadurch entsteht ein Performance-Vorteil, da nur die Widgets erzeugt werden, die sichtbar sind und benötigt werden. Um alle Widgets zu speichern, müsste die Anwendung gestartet und alle Komponenten einmal aktiviert werden. Nur dadurch könnte sichergestellt werden, dass das Test-Tool sämtliche Widgets „kennt“.

Aus diesem Grund soll für das Test-Tool ein anderes Prinzip verwendet werden. Das Test-Tool soll sich nur die Widgets „merken“ die bereits erzeugt wurden. Hier bietet Jo Widgets einen großen Vorteil, da alle Widgets an einer zentralen Stelle, der „WidgetFactory“, erstellt werden (s. Kapitel 5). Diese bietet die Möglichkeit, einen „WidgetFactoryListener“ zu registrieren. Dieser wird informiert sobald ein neues Widget erzeugt wurde. Die Idee ist, dass das Test-Tool durch diesen Mechanismus die GUI-Komponenten „kennen lernt“. Werden durch Aktionen, wie die Selektion verschiedener Tabs in einem Tabfolder, neue Widgets erstellt, wird das Test-Tool darüber benachrichtigt und kann die „gelernten“ Widgets an einer zentralen Stelle abspeichern. Die Abbildung 6.2 zeigt wie das Test-Tool über neu erstellte Widgets informiert wird.

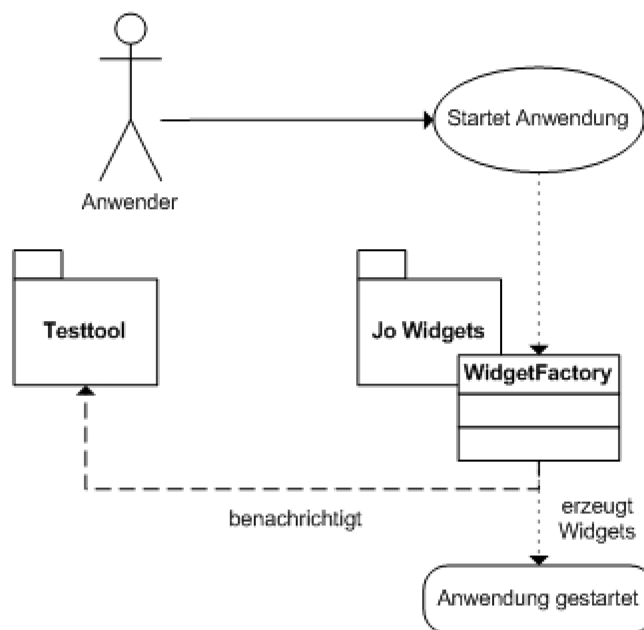

Abbildung 6.2: Wie das Test-Tool die GUI-Komponenten „kennen lernt“

[4] Lazy Initialization bedeutet, dass ein Widget erst erstellt wird, wenn es benötigt wird.

Kapitel 7

Realisierung

7.1 Entwicklung der Test-Architektur

7.1.1 Die Projektstruktur

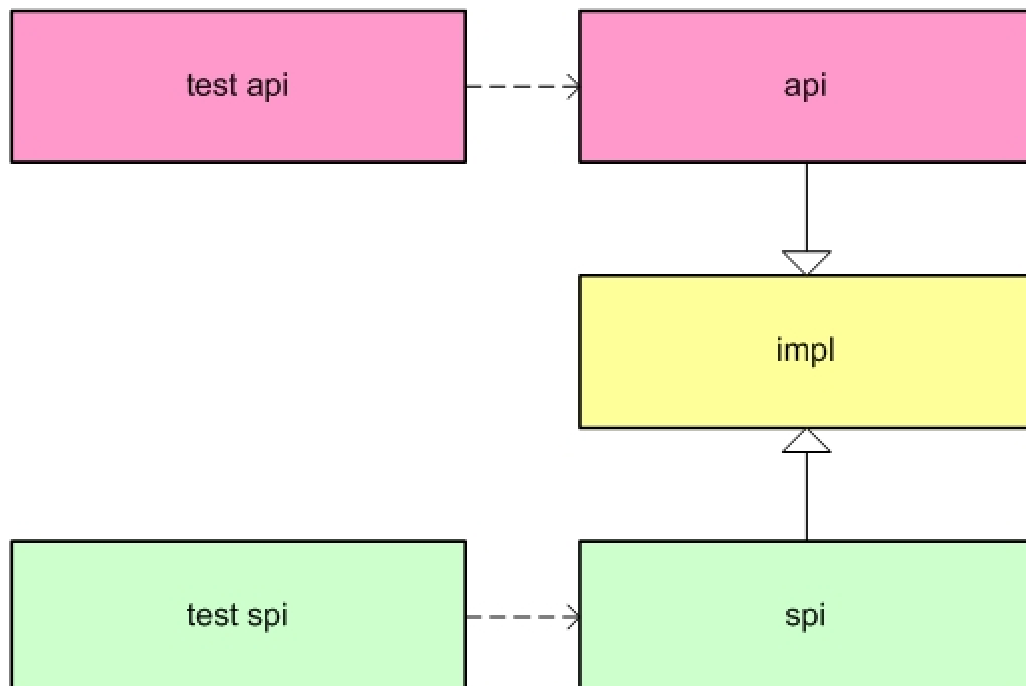

Abbildung 7.1: Die Projektstruktur der Test-Architektur

Die Abbildung 7.1 zeigt, wie die Test-Architektur in Jo Widgets integriert ist und diese erweitert. Die Darstellung der „Jo Widgets“-Architektur wurde dabei auf das Wesentliche reduziert. Die Bestandteile der Test-Architektur sind jeweils abhängig von ihren Gegenstücken in der „Jo Widgets“-Architektur. „test api“ hat z.B. eine Abhängigkeit zu „api“.

In „test api“ und „test spi“ sind in abgeleiteter Form die Interfaces von Widgets und Descriptoren (s. Kapitel 5.3) aus „api“ und „spi“ zu finden. Die Bezeichnungen der Interfaces wurden mit dem Suffix „Ui“[1] ergänzt. Zum Beispiel wird das von „IButton“ abgeleitete Interface „IButtonUi“ genannt.

Die „Ui“-Interfaces beinhalten Testfunktionalitäten, wie z.B. die Methode „push()“.

7.1.2 Umsetzung der HCI-Aspekte

Das Listing 7.1 zeigt, wie der HCI Aspekt „einen Button drücken“ im SWT Service Provider umgesetzt wurde.

```
1  @Override
2  public void push() {
3    Event e = new Event();
4
5    // Maus zum Button bewegen
6    e.type = SWT.MouseMove;
7    Point widgetPos = getUiReference().toDisplay(
8     getUiReference().getLocation().x,
9     getUiReference().getLocation().y);
10   e.x = widgetPos.x;
11   e.y = widgetPos.y;
12   Display.getCurrent().post(e);
13
14   // Button mit linker Maustaste drücken
15   e.type = SWT.MouseDown;
16   e.button = 1;
17   Display.getCurrent().post(e);
18
19   // Maustaste loslassen
20   e.type = SWT.MouseUp;
21   Display.getCurrent().post(e);
22 }
```

Listing 7.1: Implementierung der push()-Methode

Über die Klasse „Display“ können in SWT Events „geworfen“ werden. Dazu wird die Methode post(...) verwendet. Diese bekommt das gewünschte Event übergeben.

Um einen Button betätigen zu können, muss sichergestellt sein, dass die Maus sich innerhalb des Buttons befindet. Deshalb ist das erste Event vom Typ „SWT.MouseMove“ (s. Zeile 6). Dies bewirkt eine Mausbewegung an die zuvor berechneten X- und Y-Koordinaten des Buttons (s. Zeile 7-12). Durch das Event vom Typ „SWT.MouseDown“ wird das Drücken einer Maustaste definiert (s. Zeile 15). Die konkrete Maustaste muss separat angegeben werden. Im Beispiel, die linke Maustaste, zu sehen in Zeile 16. Damit das Event abgefeuert wird, muss das „losgelassen“ der Maustaste simuliert werden (s. Zeile 20 und 21).

Das Beispiel zeigt, dass Events auf einem sehr niedrigen Level erzeugt werden. Dadurch werden Nutzeraktionen möglichst genau nachgebildet. Das Bewegen der Maus, zum Widget dient als Hilfe für das Test-Tool und stellt sicher, dass sich die Maus im Bereich des Buttons befindet, wenn dieser gedrückt werden soll. Eine andere

[1] ≙ User Interface.

Möglichkeit wäre es, zu versuchen für das Widget den Focus zu erhalten und das Drücken der „Return“ Taste zu simulieren. Es kann jedoch nicht sichergestellt werden, dass der Focus immer erhalten werden kann, deshalb wird diese Methode nicht verwendet.

Bei anderen Service Providern wird ähnlich vorgegangen wie in diesem Beispiel. Im Dummy Service Provider wird innerhalb der push()-Methode direkt die Logik des zuständigen „ActionListener“ aufgerufen. Momentan können folgenden Aktionen simuliert werden:

- Das Drücken eines Buttons
- Das Selektieren, Auf - und Zuklappen von Tree-Elementen
- Das Selektieren von Tabs in einem Tabfolder
- Das Öffnen und Schließen von Fenstern und Dialogen

7.1.3 Erstellen eines GUI-Unit-Tests

Eine Anforderung an die Test-Architektur ist das Erstellen von Tests ohne das Test-Tool zu verwenden. Hierfür wurden nach dem Vorbild, der „WidgetFactory“ und „BlueprintFactory“, Factories für das Erzeugen von Test-Widgets erstellt. Ein Test-Widget ist z.B. ein Button der das Interface „IButtonUi“ implementiert. Über die Test-Widgets können auf die Testfunktionalitäten zugegriffen werden. Das Listing 7.2 zeigt einen so erstellen GUI-Unit-Test.

```
1  public class GUIUnitTest {
2    private static final ITestBluePrintFactory BPF =
         TestToolkit.getTestBluePrintFactory();
3    private boolean invoked;
4
5    @Test
6    public void testPushButton() {
7      invoked = false;
8
9      // Erstellen des Hauptfensters
10     IFrameUi frame = TestToolkit.createRootFrame(BPF.
          frame());
11     IButtonUi button = frame.add(BPF.button());
12     button.setText("Klick");
13     button.addActionListener(new IActionListener() {
14
15       @Override
16       public void actionPerformed() {
17         invoked = true;
18       }
19     });
20     // Den Button drücken
21     button.push();
22     Assert.assertTrue(invoked);
23   }
24 }
```

Listing 7.2: Beispiel eines GUI-Unit-Tests

Dieses Listing zeigt, wie ohne das Test-Tool ein GUI-Unit-Tests erstellt werden kann. In diesem Test wird getestet, ob das simulierte Drücken eines Buttons funktioniert. Die Klasse „TestToolkit“ ist eine Hilfsklasse, über die die „TestBluePrintFactory“ erhalten werden kann. Diese erzeugt die Test Widgets.

Durch das Einfügen des Dummy Service Providers in die Projekt POM kann dieser Test automatisiert auf CI Servern wie dem Hudson ablaufen.

7.2 Entwicklung des Test-Tools

In diesem Abschnitt wird auf die Probleme und Lösungen bei der Umsetzung des Test-Tools eingegangen.

7.2.1 Einbinden und Starten des Test-Tools

Das Starten und Einbinden des Test-Tools soll möglichst einfach sein. Wie schon im Kapitel „Entwurf“ beschrieben, wäre es ideal, wenn nur eine Abhängigkeit zum Test-Tool in das zu testende Projekt eingefügt werden müsste. Dies kann aktuell nicht umgesetzt werden. Grund ist, dass sichergestellt werden muss, dass das Test-Tool vor der eigentlichen Anwendung gestartet wird.

Um die Verwendung des Test-Tool dennoch einfach zu gestalten, gibt es die Klasse „TestToolRunner“. Diese bietet Methoden für das Ausführen des Test-Tools an:

- „run()“ startet das Test-Tool mit der Test-Tool-GUI. In diesem „Modus“ können Tests aufgezeichnet und gespeichert werden. Wird das Test-Tool in diesem „Modus“ verwendet, muss der „TestToolRunner“ in einer Klasse mit einer statischen Methode gestartet werden (überlicherweise die Main-Methode). Diese Klasse dient als Einstiegspunkt für die Java Anwendung. Das Listing 7.3 zeigt dies.

```
1 public static void main(final String[] args){
2   new TestToolRunner(new DemoApplication()).run();
3 }
```

Listing 7.3: Das Test-Tool mit GUI ausführen

- „runAsUnitTest(...)“ startet das Test-Tool ohne GUI. Als Parameter wird der Methode das Verzeichnis der Tests oder direkt eine Testdatei angegeben. Dieser „Modus“ dient dazu, die Tests automatisiert als JUnit-Tests ablaufen zu lassen. Wird das Test-Tool in diesem „Modus“ verwendet, muss der „TestToolRunner“ innerhalb einer JUnit-Test-Methode aufgerufen werden. Das Listing 7.4 zeigt dies.

```
1 @Test
2 public void testApplicationButtonTest() {
3   new TestToolRunner(new DemoApplication()).
        runAsUnitTest(sampleFileName);
4 }
```

Listing 7.4: Das Test-Tool ohne GUI starten

In Jo Widgets implementieren Anwendungen entweder das Interface IApplication[2] oder IWorkbench[3]. Hierfür bietet der „TestToolRunner“ entsprechend parametrisierte Konstruktoren an. Der „TestToolRunner“ sorgt auch dafür, dass sich das Test-Tool nicht selbst aufzeichnen kann. Hierzu wird die GUI des Test-Tools vor dem registrieren des „WidgetFactoryListeners“ gestartet. Das Test-Tool „lernt“ dadurch nicht die eigenen GUI Komponenten.

7.2.2 Die Benutzeroberfläche des Test-Tools

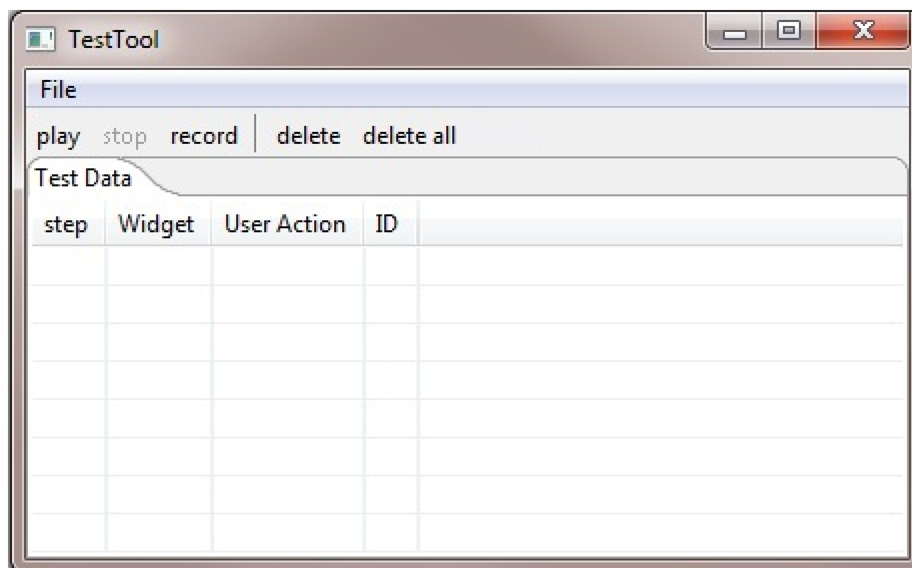

Abbildung 7.2: Die Test-Tool-GUI

In der Abbildung 7.2.2 ist die Benutzeroberfläche des Test-Tools zu sehen. Den Hauptteil der GUI macht die Tabelle für die Darstellung der einzelnen Test-Schritte aus. Ein Test-Schritt besteht aus dem Typ des Widgets, die Aktion des Nutzers und der ID. Die erste Spalte „step“ dient dazu, die Schritte zu nummerieren.

In der Toolbar sind typische Funktionen für Player/Recorder, wie „play“, „stop“ und „record“ zu finden. Diese dienen dazu, den Test aufzuzeichnen bzw. abzuspielen.

Der Menüpunkt „File“ bietet Menüpunkte für das Speichen/Laden eines Tests-Skripts als XML-Datei.

[2]Standard Java Anwendung implementieren dieses Interface.

[3]Anwendung, die den Workbench von Jo Widgets verwenden wollen, implementieren dieses Interface. Der Workbench ist vergleichbar mit der Eclipse IDE.

7.2.3 Generierung der Widget-ID

Für die Erzeugung der Widget-ID wird die Position des Widgets in seiner Parent-Child-Hierarchie verwendet. Diese stellt die Vater-Kind-Beziehung des Widgets dar. Die Abbildung 7.2.3 zeigt eine Beispiels Hierarchie.

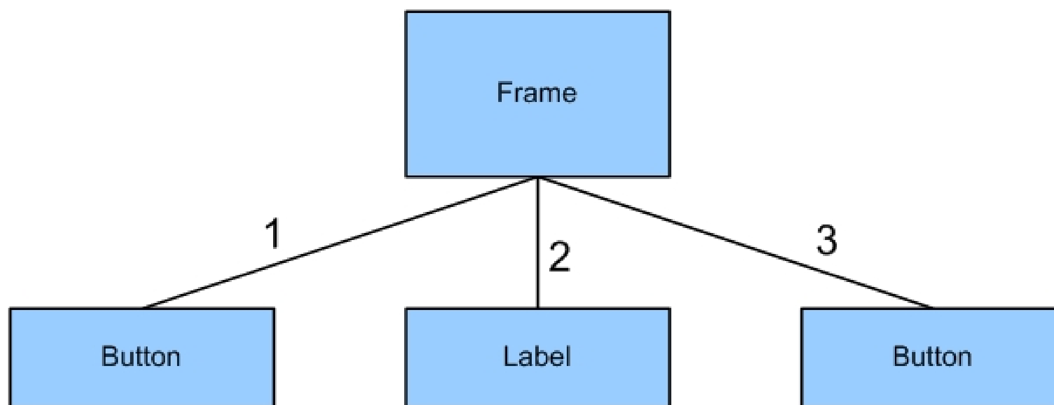

Abbildung 7.3: Beispiel einer Parent-Child-Hierarchie

Das Beispiel zeigt ein Frame, mit zwei Buttons und einem Label als Kind-Widgets. Die Zahlen geben die Stelle an, an der sich das Kind befindet.

Um die ID eindeutig und lesbar zu gestalten wird der Klassenname des Widgets und eine oder mehrere Properties hinzugefügt.

Eine Beispiels ID ist in dem Listing 7.5 zu sehen.

```
1 FrameImpl:MeinFenster/1_ButtonImpl:KlickMich
```

Listing 7.5: Beispiel einer Widget-ID

Diese ID ist eindeutig und aussagekräftig. Ein Doppelpunkt dient als Trennzeichen zwischen der Klassennamen und der zugehörenden Properties. Mit einem Trennstrich wird das nächste Element angegeben und wenn möglich dessen Positon in der Hierarchie.

Properties wie die Positon eines Widgets, sollten nicht verwendet werden. Bereits ein Verändern der Position der Anwendung würde dazu führen, dass das Widget eventuell nicht wiedergefunden werden könnte.

Ein Problem der ID ist, dass sich die Position der Widgets in der Hierarchie, z.B. durch Drag&Drop des Benutzers, ändern kann. Deshalb ist die ID nur für den aktuellen Status der GUI gültig. Wird dieser verändert, muss die ID neu generiert werden. Dies ist jedoch nur notwendig, wenn nach dem Widget gesucht wird.

7.2.4 Ein Widget wiederfinden

Wie in Abschnitt 7.2.1 beschrieben wird das Test-Tool über neu erstellte Widgets informiert. Die Referenzen auf die Widgets werden intern im Test-Tool in der „WidgetRegistry“ gespeichert. Die „WidgetRegistry“ ist ein „HashSet“ aus Widgets, die das Interface „IWidgetCommon“ implementieren. Damit ist sie kompatibel zu allen Widgets in Jo Widgets.

Um sicherzustellen, dass im Test-Tool stets mit der gleichen Instanz der „WidgetRegistry“ gearbeitet wird, ist sie nach dem Singleton Pattern[4] aufgebaut. Da sie von mehreren Threads verwendet werden kann (z.B. im Replay-Modus), sind die Zugriffsmethoden synchronisiert.

Das Listing 7.6 zeigt die ID eines gesuchten Widgets.

```
1 FrameImpl:MeinFenster/3_ButtonImpl:KlickMich
```

Listing 7.6: Die ID des gesuchten Widgets

Um ein Widget zu finden wird zu Beginn der Suche das Ende der ID untersucht. In dem Beispiel „ButtonImpl:KlickMich“. Gesucht ist eine Instanz von „ButtonImpl“ mit der Property „KlickMich“. Hierzu werden von der „WidgetRegistry“ alle Instanzen von „ButtonImpl“ geladen und in einer Liste gespeichert. Nun werden für die Elemente der Liste nacheinander die IDs erzeugt und mit der gesuchten ID verglichen.

Eine andere Möglichkeit wäre gewesen, die ID von vorne nach hinten zu durchsuchen. Die erzeugten IDs können jedoch unter Umständen sehr lange sein. Dies würde bedeuten, dass sehr viele Schritte, in denen Elemente aus der „WidgetRegistry“ geladen und untersucht werden, notwendig wären. Dadurch wäre der Aufwand größer, als mit der Umgesetzten Methode.

An dieser Stelle könnte eingewendet werden, dass eine Map besser geeignet wäre, da direkt die ID als Key erfragt werden kann. Es wurde aber bewusst keine Map gewählt, da innerhalb der Map ein Widget mit verschiedenen IDs gespeichert werden kann. D.h. es wäre möglich durch eine aufgrund des geänderten GUI Status, ungültig gewordene ID auf ein Widget zuzugreifen. Dies wird durch ein Set verhindert, da dort nur die Instanz des Widgets, nicht aber die ID gespeichert wird. Ein weiterer Vorteil ist, dass das Test-Tool bei dem „kennen lernen“ der GUI keine IDs generieren muss und die „WidgetRegistry“ nicht größer werden kann als die max. Anzahl der vorhandene GUI Elemente.

7.2.5 Aufzeichnen einer Nutzeraktion

Um Nutzeraktionen aufzuzeichnen, registriert das Test-Tool an den Widgets verschiedene „Listener“. Bei einem Button wird z.B. ein „ActionListener“ registriert. Erhält dieser ein Event, speichert das Test-Tool die ID des Widgets, den Typ und

[4] Das Singleton-Entwurfsmuster stellt sicher, dass von einer Klasse nur eine Instanz erzeugt werden kann. Für weitere Infos siehe http://javapapers.com/design-patterns/singleton-pattern/

die durchgeführte Aktion, in einem „TestDataObject“ ab. Das „TestDataObject“ stellt eine Nutzeraktion dar.

Anhand des registrierten „Listener“ wird entschieden, welche Aktion vom Nutzer ausgeführt wurde. Im Fall des „ActionListener“, die Aktion „UserAction.Click“. Dies entspricht einem Klick mit der linken Maustaste.

Nutzeraktionen wie die Beschriftung des Buttons „lesen“ oder die Hintergrundfarbe eines Widgets „anschauen“ können damit nicht ermöglicht werden. Auf diesen Punkt wird im Kapitel 8, „Erweiterungsmöglichkeiten/Ausblick“, eingegangen.

Schwierigkeiten bei der Aufzeichnung bereiteten modale Dialoge. Diese verfügen über einen eigenen Event-Loop[5], der gestartet wird, sobald ein Dialog sichtbar ist. Dadurch kann es passieren, dass Events verspätet aufgezeichnet werden. Das liegt daran, dass der Event-Loop der Hauptanwendung blockiert wird bis der Dialog beendet ist, und evtl. die Events noch nicht alle verarbeitet hat.

Ein Beispiel verdeutlicht dieses Problem: Ein Button, der einen Dialog öffnet, soll aufgezeichnet werden. Das Erstellen und Öffnen des Dialogs, geschieht über einen registrierten “ActionListener“. Für die Aufzeichnung der Aktionen registriert das Test-Tool einen „ActionListener“ am Button und einen „WindowListener“ am Dialog. Der Button verfügt damit über einen „ActionListener“, der den Dialog erzeugt, und einen für die Aufzeichnung. Wird nun der Aufzeichnungsmodus im Test-Tool gestartet und der Button gedrückt, können verschiedene Situationen entstehen:

- Das Test-Tool zeichnet zuerst das Drücken des Buttons auf und anschließend das Öffnen und Schließen des Dialogs. Dies ist die gewünschte Reihenfolge.
- Der „ActionListener“ für das Öffnen des Dialogs wird vor dem des Test-Tools benachrichtigt. Dadurch wird das Drücken des Buttons erst nach dem Schließen des Dialogs aufgezeichnet und die Reihenfolge der Nutzeraktionen verfälscht.

Um dieses Problem zu lösen, wurden spezielle „Listener“ innerhalb der Test-Architektur erstellt. Diese werden vor den sonstigen „Listener“ eines Widgets benachrichtigt.

Eine weitere Schwierigkeit stellen Widgets dar die nicht über die „WidgetFactory“ erzeugt werden. Ein Beispiel ist ein „TreeNode“ oder ein „MenuItem“. Das Test-Tool kennt nur die Parents dieser Widgets, welche über die „WidgetFactory“ erzeugt wurden. Damit das Test-Tool auch die Kind-Objekte „kennt“, wurden „Listener“ der Test-Architektur hinzugefügt, die über jedes hinzugefügte oder entfernte Kind-Objekt benachrichtig werden.

7.2.6 Serialisierung des aufgezeichneten Materials

Die aufgezeichneten Tests werden als XML-Datei gespeichert. Hierfür wird das in der JRE enthaltene JAXB[6] verwendet. Dies wurde ausgewählt, da es einfach zu

[5] Der Event-Loop ist für die Verarbeitung der Events zuständig.
[6] ≙ Java Architecture for XML Binding.

verwenden ist und keine neuen Abhängigkeiten in das GUI-Framework Jo Widgets mit einbringt.

Die Abbildung 7.4 zeigt, wie ein durch das Test-Tool erstellter Test, in XML aussieht.

```
<userTestData>
    <TestData>
        <action>CLICK</action>
        <id>FrameImpl:test/3_ButtonImpl:test</id>
    </TestData>
    <TestData>
        <action>CLOSE</action>
        <id>FrameImpl:test/FrameImpl:test</id>
    </TestData>
    <TestData>
        <action>CLICK</action>
        <id>FrameImpl:test/3_ButtonImpl:test</id>
    </TestData>
    <TestData>
        <action>CLOSE</action>
        <id>FrameImpl:test/FrameImpl:test</id>
    </TestData>
</userTestData>
```

Abbildung 7.4: Das mit dem Test-Tool erstellte Skript.

Gespeichert werden die durchgeführte Aktion und die ID des Widgets. In diesem Beispielskript wird ein Test für das Öffnen und Schließen eines Dialogs mit Hilfe eines Buttons dargestellt.

Kapitel 8

Erweiterungen/Ausblick

Dieses Kapitel geht auf die geplanten Features des Test-Tools ein.

8.1 Weitere Nutzeraktionen

Um Nutzeraktionen wie „Lesen“ oder „Anschauen“ zu simulieren, ist das Erweitern der Test-Tool-GUI um Werkzeuge für „Drücken“, „Lesen“ und „Anschauen“ geplant. Je nach ausgewähltem Werkzeug, befindet sich das Test-Tool in einem entsprechenden Modus (z.B. Lesemodus bei Verwendung des „Lese-Werkzeugs“). Der Modus legt fest, welche Aktion im „Listener“ des betreffenden Widgets, beim Erhalt eines Events stattfinden soll.

8.2 Checkpoints

Um sinnvolle Tests erstellen zu können, genügt es nicht, nur die Aktionen des Nutzers aufzuzeichnen. Es ist wichtig Bedingungen für die Richtigkeit des Tests festlegen zu können. Hierfür sollen „Checkpoints“ im Test-Skript eingefügt werden können. Diese sollen sich auf die zuvor durchgeführte Nutzeraktion beziehen und Möglichkeiten bieten Bedingungen zu definieren. Ein Beispiel ist die Bedingung: „Das Fenster muss sichtbar sein“.
Eine Methode hierfür könnte folgende Signatur haben:

```
1 public void validate(String prop, boolean value);
```

Listing 8.1: Beispiels Signatur einer Checkpoint-Methode

Die Methode erhält die zu untersuchende Property (bzw. den Methodennamen des „Getters“[1] in String-Form) und den gewünschten Wert:

[1]Abkürzung für die Get-Methode einer Property.

```
1 validate("isVisible",true);
```

Listing 8.2: Beispiel für einen Checkpoint

Wird die Bedingung nicht erfüllt, gilt der Test als nicht bestanden und der Tester erhält eine entsprechende Fehlermeldung.

8.3 Mausanzeigeunterstützung

Das Bewegen der Maus kann aktuell noch nicht simuliert werden. Grund ist, dass hierfür ein UI-Technologie-spezifischer Code notwendig ist. Das Test-Tool soll aber unabhängig von der verwendeten Technologie sein.

Eine Möglichkeit wäre die Verwendung von JNI[2]. Dadurch wäre das Test-Tool jedoch nur unter Windows benutzbar. Eine andere Möglichkeit wäre eine Art Addon für das Simulieren von Mausbewegungen, das pro Service Provider implementiert werden müsste. Die Folge davon ist, dass ein neuer Service Provider ein Addon benötigt um eine Mausanzeigeunterstützung zu erhalten. Hier muss noch entschieden werden, welche Methode zum Einsatz kommen soll.

8.4 Eigene Properties festlegen

Momentan werden für die Generierung der Widget ID die Default-Property der Widgets verwendet. Diese ist z.B. der Text bei einem Button. Um Widget-IDs individuell gestalten zu können, soll es möglich sein, eigene Properties auszuwählen. Die Idee ist, dass wie bei den Checkpoints, der Name des „Getters“ der gewünschten Property übergeben wird.

```
1 createId("getTooltip",widget);
```

Listing 8.3: Beispiel für das Erzeugen einer ID mit Property

Das Verwenden der „Getter“-Bezeichnung bietet den Vorteil, dass innerhalb der ID Generierung unterschieden werden kann, welcher „Getter“ aufgerufen werden muss. Dies ist wichtig für das Wiederfinden des Widgets. Für diese Änderung muss die Persistierung der Testdaten als XML Datei um „Tags“ für die Methodenbezeichnung und des Wertes der Property erweitert werden.

[2] ≙ Java Native Interface.„Mit Hilfe von JNI können aus der JVM heraus plattformspezifische Funktionen verwendet werden.“[Ull05, Kapitel 24], besucht am 17.05.2011

Kapitel 9

Fazit

Ziel dieser Arbeit war das Entwickeln eines Prototyps einer Test-Bibliothek zum GUI-Framework Jo Widgets für automatisierte Tests. Hierzu wurden die verschiedenen Techniken für die Testautomatisierung untersucht und miteinander vergleichen. Dabei stellte sich heraus, das die Capture&Replay-Technik am besten für das Test-Tool geeignet ist. Diese unterstützt nicht nur das manuelle Testen indem Aktionen genau wiedergegeben werden, sondern ermöglicht es auch Tests aufzuzeichnen, die automatisiert durchgeführt werden können. Für die Erstellung von Tests, ohne die Verwendung des Test-Tools wurde die GUI Unit Test Technik ausgewählt. Mit dieser können funktionale Tests durchgeführt werden. Dies ermöglicht es, dass Module einer Anwendung bereits getestet werden können, auch wenn diese noch nicht voll funktionsfähig ist.

Mit dem vorliegenden Prototyp sind die Kernprobleme, wie die Generierung der ID, das Wiederfinden eines Widgets und wie das Test-Tool die GUI-Komponenten „kennenlernt“ gelöst worden. Er zeigt damit auf wie eine mögliche Lösung aussehen kann. Weiterhin werden für Schwierigkeiten wie modale Dialoge oder Mausanzeigeunterstützung im Single-Source-Umfeld Lösungen aufgezeigt. Ein Großteil der geplanten Features wurde bereits umgesetzt. Die GUI des Prototypen ist noch nicht final, aber bietet schon jetzt die wichtigsten Funktionen und stellt den Testablauf übersichtlich dar. Diese wird Schritt für Schritt mit der Erweiterung des Prototyps um neue Funktionen erweitert. Der Prototyp selbst steht wie Jo Widgets unter der Open-Source-Lizenz „New BSD License“. Dies ist ein Ansporn für den Autoren, das Projekt noch über diese Arbeit hinaus weiterzuentwickeln.

Der Autor ist überzeugt, dass eine weiterentwickelte Version der Test-Bibliothek sinnvoll in der Praxis eingesetzt werden kann.

Abkürzungsverzeichnis

API .... Application Programming Interface
AWT .... Abstract Window Toolkit
GUI .... Graphical User Interface
HCI .... Human Computer Interaction
POM .... Project Object Model
SPI .... Service Provider Interface
SWT .... Standard Widget Toolkit
UI .... User Interface
UID .... User Interface Dummy Component
XML .... Extensible Markup Language

Abbildungsverzeichnis

4.1 Die Benutzeroberfläche von Abbot 23

4.2 Die Beispielanwendung von Abbot. 24

4.3 Das erstellte Testskript 25

5.1 Beispiel einer einfachen Architektur für Single Sourcing 31

5.2 Die erweiterte Architektur 32

5.3 Die Architektur von Jo Widgets 33

5.4 Wie ein Widget erstellt wird 35

6.1 Beispiel einer möglichen Test-Tool-GUI 39

6.2 Wie das Test-Tool die GUI-Komponenten „kennen lernt“ 40

7.1 Die Projektstruktur der Test-Architektur 41

7.2 Die Test-Tool-GUI 45

7.3 Beispiel einer Parent-Child-Hierarchie 46

7.4 Das mit dem Test-Tool erstellte Skript. 49

Listings

2.1 Beispiel JUnit Test 16

4.1 Beispiel-GUI-Unit-Test mit Abbot 22

5.1 Beispiel erzeugen eines Blueprints mit der BlueprintFactory 35

5.2 Beispiel erzeugen eines Widgets durch Übergabe eines Descriptors . 36

7.1 Implementierung der push()-Methode 42

7.2 Beispiel eines GUI-Unit-Tests 43

7.3 Das Test-Tool mit GUI ausführen 44

7.4 Das Test-Tool ohne GUI starten 45

7.5 Beispiel einer Widget-ID 46

7.6 Die ID des gesuchten Widgets 47

8.1 Beispiels Signatur einer Checkpoint-Methode 51

8.2 Beispiel für einen Checkpoint 52

8.3 Beispiel für das Erzeugen einer ID mit Property 52

Tabellenverzeichnis

4.1 Beispiel Testdaten für einen Login-Dialog 26

4.2 Beispiel Test eines Login Dialogs 27

4.3 Techniken für Testautomatisierung im Vergleich 28

Literaturverzeichnis

[Bec05] K. Beck. *JUnit kurz & gut*. O'Reilly Vlg. GmbH & Co., 2005.

[CH08] P. Cauldwell and S. Hanselman. *Code Leader: Using People, Tools, and Processes to Build Successful Software*. Programmer to Programmer. John Wiley & Sons, 2008.

[Chr11] Aniszczykzx Chris. Vortrag Single-Sourcing RAP and RCP - http://www.slideshare.net/caniszczyk/single-sourcing-rcp-and-rap. In *Single-Sourcing RAP and RCP*. Eclipse Source, besucht am: 8.04.2011.

[Gro11a] Michael Grossmann. Interne Dokumentation innoSysTec GmbH, 2011.

[Gro11b] Michael Grossmann. Jo Widgets - https://code.google.com/p/jo-widgets/, besucht am: 19.04.2011.

[Ham04] P. Hamill. *Unit test frameworks*. O'Reilly Series. O'Reilly, 2004.

[LW04] K. Li and M. Wu. *Effective GUI test automation: developing an automated GUI testing tool*. SYBEX, 2004.

[MfMRWF07] G. Mozdzynski and European Centre for Medium Range Weather Forecasts. *Use of high performance computing in meteorology: proceedings of the Twelfth ECMWF Workshop, Reading, UK, 30 October - 3 November 2006*. World Scientific, 2007.

[Mrs08] S.Tiple Mrs.Jyoti, J.Malhotra und Mrs.Bhavana. *Software Testing and Quality Assurance*. Nirali Prakashan, 2008.

[Nag11] Carl Nagle. *SAFS - http://safsdev.sourceforge.net/FRAMESDataDrivenTestAutomationFrameworks.htm*. SourceForge, besucht am: 20.04.2011.

[Ora11] Oracle. *Java Doc - http://download.oracle.com/javase/6/docs/api/*. Oracle, besucht am: 01.04.2011.

[SBD+10] A. Schatten, S. Biffl, M. Demolsky, E. Gostischa-Franta, T. A-Streicher, and D. Winkler. *Best Practice Software-Engineering: Eine praxiserprobte Zusammenstellung von komponentenorientierten Konzepten, Methoden und Werkzeugen*. Spektrum Akademischer Verlag, 2010.

[SHBA11] A. Sillitti, O. Hazzan, E. Bache, and X. Albaladejo. *Agile Processes in Software Engineering and Extreme Programming: 12th International Conference, XP 2011, Madrid, Spain, May 10-13, 2011, Proceedings.* Lecture Notes in Business Information Processing Series. Springer London, Limited, 2011.

[Tea11a] Apache Maven Project Team. Maven - http://maven.apache.org/guides/introduction/introduction-to-the-pom.html, besucht am: 20.04.2011.

[Tea11b] Apache Maven Project Team. Maven - http://maven.apache.org/index.html, besucht am: 20.04.2011.

[Ull05] Christian Ullenboom. *Java ist auch eine Insel.* Galileo Computing - http://openbook.galileodesign.de/javainsel5/, 2005.

[Wal11] Timothy Wall. Abbot - http://abbot.sourceforge.net/doc/overview.shtml, besucht am: 19.04.2011.