Universität Hannover
Fachgebiet Software Engineering
Institut für Angewandte Systeme
Fachbereich Informatik

Erstellung eines Usability-Layers für XP-Projekte

Bachelorarbeit

im Studiengang Informatik

von

Dennis Hardt

Prüfer: Prof. Dr. Kurt Schneider
Zweitprüfer: Dr. Matthias Becker

Hannover, 8. August 2005

Danksagungen

Ich danke Herrn Daniel Lübke für die hervorragende Betreuung dieser Arbeit.

Dem Fachgebiet Software Engineering der Universität Hannover danke ich für die zur Verfügung gestellten Räumlichkeiten.

Ich danke Herrn Andreas Füllgrabe und Herrn Sebastian Meyer für das Korrekturlesen einer früheren Fassung.

Zusammenfassung

Extreme Programming (kurz XP) erfreut sich in der heutigen Softwareentwicklung immer größerer Beliebtheit. Im Kern macht es das Programmieren zur Schlüsseldisziplin und setzt neben einer Auswahl von Verfahren vor allem darauf, dass Entwickler Spaß an ihrer Arbeit haben und Software hoher Qualität erstellen. Dennoch sehen die Verfahren nicht die Benutzungsfreundlichkeit bzw. Usability von Systemen vor. Ziel dieser Arbeit ist es in automatisierten Tests auf Regeln zu prüfen, wie sie in Guidelines zu finden sind, so dass Benutzer keine Opfer mehr von bekannten Usability Fehlern werden. Jedes XP-Team kann während der Entwicklung einer Oberfläche diese Tests zusammen mit seinen Tests nutzen, so dass benutzerfreundlichere Systeme entstehen.

Inhaltsverzeichnis

1. Einleitung

1.1. Motivation

Vor einiger Zeit war ich auf der Suche nach einer Software, die Daten auf meinem heimischen Rechner sichert. Dabei sollte die Lösung für mich so einfach wie möglich sein. Jeden Tag zur selben Tageszeit sollte ein ganz bestimmter Satz von Daten auf eine externe Festplatte kopiert oder auf DVD gebrannt werden. Natürlich gibt es auf dem Markt solche Programme in einer Vielzahl von Variationen, jedoch das Richtige zu finden entpuppte sich als sehr knifflig.

Jedes Programm wurde gleich wieder von meinem Computer verbannt, wenn die Einrichtung länger als fünfzehn Minuten dauerte. Nimmt man im Nachhinein die Summe der Zeit, die ich für alle Programme aufbrachte, so hätte ich mich doch lieber auf eins beschränken sollen. In den vier Stunden hätte ich auch dieses so konfiguriert, wie ich es brauchte. Schließlich offenbarte sich endlich meine Musterlösung. Ich wurde nur nach „Was? Wann? Wohin?“ gefragt, mehr nicht. Ich musste nichts in einer externen Datei konfigurieren oder Eingaben auf einer Kommandozeile tätigen, es gab nur die Möglichkeit einem Assistenten Schritt für Schritt zu folgen. Jeder möchte, dass ein System einfach zu bedienen ist. Aber warum ist es so schwierig ein System einfach bedienbar zu machen?

Durch die Benutzung von Software entsteht zwangsläufig ein neuer Qualitätsaspekt, nämlich Usability[1] - das Maß für die Benutzungsfreundlichkeit von Systemen. Usability ist wichtig, weil Benutzer mit einer Applikation Aufgaben erledigen möchten. Je schwerer dies ist, umso mehr Zeit müssen sie investieren. Am Ende sind sie frustriert und verlieren jede Motivation mit einer Software zu arbeiten. Ein Unternehmen spart durch gute Usability sogar Geld [Nielsen93].

Software wird vorranig von einem Team aus Entwicklern geschrieben. Dabei herrscht dort kein chaotisches Vorgehen, denn man folgt Prozessen und Methoden. Solch ein Vorgehensmodell oder vielmehr eine Disziplin ist Extreme Programming (kurz **XP**) [Beck03], eine sehr bekannte agile Entwicklungsmethode. Dort greifen viele einzelne Verfahren ineinander, um die Schwächen des Einen durch die Stärken des Anderen auszugleichen. So entstehen z.B. Storycards während des Planungsprozess, die letztendlich die Anforderungen darstellen. Sie beschreiben einzelne Eigenschaften, die vom Kunden gewünscht sind. Aber es gibt einen Mangel in XP: Die Benutzungsfreundlichkeit von Systemen wird nicht berücksichtigt. Es existiert weder ein Verfahren, das ausschließlich für Usability gedacht ist, noch ist Usability in die anderen Verfahren eingeflossen.

[1] Eine ausführliche Beschreibung von Usability folgt in Abschnitt 2.2.

Die Technische Universität Kaiserslautern hat einen Versuch zur Integration von Usability in XP unternommen [Carbon04]. Zusammen mit einem Usability Experten wurden Storycards um Eigenschaften erweitert, die eine gute Benutzungsfreundlichkeit von Software auszeichnen. So soll der Benutzer z.B. immer die Möglichkeit haben eine Aktion ohne Konsequenzen abzubrechen. Eine Storycard galt erst als erledigt, wenn durch Akzeptanztests festgestellt wurde, dass auch die Usability Aspekte erfüllt waren. Neben der Funktionalität trat Usability ins Rampenlicht.

Der Ansatz über das Einbringen von Eigenschaften, die gute Usability begünstigen, soll hier erweitert werden. Experten haben in Guidelines schon Regeln bezüglich Usability schriftlich fixiert. Wünschenswert wäre es, wenn eine Oberfläche auf diese Regeln automatisiert getestet werden kann. Bewahrt man diese Tests auf, kann jedes XP-Team seine entwickelten Oberflächen prüfen.

1.2. Problemstellung

Ziel dieser Arbeit ist das automatisierte Testen auf Usability Aspekte, wie sie in Guidelines niedergeschrieben sind. Dazu sind zunächst aus Guidelines Aspekte zu sammeln, die potenziell testbar sind. Neben einer angemessenen Dokumentation sind diese Aspekte in Tests umzusetzen, die das JUnit Framework[2] benutzen.

Agieren sollen die Tests auf den Komponenten der Oberflächenbibliothek Java Swing[3]. Mitunter ist aber noch keine sofortige Testbarkeit auf Usability gegeben. Deshalb sind in einem Usability Layer die einzelnen Klassen aus Swing so zu erweitern, dass Testbarkeit gewährleistet ist. Weiterhin soll im Usability Layer versucht werden, Usability Regeln direkt zu erfüllen.

1.3. Struktur der Arbeit

Diese Arbeit ist in folgende Kapitel gegliedert:

- Kapitel 2 gibt eine Einführung in die Grundlagen von Graphical User Interfaces (GUI) und Usability. Es schließt mit einem Blick auf die Strukturen in Java Swing, sowie einem kurzen Überblick über JUnit ab.
- Kapitel 3 erläutert Extreme Programming (XP) nach Kent Beck [Beck03] und wieso Usability in XP so schwierig ist.
- Kapitel 4 zieht Konsequenzen aus dem Testen auf Usability. Neben einer neuen Teststrategie, werden auch Grenzen der Testbarkeit und andere Probleme dargestellt.

[2] http://www.junit.org
[3] http://www.sun.com

- In Kapitel 5 folgt die Realisierung des Usability Layers und der Umsetzung der Teststrategie. Dazu wird auch ein Dokumentationswerkzeug zum Erfassen von Usability Guidelines präsentiert.

Abschließend wird in Kapitel 6 und 7 ein kritischer Rück- und Ausblick gegeben.

2. Usability in Graphical User Interfaces

In ihrer täglichen Arbeit interagieren Benutzer über eine Mensch-Maschine-Schnittstelle mit Anwendungsprogrammen, um ihre Aufgaben zu erledigen. Diese Schnittstelle existiert in vielen unterschiedlichen Varianten, wie z.B. einer Kommandozeile, in der Benutzer auszuführende Befehle angeben und den abzuarbeitenden Datensatz durch weitere Parameter bereit stellen. Besonders in UNIX-Systemen ist dieses Prinzip noch weit verbreitet.

Betrachtet man heutzutage einen Arbeitsplatz im Büro, so fällt neben einer Tastatur auch eine Maus ins Auge. Diese Maus wird für eine andere Art der Benutzerinteraktion benötigt, nämlich den grafischen Benutzerschnittstellen (*Graphical User Interfaces*).

2.1. Graphical User Interfaces

Graphical User Interfaces (kurz **GUI**) haben sich über die Jahre stark verbreitet. Eine ganze Schar von Betriebssystemen stützt sich heute auf diesen Ansatz. Zu nennen sind z.B. Microsoft Windows [1] oder MacOS [2]. Auch Oberflächenbibliotheken sind oftmals auf GUIs ausgerichtet und wurden dafür konzipiert. Java Swing bietet z.B. Klassen für Fenster, Komponenten und die Benutzerinteraktion an.

Charakteristisch für ein GUI sind Fenster, Menüs, Icons und ein Zeiger [Nielsen93]. Abbildung 2.1 zeigt dazu ein Beispiel.

Fenster bilden die äußere Hülle um eine Vielzahl von Komponenten. Sie lassen sich häufig frei auf dem Bildschirm positionieren, sind in ihrer Größe veränderbar und können geöffnet oder geschlossen werden. In modernen Betriebssystemen ist es kein Problem mehrere Fenster gleichzeitig übereinander darzustellen. Doch nicht immer sind alle Fenster vom gleichen Typ, man unterscheidet zwischen vier Grundtypen [Sun01]:

- *Primäre Fenster*: Hier findet die hauptsächliche Interaktion des Benutzers mit Daten und Dokumenten statt.
- *Sekundäre Fenster* (hauptsächlich *Dialoge*): Hier werden weitere Eingaben vom Benutzer abgefragt oder Daten speziell ausgegeben. So wird z.B. vom Benutzer eine Bestätigung für eine Operation erfragt.

  Ein sekundäres Fenster ist immer abhängig von einem primären Fenster. Wird das primäre Fenster geschlossen oder minimiert, so ist auch das sekundäre

[1] http://www.microsoft.com
[2] http://www.apple.com

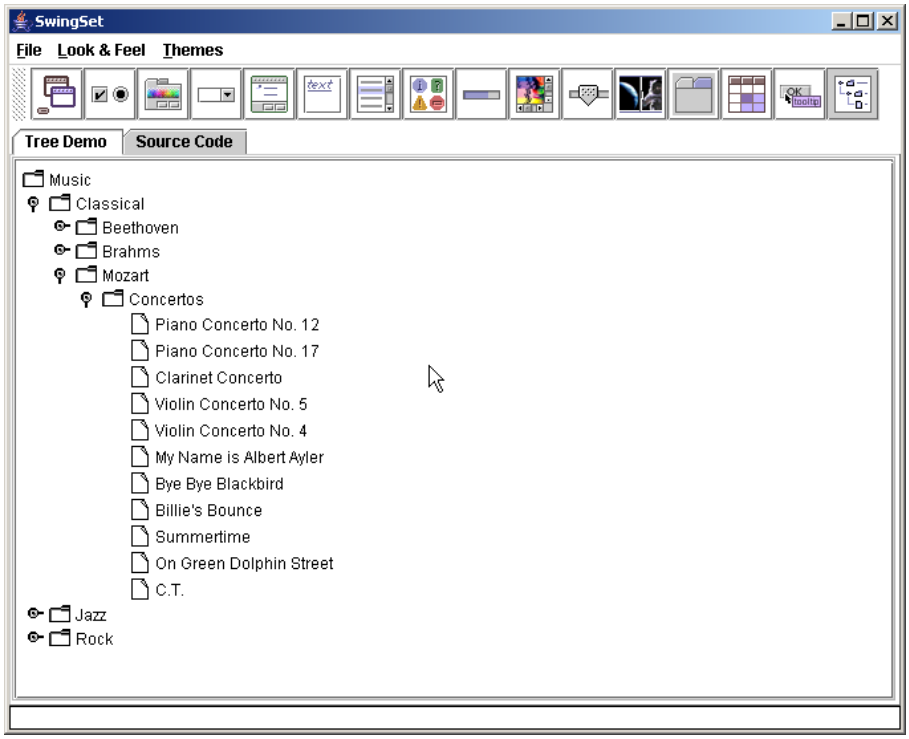

Abbildung 2.1.: Ein Graphical User Interface

Fenster zu schließen oder zu verbergen.

- *Utility:* Die Inhalte eines solchen Fensters betreffen immer ein aktives primäres Fenster. So werden hier häufig auswählbare Operationen angeboten. Im Gegensatz zu den sekundären Fenstern müssen diese Fenster nicht zwingend geschlossen werden, wenn das primäre Fenster geschlossen wird - sie sind eigenständig.
- *Plain:* Ein einfaches Fenster, meist eingesetzt für so genannte „Splash-Screens“[3].

Innerhalb des Fensters befinden sich dann Komponenten einer Oberflächenbibliothek, mit denen ein Benutzer interagiert, um Daten zu manipulieren. Java Swing bietet eine große Auswahl von verschiedenen Komponenten an. Diese erstrecken sich über ein Spektrum von z.B. einfachen Buttons bis hin zu komplexen Tabellen.

2.2. Usability

Das Benutzen von Software kann Spaß machen, aber auch frustrieren. Jeden Tag ist man Anwendungsprogrammen ausgeliefert, um seine Arbeit zu erledigen. Manchmal

[3] Ein Splash-Screen wird während des Startvorgangs der Applikation angezeigt und kann weitere Informationen über das Produkt enthalten.

geht es schnell von der Hand, manchmal kostet es viel Zeit. Dabei wünscht sich niemand schlecht benutzbare Systeme. Wird der Zustand in dem etwas fertig ist, später erreicht als nötig, hat man Zeit sinnlos verschwendet. Daraus ist ein neuer Qualitätsaspekt an Software entstanden, der die Benutzungsfreundlichkeit repräsentiert: *Usability*.

Doch ist Usability kein eindimensionales Merkmal einer Benutzerschnittstelle. Es ist vielmehr eine Unterteilung in fünf Eigenschaften [Nielsen93]:

- *Learnability*: Ein System sollte einfach und schnell zu erlernen sein. Software soll einen produktiven Nutzen bringen - ein Zustand, der so schnell wie möglich zu erreichen ist. Dabei sind Schulungen und Handbücher Maßnahmen, um Menschen ein System zu vermitteln. Hält man sich zusätzlich an Standards, können sich Benutzer immer wieder durch ihr eigenes Wissen schnell einarbeiten.
- *Efficiency*: Das System sollte effizient nutzbar sein und einen hohen Grad an Produktivität erreichen. Wird z.B. während der Benutzung festgestellt, dass immer die gleichen Teilschritte zum Abschließen einer Aufgabe durchgeführt werden müssen, dann lassen sich diese auch zusammenfassen, um den Benutzer zu entlasten.
- *Memorability*: Benutzer sollten sich schnell an die Funktionen eines Systems erinnern können. Häufig schreiben sich Benutzer einzelne Schritte auf, wenn sie eine Aufgabe erfolgreich erledigt haben. Sie wollen bei ähnlichen Arbeiten einfach sofort wieder zum Ergebnis kommen, ohne den Prozess selbst noch mal zu erarbeiten.
- *Errors*: Nutzerfehler sollten minimiert werden. Tritt ein Nutzungsfehler auf, so sollte dieser korrigierbar sein. Dabei dürfen keine katastrophalen Fehler, wie z.B. Datenverlust, auftreten.
- *Satisfaction*: Das Benutzen eines Systems sollte angenehm sein, so dass Benutzer subjektiv zufrieden sind. Ein Punkt, der bei Software besonders wichtig ist, die nicht zur Arbeit eingesetzt wird, z.B. *home computing* oder Computerspiele.

Weist Software Mängel in einer der fünf Eigenschaften auf, verlieren Benutzer vielleicht wertvolle Zeit. Zeit, die in einer Firma gleichbedeutend mit Produktivität ist. Am Ende kostet schlechte Usability sogar Geld. Software möglichst gut benutzbar zu machen, sollte daher im Sinne eines jeden Entwicklers liegen.

2.2.1. Guidelines

Man kann sich viele Maßnahmen überlegen, um die im letzten Abschnitt genannten Eigenschaften möglichst vorbildlich abzudecken. *Guidelines* sind dabei eine Möglichkeit, wo Regeln schriftlich festgehalten wurden, um bekannte Fehler in der Usability zu beschreiben. Durch ihre Kenntniss können Entwickler Fallen umgehen und dem Benutzer leichter bedienbare Software liefern. Unterschieden wird zwischen *general*

guidelines, die für alle Benutzerschnittstellen gelten, *category-specific guidelines*, die z.B. nur für GUIs gelten und *product-specific guidelines*, die ein individuelles Produkt betreffen [Nielsen93].

Nielsen beschreibt in [Nielsen-b] zehn *Heuristiken*, die man als *general guidelines* bezeichnen kann. Guidelines, wie z.B. [Sun01], sind *category-specific guidelines*, weil sie sich ausschließlich auf GUIs beziehen. Zusätzlich stellen sie sehr viel feinere Regeln auf, die sich zum Großteil von *general guidelines* ableiten lassen. So soll z.B. jedes Fenster einen Titel haben [Sun01]. Dies ist eine Konsequenz aus der Heuristik, dass man sich an Konventionen der Plattform halten soll [Nielsen93], um Konsistenz zu erzielen.

Anwender haben eine bestimmte Sichtweise auf ein System. Sie sehen nur die Oberfläche, die Entwickler zur Interaktion vorgesehen haben. Sie sehen weder den Code noch kennen sie interne Strukturen. Auch Experten werden zu Benutzern, wenn sie eine Oberfläche begutachten. Stellen sie zusätzlich Regeln an ein GUI auf, verändert sich die Sichtweise nicht.

2.2.2. Heuristic Evaluation

Heurisitic Evaluation ist eine Methode, um Usability Fehler in einer Benutzerschnittstelle zu finden. Dabei inspizieren Prüfer eine Oberfläche und notieren, wenn ein vorher ausgewählter Satz von Heuristiken nicht eingehalten wurde [Nielsen-a].

Entscheidend ist, dass diese Methode von einer Gruppe von drei bis fünf Personen auszuführen ist [Nielsen-a]. Ein einzelner Prüfer würde zwar sehr offensichtliche Usability Fehler finden, jedoch nur eine Auswahl von gut versteckten Problemen. Erst durch die Summe von verschiedenen Befunden erreicht man eine fast vollständige Fehlerabdeckung. Der Prozess an sich ist aber alleine durchzuführen, um Ergebnisse nicht zu vermischen.

Zusätzlich kann ein Beobachter eingesetzt werden, der die Dokumentation der Befunde für jeden Prüfer übernimmt. Auch das anschließende Zusammentragen aller Ergebnisse übernimmt dann der Beobachter.

Eine Prüfungssitzung dauert ungefähr ein bis zwei Stunden, in der die Prüfer eine Oberfläche schrittweise durchgehen und alle Komponenten mit einer Liste von Heuristiken abgleichen. Zusätzlich ist es natürlich erlaubt, auch Heuristiken anzugeben, die nicht explizit aufgelistet werden und dem Prüfer spontan einfallen. Als Ergebnis entsteht eine Liste von Problemen, die aufzeigt, wo gegen Heuristiken verstoßen wurde. Durch die *Heuristic Evaluation* werden jedoch weder Lösungsansätze gegeben, noch die Qualität von zukünftigen Änderungen bewertet.

2.3. Umsetzen von Graphical User Interfaces mit Java Swing

In Java existiert zum Umsetzen von GUIs u.a. die Oberflächenbibliothek Swing. Mit ihr lassen sich Fenster strukturiert aufbauen, um Benutzern den Zugang zu einer Applikation zu ermöglichen. Dabei sind die in den folgenden Abschnitten beschriebenen Grundlagen wichtig für diese Arbeit.

2.3.1. Das Model-View-Controller Paradigma in Swing

Bei der Entwicklung einer Applikaton mit einem GUI bietet sich eine Trennung nach dem *Model-View-Controller* (**MVC**) Paradigma [MVC] an. Den Kern bilden dabei drei Objekte. Das *Model* ist Repräsentant der Daten der Applikation. Dies kann eine Datenbank sein oder ist durch Klassen realisiert. Die *View* ist die entsprechende Präsentation auf dem Bildschirm und der *Controller* die Art der Benutzerinteraktion. Im Zusammenspiel können sich Entwickler auf einzelne Teile beschränken. Die einzelnen Schritte aus Abbildung 2.2 illustrieren das MVC-Prinzip in vereinfachter Form:

1. Die *View* reicht Benutzereingaben an den *Controller* weiter.
2. Der *Controller* setzt Eingaben in Änderungen am *Model* um.
3. Änderungen im Zustand des *Model* werden an die *View* weitergegeben.

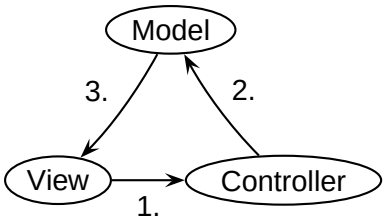

Abbildung 2.2.: Model-View-Controller

MVC hat auch in Java Swing Einzug gehalten, wurde jedoch modifiziert. View und Controller sind zu einem *user-interface object* (UI object) verschmolzen [Fowler]. Nur die Trennung zum Model besteht weiter, damit es Applikationsspezifisch entwickelt werden kann. Abbildung 2.3 zeigt den neuen Sachverhalt.

Der *UI Manager* bildet eine Instanz, in der Informationen über das Aussehen von Komponenten abgefragt werden. So lässt sich die Beschaffenheit einer jeden Komponente durch Änderung des *UI Object* zur Laufzeit auswechseln und je nach Plattform variieren.

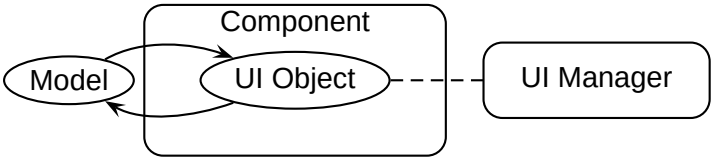

Abbildung 2.3.: Model-View-Controller in Swing

2.3.2. Struktur eines Graphical User Interface in Swing

In Java Swing erweitert eine umfassende Vererbungshierarchie jede Komponente um verschiedene Funktionalitäten, beginnend mit der Basisklasse `Component` aus dem Paket `java.awt`. Sie stellt Objekte bereit, die eine auf dem Bildschirm darstellbare Präsentation haben und Benutzerinteraktionen ermöglichen [Java API]. Als Unterklasse folgt die Klasse `Container`, ebenfalls aus dem Paket `java.awt`. Durch sie erhalten alle Komponenten die Möglichkeit, andere Komponenten als Kinder zu fassen. Es lässt sich dadurch eine Baumstruktur aufbauen, die repräsentativ für das GUI ist und entsprechend auf dem Bildschirm angezeigt wird. Dabei bilden Fenster die Wurzel in solch einem Baum, die Inhalte folgen als Kinder. Swing setzt diese Vererbungshierarchie durch Klassen, wie z.B. `JComponent`, weiter fort, bis die Komponenten selbst folgen. In Swing gilt als Namenskonvention, dass der Name einer Komponente mit einem `J` beginnt und mit dem Typ der Komponente abschließt.

Aufgebaut wird ein GUI dadurch, dass einer Komponente weitere Komponenten hinzugefügt werden. Abbildung 2.4 zeigt die Anatomie eines Fensters. Es soll verdeutlicht werden, dass es nicht immer eine einheitliche Baumstruktur ist. So kann eine Trennung zwischen allen untergeordneten Komponenten existieren, obwohl die verschiedenen Bereiche auch nur Kinder sind. Die Inhalte (in Abbildung 2.4 rechts unten angedeutet) sind dann Komponenten, die durch die `Container`-Hierarchie strukturiert werden.

Existiert ein Baum für ein GUI, so sind die Zusammenhänge zwischen den Komponenten bekannt. Dennoch fehlen Informationen darüber, wie jede Komponente genau darzustellen ist. Zu diesem Zweck existieren neben dem in Abschnitt 2.3.1 schon erwähnten *UI Object* und *UI Manager*, noch `LayoutManager`. Im *UI Object* wird festgelegt, wie eine Komponente aussieht, d.h. welche Farben, Schriftarten, Liniendicken, etc. sie hat. `LayoutManager` hingegen platzieren Komponenten auf dem Bildschirm. Wird z.B. im `ContentPane` eines Fensters ein Button hinzugefügt, wird keine Aussage darüber getroffen, wo er angezeigt werden soll. Zentriert, linksbündig, rechtsbündig, oben oder unten - es ist nicht bekannt. `LayoutManager` bieten entsprechende Schnittstellen an, um diesen Mangel zu beseitigen. Wird einem Elternelement ein `LayoutManager` zugewiesen, werden alle Kinder durch ihn positioniert und evtl. auch weitergehend verändert (z.B. in ihrer Größe). Swing stellt schon einen Basissatz von *Layouts* bereit. Zu nennen sind unter anderem das `FlowLayout` und `BorderLayout` [Java API], die Komponenten hintereinander bzw. in vordefinierten Regionen im `Container` platzieren.

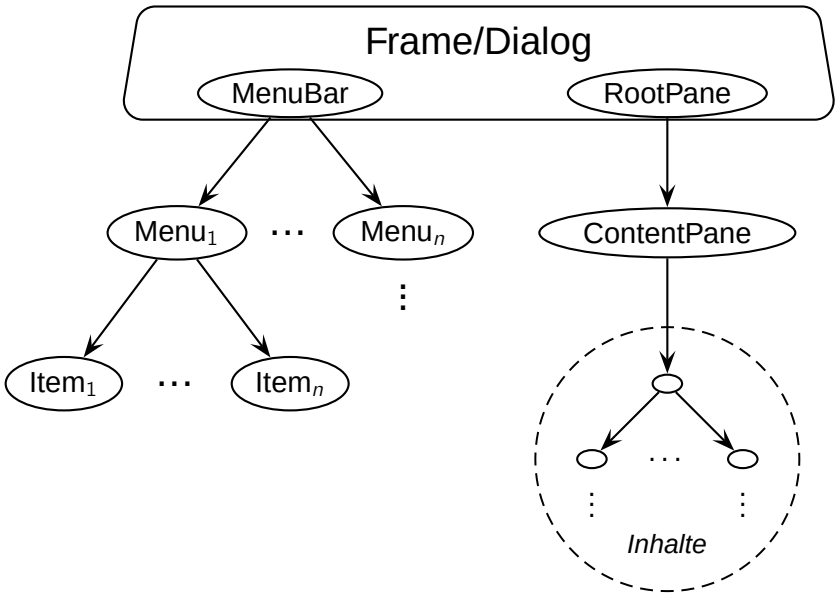

Abbildung 2.4.: Anatomie eines Frame bzw. Dialog in Java Swing

Greifen alle Aspekte von Swing ineinander, also die Baumstruktur, das Austauschen des Aussehens und die Positionierung, lassen sich plattformunabhängige GUIs realisieren.

2.4. Automatisierte Tests mit JUnit

Zum Schreiben von automatisierten Tests benötigt man ein geeignetes Werkzeug zur Formulierung der Tests selbst, aber auch um diese aufzubewahren und die Ergebnisse gesammelt darzustellen. Das JUnit Framework[4] stellt all diese Funktionen für Java zur Verfügung.

Unterklassen der Klasse `TestCase` bilden die eigentlichen Testfälle. Dort werden in öffentlichen Methoden ohne Rückgabewert, deren Name mit `test` beginnt, die Bedingungen eines Tests aufgestellt. Dazu steht eine Sammlung von *assert*-Methoden zur Verfügung. `assertTrue(boolean b)` prüft z.B., ob `b` wahr ist.[5] Wäre `b` falsch, schlägt der Test fehl. Ein einfaches Beispiel für einen Test, der immer erfüllt ist:

```
public void MyTest extends TestCase {
  public void testTrue() {
    assertTrue(true);
  }
}
```

[4] http://www.junit.org

[5] Weitere Methoden bilden Verfeinerungen von `assertTrue`. Es gibt z.B. das direkte Gegenstück `assertFalse`, aber auch `assertEquals`, `assertNotNull`, etc. [Hamill04]

Schlägt während der Ausführung eine `assert`-Methode fehl, so wird die zugehörige `test`-Methode abgebrochen. Alle anderen Methoden werden aber weiter ausgeführt. Dies erlaubt stufenweise einen bestimmten Aspekt zu prüfen und Testfälle in Methoden zu organisieren.

Um nicht alle Tests in einem einzigen `TestCase` unterzubringen, bietet sich eine Verteilung auf mehrere `TestCases` an. Zunächst müsste jedoch jeder `TestCase` selbst ausgeführt werden, eine Zusammenfassung erfolgt erst durch eine `TestSuite`. Ihr können, in Anlehnung an das Composite Pattern [Gamma96], `TestCases` und weitere `TestSuites` hinzugefügt werden. Damit eine TestSuite ausführbar wird, muss die Methode `public static Test suite()` implementiert werden, in der die TestSuite gebaut und zurückgegeben wird[6]. Dazu ein Beispiel, in dem alle Testmethoden von `MyTest` einer TestSuite hinzugefügt werden:

```
public void MySuite extends TestSuite {
  public static Test suite() {
   return new TestSuite(MyTest.class);
  }
}
```

Ausgeführt wird ein `TestCase` oder eine `TestSuite` durch Übergabe an einen `TestRunner`. Dabei werden alle Tests durchgeführt, die Ergebnisse gesammelt und durch unterschiedliche Oberflächen ausgegeben. In dem z.B. auf Swing basierenden GUI wird zusätzlich durch einen Balken signalisiert, ob die Tests erfolgreich (*grün*) waren oder ob ein Test fehlgeschlagen (*rot*) ist.

[6] Für nähere Details siehe [Hamill04].

3. Usability Guidelines in Extreme Programming

3.1. Extreme Programming

Der Bereich des Software Engineering bietet eine Vielzahl von Möglichkeiten zur Softwareentwicklung an. Extreme Programming (**XP**), beschrieben im gleichnamigen Buch von Kent Beck [Beck03], ist ein sehr bekannter Vertreter der agilen Entwicklungsmethoden. Neben einem großen Angebot von Verfahren, geht es in XP aber auch um Einstellungen und Werte, die zu qualitativ hochwertiger Software führen. Dabei soll das Grundproblem der Softwareentwicklung, nämlich Risiken [Beck03], unter Kontrolle gebracht werden. Es ist Realität, dass sich Termine verzögern können oder zu einem späten Zeitpunkt im Projekt noch Änderungen einfließen. Insgesamt zielt XP darauf ab, auf die Probleme einzugehen, indem es einen eigenen Stil der Softwareentwicklung darstellt. Änderungen sind z.B. in XP gewünscht und auch jederzeit willkommen - sie stellen kein Problem mehr da.

Dieser Stil folgt vier Werten: Kommunikation, Einfachheit, Feedback und Mut:

- Kommunikation ist wichtig, damit keine Informationen verloren gehen oder niemals ausgesprochen werden. Programmierer, Manager und Kunden müssen sich mitteilen, damit ihr Wissen weitergegeben wird.
- Einfachheit zielt darauf ab, dass immer die einfachste Lösung anzustreben ist. Man soll sich nicht mit Dingen von morgen belasten, sondern ein Problem einfach lösen - jetzt.
- Der dritte XP-Wert ist das Feedback. Nur durch Rückmeldungen weiß man, ob sich das Projekt in die richtige Richtung entwickelt und fördert die Kommunikation und Einfachheit.
- Mut wird beim Treffen von Entscheidungen benötigt. Wirft man Code lieber weg oder versucht man ihn zu vereinfachen. Beides erfordert viel Mut, weil niemand weiß, was Änderungen alles bewirken können.

Aus den Werten und Prinzipien[1] folgen in XP vier grundlegende Arbeitsschritte: Programmieren, Testen, Zuhören und Designentwurf. Es wird programmiert, um etwas zu leisten. Es wird getestet, um zu wissen, wann man mit Programmieren fertig ist. Man hört zu, weil man sonst nicht weiß was zu Programmieren und zu Testen ist. Und man entwirft ein Design, damit man unendlich lange programmieren, testen und zuhören kann [Beck03].

[1] Für die Prinzipien siehe [Beck03].

Um Risiken zu kontrollieren, Werte zu vereinigen und den vier Arbeitsschritten zu folgen, bietet XP eine große Sammlung von Verfahren an. Verfahren, die sich gegenseitig stützen, um die Schwächen eines anderen Verfahrens durch eigene Stärken auszugleichen. Abbildung 3.1 zeigt eine Übersicht über alle Verfahren.

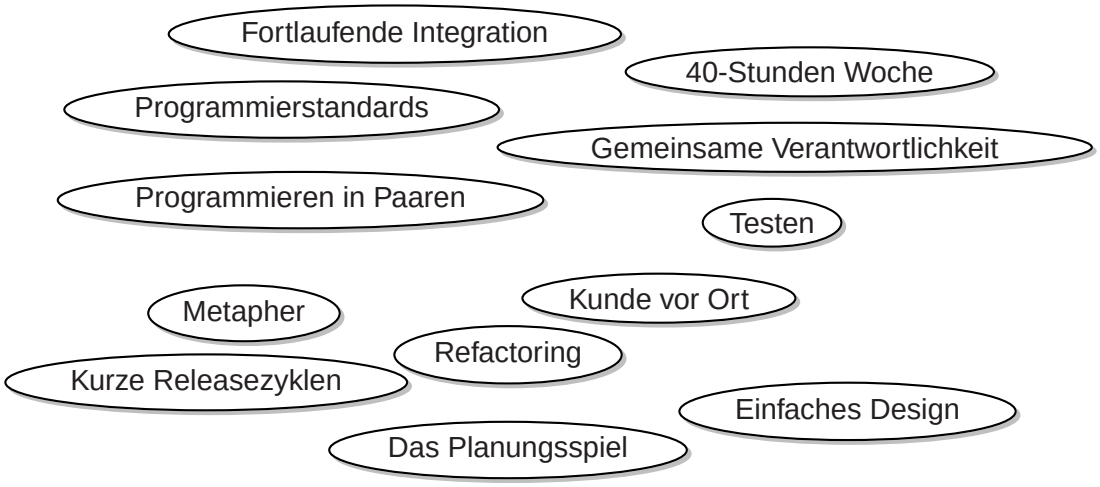

Abbildung 3.1.: Extreme Programming Verfahren als Übersicht

Im weiteren Verlauf dieser Arbeit sind nur einige Verfahren von besonderer Bedeutung. Sie seien hier etwas ausführlicher erläutert:

- *Kurze Releasezyklen:* Ein System soll sehr schnell in Produktion gehen und möglichst früh eine neue Version folgen.

- *Testen*: In XP werden automatisierte, isolierte Tests geschrieben. Programmierer schreiben Komponententests, um von der Korrektheit ihres Codes überzeugt zu sein. Komponententests müssen dabei immer zu 100% erfüllt sein. Kunden schreiben Akzeptanztests, um zu bestätigen, dass gewüschte Funktionalitäten auch wirklich implementiert wurden. Akzeptanztests fallen nicht unter die 100%-Regel, doch sollten zum Projektende nahezu 100% angestrebt werden.

- *Fortlaufende Integration*: Der Code wird spätestens nach einem Entwicklungstag integriert und getestet. Konflikte und fehlgeschlagende Tests sind in einem Wechselspiel zwischen Testen und Implementieren solange zu beheben, bis der integrierte Code zu 100% fehlerfrei läuft.

- *Gemeinsame Verantwortlichkeit*: Von jedem im Team wird erwartet, dass er dem Code jederzeit etwas Sinnvolles hinzufügt. Sieht er eine Möglichkeit zur Verbesserung, dann nimmt der Entwickler sie vor, es gibt keine individuelle Verantwortlichkeit.

3.2. Das Problem mit der Usability in Extreme Programming

In XP wettet man darauf, dass es besser ist *„heute etwas Einfaches zu tun und morgen etwas mehr zu zahlen, wenn man es ändern muss, statt heute etwas Kompliziertes zu tun, dass vielleicht niemals eingesetzt wird“* [Beck03]. Hier bezieht sich Kent Beck zwar auf das Programmieren, doch lässt sich dieser Satz auch auf Usability übertragen. Selbst beim Entwurf eines GUI kann man sich die Frage nach der einfachsten Lösung stellen, denn ein Benutzer möchte primär nur seine Arbeit mit einem System erledigen. Wird es zu kompliziert, nutzt er es am Ende vielleicht gar nicht. Nimmt man an, dass in XP der Wunsch vorhanden ist ein System mit einem hohen Maß an Usability zu erstellen, ist für diese Forderung zunächst wenig Platz. Kein Verfahren sieht Usabiliy direkt vor. *„XP erklärt das Programmieren zur Schlüsselaktivität während der gesamten Dauer eines Softwareprojekts.“* [Beck03, Vorwort Gamma]

Usability Guidelines haben die besondere Eigenschaft allgemein gültig zu sein und sollten daher in jedem GUI Einzug erhalten. Sie bilden einen Schatz von Wissen, dessen Anwendung z.B. durch die Heuristic Evaluation sichergestellt werden soll. Dennoch passt diese Methode direkt nicht in XP. Hier liegt ein sehr evolutionärer Designprozess vor, wo Releasezyklen kurz sind, täglich integriert wird und Tests Vertrauen aufbauen. Eine Sitzung zur Überprüfung von Heuristiken hingegen ist ein einmaliger Prozess, angewandt auf etwas nahezu abgeschlossenes. Jede Änderung am GUI könnte dazu führen, dass Regeln aus Guidelines nicht mehr eingehalten werden. Jede Heuristic Evaluation kostet Geld und benötigt Prüfer. Ihr resultierendes Feedback würde einige Zeit brauchen, um in die Entwicklung einzufließen.

In XP kann man versuchen die Bedeutung von Guidelines zu stärken. Nimmt man z.B. an, dass die Entwickler selbst auf die Einhaltung von Heuristiken achten sollen, dann müssten sie entsprechend geschult werden und zwar jeder von ihnen. Aufgrund der gemeinsamen Verantwortlichkeit ist es schlecht einzelne Spezialisten unter den Entwicklern zu haben, besonders wenn diese das Team verlassen. Doch was kann man alternativ tun?

„XP ist eine Disziplin der Softwareentwicklung. Es ist eine Disziplin, weil es bestimmte Dinge gibt, die man tun muss, wenn man XP einsetzen will. Man kann es sich nicht aussuchen, ob man automatisierte Tests schreibt oder nicht - wenn man es nicht tut, dann ist es auch kein XP; Ende der Diskussion.“ [Beck03, S. xvii] XP umfasst also einen Satz von Werten, Prinzipien und Verfahren und alles wird immer angewandt - sonst wäre es kein XP. Vielleicht gelingt durch Erweiterung eines Verfahrens die Einhaltung von Guidelines. Dies ist das Ziel dieser Arbeit.

4. Erfüllen von Usability Guidelines

4.1. Lösungsansatz

Zur Integration von Usability Guidelines in XP soll ein Verfahren gezielt erweitert werden. Vielleicht besteht sogar die Möglichkeit, die Heuristic Evaluation (siehe Abschnitt 2.2.2) in einer Form umzusetzen, so dass XP davon profitieren kann.

Unter der Annahme, dass ein XP-Team ein GUI entwickelt hat, sei eine Heuristic Evaluation durchzuführen. Das GUI liegt damit in einer ausführbaren Version, aber auch als Code, vor. Prüfer würden mit einer Liste von Heuristiken das GUI inspizieren und Komponenten notieren, die gegen eine Heuristik verstoßen. Ein ähnliches Vorgehen existiert durch Tests bereits in XP. Unterschiedlich ist nur die Beschaffenheit: Tests sind automatisiert und isoliert. Beim Testen werden für Objekte Eigenschaften geprüft, die Ergebnisse gesammelt und schließlich angezeigt. In dieser Arbeit werden Heuristiken in automatisierte Tests überführt, so dass die Heuristic Evaluation in einer anderen Form vorliegt. Jedes XP-Team kann dadurch unabhängig und automatisert auf Usability testen. Immer wenn programmiert wird, lässt sich sofort ein GUI überprüfen. Das Feedback wird somit schneller, direkter und vor allem wieder verwendbar.

Als Grundlage für diese Tests dienen die Regeln aus Guidelines, so dass Entwickler verstärkt auf deren Einhaltung achten. In XP werden immer Tests geschrieben, bevor programmiert wird, auch *„Test-First“* genannt. Testet man Teile eines GUI bevor es überhaupt existiert, geben die Tests schon Hinweise über das zu programmierende Produkt. Es gibt in Guidelines Regeln, die bestimmte Eigenschaften von Komponenten verlangen und es gibt Regeln, die Verbote beschreiben. Tests, die Usability Regeln prüfen, werden in dieser Arbeit als *Usability Unit Tests* definiert.

Um überhaupt wieder verwendbar auf Usability zu testen, stellen sich zunächst zwei Fragen:

- Was wird getestet? Es ist eine Datenbasis zu finden, die in jedem Projekt gleich ist, wenn ein GUI aufgebaut wird.

  Ein möglicher Ansatz wäre die benutzte Oberflächenbibliothek, hier *Java Swing*. Gelingt es Guidelines in Usability Unit Tests zu überführen und diese entsprechenden Komponenten aus Swing zuzuordnen, lässt sich auf Usability testen. Mitunter ist durch die Beschaffenheit von Usability Regeln jedoch nicht direkt Testbarkeit auf Usability gegeben, so dass Swing erweitert werden muss. Es entsteht ein neuer *Usability Layer*, der jede Komponente aus Swing gezielt ausbaut.

- Wie wird getestet? Es muss festgestellt werden, welche Teststrategie Guidelines

fordern. Dazu zwei Beispiele:

1. Zwei Dialoge dürfen nicht den selben Titel aufweisen [Johnson00]. Ein entsprechender Usability Unit Tests agiert hier auf einem GUI, das ein XP-Team entwickelt hat. Dabei werden die Daten aller Dialoge, die ein Benutzer im Betrieb sehen kann, geprüft.

   Einige Tests arbeiten folglich mit einem konkreten GUI. Der Grund dafür ist, dass Guidelines aus der Sicht von Benutzern geschrieben wurden. Sie beschreiben z.T. das Verhalten, die Daten und Zusammenhänge von Komponenten im Betrieb. Also:

   Ein GUI ist so zu testen, wie es der Benutzer auf dem Bildschirm wirklich sieht.

2. In Dialogen sollen alle Kontrollbuttons die selbe Breite aufweisen [Sun01]. Auch dies lässt sich bei einem konkreten GUI überprüfen. Dennoch gilt diese Regel für alle Instanzen eines Kontrollbuttons gleichermaßen, sie ist nur abhängig von dem zur Verfügung gestellten `LayoutManger` (siehe Abschnitt 2.3.2). Folglich lässt sich ein Usability Unit Test schreiben, der einen entsprechenden `LayoutManager` spezifiziert.

Abbildung 4.1 illustriert den hier verfolgten Testansatz.

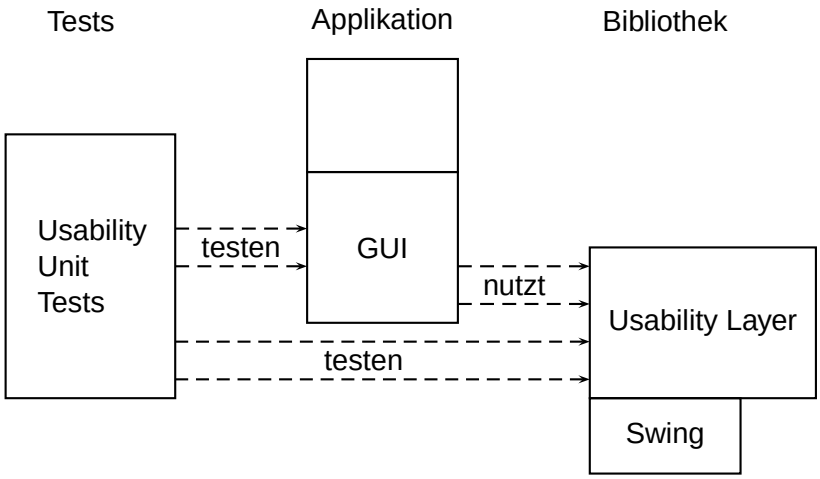

Abbildung 4.1.: Automatisiertes Testen auf Usability

Letztendlich werden in dieser Arbeit Regeln aus Guidelines in automatisierte Tests umgesetzt: „Eine Programmeigenschaft, für die es keinen automatisierten Test gibt, existiert einfach nicht.“ [Beck03, S. 57]

4.2. Usability Unit Tests

Abschnitt 2.3.1 hat den MVC-Ansatz aus Swing erläutert. Beim Schreiben von Usability Unit Tests genügt eine Beschränkung auf die View und den Controller, welche durch die Komponenten selbst repräsentiert werden. Dabei ist das Model zu vernachlässigen, weil es vom XP-Team z.T. noch entwickelt wird, aber auch weil sich Guidelines nur selten darauf beziehen.

In XP bilden Akzeptanz- und Komponententests den Kern der Teststrategien [Beck03, S. 119]. Daher bietet sich für *Usability Unit Tests* zunächst ein Vergleich an:

- Ein Akzeptanztest ist ein Test, der aus der Perspektive des Kunden geschrieben wird [Beck03]. Er definiert Leistungsmerkmale, die vom XP-Team zu implementieren sind.

  Akzeptanztests haben die Eigenschaft, dass sie nicht immer zu 100% fehlerfrei laufen müssen. Schlägt ein Test fehl, so wurde eine Funktionalität nur noch nicht implementiert, der Kunde kann letztendlich entscheiden, ob dies getan werden muss oder nicht. Usability Unit Tests sind keine Akzeptanztests, weil die Regeln aus Guidelines allgemeine Gültigkeit haben und somit in jedem GUI einfließen sollten.

- Ein Komponententest ist ein Test, der aus der Perspektive der Programmierer geschrieben wird [Beck03]. Er verifiziert das Verhalten einer Komponente (z.B. einer Klasse) und stellt in XP die Anforderungen dar. Dabei wird solange programmiert bis der Komponententest fehlerfrei läuft.

  Wären Usability Unit Tests Komponententests, so würde ein Entwickler in einem Test eine Instanz einer Klasse erzeugen und dann auf eine Usability Regel prüfen. Am Beispiel der `LayoutManager` wurde in Abschnitt 4.1 bereits angedeutet, dass es Usability Regeln gibt, wo diese Art des Testens funktioniert. Folglich können Usability Unit Tests Komponententests sein.

Die charakteristischen Forderungen für wieder verwendbare Usability Unit Tests sind:

- Kein Usability Unit Test schlägt fehl. Guidelines legen den allgemeinen Vertrag von Komponenten fest. Wer sich nicht daran hält, muss nachbessern.
- Es werden nur die Standardkomponenten einer Oberflächenbibliothek getestet, um wieder verwendbare Tests zu schreiben. Werden zusätzlich neue Komponenten realisiert, so müssen evtl. auch neue Usability Unit Tests geschrieben werden. Je nachdem ob Guidelines für diese Komponente existieren oder nicht.

Die Beschaffenheit von Usability Regeln fordert zusätzlich eine neue Teststrategie. Z.B. soll *jedes* dem Benutzer sichtbare Fenster einen Titel haben [Sun01], so dass ein konkretes GUI im Betrieb zu testen ist. Hier entstehen **Kontexttests**, die alle instanziierten Komponenten im Kontext prüfen. Im Grunde entsprechen solche Tests immer einem bestimmtem Zustand, als wenn das GUI mit einer Kamera fotografiert wurde

und anschließend ausgewertet wird. Der folgende Abschnitt wird Kontexttests näher erläutern.

Abschließend ist zu sagen, dass Usability Unit Tests immer in einem *Komponententest* oder *Kontexttest* resultieren.

4.2.1. Kontexttests

Kontexttests stellen eine neue Form des Testens dar, um ein GUI so zu testen, wie es der Benutzer auf dem Bildschirm sieht. In Abschnitt 2.3.2 wurde die Struktur eines GUI vorgestellt. Wäre z.B. ein Fenster im Betrieb zu testen, so sind zunächst für jede Komponente Kontexttests zu schreiben. Im Gegensatz zu Komponententests wird bei Kontexttests das zu testende Objekt nicht im Test selbst erzeugt. Komponente und Test sind erst einander bekannt zu machen, indem ein äußerer Mechanismus die baumähnliche Struktur eines GUI auswertet und alle erreichten Komponenten in zugehörigen Kontexttests setzt. Dabei prüft ein Kontexttest immer genau *eine* instanziierte Komponente auf Usability Regeln. Abbildung 4.2 zeigt dazu ein Beispiel, in dem ein Baum schrittweise ausgewertet wird. Dabei ist Kontexttest P ein Test für Komponenten vom Typ `JPanel` und die Kontexttests B_1 und B_2 von der selben Testklasse für Komponenten vom Typ `JButton`.

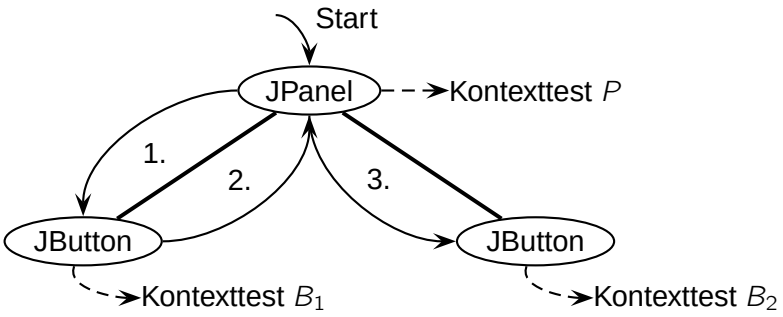

Abbildung 4.2.: Kontexttests für ein primäres Fenster

Als Kernaussage folgt für diese neue Teststrategie: *Jede Instanz einer Komponente eines GUI ist von einem zugehörigen Kontexttest zu überprüfen.* Dies ist ein Ansatz, der neue Tests ermöglicht, aber auch neue Probleme verursacht, auf die in Abschnitt 4.2.3 eingegangen wird.

Die Art und Weise wie Komponenten bei einem Kontexttest benutzt werden, ist nicht immer gleich. Der Kontext einer Komponente leitet sich unterschiedlich ab:

- Eine Guideline kann ein einzelnes Element isoliert betreffen, wie z.B. ein Fenster, einen Schieberegler oder einen Button. Wichtig ist, dass sich der Kontext nur durch den Typ der Komponente ableitet. Solche Guidelines beschreiben Eigenschaften, die ein Element aufweisen muss, wenn es dem Benutzer präsentiert

wird. Ein Dialog muss z.B. immer einen Titel haben, um herauszustellen welchen Zweck er erfüllt [Sun01]. Ist kein Titel gesetzt, so muss ein entsprechender Kontexttest diesen Mangel erkennen und den Entwickler darauf hinweisen.

- Eine Guideline kann sich auf eine Gruppe von Komponenten und deren Zusammenspiel beziehen. Betrachtet man als Beispiel die Menüstruktur eines Hauptfensters, so ist eine recht offensichtliche Regel, dass in einem Menü nicht auch ein Menüeintrag mit demselben Namen existieren darf [Sun01]. Diese Regel stellt eine Gruppenbeziehung dar, wo *ein* Menü mit allen *n* Menüeinträgen zu überprüfen ist.

Setzt man Kontexttests ein, so folgt für den Entwicklungsprozess, dass ein Teil der Tests dem Test-First Ansatz entspricht. Fehlen für eine Komponente Daten, so wird dies bemängelt und die Entwickler können Stück für Stück die Tests erfüllen. Zusätzlich gibt es Tests, die erst nachträglich sichtbar werden. Beschreibt ein Usability Unit Test z.B. ein Verbot, dann schlägt dieser Test erst fehl, wenn die Regel verletzt wird. D.h. es kann ein Zustand existieren, wo alle Tests erfüllt sind und erst durch weitere Änderungen ein Fehlschlag entsteht.

4.2.2. Testaspekte und Grenzen

Das Schreiben von Usability Unit Tests mag zunächst Furcht einflößend sein, da die Guidelines auf einer sehr menschlichen Ebene geschrieben sind. Manche Dinge werden daher einfach zu testen sein, manche Dinge sehr schwierig. Aber: *„Es ist unmöglich, absolut alles zu testen, wenn die Tests nicht genauso kompliziert und fehleranfällig wie der Code werden sollen. Es ist Selbstmord, nichts zu testen (im Sinne isolierter automatisierter Tests). Was soll man also von all dem testen, was man theoretisch testen könnte? Man sollte all das testen, in das sich Fehler einschleichen können.“* [Beck03, S. 116]

Niemand kann alle Usability Regeln testen! Schon allein aufgrund der Tatsache, dass man nicht die Zeit hat, aber auch weil es automatisiert einfach zu kompliziert wäre. Usability handelt von Menschen, die denken und fühlen. Dabei ist eine mitunter einfache Formulierung in Guidelines nur schwer in Code umzusetzen. Hier sollen zwei Beispiele zeigen, wo die Grenzen der Testbarkeit liegen.

Nielsen beschreibt in [Nielsen93] zehn Regeln, die in jedem Fall einzuhalten sind

„Simple and natural dialogue: Dialogues should not contain information that is irrelevant or rarely needed. Every extra unit of information in a dialogue competes with the relevant unit of information and diminishes their relative visibility. All information should appear in a natural and logical order.“ [Nielsen93]

Unter dem Gesichtspunkt der Testbarkeit folgt für die beiden Kernaussagen dieser Regel:

1. Dialoge sollen keine Informationen enthalten, die unwichtig sind oder nur selten benötigt werden. Dieser Aspekt ist nicht testbar, weil der Entscheidungsprozess

fehlt, der festlegt welche Informationen letztendlich unwichtig sind. Diesen kann nur ein Mensch übernehmen. Den Sinn und Unsinn von Inhalten zu Hinterfragen ist somit nicht möglich.

2. Alle Informationen in Dialogen sollen in einer natürlichen und logischen Reihenfolge präsentiert werden, was zunächst auch als nicht testbar wirkt. Nur ein Mensch kann sagen, was eine natürlich Reihenfolge von Informationen ist. Doch werden diese Informationen durch Komponenten einer Oberflächenbibliothek dargestellt und es entsteht z.T. Testbarkeit. So hat jedes Land eine bestimmte Leserichtung, in den westlichen Ländern ist dies von links nach rechts, von oben nach unten. Kommt man aus einem dieser Länder und betrachtet ein Eingabefeld für Text, so würde man die Beschreibung der einzugebenden Information links erwarten [Sun01]. Diese Positionierung ist testbar. Weiterhin gibt es für alle Eingabekomponenten eine *„tab-traversal-order“*. Diese besagt, wie durch alle Komponenten in einem Dialog zu springen ist, wenn die Tab-Taste gedrückt wird. Als Regel muss diese Reihenfolge mit der Leserichtung des Landes übereinstimmen [Sun01]. Auch dieser Aspekt ist testbar.

Das letzte Beispiel sollte verdeutlichen, dass Interpretationen, wie sie nur ein Mensch vornehmen kann, nicht testbar sind. Aber manche Aspekte, seien sie noch so abstrakt geschrieben, lassen sich auf einen automatisierten Test abbilden. Eine andere Regel:

„Feedback: The system should always keep users informed about what is going on, through appropriate feedback within reasonable time.“ [Nielsen93]

Wird z.B. ein Forschrittsbalken eingeblendet oder der Mauszeiger ändert sich zu einer Sanduhr, erhält der Benutzer Rückmeldungen über den aktuellen Zustand eines Systems. Hier ist schwer zu testen, in welcher Situation Rückmeldungen überhaupt angezeigt werden müssen. Es lässt sich nicht realisieren alle Rechenoperationen im System zu simulieren und beim Überschreiten einer Messzeit einen Test fehlschlagen zu lassen, wenn dem Benutzer nichts angezeigt wird. Nicht nur, dass dieser Aspekt vom Zielsystem abhängen könnte, er ist auch zu komplex, um ihn automatisiert abbilden zu können. Das Testen auf Regeln aus Guidelines entspricht niemals der Simulation eines Benutzers.

Es gibt noch eine Vielzahl anderer Aspekte, die sich nicht automatisiert überprüfen lassen. Als Grundregel sind jedoch interpretations- und systemabhängige Aspekte nur bedingt testbar. Die zehn Regeln [Nielsen93] beschreiben eine recht allgemeine Auswahl von Heuristiken. Nur wenn man sie verfeinert und Konsequenzen zieht, erhält man manchmal Testaspekte. Solche Ableitungen stehen oft in Guidelines, in denen schon nach Komponenten aufgeteilt wurde und Eigenschaften sehr viel feiner beschrieben werden.

4.2.3. Zustände und Konstruktionsprozesse

Die Forderung alles so zu testen, wie es der Benutzer auf dem Bildschirm sieht, bringt zwei Probleme mit sich. Zum einen kann sich die Oberfläche zur Laufzeit ändern, da

sie nicht statisch ist. In modernen Applikationen lassen sich Elemente frei verschieben, sogar ein- und ausblenden. Das zweite Problem ist, dass Fenster die Wurzel in dem Baum der Komponenten sind. Nun können Fenster jedoch Fenster öffnen, wobei die Quelle des neuen Fenster unbekannt bleibt. Für Kontexttests ergibt sich daraus folgende Wunschvorstellung:

Alle Instanzen von Komponenten des GUI, die jemals zur Laufzeit auftreten, sind in allen Zuständen bekannt und zugänglich.

Dieser Wunsch ist nur schwer in die Realität umzusetzen. Das Problem ist der Prozess, wie Fenster und Zustände entstehen. Sie sind *irgendwo* im Code *irgendwie* hinterlegt. Man müsste zunächst die Stelle einer solchen Änderung erreichen und anschließend ausführen. Welche Daten dazu bereitgestellt werden müssen, weiß ein automatisiertes System nicht, weil eine auswertbare Struktur evtl. fehlt. Dieser Zustand ist für das Schreiben von Kontexttests sehr unbefriedigend.

Eine Regel lautet z.B. *„avoid more than two levels of dialog boxes.“* [Johnson00]. Um auf diese zu testen, müssten alle erzeugten Dialoge bekannt sein. Ein Benutzer würde einen neuen Dialog z.B. durch Drücken eines Buttons erhalten. Ein automatisiertes System kennt diesen Button jedoch nicht. Deshalb muss der Usability Layer die Möglichkeit anbieten, dass erzeugte Dialoge identifizier- und nachbaubar sind. Einmal um Dialoge in einen Kontext einzuordnen, damit Regeln testbar werden, zum Anderen um auch möglichst alle Fenster, die jemals entstehen, zu testen.

Für Zustände ist ein ähnlicher Schluss möglich, doch entsteht hier noch ein weiteres Problem, jenseits der Erreichbarkeit von Komponenten. Es gibt Regeln, die Zustandsänderungen beschreiben. Z.B. darf sich die Menüleiste nicht ändern, egal in welchem Zustand sich die Applikation befindet [Johnson00]. Nur ganze Menüs können entfernt werden, einzelne Menüeinträge niemals - sie sind bei Inaktivität nur auszugrauen. Um dies zu testen, müsste überprüft werden, ob an irgendeiner Stelle im Code eine Löschoperation genutzt wird. Man könnte die Idee haben, die Methode `remove` in den Menüs zu überschreiben, so dass sie nicht mehr verfügbar ist. Doch bringt solch eine Änderung neue Probleme mit sich, auf die in Abschnitt 5.5 eingegangen wird.

4.3. Konsequenzen für den Usability Layer

Der Usability Layer ist eine Erweiterung von Swing mit zwei Zielen:

- Ermöglichen von Kontexttests und erhöhen der Testbarkeit auf Usability.
- Erfüllen von Guidelines im Usability Layer, soweit es durch Implementierung möglich ist. Dies ist der Fall, wenn eine Usability Regel durch einen Komponententest beschrieben wird. Manchmal lassen sich Fehler schon vorher ausschließen, weil sie für alle Komponenten gleichermaßen gelten.

4.3.1. Testbarkeit herstellen

Die Sichtweise des Benutzers soll bei den Tests nachgebildet werden. Dabei liegt das GUI intern als eine Baumstruktur vor, aus dem die Darstellung auf dem Bildschirm gewonnen wird. Diese Baumstruktur ist jedoch nicht immer einheitlich (siehe Abschnitt 2.3.2). Deshalb muss im Usability Layer jede Komponente aus Swing um eine Schnittstelle erweitert werden, so dass eine automatisierte Auswertung möglich wird.

Zusätzlich ist die in Abschnitt 4.2.3 vorgestellte Problematik der Fenster und Zustände zu lösen. Die Baumstruktur darf nicht enden, wenn ein Blatt noch ein weiteres Fenster erzeugt.

Jede Komponente hat durch sein Aussehen und die Funktion eine spezielle semantische Einordnung. Swing bildet schon einen sehr mächtigen Grundstock, doch sind manche Komponenten noch nicht direkt auf Usability testbar. Diese müssen dann im Usability Layer erweitert werden. Dabei kann es sein, dass neue Funktionalitäten zu schreiben sind oder einfach nur die Semantik verfeinert werden muss. So hat jeder Dialog z.B. einen Button, mit dem sich der Dialog ohne Auswirkungen auf das Model schließen lässt. Dieser muss von außen eindeutig identifizierbar sein.

4.3.2. Usability Guidelines integrieren

Java ist ein Vorreiter für Systemunabhängigkeit. Es gibt die *Virtual Machine* für eine Vielzahl von Betriebssystemen. Auch Swing wurde so konzipiert, dass es überall einsetzbar ist [Fowler]. Dabei spielt selbst die Auflösung und Farbanzahl keine Rolle, wodurch `LayoutManager` auf fast jedem System unterschiedliche Ergebnisse liefern.

Beschreiben Usability Regeln bestimmte Abstände und Positionierungen, so entsteht immer ein Komponententest. Deshalb lassen sich für den Usability Layer `LayoutManager` schreiben, die spezifizierte Vorgaben erfüllen. Abschnitt 5.3.1 zeigt ein Beispiel, wo für Dialoge ein `LayoutManager` für Buttons vorgestellt wird.

5. Realisierung

5.1. Vorgehen beim Erfüllen von Guidelines

5.1.1. Abstract Test Sheets

Regeln aus Guidelines sind zunächst systematisch zu sammeln, bevor sie in Tests überführt werden können. Die Formulierung einer Regel ist dabei *abstrakt* und nicht direkt im Code verwendbar. Wie in Abschnitt 4.2.2 bereits gezeigt, gibt es auch Regeln, die nicht automatisiert testbar sind. Folglich beginnt man verschiedene Guidelines zu durchsuchen und schreibt potentielle Testkandidaten nieder. Wichtig ist auch die Angabe der Quelle zur besseren Nachvollziehbarkeit.

Hat man eine Sammlung aufgebaut, müssen die entsprechenden Komponenten aus dem Usability Layer identifiziert werden. Betrachtet man z.B. die *mnemonics*[1] im Menü einer Applikation, so darf das *mnemonic* eines Menüs nicht auch von einem Menüeintrag verwendet werden [Sun01]. Diese Regel bezieht sich auf die Klassen `JMenu` und `JMenuItem`.

Wurden für alle Regeln die entsprechenden Komponenten identifiziert, folgt eine Unterteilung in zwei verschiedene Rollen. Zunächst gibt es immer einen *Betroffenen*, also eine Klasse, für die der Test geschrieben werden muss. Sieht man in der Baumstruktur des GUI eine Komponente und ihre Klasse ist Betroffener, so müssen alle hinterlegten Regeln geprüft werden. Ein Beispiel:
Sei die Regel, dass *jedes primäre Fenster einen Titel hat* gegeben [Sun01]. Dann ist die *betroffene* Komponente vom Typ `JFrame`. Ein Mensch würde diese Regel prüfen, indem er für jedes ihm sichtbare primäre Fenster nach einem vorhandenen Titel schaut. Ein automatisiertes System würde dasselbe über eine Schnittstelle prüfen, sobald es einen `JFrame` erhält.

Ebenfalls identifiziert durch ihre Klassen, bilden weitere *Beteiligte* die andere Rolle. Hier bezieht eine Regel andere Komponenten ein, um Beziehungen zwischen ihnen aufzubauen. Doch ist die Auswahl der Rollen mit großer Sorgfalt durchzuführen. Man trifft schon eine erste Entscheidung über die Implementierung. Für das obige Beispiel mit der Einzigartigkeit der *mnemonics* könnte gewählt werden:

- Sei das `JMenu` der *Betroffene*, dann müsste die Heuristik überprüft werden, sobald man ein `JMenu` „sieht“. Von dort aus ist ein Abgleich mit den *Beteiligten*, also allen Menüeintragen vom Typ `JMenuItem` durchzuführen. Sie müssten durch-

[1] mnemonic: „Eselsbrücken“, die bei einer Komponente eine Alternative zur Auswahl mit der Tastatur anbieten. Der entsprechende Buchstabe wird im Namen der Komponente unterstrichen.

laufen werden, um das *mnemonic* eines jeden Eintrags mit dem des Menüs zu vergleichen.

- Alternativ können die Rollen auch vertauscht werden. Seien die *Betroffenen* vom Typ `JMenuItem`. Dann müsste die Regel erst geprüft werden, wenn jeder Eintrag nach Öffnen des Menüs „sichtbar“ wird. Durch Auswertung der Elternkomponente ist noch die Beziehung vom `JMenuItem` zum beteiligtem `JMenu` herzustellen und die *mnemonics* können ohne Durchlauf verglichen werden.

Eine angemessene Dokumentation dieser Informationen lässt sich durch die in dieser Arbeit entwickelten *Abstract Test Sheets* realisieren. Sie stellen tabellarisch die eben genannten Beziehungen dar und formen die Sammlung der zu implementierenden Regeln. Dabei bilden die Betroffenen zweckmäßigerweise die erste Spalte, um schnell zu überblicken, welche Regeln für eine Komponente überprüft werden. Danach folgen die Beteiligten und schließlich die Regel selbst. Abbildung 5.1 zeigt ein Abstract Test Sheet.

Betroffene Klasse	*Beteiligte* Klassen	Usability Regel
`JMenu`	`JMenuItem`	Die *mnemonics* aller Menüeinträge sind in einem Menü einzigartig. [Sun01]

Abbildung 5.1.: Ein Abstract Test Sheet

Ein Abstract Test Sheet ist ein wichtiges Werkzeug vor der Implementierung eines Tests. Seine Hauptfunktionen sind:

- Sammeln von Regeln, die in automatisierte Tests umzusetzen sind, und Angabe der Quelle.
- Zuordnen von Regeln zu Komponenten aus dem Usability Layer.
- Herstellen von Beziehungen zwischen verschiedenen Komponenten.

Im Anhang A befinden sich die Abstract Test Sheets zu allen Regeln, die im Laufe dieser Arbeit als Test implementiert wurden.

5.1.2. Kontexttests

Abschnitt 4.2 hat bereits gezeigt, dass Usability Unit Tests entweder Komponententests oder Kontexttests sind. Dabei stützt sich ein Kontexttest darauf, dass in ihm ein Testobjekt von außen gesetzt wird. Dazu wurde in dieser Arbeit die Klasse `TestCase` durch eine Unterklasse `UsabilityTestCase` erweitert, um das Testobjekt zu referenzieren. Ein Kontexttext wäre z.B.:

```
public void DemoTest extends UsabilityTestCase {
  public void testX() {
    assertTrue(getTestObject().getX() > 10);
  }
}
```

Hier wird überprüft, ob die x-Koordinate des Testobjekts größer als `10` ist. Wichtig ist, dass Kontexttests alleine nicht lebensfähig sind. Sie würden bei direkter Ausführung irgendwann eine `NullPointerException` werfen. Man benötigt immer einen äußeren Mechanismus, der jeder Komponente einen `UsabilityTestCase` zuordnet und das Testobjekt setzt. In Abschnitt 5.2.1 wird eine Klasse mit dieser Funktionalität vorgestellt.

5.1.3. Umsetzen von Abstract Test Sheets

Existiert durch *Abstract Test Sheets* eine Sammlung von Regeln, müssen die dort hinterlegten Informationen jetzt in Code umgesetzt werden. Dabei haben sich drei Arbeitsschritte herauskristallisiert, die nacheinander zu durchlaufen sind:

1. Prüfen, ob die Regel durch Swing schon erfüllt wird. Zur Vereinfachung werden diese Regeln nicht weitergehend dokumentiert oder näher behandelt.

   Beispiel: *Buttons sollen sich sichtlich ändern, wenn mit ihnen interagiert wird* [Johnson00]. Wird in Swing ein Button gedrückt, so sieht es aus, als ob er wirklich runtergedrückt wurde. Bei manchem *Look&Feel* ändert sich der Button sogar, wenn man mit dem Mauszeiger über ihn fährt.

2. Schreiben eines Tests, der prüft, ob eine Regel verletzt wird. Dies kann ein Komponenten- oder Kontexttest sein.

   Beipiel (Kontexttest): *Jedes primäre Fenster hat einen Titel.* [Sun01] Wurde das aktuelle primäre Fenster als Testobjekt gesetzt, dann bietet es sich an zu prüfen, ob der Titel des Fensters nicht `null` ist:

   ```
   assertNotNull(getTestObject().getTitle());
   ```

   Jetzt ist sichergestellt, dass ein Titel gesetzt ist. Doch reicht diese Überprüfung noch nicht aus, weil ein `String` auch leer sein kann. Man erweitert den Test um:

   ```
   assertTrue(getTestObject().getTitle().length() > 1);
   ```

   Bis diese Regel nahezu vollständig getestet ist, benötigt man noch weitere Überprüfungen, auf die hier nicht näher eingegangen wird.

   Beispiel (Komponententest): *Die Kontrollbuttons in Dialogen sollen vom Inhalt sichtlich getrennt sein.* [Johnson00] Diese Regel endet in einem Komponententest. Am Anfang prüft man die Positionen von Buttons und die Abstände zum

Inhalt. Die Frage, was der Inhalt dieses Tests ist, lässt sich zu diesem Zeitpunkt noch nicht beantworten. Aber Komponententests haben die Eigenschaften, dass sie etwas für einen dritten Schritt spezifizieren.

3. Wurde zum Prüfen einer Regel ein Komponententest verfasst, so ist der Usability Layer solange zu erweitern bis dieser Test zu 100% fehlerfrei läuft.

   Beispiel: *Die Kontrollbuttons in Dialogen sollen vom Inhalt sichtlich getrennt sein.* [Johnson00] Wie oben bereits erwähnt, ist es schwierig den eigentlichen Inhalt eines Dialogs festzustellen. Dieser Mangel lässt sich jedoch beheben, indem man den `ContentPane` des Dialogs auftrennt. Fügt man einen neuen *Pane* für die Inhalte und einen neuen *Pane* für die Buttons ein, wurde die Struktur getrennt. Eine sichtliche Trennung lässt sich dann durch `LayoutManager` erreichen. Eine Regel existierte zunächst als Komponententests, wenn auch vage. Durch die Implementierung im Layer wurde sie erfüllt.

Zusammenfassend zeigt Abbildung 5.2 den Umsetzungsprozess einer Guideline in Tests.

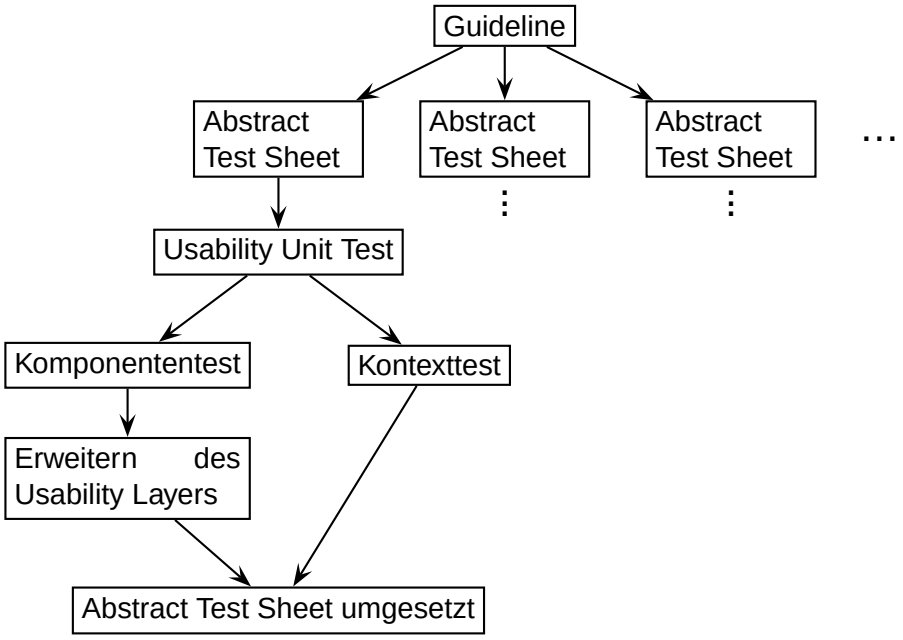

Abbildung 5.2.: Umsetzen einer Guideline in Tests

5.2. Ermöglichen von Kontexttests

Wie bereits in Abschnitt 5.1.2 erwähnt, sind Kontexttests alleine nicht lebensfähig. In dem Paket `ul.test` wird alles zur Verfügung gestellt, um diese spezielle Art des Testens zu realisieren. Der Usability Layer wird durch das Paket `ul.layer` repräsentiert, in dem alle Komponenten aus Swing in einer erweiterten Form vorliegen. Dabei wurde der erste Buchstabe von allen Komponenten von `J` nach `U` abgeändert, um die Erweiterungen zu verdeutlichen. Ein `JFrame` heißt jetzt entsprechend `UFrame`.

Die Klasse `UsabilityTestCollector` aus dem Paket `ul.test` bildet das Herz der Kontexttests, indem sie drei wichtige Funktionen bereitstellt:

1. Alle Komponenten des GUI müssen erreicht werden. Ein Mensch sieht diese, indem er mit dem System interagiert. Ein automatisiertes System muss eine ähnliche Auswertung durchführen.
2. Abbilden einer Komponente auf alle hinterlegten Kontexttests. Sieht ein Mensch eine Komponente, so würde er prüfen, ob alle zugeordneten Regeln erfüllt werden.
3. Setzen des Testobjekts in allen zugeordneten Kontexttests.

5.2.1. Sammeln von Kontexttests für Komponenten

Der `UsabilityTestCollector` verwaltet intern eine `Map`, um Komponenten auf Kontexttests abzubilden. Der *key* der `Map` ist dabei die Klasse einer Komponente aus dem Usability Layer. Das *value* wird durch eine Liste repräsentiert, die Kontexttests enthält. Während der Konstruktion eines Sammlers, wird die `Map` mit einem Datensatz gefüllt, der die bereits implementierten Kontexttests auf die jeweiligen Komponenten abbildet. Eigene Tests können nachträglich hinzugefügt oder entfernt werden.

In Abschnitt 4.2.1 wurde vorgestellt, dass ein Sammler die baumähnlichen Datenstruktur der `Container` auszuwerten hat. Dabei arbeitet er ähnlich wie ein Sieb, indem rekursiv alle Kinder eines `Container` erneut an ihn übergeben werden. In jedem Iterationschritt werden neue Komponenten gefunden, bis das gesamte GUI abgearbeitet wurde.

Die Schnittstelle `collectTests(Container testObject)` ermöglicht einen Sammelprozess für eine Wurzel (i.d.R. ein Fenster) zu starten und so lange zu durchlaufen bis nur noch Blätter vorliegen. Als Produkt entsteht eine `UsabilityTestSuite`[2], die Kontexttests für alle Komponenten ausgehend von der Wurzel enthält.

Abbildung 5.3 illustriert die vier durchgeführten Schritte:

1. Abbildung aller neuen Komponenten auf hinterlegte Kontexttests vom Typ `UsabilityTestCase` und setzen der Komponente als Testobjekt.

[2] `UsabilityTestSuite` ist eine Erweiterung der Klasse `TestSuite`. Genauere Informationen können der JavaDoc-Dokumentation zu dieser Arbeit entnommen werden.

2. Hinzufügen von allen neuen Tests aus Schritt 1. zu einer `UsabilityTestSuite` *t*.

3. Durch alle Kinder der neuen Komponenten aus Schritt 1. iterieren.

4. Übergeben jeder neu erreichten Komponente aus Schritt 3. an den `UsabilityTestCollector` - der Prozess startet rekursiv wieder bei Schritt 1.

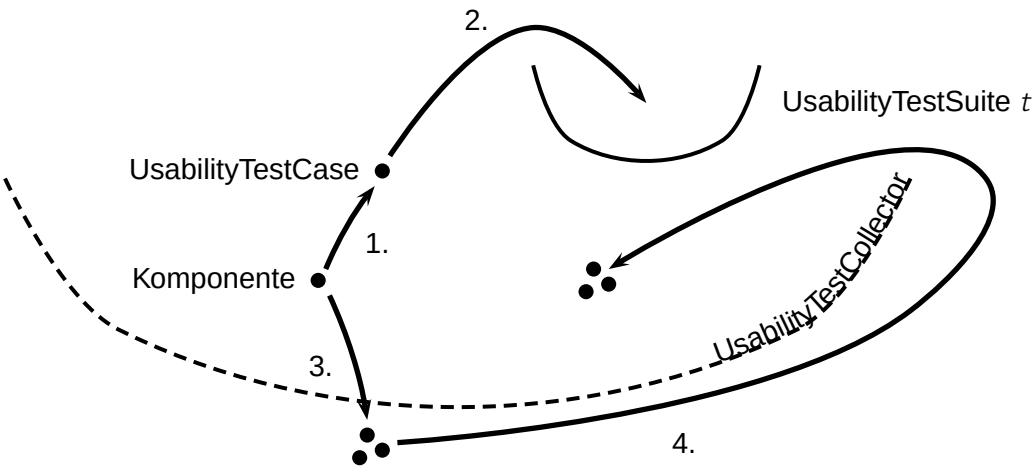

Abbildung 5.3.: Sammelprozess für alle Komponenten

Swing verhindert beim Hinzufügen von Komponenten zu Komponenten schon effektiv Schleifen, d.h. einer Komponente kann keine Komponente hinzugefügt werden, die bereits im Baum existiert. Dieser Sonderfall ist somit zu vernachlässigen.

Zum Auslesen der Kinder einer Komponente existiert nach Abschnitt 2.3.2 keine einheitliche Schnittstelle. Bei einem `UPanel` sind die Kinder einfach alle direkten Kinder, die man durch Aufruf von `getComponents()` erhält. Bei einem `UFrame` hingegen bestehen die Kinder aus den Komponenten im `ContentPane` und dem `UMenuBar`. Deshalb wird das Interface `UsabilityContainer` von allen Komponenten im Usability Layer implementiert und die Methode `getAllChildren()` stellt die Kinder als Array zur Verfügung.

5.2.2. Verfolgen der Dialogerzeugung

Abschnitt 4.2.3 hat bereits gezeigt, dass ein GUI zur Laufzeit nicht statisch ist. Durch die Interaktion des Benutzers können neue Dialoge und Zustände entstehen. Besonders problematisch ist, dass diese Änderungen erst nach Ausführung einer Aktion sichtbar werden. Ein automatisiertes System kann nicht immer die richtigen Daten zur Verfügung stellen, um einen Übergang zu erzwingen, oder die Konsequenzen einer Interaktion kennen. Der Sammelprozess von Kontexttests betrifft somit immer einen

ganz bestimmten Zustand. Den Entwicklern ist frei gestellt, ob sie ein GUI ändern und anschließend erneut Tests sammeln.

Ein vereinfachtes Modell für die Verfolgung von Dialogen, die zur Laufzeit entstehen, ist dennoch zweckdienlich, weil auch neue Regeln testbar werden. Interessant sind jedoch nicht die Standarddialoge [Johnson00], wie z.B. ein Dialog zur Auswahl einer Datei. Wichtig sind Dialoge, die vom Entwickler realisiert wurden, z.B. ein Optionsdialog oder ein Dialog mit weiteren Abfragen vor einer Einfügeoperation.

In dieser Arbeit entstand ein Modell, das einem Sammler bzw. Test die Identifikation von erzeugten Dialogen erlaubt. Unter Beschränkung auf die Benutzerinteraktionen mit sichtbaren Komponenten sind zwei Eigenschaften von Bedeutung:

- Nur einige Komponenten können ihrer Semantik nach Dialoge öffnen. Deshalb sind diese auszuwählen und gezielt um das angestrebte Modell zu erweitern. In dieser Arbeit wurden die Klassen `UButton`, `UMenuItem` und `UTextField` ausgewählt, weil der Benutzer durch einen Klick oder das Betätigen der Enter-Taste etwas vom System fordert. Ein Dialog wäre nur eine mögliche und erwartete Antwort.
- Die Anzahl der Dialoge, die eine Komponente erzeugt, ist nicht immer bekannt. Sie kann für verschiedene Zustände im *Model* variieren. Ein Entwickler sollte sich jedoch darum bemühen, dass eine Komponente immer den selben Dialog öffnet. Andernfalls würde er die Nachvollziehbarkeit der auszuführenden Aktion erschweren. *„When software changes too much on its own initiative, users become disoriented and annoyed.“* [Johnson00]

Interagiert ein Benutzer mit Instanzen eines `UButton`, `UMenuItem` oder `UTextField`, wird eine hinterlegte Aktion ausgeführt. Bei allen registrierten Elementen vom Typ `ActionListener` wird die Methode `actionPerformed(...)` aufgerufen. Dabei ist einem Sammler nur die Methodensignatur bekannt, ob ein Dialog geöffnet wird, bleibt verborgen.

Zur Verbesserung der Nachvollziehbarkeit sind Aktionen in geeignete Teilschritte zu zerlegen. Das Einfügen einer Methode `createDialog(...)` würde erzeugte Dialoge explizit auszeichnen. Unter Verwendung des *Command Pattern* [Gamma96] sind Aktionen so zu kapseln, dass die einzelnen Teilschritte durch entsprechende Methoden repräsentiert werden. Eine anschließende Abarbeitung in ausgewählter Reihenfolge ermöglicht, dass die auszuführende Aktion gleich bleibt. Dabei ist einem Pendant zur Methode `actionPerformed(...)` der erzeugte Dialog zu übergeben. Abbildung 5.4 zeigt diesen Ansatz in UML.

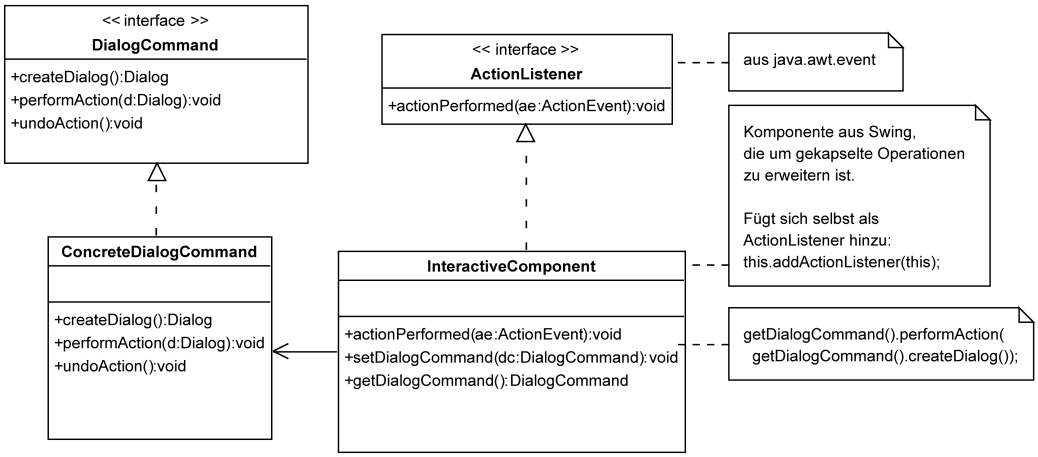

Abbildung 5.4.: Kapselung von Aktionen mit dem Command Pattern

Einem automatisiertem System wird durch die vorgestellten Änderungen erlaubt einzelne Schritte einer Aktion gezielt auszuführen. Der Sammler und jeder Test kann durch Auswertung der Methode `createDialog(...)` Dialoge nachbauen. Der entsprechende Baum zu einem Fenster endet nicht, wenn alle Komponenten erreicht wurden. Entsteht ein Wert, der von `null` verschieden ist, kann auch dieser Dialog rekursiv getestet werden. Zusätzlich werden auch neue Tests möglich und man kann z.B. prüfen, ob sich zuviele Dialoge übereinander aufbauen: *„Avoid more than two levels of dialog boxes.“* [Johnson00]

Nachteil dieser Änderung ist, dass das System komplexer wird, weil für jeden `UButton`, jedes `UMenuItem` und jedes `UTextField` eine gekapselte Aktion geschrieben werden muss. Weiterhin ist im `UsabilityTestCollector` jetzt auf Schleifen zu achten, wenn z.B. ein Button in einem Dialog den selben Dialog noch mal erzeugt. Irgendwann würde jedoch die Regel verletzt, dass die Hierarchie der Dialoge nicht zu tief werden darf (siehe oben). Erweitert man den Sammler so, dass diese Regel implizit geprüft wird, werden Schleifen effektiv verhindert.

Ein ähnliches Modell für die Problematik der Zustände zu finden ist sehr viel schwieriger. Ein Zustandswechsel kann z.B. sein, dass ein Fenster nachträglich dekoriert wird, aber auch, dass das Model seinen internen Zustand ändert und damit die View beeinflusst. Aufgrund der Komplexität und mangelnden Beschreibbarkeit wird hier kein Modell zum Verfolgen von Zuständen angeboten. Möchten die Entwickler einzelne Zustände testen, ist die Applkation in diese zu überführen und dann ein neuer Sammelprozess zu starten.

Abschließend sei angemerkt, dass eine Berücksichtigung von Zuständen auch den Testvorgang an sich erschwert. Man müsste Buch führen darüber, welcher Test gerade gültig ist und welcher nicht. Weiterhin müsste der Zustand für den Entwickler immer ersichtlich sein, um im Fehlerfall die Komponente zu identifizieren, die gegen eine Regel verstößt. Der folgende Abschnitt wird zeigen, dass schon bei einem Ansatz ohne Zustände die Nachvollziehbarkeit eine wichtige Rolle spielt.

5.2.3. Rückmeldungen an die Entwickler

Wird ein Komponententest geschrieben, dann wird ein konkretes Testobjekt im Test selbst erzeugt und anschließend getestet. Schlägt dieser Test fehl, weiß der Entwickler genau was getestet wurde und er kann die Aussagen des Tests nachvollziehen. Schreibt man hingegen Kontexttests ist zunächst die verletzte Regel aussagekräftig anzugeben.

Um die Ausgabe von Fehlermeldungen zu ermöglichen kann jeder *assert*-Methode eines `TestCase` zusätzlich ein `String` übergeben werden [Hamill04]. Die Klasse `UsabilityTestCase` ermöglicht über die Methode `createGuideline(String key)` hinterlegte Fehlermeldungen an *assert*-Methoden weiterzureichen. Sei ein `UFrame` als Testobjekt bekannt, dann würde eine Testzeile z.B. wie folgt lauten:

```
assertNotNull(createGuideline("UFrame.TitleMissing"),
              getTestObject().getTitle());
```

Schlägt der Test fehl, erhält der Entwickler als Fehlermeldung, dass jeder „Frame“ einen Titel benötigt. Dazu liegen die Regeln selbst in einer *property*-Datei[3], welche als `ResourceBundle` eingebunden wird. Als Konvention empfiehlt sich für den *key* zunächst die *betroffene* Klasse zu nennen und anschließend eine sehr kurze Beschreibung der verletzten Regel. Insgesamt sind durch diesen Ansatz die Aussagen von Regeln leichter änder- und lokalisierbar.

Aufgrund der Beschaffenheit von Kontexttests entsteht auch ein Problem. Der Entwickler weiß nie, was gerade getestet wurde. Er kann den eingeschlagenen Weg im Baum nicht nachvollziehen. Zwar gibt die Fehlermeldung Hinweise auf den Typ der Komponente, doch ist die Komponente selbst nicht bekannt. Die Fehlersuche wird dadurch stark erschwert.

Zur Lösung dieses Problems, muss der gesamte Sammelprozess bekannt sein. Man könnte zwar die Methode `getParent()` vom Testobjekt aus solange verfolgen, bis eine Wurzel erreicht wird, doch liefern neu erzeugte Fenster keine Elternkomponente. Manche Komponenten werden sogar intern umgewandelt und das Ergebnis wäre kaum nachvollziehbar. Die Abbildung muss demnach der Vorstellung des Entwicklers entsprechen. Neben der Echtheit von Komponenten, muss die Art der Herkunft einer Komponente sichtbar sein. Abbildung 5.5 zeigt einen gewünschten Baum.

Der Sammler erzeugt für jede Komponente, die er während des Sammelprozesses erreicht einen eigenen `UsabilityTestCase`. Kennt dieser Test seinen zugehörigen Knoten im Baum aus Abbildung 5.5, lässt sich die Herkunft einer Komponente durch Auswertung aller übergeordneten Knoten bestimmen. Durch diesen *Stammbaum* ist bei Fehlschlag eines Tests das Testobjekt immer eindeutig identifizierbar.

Zum Nachbau des Stammbaums wird das *Composite Pattern* [Gamma96] verwendet. Die Klasse `HistoryNode` repräsentiert dabei einen Knoten, dem weitere Knoten als Kinder hinzugefügt werden. Dieser Baum bildet einen eigenen Datensatz, unab-

[3] konkret UsabilityGuidelines.properties

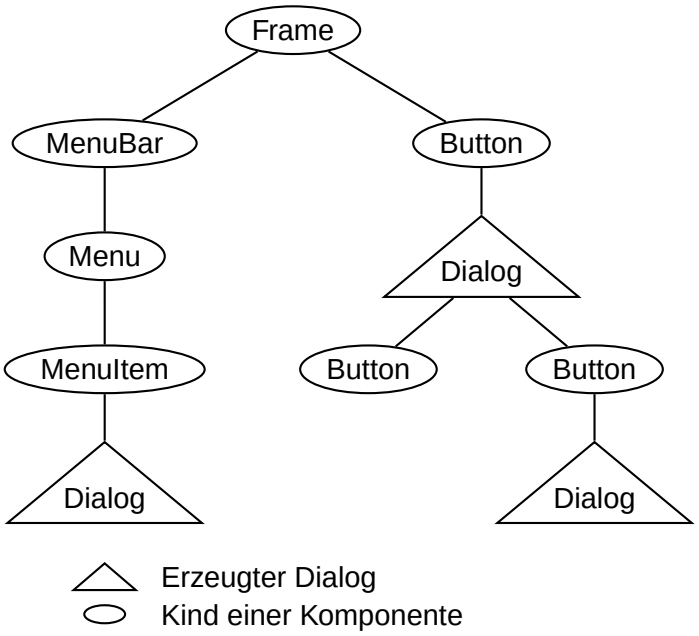

Abbildung 5.5.: Ein einheitlicher Baum eines GUI

hängig vom `Container`-Aufbau der Komponenten und stellt durch die Knoten selbst verschiedene Übergänge dar. Abbildung 5.6 zeigt diesen Ansatz in UML.

Abbildung 5.6.: Realisierung eines Stammbaums in UML

Zur Repräsentation der Art der Herkunft werden für jeden Knoten eigene Klassen verwendet, die `HistoryNode` implementieren. So existiert eine Klasse für die Wurzel,

eine für einen Übergang, wenn ein direktes Kind gefunden wurde, und eine für erzeugte Dialog. Dadurch lässt sich für jeden Knoten eine Geschichte nachbauen, die den im Baum eingeschlagenen Weg aufzeigt. Als Operation im *Composite* existiert die Methode `composeHistory()`, die eine lineare Projektion als `History` erstellt.

Schlägt ein Test fehl, kann durch Auswertung der `History` festgestellt werden, welche Komponente genau betroffen ist. Die derzeitige Implementierung ist sehr einfach gehalten und liefert als Ergebnis einen String, der den im Baum eingeschlagenden Weg linear darstellt.

5.2.4. Durchlaufstrategien als Folge aus Abstract Test Sheets

Ist auf Usability zu testen, genügt es nicht immer jede Komponente einzeln und isoliert zu betrachten. Sobald im Abstract Test Sheet ein Beteiligter gesetzt ist, sind Beziehungen untereinander auszuwerten. Ist z.B. zu prüfen, ob die Namen aller Menüeinträge in einem Menü einzigartig sind [Sun01], können die Einträge nicht als Betroffene gesetzt werden. Denn ein `UMenuItem` kennt nur sich selbst, so dass `UMenu` als Betroffener zu setzen ist. Dabei entsteht ein Durchlauf, wo m Kinder mit n anderen Kindern auf gleicher Ebene verglichen werden.

Dies ist nur ein Beispiel für eine gleichbleibende Struktur. In einem Durchlauf wird ein festes Element, die *Referenz*, mit einem *durchlaufendem* Element verglichen. Kapselt man diesen Vergleich nach dem *Command Pattern* [Gamma96] in einem Objekt, so lassen sich einheitlich Regeln spezifizieren. Sei dazu das *Interface* `Rule` so gegeben, dass es in einer Methode `check(...)` den Wert `true` liefert, wenn eine Regel verletzt wird. Für das Beispiel mit den Menüeinträgen folgt:

```
public class ItemUniqueRule implements Rule {

  ...

  public boolean check(Component reference, Component current) {
    UMenuItem ref = (UMenuItem) reference;
    UMenuItem cur = (UMenuItem) current;
    return ref.getText().equals(cur.getText());
  }
}
```

Hier fehlt noch der Durchlauf, welcher diese Regel für alle Menüeinträge prüft. Dabei gibt es sehr viele nützliche Varianten, so dass eine ganze Familie von Durchlaufstrategien entsteht. Realisiert man dieses Prinzip über das *Strategy Pattern* [Gamma96], repräsentiert eine Klasse selbst, wie letztendlich verglichen wird. Von außen ist nur eine Regel anzugeben und der Betroffene. Abbildung 5.7 zeigt die Realisierung in UML.

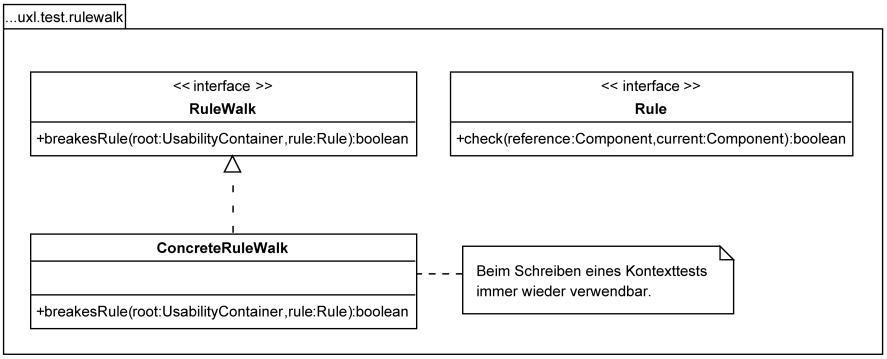

Abbildung 5.7.: Durchläufe realisiert als Strategien in UML

In dieser Arbeit werden drei unterschiedliche Durchlaufstrategien eingesetzt:

- Ein Durchlauf `RootNextNodeWalk`, der die übergebene Wurzel mit allen Kindern der Wurzel selbst prüft. *Anwendungsbeispiel:* Prüfen, ob der Name eines Menüs auch von einem Menüeintrag genutzt wird.
- Ein Durchlauf `NodeSameNodeWalk`, der alle Kinder der übergebenen Wurzel untereinander prüft. *Anwendungsbeispiel*: Prüfen, ob der Name eines Menüeintrags innerhalb eines Menüs einzigartig ist.
- Ein Durchlauf `NodeAllNodeWalk`, der alle Knoten unterhalb der übergebenen Wurzel untereinander prüft. *Anwendungsbeispiel*: Prüfen, ob die Aktion eines Menüeintrags innerhalb der gesamten Menüleiste einzigartig ist.

5.3. Erfüllen von Guidelines im Usability Layer

5.3.1. Ein LayoutManager für Buttons

Ist das Aussehen eines Fensters zu prüfen, erweist sich die freie Manipulierbarkeit als problematisch. Ein Fenster kann immer anders aussehen, z.B. bedingt durch das System, die Auflösung oder aktuelle Größe. `LayoutManager` übernehmen die Aufgabe einzelne Komponenten zu platzieren und weitergehend anzupassen (siehe Abschnitt 2.3.2). Testbarkeit im Kontext ist daher kaum möglich, so dass immer Komponententests entstehen, die sich im Usability Layer durch Programmieren erfüllen lassen.

In Abschnitt 5.1.3 wurde bereits auf die Regel eingegangen, dass in einem Dialog die Kontrollbuttons sichtlich vom Inhalt zu trennen sind. Weiterhin sollen *alle Kontrollbuttons in einem Dialog die Breite des Buttons haben, der den längsten Text aufweist.* [Sun01]. Ein entsprechender Komponententest lässt sich sehr schnell schreiben, indem man zunächst die maximale Breite bestimmt und anschließend alle Buttons mit diesem Wert vergleicht.

Würde man auf diese Eigenschaft im Kontext prüfen, muss das XP-Team einen entsprechenden `LayoutManager` für alle Dialoge schreiben. Aber er kann auch schon im Usability Layer selbst existieren, indem er letztendlich dasselbe tut, wie der Test. Nachdem die maximale Breite bestimmt wurde, ist sie in allen Buttons zu setzen. Die Klasse `ButtonPaneLayout` aus dem Paket `ul.layer` führt diese Operationen z.B. durch.

5.4. Anwendung und Folgen

Hat sich ein XP-Team dazu entschlossen mit Hilfe dieser Arbeit auf Usability zu testen, müssen für das GUI schrittweise Tests gesammelt werden. Am Ende soll ein grüner Balken signalisieren, dass keine hinterlegte Regel verletzt wurde.

Man beginnt zunächst damit einen Test zu schreiben. Hier wird es eine `TestSuite`:

```
public class MyUsabilityTests extends TestSuite {
  public static Test suite() {
    return new TestSuite();
  }
}
```

Diese Suite ist jedoch noch nicht testfähig, es mangelt ihr an Inhalten. Da ein GUI zu testen ist, benötigt man einen Sammler. Mit diesem sammelt man z.B. für ein primäres Fenster `MyFrame` Tests:

```
  public static Test suite() {
    UsabilityTestCollector utc = new UsabilityTestCollector();
    return utc.collectTests(new MyFrame());
  }
```

Nach Beseitigung des Kompilierfehlers, dass `MyFrame` nicht existiert, wird der Test ausführbar. Er liefert einen roten Balken und listet einige Fehler auf, wo bestimmte Daten[4] noch zur Verfügung gestellt werden müssen. Im Wechselspiel zwischen Testen und Programmieren wird `MyFrame` solange weiterentwickelt bis schließlich kein Test mehr fehlschlägt. Jetzt müssen die Entwickler eine Aufgabe einer Storycard erfüllen, wonach durch einen Parameter im Konstruktor zwei Zustände darzustellen sind. Konkret soll dies ein `boolean` realisieren, indem bei `false` der Zustand wie er jetzt ist beibehalten wird und bei `true` ein neuer Zustand eintritt. Der Testfall wächst zu:

```
  public static Test suite() {
    UsabilityTestCollector utc = new UsabilityTestCollector();
    TestSuite tests = new TestSuite();
    tests.add(utc.collectTests(new MyFrame(false)));
```

[4] konkret ein Titel und eine Menüleiste

```
    tests.add(utc.collectTests(new MyFrame(true)));
    return tests;
  }
```

Hier wird schon ein Unterschied im Testansatz deutlich, denn die selbe Komponente wird in jeweils unterschiedlichen Zuständen getestet. Es ist ein sehr künstliches Beispiel, doch zeigt es, dass die Aufgabe der Entwickler in der Identifikation von Komponenten liegt. Man weiß dabei nie genau, wann man fertig ist; in XP gilt:

„Anhand der Tests erkennen Sie, wann Sie fertig sind - wenn die Tests fehlerfrei ausgeführt werden, sind Sie vorerst einmal mit dem Programmieren fertig. Wenn Ihnen keine Tests mehr einfallen, die Fehler aufdecken könnten, dann sind Sie wirklich fertig.“ [Beck03, S. 45]

Auch hier ist man mit Programmieren fertig, wenn der Balken grün wird, doch ist man noch nicht wirklich fertig. Immerhin schreibt man keine Tests, sondern nutzt sie. Kontexttests haben die Eigenschaft auf Regeln zu basieren, die jemand anders in einem Dokument formuliert hat. Natürlich lassen sich diese Tests erweitern, doch sollte sich das XP-Team lieber um das Programmieren kümmern.

Der Sammler kann nur abbilden, wenn ihm ein `Container` zur Verfügung gestellt wird. Die Aufgabe der Entwickler ist es, diese vorzubereiten und dann zu testen. Dabei ist man erst fertig, wenn man keine Komponente mehr findet, die man *irgendwie* testen kann. Es müssen nicht immer ganze Fenster übergeben werden, manchmal genügt es auch für Varianten eines *Toolbar*s oder *Button*s zu sammeln.

Entwickler sollten den Sammelprozess kennen und auch ausnutzen. Nach Test-First würde zunächst nichts dagegen sprechen, dass der Sammler nur eine Abbildung auf Kontexttests liefert. Es ist immer möglich zunächst für einen Frame und anschließend noch für ein hinzugefügtes Menü seperat zu sammeln. Doch ist Testen dann nicht mehr einfach, es wird sehr kompliziert. Man müsste für jeden Button einzeln sammeln, er müsste sogar von außen zugreifbar sein. Ändert man einen globalen Zustand, reicht es nicht aus einfach den Frame wie oben neu zu sammeln, jede Komponente selbst müsste erneut abgebildet werden. Eigentlich würde man nur den Sammelprozess manuell durchführen.

5.5. Entwurfsalternativen

Wird eine Methode nicht mehr benötigt, lässt sich diese so überschreiben, dass durch Werfen einer `UnsupportedOperationException` keine Verfügbarkeit mehr vorliegt. In Abschnitt 4.2.3 wurde bereits erwähnt, dass sich die Menüleiste einer Applikation zur Laufzeit nicht ändern darf. Als Lösung kann die Methode `remove` wie eben beschrieben geändert werden. Aber in Swing sind intern alle Schnittstellen wohlbekannt und werden demnach auch selbstverständlich benutzt. Eine nachträgliche Änderung, auch wenn ein neuer Usability Layer geschrieben wird, würde nicht nur die Kompatiblität einschränken, sondern auch die Funktionalität gefährden.

Manchmal fordern Kontexttests die Existenz von bestimmten Daten in Komponenten - sie stellen Mindestanforderungen auf. Zur Lösung dieses Problem können die Konstruktoren z.B. so geändert werden, dass die Komponente ohne diese Daten nicht erzeugt werden kann. Aber es gibt vier Gegenargumente:

- Das Ändern einer Schnittstelle nach außen, kann zu Problemen führen und müsste sehr sorgfältig geprüft werden. Immerhin würde ein parameterloser Konstruktor nicht mehr existieren.
- Alle Daten müssen vor dem Konstruktionsprozess eines Elements vorliegen.
- Die Signatur der Konstruktoren wird komplexer. Bei Methoden sollte als Regel gelten, dass nie mehr als drei Parameter von Nöten sind [Bloch02]. Dies lässt sich auf Konstruktoren übertragen. Die Signatur würde nicht nur länger werden, sondern oftmals auch viele Parameter vom selben Typ fassen.
- Regeln können erweitert werden, so dass die Mindestanforderungen evtl. weiter wachsen. Jedes Mal wären die Konstruktoren der betroffenen Klasse zu ändern.

Abschließend ist zu sagen, dass es immer einfacher ist einen Test zu schreiben, statt etwas an einer Bibliothek zu ändern und alle Konsequenzen zu prüfen. Dies gilt auch, wenn man alternativ Überprüfungen zur Laufzeit vornimmt. Hier kommt jedoch erschwerend hinzu, dass diese Stellen im Code zunächst erreicht werden müssen, bevor die Entwickler das Problem sehen. Bei einem Test wird nie eine Regel ausgelassen und Rückmeldungen werden sofort gegeben.

6. Kritische Würdigung

6.1. Kritischer Rückblick auf den Erarbeitungsprozess

Um jede Regel einer Guideline in einem automatisierten Test umzusetzen, wird mit dem Studieren von verschiedenen Werken begonnen. Dabei stellen sich zwei Probleme:

- Es ist schwer festzulegen, wann eine Regel nahezu vollständig in einem Test erfasst ist.
- Es ist schwer herauszufinden, wie eine Regel überhaupt getestet werden kann. Besonders wenn erst noch Erweiterungen im Usability Layer vorgenommen werden müssen.

Findet man dennoch potentiell testbare Regeln, so entsteht eine Liste, die schon stark *Abstract Test Sheets* ähnelt. Doch sieht man zu diesem Zeitpunkt noch nicht den Sinn einer Rollenverteilung nach Betroffener und Beteiligte. Würde man zu Programmieren beginnen, sind die resultierenden Tests mitunter nicht immer einfach. In Abschnitt 5.1.1 wurde bereits das Beispiel mit der Einzigartigkeit von *mnemonics* in Menüs und Menüeinträgen angeführt. Vom Menüeintrag aus zu prüfen, ist durch die Auswertung der Elternkomponente nicht nur einfacher zu verstehen, der Code wird auch sehr viel kompakter. Das Resultat entspricht mehr den Werten von XP.

Die Beschaffenheit der Regeln deutete schon sehr früh daraufhin, dass manche automatisierten Tests keine Komponententests sind. Hier ist Vorsicht geboten, dass man nicht in die Utopie der vollständigen Benutzersimulation abrutscht. Neben der Aufgabe einfache Modelle zu finden, ist auch die Beschaffenheit von Kontexttests zu verinnerlichen. Es wird immer ein noch unbekanntes GUI getestet.

Hat man erfolgreich Testbarkeit im Layer selbst und außerhalb hergestellt, so tritt man beim ersten Schreiben von Kontexttests schon in die nächste Falle. Jeder Fehlschlag endet nur in einem Hinweis, anhand dessen man nicht nachvollziehen kann, welche Komponente er betrifft. Die Idee einer Buchführung in Form eines Stammbaums löste schließlich das Problem.

Die ersten geschriebenen Kontexttests können z.T. sehr kompliziert sein. Sind Daten zu prüfen, die erst zur Laufzeit entstehen, wie z.B. die Titel von allen erzeugten Dialogen, ist die einfachste Lösung in einer entsprechenden Klasse *statisch* diese Daten zu speichern. Wird ein Dialog im Kontext getestet und der Titel ist in einer statischen Liste bereits vorhanden, schlägt der Test fehl, ansonsten wird der Titel zur Liste hinzugefügt. Dieses Beispiel nutzt den Mechanimus, der Kontexttests zu Grunde liegt,

vollständig aus, weil die benötigten Daten erst durch den Sammelprozess zur Laufzeit langsam entstehen. Nur wer die besondere Art des Testens versteht, wird auch gute Tests schreiben.

Fenster und Zustände sind der wohl kritischste Aspekt dieser Arbeit, denn nur die Entwickler wissen, ob z.B. bei einem bestimmten Netzwerk- oder Zeitevent ein Dialog angezeigt wird oder nicht. Folglich können auch nur die Entwickler diese Dialoge testen. Das in dieser Arbeit vorgestellte Modell, mit dem die Erzeugung von Dialogen nachvollzogen werden kann, muss nicht zwingend alle Dialoge liefern. Aber das Modell reicht zum Testen aus, wenn es z.B. um die Eigenschaft *„avoid more than two levels of dialog boxes“* [Johnson00] geht, weil der Benutzer an die erweiterten Komponenten[1] eine direkte Forderung stellt. Er wird eine Antwort erwarten, weil er sie selbst provoziert hat. Alle anderen Dialoge treten aus diesem Blickwinkel eher zufällig auf.

Abschließend sei gesagt, dass man sich beim automatisierten Testen auf Usability immer auf seinen gesunden Menschenverstand besinnen sollte. Es gibt Aspekte, die sich nicht testen lassen und die akzeptiert werden müssen.

6.2. Probleme und Grenzen des ausgewählten Ansatzes

Beschränkt man sich auf Regeln und möchte darüber ein hohes Maß an Usability erreichen, lässt man den Benutzer selbst nahezu vollständig außer acht. Insbesondere wenn nur ein automatisiertes Systen auf Regeln prüft, kann eine trügerische Sicherheit entstehen. Denn Feedback über die z.B. angemessene Präsentation des GUI entsteht nicht. Die Fragen, ob es den Aufgaben des Benutzers genügt oder ob ihm die Arbeit mit diesem System Spaß macht, bleiben offen. Man kann z.B. mitnichten prüfen, ob eine Fehlermeldung aussagekräftig ist, sie könnte theoretisch immer von der Gestalt sein: „Es ist ein Fehler aufgetreten. Bitte wenden Sie sich an Ihren Systemadministrator.“ und keinem Benutzer wäre geholfen. Das Überprüfen von Usability geht manchmal schlecht ohne Menschen, weil es gerade um Menschen geht.

Bei der Erweiterung einer Oberflächenbibliothek muss man sehr vorsichtig sein und fühlt sich manchmal sogar unwohl. Schon Java Swing ist eine sehr umfangreiche Bibliothek, so dass es zu mehreren Problemen kommen kann:

- Man hat keine Tests. Jede Erweiterung an Swing ist *„blind“*, weil man nur hoffen kann, dass keine Seiteneffekte auftreten, an die nicht gedacht wurde.
- Die Unterklassen im Usability Layer müssen immer gemeinsam mit Swing weiterentwickelt werden [Bloch02].

Ein XP-Team kann von den Usability Unit Tests nur profitieren, wenn ein GUI mit dem Usability Layer aufgebaut wird. D.h. ein Einsatz in bestehenden Projekten ist erst nach Umschreiben des GUI möglich.

[1] d.h. `UButton`, `UMenuItem` und `UTextField`

Um diese Problematik abzumildern, könnte man versuchen ohne den Usability Layer direkt mit Swing zu arbeiten. Aber es fehlt dadurch eine einheitliche Schnittstelle zum Auslesen der Kinder einer Komponente, sowie die Möglichkeit Fenster nachzubauen, Regeln implizit im Layer zu erfüllen und die Semantik zu verfeinern. Dazu soll hier ein alternativer Lösungsansatz angedacht werden:

- Eine einheitliche Schnittstelle für jede Komponente lässt sich auch über das Adapter Pattern [Gamma96] realisieren. Auf jede Komponente müsste nur der entsprechende Adapter abgebildet werden, um anschließend über ihn alle Kinder auszulesen.
- Zur Verfeinerung der Semantik und zum Nachbau des Fensters wird jede Komponente extern mit einem *„Etikett“* versehen. D.h. es gibt Etiketten, die für jede Komponente anders strukturiert sind. Während der Implementierung mit Daten gefüllt, lassen sie sich an eine Komponente als *property*[2] hängen. Dadurch könnte ein Test den Kontext von Komponenten auslesen - je nach Struktur des Etiketts. So stände z.B. die Semantik von einem Button, der nur einen Dialog schließt, im Etikett selbst[3] - eine Implementierung im Usability Layer wäre nicht mehr nötig.
- Regeln lassen sich durch Wegfall des Usability Layers nicht mehr implizit erfüllen. Alle Komponententests sind in Kontexttests zu überführen, damit das XP-Team entsprechend reagieren muss.

Zusätzlich sei gesagt, dass für diesen Ansatz nicht immer zwingend mit den Komponenten einer Oberflächenbibliothek zu arbeiten ist. Letztendlich wird nur eine Beschreibung des GUI benötigt, die sich automatisiert auswerten lässt. Dabei können die internen Datenstrukturen eines GUI-Editor genauso zum Testen genutzt werden, wie eine Beschreibung in XML. Vielleicht gelingt es in anderen Formen der Beschreibung sogar mehr Informationen unterzubringen, so dass mehr Regeln testbar werden.

[2] In Swing kann jede Komponente mit beliebigen Eigenschaften (den *properties*) von außen dekoriert werden.

[3] z.B. durch eine Methode `getCancelButton()`

7. Zusammenfassung und Ausblick

Benutzer vor Fehlern zu schützen, die längst bekannt sind und sogar niedergeschrieben wurden, ist ein wichtiger Grundpfeiler, um die Usability von Software zu erhöhen.

Ziel dieser Arbeit war das automatisierte Testen auf Usability Regeln aus Guidelines. Dazu stellt ein Usability Layer, der auf Java Swing basiert, Testbarkeit her. Im Anschluss entstand ein Satz von ausgewählten Regeln als Usability Unit Tests (siehe Anhang A für die umgesetzten Regeln). Neben der neuen Teststrategie der Kontexttests, stellte sich auch die Dokumentationsmethode über Abstract Test Sheets als sehr viel versprechend heraus. Insgesamt lassen sich über den erarbeiteten Prozess aus Abschnitt 5.1.3 Guidelines in Tests umsetzen.

Jedes XP-Team kann mit dem erstellten Usability Layer ein GUI realisieren und anschließend sofort testen, ob Regeln nicht eingehalten werden. Dabei muss das Team nicht mal Guidelines kennen, sondern nur ein automatisiertes System gekonnt nutzen, das durch JUnit in einer Umgebung arbeitet, die XP-Projekte gewöhnt sind. Neben Storycards sind Anforderungen entstanden, die in jedem Projekt umzusetzen sind, z.T. aber auch schon im Usability Layer erfüllt werden, wenn ein Komponententest entstand.

Kritisch ist im Verlauf dieser Arbeit die Notwendigkeit von Kontexttests gewesen. Aus der Beschaffenheit von vielen Usability Regeln folgt zwangsläufig, dass ein konkretes noch unbekanntes GUI zu testen ist. Zusätzliche Informationen und Strukturen können dabei nur bedingt über den Usability Layer zur Verfügung gestellt werden. Auch wenn es z.B. gelang für erzeugte Dialoge ein einfaches Modell zu entwickeln, ist dies für Zustände schwieriger. Manche Daten in einem GUI entstehen erst durch Benutzerinteraktionen, die bei einer automatisierten Auswertung vollständig fehlen.

Den Nutzen des erstellten Usability Layers und der Tests wird nur eine Erprobung in der Praxis zeigen können. Insbesondere wäre der Umgang mit der Problematik der Zustände interessant und ob vielleicht sogar Konsequenzen für das Design entstehen. Ein XP-Team sucht die einfachste Lösung, ein Benutzer tut genau das selbe. Für diesen Praxiseinsatz stellt die vorliegende Arbeit ein voll funktionfähiges Werkzeug zur Verfügung. Durch seine flexible Struktur und der Dokumentation können Änderungen, die aus der Einsatzerfahrung sinnvoll erscheinen, schnell umgesetzt werden.

Literaturverzeichnis

[Beck03] Kent Beck,
Extreme Programming,
Addison-Wesley Verlag, 2003

[Bloch02] Joshua Bloch
Effektiv Java programmieren,
Addison-Weskey Verlag, 2002

[Carbon04] Ralf Carbon, Jörg Dörr, Marcus Trapp,
Focusing Extreme Programming on Usability
erschienen in *Informatik 2004 - Informatik verbindet, Band 2*
Hrsg. Peter Dadam, Manfred Reichert; Gesellschaft für Informatik

[Fowler] Amy Fowler,
A Swing Architecture Overview,
http://java.sun.com/products/jfc/tsc/articles/architecture

[Gamma96] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides,
Entwurfsmuster,
Addison-Wesley Verlag, 1996

[GNOME] The GNOME Usability Project,
GNOME Human Interface Guidelines 2.0,
http://developer.gnome.org/projects/gup/hig/

[Hamill04] Paul Hamill,
Unit Test Frameworks,
O'Reilly Media, Inc.,2004

[Java API] Sun Microsystems, Inc.,
Java 2 Platform Standard Edition 5.0 API Specification,
http://java.sun.com/j2se/1.5.0/docs/api/

[Johnson00] Jeff Johnson,
GUI bloopers: dont's and do's for software developers and Web designers,
Morgan Kaufmann Publishers, 2000

[MVC] Sun Microsystems, Inc.,
Model-View-Controller,
http://java.sun.com/blueprints/patterns/MVC-detailed.html

[Nielsen-a] Jakob Nielsen,
How to Conduct a Heuristic Evaluation,
http://www.useit.com/papers/heuristic/heuristic_evaluation.html

[Nielsen-b] Jakob Nielsen,
Ten Usability Heuristics,
http://www.useit.com/papers/heuristic/heuristic_list.html

[Nielsen93] Jakob Nielsen,
Usability Engineering,
Morgan Kaufmann Publishers, 1993

[Smith86] Sidney L. Smith, Jane N. Mosier
Guidelines for designing user interface software, 1986,
ftp://archive.cis.ohio-state.edu/pub/hci/Guidelines/

[Sun01] Sun Microsystems, Inc.,
Java Look and Feel Design Guidelines, second edition, 2001,
http://java.sun.com/products/jlf/ed2/book/index.html

A. Umgesetzte Abstract Test Sheets

Betroffene Klasse	*Beteiligte* Klassen	Usability Regel
`UDialog`	`UMenuBar`	Ein Dialog hat keine Menüleiste. [Johnson00], [Sun01]
`UDialog`		Ein Dialog hat immer einen Titel. [Sun01]
`UDialog`		Zwischen den Kontrollbuttons und dem Inhalt eines Dialogs muss eine sichtliche Trennung vorliegen. Insbesondere sind die Kontrollbuttons von anderen Buttons zu trennen. [Johnson00]
`UDialog`	`UButton`	Ein Hilfe-Button ist in der Reihenfolge der Kontrollbuttons als letzter Button in der Leserichtung des jeweiligen Landes anzuzeigen. [Sun01]
`UDialog`	`UButton`	Ein ausführender Button, im Sinne von Schließen des Dialogs und Ausführen einer Operation, ist in der Reihenfolge der Kontrollbuttons als erster Button in der Leserichtung des jeweiligen Landes anzuzeigen. [Sun01]
`UDialog`		Dialoge dürfen nicht in Fallen führen. D.h. es gibt immer einen Button, der einen Dialog ohne weitere Auswirkungen schließt. [Johnson00]
`UDialog`		Die Hierarchie der Dialoge darf nicht eine Tiefe über zwei Ebenen erreichen. D.h. ein primäres Fenster darf einen Dialog *A* erzeugen. Der Dialog *A* darf auch einen Dialog *B* erzeugen. Dialog *B* darf jedoch keine Dialoge mehr erzeugen. [Johnson00]
`UDialog`		Mehrere Dialoge dürfen nicht den selben Titel haben. [Johnson00]
`UDialog`		Jeder Dialog hat eine *tab traversal order*, die der Leserichtung des jeweiligen Landes entspricht. [Sun01]
`UFrame`		Ein primäres Fenster hat immer eine Menüleiste. [Johnson00]
`UFrame`		Ein primäres Fenster hat immer einen Titel. [Sun01]

Betroffene Klasse	*Beteiligte* Klassen	Usability Regel
`UGroupBox`		Ein Box zum Gruppieren von Elementen hat immer einen Titel. [Johnson00]
`UGroupBox`		Ein Box zum Gruppieren von Elementen enthält immer mehr als ein Element. [Johnson00]
`UGroupBox`		Boxen zum Gruppieren von Elementen sind nicht zu schachteln. [Johnson00]
`UMenuBar`	`UMenu`	Eine Menüleiste enthält immer mindestens ein Menü. [Johnson00]
`UMenuBar`	`UMenuItem`	Einem Menüeintrag hinterlegte Aktionen sind in der gesamten Menüleiste einzigartig. [Johnson00]
`UMenuBar`	`UMenu`	Der Titel eines Menüs ist in einer Menüleiste einzigartig. [Sun01]
`UMenuItem`		Ein Menüeintrag hat einen Namen. [Johnson00]
`UMenuItem`		Ein Menüeintrag muss über die Tastatur erreichbar sein, d.h. er hat ein *mnemonic*. [Johnson00], [Sun01]
`UMenuItem`		Das *mnemonic* eines Menüeintrags muss auf internationalen Tastaturen verfügbar sein. [Sun01]
`UMenuItem`	`UDialog`	Endet der Name eines Menüeintrags mit „...“, so öffnet dieses Element in jedem Fall einen Dialog bei Interaktion. [Sun01]
`UMenu`	`UMenu`	Wird einem Menü ein weiteres Menü hinzugefügt, so darf das hinzugefügte Menü keine Menüs mehr enthalten. D.h. Untermenüs sind nicht zu schachteln. [GNOME]
`UMenu`		Ein Menü enthält mindestens einen Menüeintrag. [Johnson00]
`UMenu`	`UMenuItem`	Der Name des Menüs ist von keinem Menüeintrag zu verwenden. [Johnson00]
`UMenu`	`UMenuItem`	Der Name eines Menüeinträgs ist innerhalb eines Menüs einzigartig. [Johnson00]
`UMenu`	`UMenuItem`	Das *mnemonic* eines Menüeinträgs ist innerhalb eines Menüs einzigartig. [Sun01]
`UMenu`		Ein Menü hat immer einen Namen, der aus nur einem Wort besteht. [Sun01]
`UMenu`		Ein Menü muss über die Tastatur erreichbar sein, d.h. es hat ein *mnemonic*. [Johnson00], [Sun01]
`UMenu`		Das *mnemonic* eines Menüs muss auf internationalen Tastaturen verfügbar sein. [Sun01]
`UMultiple-Choice`		Eine 1-zu-N Auswahl hat immer einen Startwert. [Johnson00]

Erklärung der Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit selbständig und ohne fremde Hilfe verfasst und keine anderen als die in der Arbeit angegebenen Quellen und Hilfsmittel verwendet habe. Die Arbeit hat in gleicher oder ähnlicher Form noch keinem anderen Prüfungsamt vorgelegen.

Hannover, den 8. August 2005

_____

Dennis Hardt