Ansätze zur Vereinigung von Komplexität und Benutzbarkeit in Anwendungssoftware

Julian Fietkau

7. Februar 2011

Seminararbeit im Modul

„Interaktive Systeme“

im Wintersemester 2010/2011

Dozenten: Prof. Dr. Horst Oberquelle, Prof. Dr.-Ing. Steffi Beckhaus

Fachbereich Informatik
Universität Hamburg

Inhaltsverzeichnis

Zusammenfassung

Moderne Anwendungssoftware hat mit steigenden Anforderungen sowohl an Funktionalität als auch an Benutzbarkeit zu kämpfen. In dieser Arbeit wird untersucht, wie Softwarefunktionalität und die Komplexität grafischer Schnittstellen miteinander zusammenhängen. Weiterhin werden einige Ansätze dargestellt, wie selbst für komplexe Anwendungssoftware eine benutzergerechte Schnittstelle entwickelt werden kann.

1. Einführung

Im Bereich der Entwicklung von Benutzungsschnittstellen für gebrauchstaugliche Software gibt es wiederkehrende Bestrebungen, eine Vereinfachung für die Benutzer zu erreichen[Kin07]. Die Argumentation, man müsse „einfache“ Software anstreben, hat zu einem breiten Trend in Richtung Reduktion von Softwarekomplexität geführt. Gleichzeitig steigen die Anforderungen an die Funktionalität von alltäglich genutzter Software in dem Maße, in dem sie das Leben von Nutzern, deren Zahl stetig steigt, zunehmend prägt – Software soll immer mehr können und immer mehr Features bieten. Dies erscheint auf den ersten Blick wie ein nicht auflösbarer Widerspruch.

Die Entwickler von Anwendungssoftware und die Designer ihrer Schnittstellen müssen mit der Herausforderung umgehen, Wege zur Vereinigung von vielfältig fähiger und vielseitig nutzbarer sowie leicht erlernbarer und begreifbarer Software zu finden.

1.1. Ziele dieser Arbeit

Diese Arbeit soll das Problem von Komplexität und Benutzbarkeit bezogen auf Software einerseits verständlich darlegen und andererseits Lösungsansätze liefern, wie der scheinbare Zwiespalt angegangen werden kann. Dazu werden traditionelle Ansätze zur Konzeption und Beherrschung von Schnittstellenkomplexität dargestellt und aktuelle Ideen und Impulse thematisiert, wie vielfältige Softwarefunktionalität Anwendern angemessen zugänglich gemacht werden kann.

1.2. Begriffe

Anwendungssoftware

Hierbei handelt es sich um ein auf einem Computer lauffähiges Programm, welches von einem Benutzer bedient wird um eine spezifische Funktion durchzuführen, deren Ergebnis ein externes Ziel erfüllt. Beispiele sind Programme zur Textverarbeitung oder Tabellenkalkulation. Anwendungssoftware steht in Abgrenzung von Systemsoftware (Betriebssystem, Dienstprogramme, Diagnose- und Wartungssoftware etc.), welche ausschließlich die Funktionsfähigkeit des Rechners sicherstellt und ansonsten keinen externen Nutzen bietet. *Spiele*, deren Verwendung in aller Regel abgesehen von Spaß und Kurzweil keinen bleibenden Ergebnisse produziert, gelten nach der gängigen Definition[Wir11] ebenfalls als Anwendungssoftware. Die Unterschiede im Nutzungsverhalten bei Spielen im Gegensatz zu zielorientiert eingesetzter Anwendungssoftware

ist jedoch so groß, dass in dieser Arbeit zwischen Spielen und sonstiger Anwendungssoftware explizit unterschieden wird. Wenn im Folgenden von Anwendungssoftware die Rede ist, ist damit zielorientierte Anwendungssoftware gemeint.

Komplexität
In Anlehnung an Don Norman[Nor10] wird Komplexität in dieser Arbeit als Maß für die strukturelle Differenziertheit eines Systems verwendet. Je mehr Eigenschaften und Merkmale ein System hat, desto komplexer ist es. Komplexität ist im Gegensatz zur →*Kompliziertheit* eine objektive Eigenschaft eines Systems. Bezogen auf Benutzungsschnittstellen: Je mehr Handlungsmöglichkeiten, Bedienelemente und Befehle existieren, desto komplexer ist die Oberfläche. Je mehr Aktionen möglich und Features vorhanden sind, desto komplexer ist die Software. In dieser Arbeit ist ausschließlich die Rede von *äußerer* Komplexität von Software. Die innere Komplexität, die sich mit Fragen der Modularisierung und Strukturierung von Quellcode befasst, ist für den Benutzer höchstens indirekt relevant und wird nicht näher betrachtet. Ein Beispiel für eine komplexe Schnittstelle einer Anwendungssoftware findet sich in Abbildung 1.

Kompliziertheit
Als subjektives Pendant zur →*Komplexität* bezeichnet die Kompliziertheit eines Systems (ebenfalls nach Norman[Nor10]) das Ausmaß, indem es auf Klienten bzw. Benutzer verwirrend und undurchschaubar wirkt. Kompliziertheit ist in diesem Sinne ein psychologisches Phänomen und entsteht zum Beispiel, wenn zu einer Software nicht leicht ein passendes mentales Modell gebildet werden kann. Komplexität und Kompliziertheit sind weder identisch noch vollständig orthogonal – Komplexität kann sehr wohl Kompliziertheit nach sich ziehen, einen zwingenden Zusammenhang gibt es jedoch nicht. Eine mögliche Situation für die Wahrnehmung von Kompliziertheit ist wiederum die Oberfläche in Abbildung 1.

Abbildung 1: Eine typische Textverarbeitungssoftware. An diesem Beispiel zeigt sich die Komplexität der Software darin, dass eine große und vielfältige Menge von Möglichkeiten existiert, die Inhalte des Dokuments zu erzeugen und zu verändern. Diese Möglichkeiten sind durch die zahlreichen Menüs und Buttons reflektiert. Eine gewisse Kompliziertheit wohnt der Textverarbeitung dadurch inne, dass Funktionen teilweise schwierig zu finden sind oder ihr Verhalten nicht den Erwartungen entspricht.

2. Anforderungen an grafische Schnittstellen

Bekanntermaßen erhält Software ihre Fähigkeiten und „Features“, indem entsprechender Programmcode entwickelt und eingebunden wird. Dieser Code wird im Falle von Anwendungssoftware in der Regel nicht von allein ausgeführt, sondern die Benutzer führen an der Oberfläche der Software zielgerichtete Aktionen durch.

Wenn Funktionalität zwar im Code vorhanden ist, aber in der Oberfläche keine Möglichkeit existiert, die Funktionalität zu verwenden, dann wird der Code nicht ausgeführt und ist sinnlos. Die Benutzungsschnittstelle der Software ist der Ort, an dem die verfügbare Funktionalität abgelesen werden kann. Neue Funktionen in einer Software bewirken zwangsweise eine – wie auch immer geartete – Steigerung der Komplexität ihrer Benutzungsschnittstelle. Dennoch soll idealerweise die Bedienung der Software nicht schwieriger werden.

2.1. Wachsende Funktionalität

Moderne Anwendungssoftware blickt auf mehrere Jahrzehnte der Fortentwicklung und Verfeinerung typischer Anwendungsfälle zurück. Fälle, in denen ein etabliertes Feature aus späteren Versionen einer Software bewusst entfernt wird, sind die Ausnahme. Besonders „einfache“ Software wie funktional im Vergleich zur etablierten Konkurrenz eingeschränkte Textverarbeitungsprogramme[Abi11] tauchen immer wieder auf, setzen sich jedoch nicht durch und werden von der breiten Masse der Benutzer nicht angenommen.

Softwarefunktionalität lässt sich nicht einfach messen oder zählen. Dennoch ist es möglich, Trends und Richtungen durch verwandte Größen indirekt zu untersuchen. Für die konkrete Software „Microsoft®Word für Windows®“ hat Jensen Harris (ein bei *Microsoft Corp.* beschäftigter UX-Experte) beispielsweise schlicht die Anzahl der Symbolleisten über die Versionen hinweg gezählt[Har05b]. Seine Ergebnisse sind in Tabelle 1 reproduziert.

Über die Bedeutung zusätzlicher Features für den Erfolg und Fortbestand von Software sei hier die Erfahrung von Joel Spolsky[Spo06] beispielhaft wiedergegeben:

> I think it is a misattribution to say, for example, that the iPod is successful *because it lacks features*. If you start to believe that, you'll believe, among other things, that you should *take out* features to increase your product's success. With six years of experience running my own software company I can tell you that *nothing* we have *ever* done at Fog Creek has increased our revenue more than releasing a new version with more features. Nothing. The flow to our bottom line from new versions with new features is absolutely undeniable. It's like gravity. When we tried Google ads, when we implemented various affiliate schemes, or when an article about FogBugz appears in the press, we could barely see the effect on the bottom line. When a new version comes out with new features, we see a sudden, undeniable, substantial, and permanent increase in revenue.

Versionsnummer	Anzahl Symbolleisten
1.0	2
2.0	2
6.0	8
95	9
97	18
2000	23
2002	30
2003	31
2007ff.	N.A.

Tabelle 1: Die Anzahl an Symbolleisten in den verschiedenen Versionen von „Microsoft®Word für Windows®“. Das Versionierungsschema der Software hat sich mehrfach geändert, die Tabelle enthält jedoch alle erschienenen Hauptversionen bis einschließlich 2007 in der korrekten Reihenfolge. Jene Version war die erste, in der die Symbolleisten nicht mehr zum Einsatz kamen.

Ein anschauliches Beispiel für die Tendenz zu immer mehr Features ist die Online-Text- und Tabellenverarbeitung *Google Docs*™. Ursprünglich hatte sich der Dienst als leichtgewichtige Alternative zu traditionellen Textverarbeitungen beworben und kam mit sehr wenigen UI-Elementen aus. Inzwischen hat die Oberfläche mehr mit den althergebrachten Konkurrenten gemeinsam als mit der ursprünglichen Version, die Entwickler sehen sich mit den gleichen Usability-Problemen konfrontiert und die Gesamtmenge an Features steigt weiter[Goo10], ohne dass ein Ende abzusehen wäre.

2.2. Einfache Bedienbarkeit

Unabhängig von den Anforderungen an die Features der Software existiert die Maxime, dass moderne Software leicht zu bedienen sein muss. Dafür lassen sich vielerlei Gründe ausmachen.

1. **Software wird immer schneller konsumiert.** Noch vor wenigen Jahren war der Kauf von Software in den meisten Fällen mit sorgfältigem Abwägen verbunden. Benutzer entschieden sich erst nach reiflicher Überlegung für den Kauf und lasen zum Einstieg das Handbuch, die Installation einer Software konnte beträchtliche Zeiträume in Anspruch nehmen. Heute ist Software ein Gebrauchsgut – moderne „Apps“ sind für wenige Euro oder gar Cent schnell gekauft, Software kann jederzeit ad hoc aus dem Internet geladen und installiert werden. Die bisherige Speerspitze dieses Trends ist die Ein-Klick-Installation, bei der Software völlig ohne Installationsassistent oder Einrichtungsdialoge mit einem Mausklick heruntergeladen, installiert und in das System eingebunden wird. Dieser Komfort, den Linux-Benutzer bereits seit einigen Jahren genießen[Ubu11], findet aktuell über den Umweg aktueller Smartphones auch in anderen Desktop-Betriebssystemem Einzug[Mac11].

2. **Das Publikum für Software ist zunehmend heterogen.** Längst ist die Verwendung von IT nicht mehr einer technisch versierten Minderheit vorbehalten. Potenziell kann statt einer wohlhabenden Person aus Mitteleuropa genau so gut ein Kioskbesitzer in Indien oder ein Kind in Nigeria mit der Software in Berührung kommen. Menschen mit Sehschwäschen oder körperlichen Behinderungen gehören ebenso zu den Computerbenutzern wie Menschen, deren Schrift von rechts nach links läuft oder die überhaupt nicht lesen können. Schon aus ökonomischer Sicht erscheint es sinnvoll, Software für ein möglichst breites Publikum zu entwickeln. Dafür muss sie einfach zu bedienen und schnell zu erlernen sein.

3. **Es entsteht insgesamt immer mehr Software.** Moderne Programmiersprachen erleichtern die Abstraktion und Programmierung auch von praxisfähiger Software. Steigende Verfügbarkeit von Entwicklerkapazitäten führt zu steigender Verfügbarkeit von Software, wodurch der Konkurrenzdruck für einzelne Softwareprodukte steigt. Neben anderen Qualitätsmerkmalen wird so auch die Benutzbarkeit zum Auswahl- und Kaufkriterium.

Steve Krug bringt mit seinem einprägsamen „Don't make me think"[Kru05] die Denk- und Handlungsweise klar zum Ausdruck, die die heutige Interaktion mit Anwendungssoftware beschreibt. Software wird nur noch selten systematisch erlernt, sie soll am besten von Anfang an intuitiv erschließbar und schnell verständlich und benutzbar sein.

Ein bezogen auf die Praxisnähe recht verlässliches Dokument zum Thema „Benutzergerechte Gestaltung" ist die ISO 9241 Teil 110 („Grundsätze der Dialoggestaltung"). Dort werden diverse Kriterien aufgelistet, die als Bewertungsmaßstab für die Benutzbarkeit von interaktiver Software geeignet sind. Einfachheit gehört nicht dazu. Stattdessen ist unter Anderem von „Aufgabenangemessenheit", „Selbstbeschreibungsfähigkeit" und „Lernförderlichkeit" die Rede. Dies ist ein erster Hinweis, dass Einfachheit an sich zwar eine Möglichkeit ist, benutzergerechte Schnittstellen zu entwerfen, jedoch nicht zum Selbstzweck werden sollte: Einfachheit ist zunächst mal ein Weg (einer von vielen), nicht das Ziel.

3. Ebenen von Komplexität

Die Komplexität einer grafischen Schnittstelle manifestiert sich auf mehreren Ebenen. Traditionelles Interaktionsdesign befasst sich hauptsächlich mit der Komplexität und Bedienbarkeit einzelner, maximal bildschirmfüllender Dialogmasken. In diesem Bereich existiert viel Literatur, weshalb hier lediglich ein sehr kurzer Einblick gegeben wird.

Dagegen bringt die Strukturierung von Oberflächen bezogen auf eine große Anwendung eigene Herausforderungen mit sich, die etwa mit Problemen der Suche und Navigation zu tun haben. Diese übergeordnete Ebene wird gesondert behandelt.

3.1. Einzelne Dialogmasken

Die Komplexität von Dialogmasken ist dann optimal, wenn sie geeignet ist, die nötigen Aufgaben zu erfüllen, und gleichzeitig nicht komplizierter ist als nötig. Dabei gibt es ein Spektrum, in dem Entwürfe sich sowohl in Richtung übermäßige Kompliziertheit als auch in Richtung Übervereinfachung bewegen können. Einige Beispiele liefern die Abbildungen 2 bis 4.

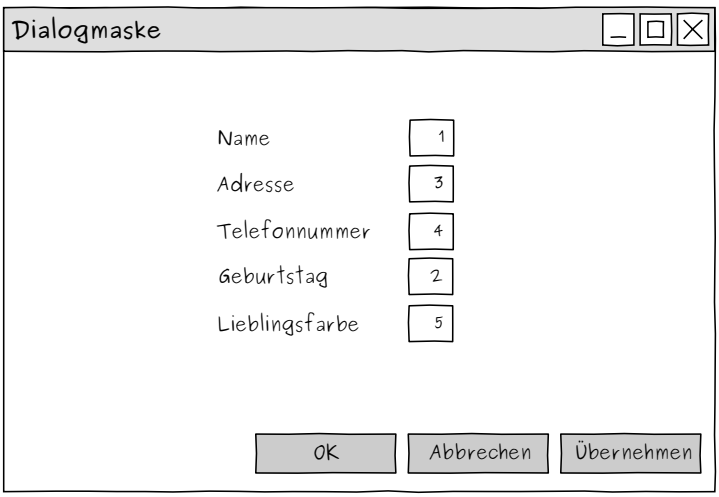

Abbildung 2: Eine einfach strukturierte Dialogmaske. Sie bietet wenige Interaktionsmöglichkeiten und eine geringe Komplexität in Anzahl und Anordnung der Elemente. Allerdings ist sie „übervereinfacht“ und der tatsächlichen Aufgabe nicht gewachsen: Die Festlegung einer Sortierreihenfolge ist nur teilweise möglich, da nicht zwischen auf- und absteigender Sortierung unterschieden wird. Weiterhin ist der Zweck des Formulars kaum aus dem Formular selbst erschließbar.

Die Forschung im Bereich des Interaktionsdesign befasst sich ausführlich mit dieser Komplexitätsebene. Es gibt viele Hilfen und Richtlinien, von denen hier beispielhaft *Gestaltgesetze* und *Interaktionsmuster* herausgegriffen werden. Zur Vertiefung existiert vielerlei hochwertige Literatur[Her06].

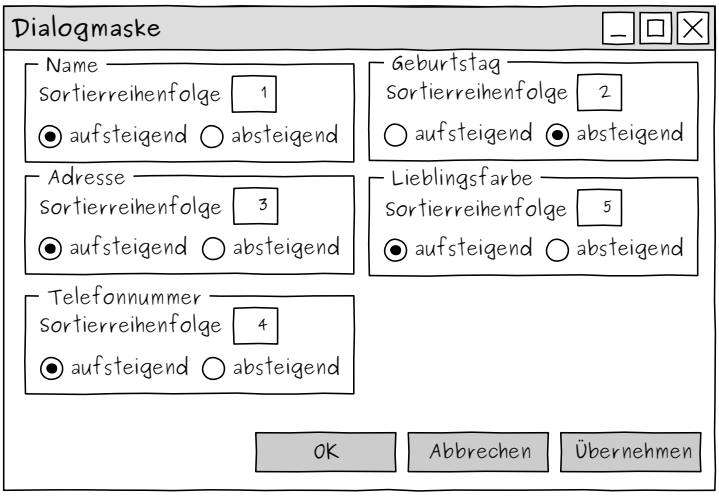

Abbildung 3: Diese Dialogmaske ist sehr kompliziert. Alle nötigen Möglichkeiten zur Interaktion sind vorhanden und die Aufgabe kann erfüllt werden, aber die Anordnung ist nicht leicht zu erschließen und die Interaktionsmethoden sind für die Aufgabe ungünstig gewählt.

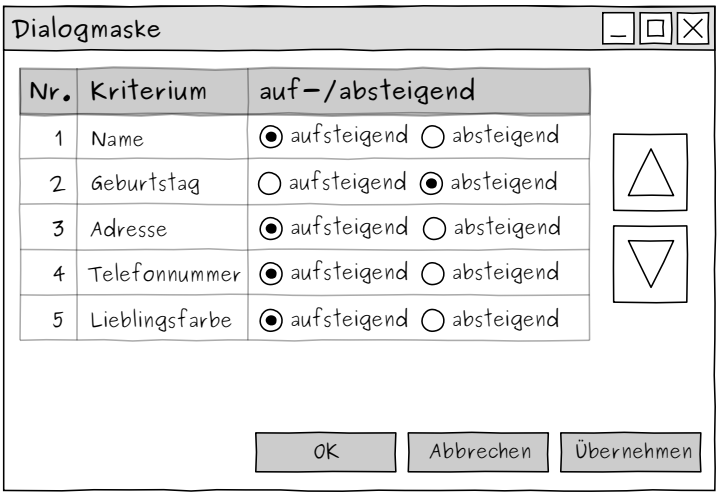

Abbildung 4: Diese Dialogmaske schafft von den drei vorgestellten Entwürfen am besten die Reduktion der Kompliziertheit bei angemessener Komplexität. Sie bedient sich der Listenmetapher, die viele Benutzer z.B. aus Musik-Player-Software kennen, und ermöglicht direkte Manipulation. Die Aufgabe kann damit erledigt werden und trotzdem hält die Kompliziertheit sich in Grenzen.

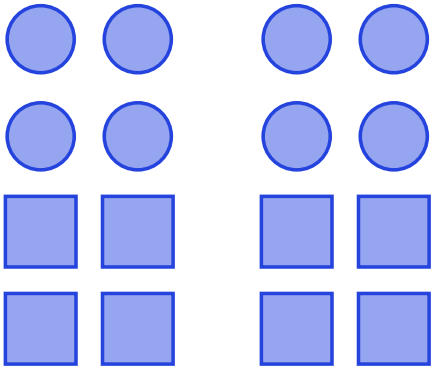

Abbildung 5: Anhand dieser Anordnung einfacher geometrischer Figuren lassen sich gleich mehre Gestaltgesetze veranschaulichen. Das Gesetz der Nähe bewirkt, dass die linken und rechten Elemente als zwei jeweils zusammenstehende Gruppen wahrgenommen werden. Würde man den mittleren Abstand schrittweise verringern, würde irgendwann stattdessen das Gesetz der Ähnlichkeit die Trennung in zwei Hauptgruppen von Kreisen und Quadraten bewirken.

3.1.1. Gestaltgesetze

Die Wahrnehmung von räumlich angeordneten Elementen aller Art ist den Gestaltgesetzen unterworfen. Sie bestimmen, wie der Betrachter (oder im Fall von interaktiver Software der Benutzer) die Elemente mental gruppiert und zuordnet.

In Abbildung 5 ist zu erkennen, wie Messgrößen wie der räumliche Abstand oder die gestalterische Ähnlichkeit einzelner Elemente bewirken, dass ein Betrachter sie als zusammenhängend wahrnimmt. Wichtige Gestaltgesetze sind das Gesetz der Nähe, das Gesetz der Ähnlichkeit oder das Gesetz der guten Fortsetzung. Es liegt am Entwickler des Designs, die Elemente so zu gruppieren, dass ihr wahrgenommener Zusammenhang zum semantischen Zusammenhang passt, um eine zielgerichtete Verarbeitung ohne Irritationen zu ermöglichen.

Durch eine sinnvolle räumliche Strukturierung der Oberfläche wird das Verstehen semantischer Zusammenhänge somit leichter. Dadurch kann die Kompliziertheit in Form von wahrgenommener Willkür der Oberfläche verringert und das Lernen erleichtert werden, ohne bezogen auf die Komplexität Abstriche machen zu müssen.

3.1.2. Interaktionsmuster

Im Bereich des Interaktionsdesigns sind Interaktionsmuster (angelehnt an die Entwurfsmuster des Software Engineering, welche wiederum konzeptuell aus der Architektur stammen) schablonenartige Lösungen für wiederkehrende Probleme. Der große Wert von Interaktionsmustern besteht darin, dass sie es ermöglichen, Wissen über Interaktion über Anwendungsgrenzen hinweg nutzbar zu machen.

Interaktionsmuster greifen schematisch etablierte Methoden zur Interaktion mit Software auf und sind dafür gedacht, als Richtlinien für Softwareentwickler zu dienen, damit die Benutzer sich leicht funktionierende mentale Modelle der Software bilden können. Ein klassisches Beispiel ist die „Undo“-Funktionalität, die heute in fast jeder gestalterischen Software enthalten ist. Über einen Menüeintrag bzw. eine Tastenkombination ermöglicht sie es, die eigenen Aktionen in umgekehrter Reihenfolge rückgängig zu machen und meist auch wiederherzustellen.

Solche Interaktionsmuster werden in Katalogen[Tid05] gesammelt, die ein wertvolles Werkzeug dafür darstellen, leicht erlernbare Software zu konzipieren.

3.2. Große Anwendungen

Bei der Gestaltung von Anwendungssoftware ist längst nicht mehr nur der Entwurf einzelner Dialogmasken entscheidend, sondern auch, wie die individuellen Ansichten und Fenster einer Software zusammenpassen und -wirken. Eine sinnvolle Strukturierung der Anwendung ist essenziell, damit Benutzer sich zurechtfinden können.

Im Bereich der Informationsarchitektur wird – bisher vor allem im Hinblick auf Webseiten – an der Strukturierung, Visualisierung und Navigierbarkeit großer Informationsmengen geforscht[RM98]. Viele der Ergebnisse lassen sich auf die Entwicklung von Anwendungssoftware übertragen.

Die Grundfrage lautet: *Ist die Struktur der Anwendung sinnvoll hinsichtlich der Aufgaben und Vorkenntnisse der Benutzer?* Eine Software kann zu einfach oder flach strukturiert sein, so dass sie nicht geeignet ist, auch komplexere Aufgaben zu erfüllen. Genau so kann sie kompliziert und verwirrend strukturiert sein, wenn die Struktur der Software nicht zu den Arbeitsabläufen passt. Darüber hinaus kann die Software ihren Benutzern das Leben schwer machen, indem z.B. unnötige Belastungen für das Kurzzeitgedächtnis verlangt werden wie im Beispiel in Abbildung 6.

Navigation und Suche sind wiederkehrende Probleme, für die vor allem in Webseiten vielfältige Lösungen entwickelt werden, die in Zukunft hoffentlich auch in Anwendungssoftware zum Einsatz kommen können um den Umgang mit großer Software zu erleichtern.

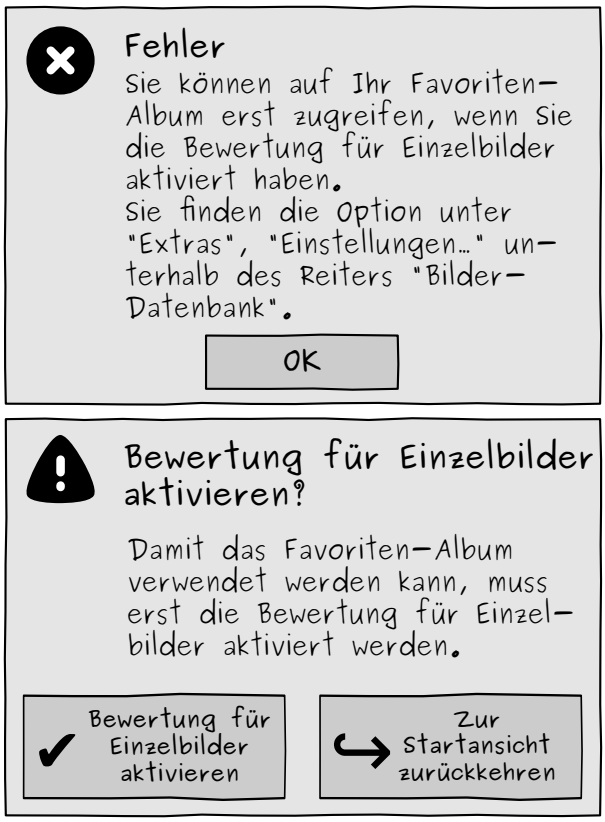

Abbildung 6: Zwei beispielhafte Entwürfe für eine Meldung in einer fiktiven Fotoverwaltungssoftware. Im ersten Entwurf fällt u.A. auf, dass die Navigationsarbeit unnötigerweise auf den Benutzer abgewälzt wird, obwohl die Software zur entsprechenden Stelle der Konfiguration auch direkt zeigen oder die Einstellung vor Ort zugänglich machen könnte wie im zweiten, besseren Entwurf.

4. Methoden und Lösungsansätze

Nachdem die Schwierigkeiten nun ausführlich dargestellt wurden, sollen in diesem Abschnitt einige Lösungsansätze präsentiert werden. Es handelt sich um Beispiele auf verschiedenen Abstraktionsebenen. Die *Ribbon*-Oberfläche ist eine konkrete Implementation, die auf interessanten empirischen Erhebungen basiert. Die adaptiven Oberflächen aus dem zweiten Unterabschnitt sind eine ganze Klasse von Methoden, die die grafische Oberfläche auf bestimmte Weisen verändern. Das danach vorgestellte *Feature Layering* ist eher im Bereich der Entwicklungsmodelle anzusiedeln und dient als Hilfe zur Erstellung benutzergerechter Software bereits vom Schritt der Konzeption an. Im letzten Unterabschnitt wird es schließlich um Lektionen aus dem Game Design gehen, die sich auf der Ebene fundamentaler Interaktionsparadigmen von etablierten Werten absetzen. Insgesamt behandeln die folgenden Unterabschnitte also Themen von aufsteigender Abstraktion.

4.1. Das Ribbon-UI

Wie in Abschnitt 2.1 bereits dargestellt wurde, stieg die Anzahl der Symbolleisten in Microsoft®Word für Windows®mit der wachsenden Funktionalität über die Jahre drastisch an, so dass das Ziel der Symbolleiste – nämlich die wichtigste Funktionalität schnell und übersichtlich zugänglich zu machen – durch dieses Interaktionsmuster nicht länger erfüllt werden konnte.

Die Entwickler entschieden sich dafür, die Interaktion von Grund auf neu zu gestalten und entwarfen die *Ribbon*-Oberfläche (siehe Abbildung 7) um das Menü und die Symbolleisten zu ersetzen. In den Entwurf flossen große Mengen quantiativer Nutzdaten, die von freiwilligen Benutzern erhoben wurden. Auf Basis dieser Nutzdaten wurde entschieden, welche Funktionen wie viel Bildschirmfläche beanspruchen dürfen und wie sie zugänglich gemacht werden.

Die alten Menüs und Symbolleisten stehen nicht länger als Option zur Verfügung, so dass Benutzer auf Erweiterungen von Drittanbietern angewiesen sind oder bei alten Versionen der Software bleiben müssen, wenn sie die traditionelle Oberfläche beibehalten möchten. Vor allem die „Power User“, die mit den alten Menüs eng vertraut sind und sich nun umgewöhnen müssten, sind unzufrieden. Für neue Benutzer scheint die neue Oberfläche allerdings tatsächlich besser geeignet zu sein; die Entwickler selbst bewerten den Ribbon als zukunftsweisendes Erfolgskonzept[Har08].

4.2. Adaptive User Interfaces

Adaptive Schnittstellen sind solche, die ihre eigene Struktur an das Verhalten des individuellen Benutzers anpassen. Die Idee ist, dass während der Benutzung Daten darüber erhoben werden, welche Funktionen verwendet werden, so dass häufiger benötigte Funktionen in der Oberfläche prominenter platziert werden können.

Als vergleichsweise unauffällige Beispiele wären z.B. kommandobasierte Schnittstellen zu nennen, die häufig benutzte Kommandos schon nach den ersten paar Zeichen

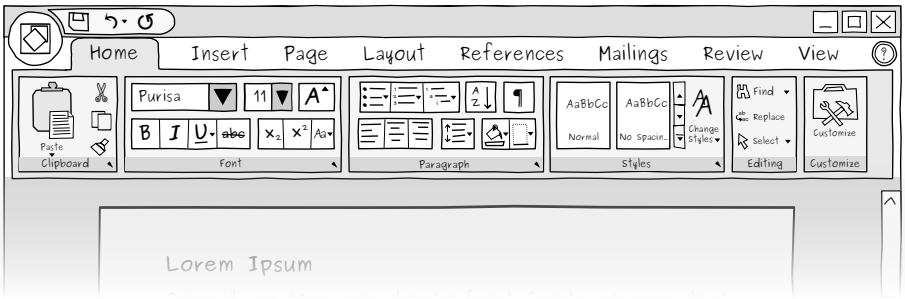

Abbildung 7: Schematische Darstellung des *Ribbon*. Diese Oberfläche kommt in den Versionen 2007 und späteren von Microsoft®Office sowie in anderen aktuellen Microsoft®-Produkten zum Einsatz und ersetzt die Menüleiste und Symbolleisten. Charakteristisch sind etwa verschieden große Buttons je nach (antizipierter) Gebrauchshäufigkeit, Tabs zur Kategorisierung der Funktionen, ausführliche Tooltips und mehr[Har05a].

zur automatischen Vervollständigung vorschlagen. Analog könnte die automatische Vervollständigung beim Verfassen von Textnachrichten auf einem Mobiltelefon ebenfalls als adaptive Schnittstelle gezählt werden, da sie sich meist an in der Vergangenheit geschriebenen Wörtern orientiert. Ein sehr verbreitetes Beispiel für einen etablierten adaptiven UI-Aspekt ist die in fast jeder dokumentbasierten Software enthaltene Liste der zuletzt verwendeten Dateien (siehe Abbildung 8a).

Auffälligere Beispiele sind etwa die Startansicht, die einige moderne Webbrowser nach Programmstart anzeigen, in der die am häufigsten besuchten Webseiten direkt zur Verfügung stehen, oder die adaptiven Menüs, die zeitweise in einigen weit verbreiteten Softwareprodukten zum Einsatz kamen, wegen Benutzbarkeitsproblemen letztlich aber wieder verschwunden sind (Abbildung 8b).

Adaptive Interfaces können den Benutzern unnötige Zeit und Arbeit sparen, die sie sonst mit Navigation verbringen würden. Die große Falle, an der adaptive Konzepte jedoch immer wieder scheitern, ist das Lern- und Gewöhnungsverhalten der Benutzer. Software, die ihre eigene Schnittstelle scheinbar grundlos verändert, verhindert das Erlernen von Abfolgen und Wegen und erschwert Gewöhnung. Aus diesem Grund muss Adaption im Bereich des Interaktionsdesigns sehr sparsam und bewusst eingesetzt werden.

4.3. Feature Layering

Dieser Ansatz liefert eine Hilfestellung zur Beantwortung der Frage, wie Software für Benutzer auf verschiedenem Expertise-Grad zugänglich gemacht werden kann[Coo07].

Die Software wird dabei aus drei archetypischen Perspektiven betrachtet, die in etwa drei Stadien des Lernvorgangs eines Benutzers entsprechen: *Intro*, *Expert* und *Meta*. Es geht nicht darum, die Benutzer in diese drei Gruppen einzuteilen oder harte Grenzen zu ziehen, sondern es soll möglich sein, auf jeder dieser drei Ebenen mit der Software zu

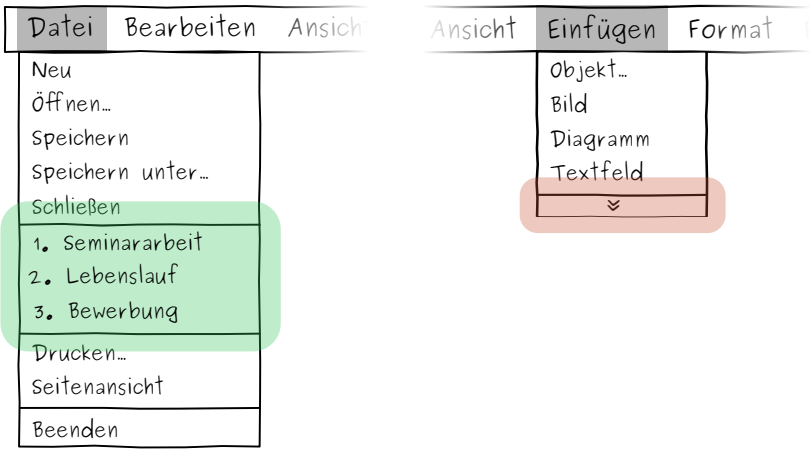

Abbildung 8: (a) Die Liste der zuletzt verwendeten Dokumente wird während der Benutzung der Software ständig verändert. Da die Funktion für die Benutzer leicht verständlich ist, gibt es kaum Akzeptanzprobleme. (b) Die adaptiven Menüs, in denen selten benutzte Einträge automatisch aus- und bei Bedarf per Mausklick wieder eingeblendet werden konnten, sind weitgehend wieder verschwunden. Das Verhalten der Software war für die Benutzer undurchschaubar und verwirrend, so dass das Feature oftmals direkt deaktiviert wurde.

arbeiten (siehe Abbildung 9). Das Modell trägt der Tatsache Rechnung, dass Benutzer eine Software im Laufe der Zeit erlernen, wobei sich die Ansprüche signifikant wandeln.

- **Intro** ist das Level, auf dem ein Benutzer sich befindet, wenn er die Software noch gar nicht oder nur selten verwendet hat. Auf diesem Level ist es entscheidend, die Bildung von mentalen Modellen zu erleichtern, z.B. indem auf allgemein bekannte Muster und Metaphern zurückgegriffen wird. Eine Tabellenkalkulation erlaubt die Arbeit auf dem Intro-Level, indem sie es gestattet, unter Verzicht auf weiterführende Features auch weitgehend genau so arbeiten zu können wie auf dem Papier.

- **Expert** ist das Level, auf dem ein Benutzer die Software als Werkzeug ohne große Schwierigkeiten verwenden kann. Einfache Aufgaben können ohne Denkaufwand gelöst werden, Prozesse sind internalisiert. Auf diesem Level müssen Möglichkeiten zur Steigerung von Effizienz und Effektivität geboten werden. Weiterführende Konzepte (wie Referenzsemantiken oder Stapelverarbeitung) müssen zur Verfügung stehen. In einer Tabellenkalkulation wären Verweise auf andere Zellen oder automatisch berechnete Formeln diesem Level zuzuordnen.

- **Meta** ist das Level, auf dem ein Benutzer alle oder nahezu alle Möglichkeiten der Software gut kennt. Zur weiteren Steigerung der Produktivität können Möglichkeiten und Werkzeuge helfen, die Software zu automatisieren und den Funktionsumfang zu erweitern, wie etwa die Möglichkeit der schnellen Entwicklung von

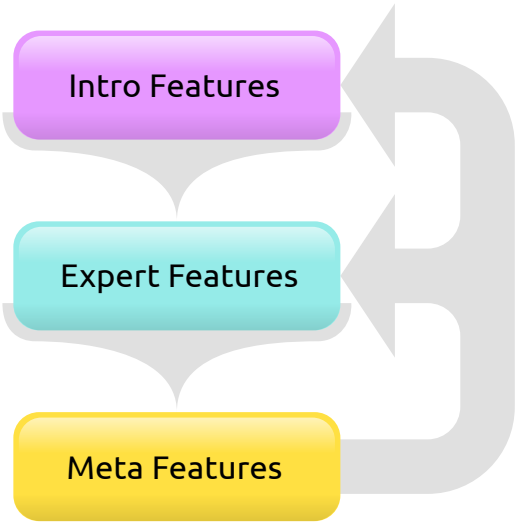

Abbildung 9: Ein Modell zur Entwicklung von Software, die sowohl mit geringer als auch mit steigender Expertise benutzbar bleibt, wobei Benutzer sich gegenseitig beeinflussen können (vgl. [Coo07]).

Plug-Ins oder Makros. Moderne Tabellenkalkulationen ermöglichen bspw. die Automatisierung oder gar die Erstellung von eigenen interaktiven Modulen auf Basis von Skriptsprachen.

Insbesondere ist es förderlich, wenn Meta-Benutzer die Möglichkeit haben, ihre Fortschritte und selbst erstellten Inhalte Benutzern auf unteren Leveln zugänglich zu machen. So gibt es nicht nur für jene Benutzer ebenfalls einen kurzfristigen Produktivitätsgewinn, sondern auch langfristig kann die Effizienz gesteigert werden, wenn durch die Vorbildfunktion die Motivation gesteigert wird, sich selbst genauer mit der Software auseinanderzusetzen.

4.4. Motivationstechniken aus dem Bereich Game Design

Im Bereich der Usability wurde lange Zeit das Ziel verfolgt, Software möglichst „glatt“ und transparent benutzbar zu machen, so dass die Benutzer durch die Software möglichst wenig gestört oder am Arbeiten gehindert werden. Erst seit vergleichsweise kurzer Zeit spielt etwas wie „Emotional Design“ eine Rolle; man entdeckt, dass die Verwendung von Software sogar Spaß machen kann[Coo06].

Mit der Vereinigung von Softwarebenutzung und Spaß hat unterdessen eine ganz andere Entwicklergemeinschaft über Jahrzehnte Erfahrungen gesammelt: Die Erkenntnisse der Spieleentwickler entpuppen sich als wertvolle Werkzeuge, Software zu entwickeln, die motiviert und Spaß macht.

Ein besonderes Augenmerk muss hierbei auf Lernprozesse gerichtet werden. Das Erlernen von Anwendungssoftware wird von den meisten Benutzern als Arbeit, als Last

empfunden. Dagegen passieren beim Spielen erstaunliche Lernprozesse, ohne dass es dem Spieler negativ auffällt. In jedem Spiel müssen Taktiken erlernt, Mechaniken verinnerlicht und Welten erkundet werden. Gute Spiele schaffen es, dass die Spieler daran Spaß haben, indem sie mit der richtigen Mischung aus Herausforderung und positiver Bestätigung nach Erfolgen aufwarten.

Sicherlich lassen sich nicht alle Aspekte aus der Spieleentwicklung direkt auf Anwendungssoftware übertragen. In Spielen ist es z.B. üblich (und nötig) den Spieler dafür zu bestrafen, wenn er eine geringe Leistung bietet, etwa durch Vorenthalten von weiteren Inhalten. Dies würde in einer typischen Anwendungssoftware als schlichte Bevormundung empfunden werden. Andere Aspekte lassen sich jedoch gut übertragen und können dabei helfen, die Lernerfahrung angenehm zu gestalten:

- **Positive Bestätigung:** Benutzer sollen für erbrachte Leistungen belohnt werden, mindestens durch einen Effizienzgewinn.
- **Sanfte Lernkurve:** Es muss stets mindestens eine offensichtliche Möglichkeit geben, etwas Neues zu erlernen. Große Herausforderungen sind so zu zerlegen, dass die Anforderungen weder trivial noch unbewältigbar sind.
- **Soziale Komponente:** Eine noch recht neue Entwicklung auch im Spiele-Bereich. Benutzer sollen die Möglichkeit haben, die eigenen Fähigkeiten untereinander zu vergleichen oder kollaborativ zu arbeiten.

Als Beispiel dafür, wie eine ansonsten als anstrengend und lästig empfundene Tätigkeit in ein Spiel verpackt werden kann, kann *Wii Fit* von Nintendo genannt werden[Coo08]. Dort wird das Abnehmen zur spaßigen Herausforderung: Die Software verteilt Belohnungen, schreckt aber auch nicht davor zurück, den Spieler bei nachlassender Leistung zu ermahnen. Das abstrakte Ziel „Abnehmen“ wird durch das In-Aussicht-Stellen von immer neuen Belohnungen und neuen freischaltbaren Mini-Spielen zu einer Folge von konreten und erreichbaren Zielen.

5. Fazit

Die Beobachtung, dass die Komplexität unserer alltäglichen Software vermutlich weiter steigen wird, ist ernüchternd, sollte aber kein Grund zur Verzweiflung sein. „Vereinfachung“ kann ein Mittel sein, um Software aufgabenangemessen und verständlich zu machen, sie ist jedoch bei weitem nicht das einzige.

Softwareentwickler und UI-Designer stehen vor der Aufgabe, die Komplexität ihrer Anwendungen kritisch zu betrachten und zu bewerten. Nicht jede Komplexität ist nötig, aber auch nicht jede ist verzichtbar. Eine radikalisierte „Einfachheits“-Bewegung, die nach weniger Buttons und weniger Funktionen schreit, ist voraussichtlich allenfalls ein kurzlebiger Trend.

Es gibt diverse Methoden, Komplexität von Software erschließbar und benutzergerecht zu gestalten. Einige davon wurden vorgestellt. Ein Wundermittel für jedes denkbare Problem gibt es nicht. Weiterhin ist die Kreativität und das Denkvermögen der Softwareentwickler und Oberflächendesigner gefragt, wenn es darum geht, einen guten Kompromiss zu finden.

Den Schlusstrich und die Zukunftsperspektive bilden an dieser Stelle die Worte von Don Norman[Nor10]:

> Complexity is here to stay. The frame of mind is essential: learn to accept complexity, but also learn to conquer it. (...) The technologies we use must match the complexity of the world: technological complexity is unavoidable.

Literatur

[Abi11] *AbiWord – Word Processing for Everyone.* http://www.abisource.com/. Version: 2011. – [Online; 17. Januar 2011]

[Coo06] Cook, Daniel: *Building fun into your software designs.* http://www.lostgarden.com/2006/12/building-fun-into-your-software-designs.html. Version: 2006. – [Online; 10. Dezember 2006]

[Coo07] Cook, Daniel: *One Billion Buttons Please: Should we build features for experts or newbies?* http://www.lostgarden.com/2007/02/one-billion-buttons-please-should-we.html. Version: 2007. – [Online; 3. Februar 2007]

[Coo08] Cook, Daniel: *What actitivies can be turned into games?* http://www.lostgarden.com/2008/06/what-actitivies-that-can-be-turned-into.html. Version: 2008. – [Online; 14. Juni 2008]

[Goo10] *Our favorite Docs things – 2010 Year in Review.* http://googledocs.blogspot.com/2010/12/our-favorite-docs-things-2010-year-in.html. Version: 2010. – [Online; 22. Dezember 2010]

[Har05a] Harris, Jensen: *Enter the Ribbon.* http://blogs.msdn.com/b/jensenh/archive/2005/09/14/467126.aspx. Version: 2005. – [Online; 15. September 2005]

[Har05b] Harris, Jensen: *Ye Olde Museum Of Office Past (Why the UI, Part 2).* http://blogs.msdn.com/b/jensenh/archive/2005/10/03/476412.aspx. Version: 2005. – [Online; 3. Oktober 2005]

[Har08] Harris, Jensen: *The Story of the Ribbon.* http://blogs.msdn.com/b/jensenh/archive/2008/03/12/the-story-of-the-ribbon.aspx. Version: 2008. – [Online; 12. März 2008]

[Her06] Herczeg, Michael: *Interaktionsdesign.* Oldenbourg, 2006

[Kin07] Kinard, Marcel: *I/T needs more ease-of-use and less complexity. More features doesn't always solve the problem.* http://www.marcelk.org/archives/21. Version: 2007. – [Online; 6. Februar 2007]

[Kru05] Krug, Steve: *Don't Make Me Think!: A Common Sense Approach to Web Usability.* 2nd ed. New Riders, 2005

[Mac11] *Mac App Store von Apple eröffnet.* http://www.apple.com/de/pr/library/2011/01/06macappstore.html. Version: 2011. – [Online; 6. Januar 2011]

[Nor10] Norman, Donald A.: *Living with Complexity.* The MIT Press, 2010

[RM98] Rosenfeld, Louis ; Morville, Peter: *Information architecture for the World Wide Web*. Sebastopol, CA, USA : O'Reilly & Associates, Inc., 1998. – ISBN 1–56592–282–4

[Spo06] Spolsky, Joel: *Simplicity*. `http://www.joelonsoftware.com/items/2006/12/09.html`. Version: 2006. – [Online; 9. Dezember 2006]

[Tid05] Tidwell, Jenifer: *Designing Interfaces*. O'Reilly Media, 2005

[Ubu11] *Ubuntu Wiki: SoftwareCenter*. `https://wiki.ubuntu.com/SoftwareCenter`. Version: 2011. – [Online; 17. Januar 2011]

[Wir11] *Definition: Anwendungsprogramm / Anwendungssoftware – Wirtschaftslexikon Gabler*. `http://wirtschaftslexikon.gabler.de/Archiv/74942/anwendungsprogramm-v6.html`. Version: 2011. – [Online; 6. Februar 2011]